果壳中的C#

C# 5.0 权威指南

C# 5.0 IN A NUTSHELL (*The Definitive Reference*)

O'REILLY®

[美] *Joseph Albahari & Ben Albahari* 著
陈昇　管学理　曾少宁　杨庆川　译

中国水利水电出版社
www.waterpub.com.cn

果壳中的C#——C# 5.0权威指南

本书将引领您达到C#的一个新高度

- 全面覆盖语法、数据类型、变量等基础知识
- 深入覆盖不安全代码、类型转换、预处理理指令等高级主题；并发、异步、代码契约、动态编程、安全性、COM互操作性等技术；LINQ相关技术；.NET相关的XML、集合、I/O、网络、存储管理、反射、属性、安全及本地互操作性等技术
- 知识点与案例无缝配合，极大降低学习难度

“本书是我放在桌面上的少数参考书之一。我推荐你阅读这本书。”

——Scott Guthrie
Microsoft公司副总裁

“无论你是初学者或一个专家，只要你想提高你的最新异步编程技术，这本书里就有你需要的内容。”

——Eric Lippert
Microsoft资深软件设计工程师

我们都被束缚在果壳中，却仍自以为是宇宙那无限空间之王

——霍金（引自莎士比亚）

O'REILLY®
oreilly.com.cn

上架建议：编程语言与程序设计/C#

O'Reilly Media, Inc.授权中国水利水电出版社出版

此简体中文版仅限于在中华人民共和国境内（但不允许在中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

定价：118.00元

果壳中的C#——C# 5.0权威指南

[美] Joseph Albahari　Ben Albahari 著
陈昇　管学理　曾少宁　杨庆川 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权中国水利水电出版社出版

中国水利水电出版社
www.waterpub.com.cn

内 容 提 要

本书是一本C# 5.0的权威技术指南，也是第一本中文版C# 5.0的学习资料。本书通过26章的内容，系统、全面、细致地讲解了C# 5.0从基础知识到各种高级特性的命令、语法和用法。本书的讲解深入浅出，同时为每一个知识点都专门设计了贴切、简单、易懂的学习案例，从而可以帮助读者准确地理解知识点的含义并快速地学以致用。本书与之前的C# 4.0版本相比，还新增了丰富的并发、异步、动态编程、代码精练、安全、COM交互等高级特性相关的内容。

本书还融汇了作者多年在软件开发及C#方面的研究及其实践经验，非常适合作为C#技术的一本通自学教程，亦是一本中高级C#技术人员不可多得的必备工具书。

©2012 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Waterpower Press, 2013. Authorized translation of the English edition, 2012 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由O'Reilly Media, Inc. 出版2012。

简体中文版由中国水利水电出版社出版 2013。英文原版的翻译得到O'Reilly Media, Inc.的授权。此简体中文版的出版和销售得到出版权和销售权的所有者 —— O'Reilly Media, Inc.的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

北京市版权局著作权合同登记号：图字：01-2013-4993号

图书在版编目（CIP）数据

果壳中的C#：C# 5.0权威指南 / （美）阿坝哈瑞，（美）阿坝哈瑞著；陈昇，管学理，曾少宁，杨庆川译. --北京：中国水利水电出版社，2013.8

书名原文：C# 5.0 in a nutshell, 5e

ISBN 978-7-5170-1084-5

Ⅰ.①果… Ⅱ.①阿… ②阿… ③陈… ④管… ⑤曾… ⑥杨… Ⅲ.①C语言—程序设计 Ⅳ.①TP312

中国版本图书馆CIP数据核字(2013)第171436号

策划编辑：周春元　责任编辑：李　炎　加工编辑：李　燕　封面设计：李　佳

书　名	果壳中的C#——C# 5.0权威指南
作　者	[美] Joseph Albahari　Ben Albahari　著 陈昇　管学理　曾少宁　杨庆川　译
出版发行	中国水利水电出版社 （北京市海淀区玉渊潭南路1号D座 100038） 网址：www.waterpub.com.cn E-mail：mchannel@263.net（万水） sales@waterpub.com.cn 电话：（010）68367658（发行部）、82562819（万水）
经　售	北京科水图书销售中心（零售） 电话：（010）88383994、63202643、68545874 全国各地新华书店和相关出版物销售网点
排　版	北京万水电子信息有限公司
印　刷	北京蓝空印刷厂
规　格	185mm×240mm　16开本　56印张　1950千字
版　次	2013年8月第1版　2013年8月第1次印刷
印　数	0001—2000册
定　价	118.00元

凡购买我社图书，如有缺页、倒页、脱页的，本社发行部负责调换

版权所有・侵权必究

O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些"细微的信号"来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬。

O'Reilly为软件开发人员带来革命性的"动物书"；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

"O'Reilly Radar博客有口皆碑。"

——**Wired**

"O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。"

——**Business 2.0**

"O'Reilly Conference是聚集关键思想领袖的绝对典范。"

——**CRN**

"一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。"

——**Irish Times**

"Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照Yogi Berra的建议去做了：'如果你在路上遇到岔路口，走小路（岔路）。'回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。"

——**Linux Journal**

前言

C# 5.0是微软旗舰编程语言的第4次重大升级，大大提升了C#语言的灵活性与功能。一方面，它实现了一些高级抽象，如查询表达式和异步延续；另一方面，它又通过自定义类型值和可选指针等设计实现了一些底层功能。

这部分增加的特性尤其值得学习。虽然诸如Microsoft的IntelliSense工具和各种在线参考文档在帮助你完成工作方面是非常好用的，但是它们需要由现有的一些概念知识来支撑。本书以简明统一的方式（而非繁杂冗长的介绍）准确到位地阐述了这些知识集。

本书是完全按照概念和用例组织的，因此无论是按顺序阅读还是随意浏览都可以。虽然只要求具备基本的背景知识，但它还是有一定的深度，因此比较适合中高级水平的读者阅读。

本书内容涵盖了C#、CLR和Framework程序集。我们之所以做出这样的选择，是为了重点讲解一些较难理解的主题，如并发性、安全性和应用程序域，同时不影响深度或可读性。C# 5.0及相关Framework的新特性已经被标注清楚，因此也可以将本书作为C# 4.0参考书使用。

目标读者

本书主要针对中高级开发人员。不要求读者具备C#知识，但是需要有一些普通编程经验。对于初学者，本书能够补充教程类编程介绍书籍，但不能替代教程类书籍。

熟悉C# 4.0的读者会发现，我们重写了关于并发性的小节，其中包括深入介绍C# 5.0的异步函数及其相关类型，并且还介绍了异步编程的原则，以及它如何能够提供效率和线程安全性。

本书是各种介绍实用技术图书的理想伴侣，如WPF、ASP.NET或WCF。这些书籍所省略的语言与.NET Framework方面的内容，本书都进行了详细介绍，反之亦然。

这本书并不会详细介绍每一种.NET Framework技术。此外，这本书也不会介绍平板电脑或Windows Phone开发的专用API。

本书的结构

本书前三章集中介绍C#语言，先介绍语法、类型和变量，然后介绍一些高级特性，如不安全代码和预处理指令。如果你是初学者，应该循序渐进地阅读这些章节。

其余各章的内容涵盖核心.NET Framework，包括LINQ、XML、集合、I/O与网络、内存管理、反射、动态编程、属性、安全性、并发、应用域和原生互操作性等主题。除了第6章和第7章之外，你

可以按任意顺序阅读，因为这两章是后续主题的基础。关于LINQ的三章内容最好也按顺序阅读。

一些章节要求读者理解并发的基础知识，这部分知识将在第14章介绍。

使用本书所需的其他材料

本书的例子需要使用C# 5.0编译器和微软.NET Framework 4.5。此外，微软的.NET文档可以帮助查找各个类型及其成员（在线版本）。

虽然在记事本中可以编写源代码和从命令执行编译器，但是为了提高效率，最好使用一个代码编辑器即时测试各个代码版本，并且使用集成开发环境（IDE）生成可执行程序和库。

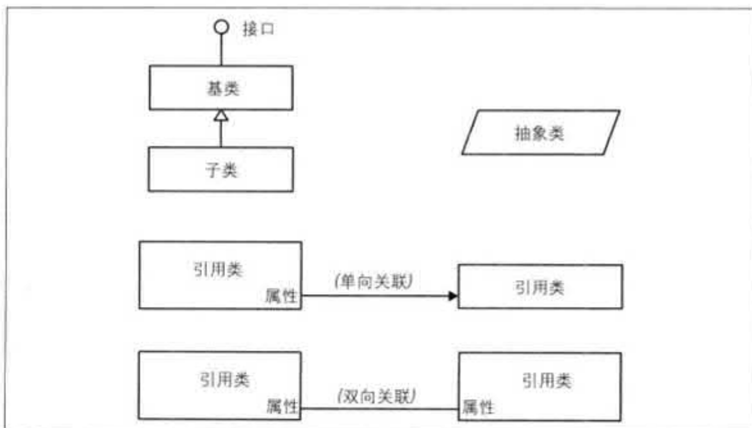
推荐从www.linqpad.net下载LINQPad 4.40或更高版本（免费）作为代码编辑器。LINQPad完全支持C# 5.0，并且由C# 5.0的作者之一维护。

对于IDE，建议下载Microsoft Visual Studio 2012：除了免费的简洁版，其他版本都适合本书介绍的内容。

提示： 第2～10章及并发性、并行编程和动态编程等章节所列代码清单都是可交互（可编辑）的LINQPad示例。

本书中使用的约定

本书使用简单的UML符号来说明类之间的关系，如图P-1所示。斜矩形表示抽象类；圆圈表示一个接口。带空心三角形的线条表示继承，其中三角形指向基类。带箭头的线条表示单向关联；不带箭头的线条表示双向关联。



图P-1：示例图

本书还使用以下的排版约定：

斜体（*Italic*）

  表示URI、文件名、目录和应该由用户提供的值所替代的文本

等宽字体（`Constant Width`）

表示C#代码、关键字与标识符以及程序输出

等宽粗体（**`Constant Width Bold`**）

突出显示部分代码

使用示例代码

本书的作用是帮助你完成工作。一般而言，你可能会在程序和文档中使用本书所提供的代码。除非必须复制大部分代码，否则不需要联系我们获得授权。例如，你不需要授权就可以使用本书的多个代码段来编写程序；销售或分发O'Reilly书籍中的示例代码CD-ROM也不需要授权；引用本书及其示例代码来回答某个问题也不需要授权；将本书的大量示例应用到你的产品文档中也不需要授权。

我们欢迎你标注内容出处，但不强制要求。一般的标注通常包括书名、作者、出版社和ISBN。例如："C# 5.0技术手册，作者：Joseph Albahari和Ben Albahari。版权所有 2010 Joseph Albahari和Ben Albahari，ISBN：978-1-449-32010-2"。

如果你认为你的代码示例使用方式超出一般用途或超出了此处的授权范围，请随时与我们联系：*permissions@oreilly.com*。

Safari® Books Online

Safari Books Online是一个随需供应的数字图书馆，你可以轻松搜索到7500多种技术和创意参考图书与视频，可以快速帮助你找到问题答案。

订阅后，你可以在图书馆中在线阅读任何页面和观看任何视频。你还可以在手机和移动设备上阅读这些图书。你可以查看未出版的新书，唯一地访问仍在编写中的书稿以及给作者发送反馈信息。你还可以复制和粘贴代码示例、整理收藏夹、下载章节、收藏关键章节、编写注解、打印内容页，以及使用无数其他可以节约时间的特性。

O'Reilly Media已经将本书上传到Safari Books Online服务。想要获得本书或O'Reilly及其他出版商的类似书籍的完整电子版，请先免费注册一个账号：*http://my.safaribooksonline.com*。

联系我们

对于本书，如果有任何意见或疑问，请按照以下地址联系本书出版商：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）
奥莱利技术咨询（北京）有限公司

本书也有相关的网页，我们在上面列出了源代码、范例以及其他一些信息。你可以访问：

http://www.albahari.com/nutshell/（英文版）

对本书做出评论或者询问技术问题，请发送E-mail至：

bookquestions@oreilly.com

希望获得关于本书、会议、资源中心和O'Reilly网络的更多信息，请访问：

http://www.oreilly.com
http://www.oreilly.com.cn

致谢

Joseph Albahari

首先，我要感谢我的兄弟和合著者Ben Albahari，感谢他在最初说服我参与这项后来非常成功的项目。我非常享受与Ben一起探究难题的过程：他不仅与我一样勇于向传统观点提出质疑，而且都具有刨根问底的精神。

我最希望感谢的还有一些优秀的技术审阅者。首先是来自Microsoft的校阅者，Stephen Toub（并行编程团队）和Chris Burrows（C#编译器团队）提供的大量信息显著地增强了关于并发性、动态编程和C#语言等章节的内容。从CLR团队，我收获了来自Shawn Farkas、Brian Grunkemeyer、Maoni Stephens和David DeWinter关于安全性和内存管理方面的非常宝贵的信息。

我极力向读者推荐Jon Skeet（《C# in Depth》的作者以及堆栈溢出的专家），他的许多宝贵建议丰富了许多章节的内容（虽然任职于谷歌公司，但是我们尊重他的选择！）。我也同样感激C# MVP Nicholas Paldino敏锐的眼光，他发现了其他工作人员未发现的一些错误和疏忽。我同时还要感谢另外两位C# MVP：Mitch Wheat和Brian Peek，以及本书所基于的3.0版本的校阅者。这里面包括了前面提到的Nicholas Paldino，他将其博大渊深的知识应用到了本书的大多数章节，以及Krzysztof Cwalina、Matt Warren、Joel Pobar、Glyn Griffiths、Ion Vasilian、Brad Abrams、Sam Gentile和Adam Nathan。

最后，我还要感谢O'Reilly团队，包括行动迅速及非常高效的编辑Laurel Ruma、宣传人员Kathryn Barrett、文字编辑Audrey Doyle以及我的家人Miri和Sonia。

Ben Albahari

由于我的兄弟在我之前写下了他的感言，他所表达的大多数内容也正是我的肺腑之言：事实上，当我们还是孩子的时候，就已经开始编写程序了（我们共用一台Apple IIe；他编写他自己的操作系统，而我则是在编写我的Hangman），因此，现在我们能一起撰写这些书籍是一件非常惬意的事。我希望我们在此书中所浓缩的经验可以丰富读者们的编程经验。

同时，我还要感谢我之前在Microsoft工作时的同事。很多人在那里工作，他们不仅智商高而且情商更高，我怀念与他们共事的时光。我还要特别感谢Brian Beckman，从他的身上我学到了很多。

目录

前言

目标读者 1
本书的结构 1
使用本书所需的其他材料 2
本书中使用的约定 2
使用示例代码 3
联系我们 3
Safari® Books Online 4
致谢 4

第1章 C#和.NET Framework简介 1

1.1 面向对象 1
1.2 类型安全性 1
1.3 内存管理 2
1.4 平台支持 2
1.5 C#与CLR的关系 2
1.6 CLR和.NET Framework 3
1.7 C#与Windows Runtime 4
1.8 C# 5.0新特性 5
1.9 C# 4.0新特性 5
1.10 C# 3.0新特性 5

第2章 C#语言基础 7

2.1 第一个C#程序 7
2.2 语法 9
2.3 类型基础 11

2.4 数值类型 19
2.5 布尔类型和运算符 25
2.6 字符串和字符 27
2.7 数组 29
2.8 变量和参数 32
2.9 表达式和运算符 40
2.10 语句 43
2.11 命名空间 51

第3章 在C#中创建类 57

3.1 类 57
3.2 继承 69
3.3 object类型 76
3.4 结构体 80
3.5 访问权限修饰符 81
3.6 接口 83
3.7 枚举类型 87
3.8 嵌套类型 91
3.9 泛化 92

第4章 C#高级特性 103

4.1 委托 103
4.2 事件 111
4.3 Lambda表达式 117
4.4 匿名方法 120
4.5 try语句和异常 121
4.6 枚举类型和迭代 128
4.7 可空类型 132
4.8 运算符重载 137
4.9 扩展方法 140
4.10 匿名类型 143
4.11 动态绑定 144
4.12 属性 151
4.13 调用者信息属性（C# 5） 152
4.14 不安全代码和指针 154

4.15 预处理指令 157
4.16 XML文档 159

第5章 框架概述 163

5.1 CLR和核心框架 165
5.2 应用技术 168

第6章 框架基础 174

6.1 字符串与文本处理 174
6.2 日期和时间 185
6.3 日期与时区 191
6.4 标准格式字符串与解析标记 202
6.5 其他转换机制 208
6.6 全球化 211
6.7 操作数字 212
6.8 枚举类型 216
6.9 元组 219
6.10 Guid结构体 220
6.11 等值比较 220
6.12 顺序比较 229
6.13 实用类 232

第7章 集合 235

7.1 枚举 235
7.2 ICollection和IList接口 242
7.3 Array类 245
7.4 复制 251
7.5 List、Queue、Stack和Set 252
7.6 字典 259
7.7 可定制的集合和委托 264
7.8 等值和顺序插入 270

第8章 LINQ查询 277

8.1 入门 277
8.2 运算符流语法 279
8.3 查询表达式 285

8.4 延迟执行 289
8.5 子查询 295
8.6 LINQ构造方式 298
8.7 映射策略 301
8.8 解释型的查询 303
8.9 LINQ to SQL 和 Entity Framework 309
8.10 查询表达式的创建 323

第9章 LINQ运算符 328

9.1 概述 329
9.2 筛选 332
9.3 映射 336
9.4 连接 347
9.5 Zip 运算符 355
9.6 排序 355
9.7 Grouping 358
9.8 集合运算符 361
9.9 转换方法 363
9.10 元素运算符 365
9.11 聚合方法 367
9.12 数量词 372
9.13 生成集合的方法 373

第10章 LINQ to XML 375

10.1 架构概述 375
10.2 X-DOM概述 376
10.3 实例化X-DOM 379
10.4 指定内容 380
10.5 导航和查询 381
10.6 更新X-DOM 386
10.7 使用Value 389
10.8 文档和声明 391
10.9 名称和命名空间 394
10.10 注解 400
10.11 将数据映射到X-DOM 400

第11章 其他XML技术 407
11.1 XmlReader 407
11.2 XmlWriter 415
11.3 使用XmlReader/XmlWriter的模式 417
11.4 XmlDocument 421
11.5 XPath 424
11.6 XSD和模式验证 428
11.7 XSLT 431

第12章 销毁和垃圾回收 432
12.1 IDisposable接口、Dispose方法和Close方法 432
12.2 自动垃圾回收 436
12.3 终止器 438
12.4 垃圾回收器如何工作 442
12.5 托管内存泄露 445
12.6 弱引用 448

第13章 诊断和代码契约 452
13.1 条件编译 452
13.2 Debug和Trace类 455
13.3 代码契约概述 458
13.4 先决条件 461
13.5 后置条件 465
13.6 断言和对象不变式 467
13.7 接口和抽象方法中的契约 468
13.8 处理契约错误 469
13.9 选择性执行契约 471
13.10 静态契约检查 472
13.11 调整器集成 473
13.12 进程和处理线程 474
13.13 StackTrace和StackFrame类 475
13.14 Windows事件日志 476
13.15 性能计数器 478
13.16 Stopwatch类 483

第14章 并发与异步 484
14.1 简介 484

14.2 线程处理......485
14.3 任务......498
14.4 异步原则......506
14.5 C# 5.0的异步函数......510
14.6 异步模式......523
14.7 旧模式......530

第15章 流与I/O......533

15.1 流体系结构......533
15.2 使用流......534
15.3 流适配器......546
15.4 压缩流......553
15.5 操作Zip文件......555
15.6 文件与目录操作......555
15.7 Windows Runtime中的文件输入/输出......565
15.8 内存映射文件......567
15.9 隔离存储区......569

第16章 网络......575

16.1 网络体系结构......575
16.2 地址与端口......577
16.3 URI......578
16.4 客户端类......579
16.5 HTTP访问......591
16.6 编写HTTP服务器......596
16.7 使用FTP......599
16.8 使用DNS......600
16.9 通过SmtpClient发送邮件......601
16.10 使用TCP......602
16.11 使用TCP接收POP3邮件......605
16.12 在Windows Runtime中建立TCP连接......606

第17章 序列化......608

17.1 序列化概念......608
17.2 数据契约的序列化......611
17.3 数据契约与集合......620

17.4 扩展数据契约 622
17.5 二进制序列化器 625
17.6 二进制序列化属性 627
17.7 使用ISerializable进行二进制序列化 629
17.8 XML序列化 632

第18章 程序集 641

18.1 程序集包含的内容 641
18.2 强名称和程序集签名 645
18.3 程序集名称 647
18.4 Authenticode签名 649
18.5 全局程序集高速缓存 652
18.6 资源和卫星程序集 654
18.7 解析和加载程序集 661
18.8 在基础文件夹外部署程序集 665
18.9 打包单个可执行文件 667
18.10 处理未引用的程序集 668

第19章 反射和元数据 670

19.1 反射和激活类型 670
19.2 反射和调用成员 677
19.3 反射程序集 688
19.4 使用属性 689
19.5 动态生成代码 694
19.6 发出程序集和类型 700
19.7 发出类型成员 703
19.8 发出泛型方法和类型 708
19.9 复杂的发出目标 710
19.10 解析IL 713
19.11 编写反编译器 713

第20章 动态编程 718

20.1 动态语言运行时 718
20.2 数字类型统一 719
20.3 动态成员重载解决方案 720

20.4 实现动态对象 726
20.5 通过动态语言交互操作 729

第21章 安全 731

21.1 权限 731
21.2 代码访问安全（CAS） 734
21.3 允许部分可信的调用程序 737
21.4 CLR 4.0中的透明模型 739
21.5 沙箱化程序集 746
21.6 操作系统安全 749
21.7 身份和角色安全 751
21.8 加密综述 752
21.9 Windows数据保护 753
21.10 散列法 754
21.11 对称加密 755
21.12 公共密钥加密和签名 759

第22章 高级线程 763

22.1 同步概述 763
22.2 排他锁 764
22.3 锁与线程安全性 771
22.4 非排他锁 775
22.5 使用事件等待处理器发送信号 780
22.6 Barrier类 787
22.7 延后初始化 788
22.8 线程本地存储 790
22.9 Interrupt和Abort 792
22.10 Suspend和Resume 793
22.11 定时器 793

第23章 并行编程 797

23.1 PFX 797
23.2 PLINQ 799
23.3 Parallel类 810
23.4 任务并行 816

23.5 处理AggregateException异常 825
23.6 并发集合 827
23.7 BlockingCollection<T> 829

第24章 应用域 833

24.1 应用域架构 833
24.2 创建和销毁应用域 833
24.3 多应用域的使用 836
24.4 DoCallBack的应用 837
24.5 应用域的监视 838
24.6 应用域和线程 838
24.7 应用域间通信 839

第25章 本地化和COM组件交互 844

25.1 调用本地库 844
25.2 类型封送 845
25.3 非托管代码的回调函数 847
25.4 模拟C共用体 848
25.5 内存共享 849
25.6 映射结构体到非托管内存区 851
25.7 COM交互 854
25.8 在C#中调用COM对象 856
25.9 内嵌互操作类型 859
25.10 主互操作程序集 859
25.11 COM中调用C#对象 860

第26章 正则表达式 861

26.1 正则表达式基础 861
26.2 量词 865
26.3 零宽度断言 866
26.4 分组 869
26.5 文本替换和拆分 870
26.6 正则表达式实例 871
26.7 正则表达式语言参考 874

第1章

C#和.NET Framework简介

C#是一种通用的类型安全且面向对象的编程语言。这种语言的目标是提高程序员的生产力，为此，需要在简单性、可表达性和性能之间实现平衡。C#语言的首席架构师从第一个版本开始就是Anders Hejlsberg（Turbo Pascal的发明者和Delphi架构师）。C#语言与平台无关，但是它能够很好地与Microsoft .NET Framework协同工作。

1.1 面向对象

C#实现了面向对象编程的广泛特性，包括封装、继承和多态。封装表示在对象周围创建一个边界，将它的外部（公开）行为与内部（私有）实现细节隔离。C#在面向对象方面的特性包括：

统一的类型系统

C#中的基础构建块是一种被称为类型的数据与函数的封装单元。C#有一个统一的类型系统，其中所有类型最终都共享一个公共的基类。这意味着所有的类型，不管它们是表示业务对象，或者像数字等基本类型，都共享相同的基本功能集。例如，任何类型都可以通过调用它的ToString方法转换为一个字符串。

类与接口

在纯粹的面向对象泛型中，唯一的类型就是类。但是C#中还有其他几种类型，其中一种是接口（类似于Java中的接口）。接口与类相似，但它只是某种类型的定义，而不是实现。在需要使用多继承时，它是非常有用的（与C++和Eiffel等语言不同，C#不支持类的多继承）。

属性、方法与事件

在纯粹的面向对象泛型中，所有函数都是方法（Smalltalk中就是这样）。在C#中，方法只是一种函数成员，也包含一些属性和事件以及其他组成部分。属性是封装了一部分对象状态的函数成员，如按钮的颜色或标签的文本。事件是简化对象状态变化处理的函数成员。

1.2 类型安全性

C#首先是一种类型安全的语言，这意味着类型只能够通过它们定义的协议进行交互，从而保证每一种类型的内部一致性。例如，C#不允许将字符串类型作为整型进行处理。

更具体地说，C#支持静态类型化，这意味着这种语言会在编译时执行静态类型安全性检查。另外一

种是动态类型安全性，.NET CLR在运行时执行动态类型安全性检查。

静态类型化能够在程序运行之前去除大量的错误。它将大量的运行时单元测试转移到编译器中，验证程序中所有类型之间都是相互适合的。这样大型程序就更容易管理、更具可预测性和健壮性。而且，静态类型化使一些诸如Visual Studio 的IntelliSense等工具有助于编写程序，因为它知道某个特定变量的类型是什么，因此也知道能够调用哪些方法来处理这个变量。

提示： C#允许部分代码通过新的`dynamic`关键字来动态指定类型。然而，C#在大多数情况下仍然是一种静态类型化语言。

C#之所以被称为一种强类型语言，是因为它的类型规则（以静态或动态的方法执行）是非常严格的。例如，不能够使用一个浮点型参数来调用一个定义时接收整数的函数，除非显式地将这个浮点数转换为整数。这有助于防止编码错误。

强类型也是C#代码能够在沙箱中运行的原因之一。沙箱指的是安全性的所有方面都由主机控制的一种环境。在沙箱中，一定要注意不能随意忽略一个对象的类型规则从而破坏其状态。

1.3 内存管理

C#依靠运行时环境来执行自动的内存管理。CLR有一个垃圾回收器，它是作为程序一部分运行的，负责回收不再被引用的对象所占用的内存。这让程序员不需要显式地释放为对象分配的内存，从而避免了诸如C++等语言中错误使用指针造成的内存问题。

C#并没有去除指针：它只是使大多数编程任务不需要使用指针。对于性能至关重要的热点和互操作性方面，还是可以使用指针，但是只允许在显式标记为不安全的代码块中使用。

1.4 平台支持

C#一般用来编写运行在Windows平台的代码。虽然Microsoft通过ECMA实现了C#语言和CLR的标准化，但是专门用来支持非Windows平台C#的资源（包括Microsoft内部和外部的）总量相对较少。这意味着，如果很注重多平台支持，那么诸如Java等语言可能是更明智的选择。因此，C#可在以下情况用于编写跨平台代码：

- C#代码运行在服务器上，生成可运行在任意平台的DHTML。这正是ASP.NET采用的方法。
- C#代码运行在一个非Microsoft Common Language Runtime的运行时环境中。最典型的例子是Mono项目，它具有自己的C#编译器和运行时环境，可运行在Linux、Solaris、Mac OS X和Windows上。
- C#代码运行在一台支持Microsoft Silverlight（支持Windows和Mac OS X）的主机上。这是一种与Adobe Flash Player类似的新技术。

1.5 C#与CLR的关系

C#依赖于一个运行时环境，它包括许多特性，如自动内存管理和异常处理。C#的设计与CLR的设计非常接近，CLR提供了这些运行时特性（但C#技术上不依赖于CLR）。而且C#的类型系统与CLR的

类型系统也非常接近（例如，都共享相同的基础类型定义）。

1.6 CLR和.NET Framework

.NET Framework由名为Common Language Runtime（CLR）的运行时环境和大量的程序库组成。这些程序库由核心库（本书主要介绍）和应用库组成，应用库依赖于核心库。图1-1是这些程序库的可视化概况（也可以作为本文导航辅助图）。

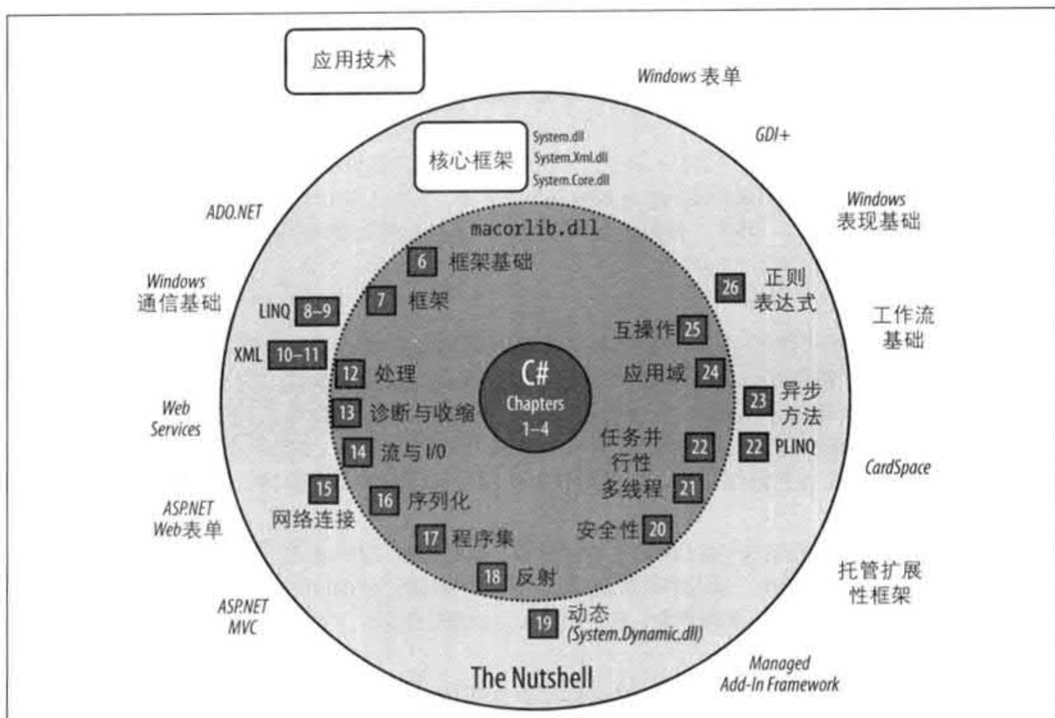


图1-1：本图说明了本文涉及的主题及其所在章节。超出本书范围的特殊框架和类库名称位于The Nutshell的边界之外

CLR是执行托管代码的运行时环境。C#是几种将源代码编译为托管语言之一。托管代码会被打包成程序集，它可以是可执行文件（*.exe*文件）或程序库（*.dll*）的形式，包括类型信息或元数据。

托管代码用Intermediate Language或IL表示。当CLR加载一个程序集时，它会将IL转换为该主机的原生代码，如x86。这个转换是通过CLR的JIT（Just-In-Time）编译器执行的。程序集几乎保留了所有源语言的结构，这使它很容易检测，也很容易动态生成代码。

提示：Red Gate的.NET Reflector是一个重要的分析程序集内容的工具（你可以将它作为反编译器使用）。

CLR是无数运行时服务的主机。这些服务包括内存管理、程序库加载和安全性服务。CLR是与语言无关的，它允许开发人员用多种语言（例如C#、Visual Basic .NET、Managed C++、Delphi.NET、Chrome .NET和J#）开发应用程序。

.NET Framework由只支持基于所有Windows平台或Web的应用程序的程序库组成。第5章概括介绍了.NET Framework程序库。

1.7 C#与Windows Runtime

C# 5.0还实现了Windows Runtime（WinRT）库的互操作。WinRT是一个扩展接口和运行时环境，它可以用面向对象和与语言无关的方式访问库。Windows 8带有这个运行时库，属于Microsoft组件对象模型或COM的扩展版本（参见第25章）。

Windows 8带有一组非托管WinRT库，它是通过Microsoft应用商店交付的支持触摸屏的Metro风格应用程序框架（术语WinRT也指这些库）。作为WinRT，这些程序库不仅可以通过C#和VB访问，也可以通过C++和JavaScript访问。

警告： 有一些WinRT库也可以用在普通的非平板机应用程序中。然而，使用WinRT要求应用程序的最低操作系统是Windows 8。（将来，使用新版本的WinRT甚至会将应用程序的最低操作系统要求提升到Windows 9。）

WinRT库支持新的Metro用户界面（用于编写沉浸式触摸优先应用程序）、移动设备专属特性（传感器、短信等）以及一系列核心功能（与.NET框架部分重叠）。由于功能上存在重叠，所以Visual Studio包含了一个Metro项目参考模板（一组.NET参考程序集），它们隐藏了.NET框架中与WinRT重叠的部分。这个模板还隐藏了.NET框架中大量与平板应用无关的部分（如访问数据库）。Microsoft的应用商店控制着消费设备的软件分发，它拒绝任何试图访问隐藏类型的应用。

提示： 参考程序集的存在仅仅是为了编译程序，并且只有有限的类型与成员集合。因此，开发者可以在计算机上安装完整的.NET框架，但是采用未完全安装的方式编写特定的项目。实际的运行时功能来自全局程序集缓存（参见第18章）的程序集，它们可能大于参考程序集。

隐藏大部分.NET框架可以简化Microsoft平台新开发者的学习过程。除此之外，还有两个更重要的目标：

- 它将应用程序限制在沙箱中（限制功能，减少恶意软件危害）。例如，禁止任意文件访问操作，以及严格限制启动或访问计算机中其他程序的行为。
- 它允许在只支持Metro的低功耗平板电脑上运行.NET框架（Metro模板），降低操作系统资源需求。

WinRT与普通COM的区别是，WinRT的程序库支持多种语言，包括C#、VB、C++和JavaScript，所以每一种语言（几乎）都将WinRT类型视为自己的专属类型。例如，WinRT将调整大小写规则，使之符合目标语言的要求，甚至还会重新映射一些函数与接口。WinRT程序集还在*.winmd*文件中包含丰富的元数据，其格式与.NET程序集文件相同，不需要特殊处理就可以无缝对接。事实上，除了命名空间存在区别之外，开发者甚至不知道使用的是WinRT，而非.NET类型。（此外，WinRT类型遵循COM风格限制，例如，它们只支持有限的继承和泛型。）

提示： WinRT/Metro并不会取代整个.NET框架。仍然推荐（和必须）将后者用于标准桌面和服务器端开发，而且具有以下优势：

- 程序不仅限于运行在沙箱中。
- 程序可以使用整个.NET框架和第三方库。
- 应用程序分发不依赖于Windows应用商店。
- 应用程序可以使用最新版框架，但是不要求用户安装最新版的操作系统。

1.8 C# 5.0新特性

C# 5.0两个较大的新特性是通过两个关键字（async和await）支持异步功能（asynchronous function）。异步功能支持使用异步延续（asynchronous continuation），从而可以简化快速响应和线程安全富客户端应用程序的编写。它们还有利于编写高度并发和高效I/O密集型应用程序，不需要为每个操作绑定一个线程资源。

第14章将详细介绍异步功能。

1.9 C# 4.0新特性

C# 4.0增加的新特性有：

- 动态绑定
- 可选参数和命名参数
- 用泛型接口和代理实现类型变化
- 改进COM互操作性

动态绑定（第4章和第20章）将绑定过程（解析类型与成员的过程）从编译时延迟到运行时，这种方法适用于一些需要避免使用复杂反射代码的场合。动态绑定还适合用于实现动态语言和COM组件的互操作。

可选参数（第2章）允许函数指定参数的默认值，这样调用者就可以省略一些参数，而命名参数则允许函数的调用者按名字指定参数，而非按位置指定。

*类型变化*规则在C# 4.0中很宽松（第3～4章），因此泛型接口和泛型代理类型参数可以标记为covariant或contravariant，从而支持更多自然类型转换。

COM互操作性（第25章）在C# 4.0中进行了3个方面的改进。第一，参数可以通过引用传递，而不需要使用ref关键字（特别适用于与可选参数一同使用）。第二，包含COM interop类型的程序集可以链接，而不需要引用。链接的interop类型支持类型等值转换，从而不需要使用主互操作程序集，并且解决了版本控制和部署的难题。第三，返回来自链接interop 类型的COM变体类型的函数可以映射到`dynamic`，而非映射为`object`，从而不需要进行强制转换。

1.10 C# 3.0新特性

C# 3.0增加的这些特性主要集中在语言集成查询功能上（*Language Integrated Query*，简称*LINQ*）。LINQ支持在C#程序中直接编写查询和以静态方式检查其正确性，并且可以同时查询本地集合（如列表或XML文档）或远程数据源（如数据库）。C# 3.0中用于支持LINQ的新特性还包括隐式类型化局

部变量、匿名类型、对象构造器、Lambda表达式、扩展方法、查询表达式和表达式树。

隐式类型化局部变量（var关键字，第2章）允许在声明语句中省略变量类型，然后由编译器推断其类型。这样可以简化代码和支持匿名类型（第4章），这是一些即时创建的类，它们常用于LINQ查询的最终输出结果中。数组也可以隐式类型化（第2章）。

对象构造器（第3章）允许在调用构造函数之后以内联方式设置属性，从而简化了对象的构造过程。对象构造器同时支持命名和匿名类型。

Lambda表达式（第4章）介绍了由编译器即时创建的微型函数，并且特别适合用于创建“流畅的”LINQ查询（第8章）。

扩展方法（第4章）使用新方法扩展现有的类型（而不需要修改类型定义），使静态方法变得像实例方法一样。

查询表达式（第8章）提供了用于编写LINQ查询的更高级语法，从而可以大大简化操作多个序列或范围变量的LINQ查询。

表达式树（第8章）介绍了描述了分配给特殊类型Expression<TDelegate>的lambda表达式的代码DOM（文档对象模型）。表达式树使LINQ查询能够远程执行（例如在数据库服务器上），因为它们可以在运行时转换和翻译（例如变成SQL语句）。

C# 3.0也增加了自动化和局部方法

自动化属性（第3章）详细讲解一种编写属性的方法，它将私有域的get/set操作进行简化，将它们交给编译器自动执行。局部方法（第3章）让一个自动生成的局部类提供可定制的钩子函数，从而不需要手工编写，而且它会在不使用时“消失”。

第2章

C#语言基础

在本章中，我们将介绍一些C#语言的基础知识。

提示：在本章和接下来的两章中，所有的程序和代码片段都可以作为可交互的例子在LINQPad中使用。阅读本书时学习这些例子可以加快你的学习进度，编辑这些例子，可以立即看到结果，而不需要像在Visual Studio中那样建立项目和解决方案。

要下载这些例子，请点击LINQPad中的Samples选项卡，然后点击“Download more samples”。LINQPad是免费程序，详见*http://www.linqpad.net*。

2.1 第一个C#程序

这是一个计算12乘以30，并把结果360打印到屏幕上的程序。双斜线“//”表示其后的内容是注释。

```
using System;                       // 导入命名空间

class Test                          // 类声明
{
  static void Main()                // 方法声明
  {
    int x = 12 * 30;                // 语句1
    Console.WriteLine (x);          // 语句2
   }                                // 方法结束
}                                   // 类结束
```

在C#中语句按顺序执行。每个语句都以分号（;）结尾。该程序的核心是以下两个语句。

```
int x = 12 * 30;
Console.WriteLine (x);
```

C#语句按顺序执行，以分号（或后面将介绍的*代码块*）结尾。

第1个语句计算表达式12*30的值，并把结果存储到整型局部变量x中。第2个语句调用Console类的WriteLine方法，把变量x的值打印到屏幕上的文本窗口中。

方法是执行一系列语句的行为。这些语句叫做语句块。语句块由一对大括号中的0个或多个语句组成。下面我们定义一个名为Main的方法：

```
static void Main()
{
  ...
}
```

编写可调用低级函数的高级函数可以简化程序。我们可以通过一个可重用的方法重构程序，计算某个整数乘以12的值：

```
using System;

class Test
{
  static void Main()
  {
    Console.WriteLine (FeetToInches (30)); // 结果：360
    Console.WriteLine (FeetToInches (100)); // 结果：1200
  }

  static int FeetToInches (int feet)
  {
    int inches = feet * 12;
    return inches;
  }
}
```

方法可以通过参数来接收调用者输入的数据，并通过返回类型给调用者返回输出数据。现在定义一个FeetToInches方法，该方法有一个用于输入英尺的参数和一个用于输出英寸的返回类型：

```
static int FeetToInches (int feet) {...}
```

上例中的字面值30和100是传递给FeetToInches方法的参数。例子中的Main方法后面只有括号，因为它没有参数，用void是因为它不给调用者返回任何值。

```
static void Main()
```

C#把Main方法作为程序的默认执行入口。Main方法也可以返回一个整型（而不是void），从而为程序执行的环境返回一个值。Main方法也可以接受一个字符串数组作为参数（数组中包含可传递给可执行内容的任何参数）。例如：

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

提示： 数组（例如string[]）代表某种特定类型，固定数量的元素的集合。数组由元素类型和它后面的方括号指定，将在“数组”小节中介绍。

方法是C#中的一种函数，另一种函数是我们用来执行乘法运算的*运算符。还有构造方法、属性、事件、索引器和终结器。

在我们的例子中，将两个方法组合到一个类中。类由函数成员和数据成员组成，形成面向对象的构建块。Console类的组成员处理命令行输入/输出功能，如WriteLine方法。我们的Test类由Main方法

和FeetToInches方法组成。类也是一种类型，我们将在“类型基础”小节中介绍它。

在程序的最外层，类型被组织到命名空间中。Using指令用来使System命名空间在我们的应用程序中有效，以使用Console类。我们能够在TestPrograms命名空间中定义我们所有的类，例如：

```
using System;

namespace TestPrograms
{
  class Test {...}
  class Test2 {...}
}
```

.NET Framework的组织方式为嵌套的命名空间。例如，以下命名空间中包含着处理文本的类型：

```
using System.Text;
```

这里的using指令仅仅是为了方便，也可以用命名空间+类型名（例如System.Text.StringBuilder）这种完全限定名称来引用某种类型。

编译

C#编译器把一系列*.cs*扩展名的源代码文件编译成程序集。程序集是.NET中的最小打包和部署单元。一个程序集可以是一个应用程序，或者是一个库。一个普通的控制台程序或Windows应用程序是一个*.exe*文件，包含一个Main方法。一个库是一个*.dll*文件，它相当于一个没有入口的*.exe*文件。库是用来被应用程序或其他的库调用（引用）的。.NET Framework就是一组库。

C#编译器的名称是*csc.exe*。可以使用像Visual Studio这样的IDE编译C#程序，也可以在命令行中手动调用csc命令编译C#程序。要手动编译C#程序，首先将程序保存成文件（例如*MyFirstProgram.cs*），然后进入命令行并按照下面的方式调用csc命令（csc位于*%SystemRoot%\Microsoft.NET\Framework\<framework-version>*下，其中*%SystemRoot%*指Windows目录）：

```
csc MyFirstProgram.cs
```

这个命令将生成名为*MyFirstProgram.exe*的应用程序。

要生成库（*.dll*），使用如下方式：

```
csc /target:library MyFirstProgram.cs
```

提示：我们将在第18章详细介绍程序集。

2.2 语法

C#的语法是基于C和C++语法的。在本节中，我们将用下面的程序介绍C#的语法元素。

```
using System;

class Test
{
  static void Main()
```

```
    {
      int x = 12 * 30;
      Console.WriteLine (x);
    }
  }
```

2.2.1 标识符和关键字

标识符是程序员为类、方法、变量等选择的名字。下面按顺序排列上例中的标识符：

```
System Test Main x Console WriteLine
```

标识符必须是一个完整的词，它是由字母和下划线开头的Unicode字符组成的。C#标识符是区分大小写的。通常约定参数、局部变量和私有字段应该由小写字母开头（例如myVariable），而其他类型的标识符则应该由大写字母开头（例如MyMethod）。

关键字是编译器保留的名称，不能把它们用作标识符。以下是上例中的关键字：

```
using class static void int
```

下面是所有的C#关键字：

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

1. 避免冲突

如果用关键字作为标识符，可以在关键字前面加上@前缀。例如：

```
class class {...}     // 不合法的
class @class {...}    // 合法的
```

@并不是标识符的一部分，所以@myVariable和myVariable是一样的。

提示：@前缀在调用其他有不同关键字的.NET语言编写的库时非常有用。

2. 上下文关键字

一些关键字是上下文相关的，它们也可以不用@前缀就能作为标识符。它们是：

add	dynamic	in	partial	where
ascending	equals	into	remove	yield
async	from	join	select	

await	get	let	set
by	global	on	value
descending	group	orderby	var

使用上下文关键字作为标识符时，应避免与上下文中的关键字混淆。

2.2.2 字面值、分隔符和运算符

字面值是静态嵌入程序中的原始数据片段。我们的例子中用到的字面值有12和30。

标点有助于划分程序的结构。下面是我们的例子中用到的标点：

```
; { }
```

分号（;）用于结束一条语句。这意味着语句也可以放在多行中，例如：

```
Console.WriteLine
  (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10);
```

大括号用于把多条语句作为一个语句块。

运算符用于改变和结合表达式。大多数C#运算符都用符号表示，例如乘法运算符*。我们将在后续章节中详细讨论运算符。我们的例子中用到的运算符有：

```
. () * =
```

点号（.）表示某个对象的成员（或数字的小数点）。括号在声明或调用方法时使用，空括号在方法没有参数时使用。而等号则用于赋值操作（双等号==才用于相等比较，我们将在之后见到）。

2.2.3 注释

C#提供了两种方式的注释：单行注释和多行注释。单行注释由双斜线开始，到本行结束为止。例如：

```
int x = 3;   // 将数字3的值赋给变量x
```

多行注释由/*开始，由*/结束。例如：

```
int x = 3;   /* 这是
                多行注释*/
```

注释也可以嵌入到XML文档标签中，我们将在第4章的“XML文档”小节中介绍。

2.3 类型基础

类型定义了值的蓝图。值是由变量或常量表示的存储位置。变量代表它的值可以改变，而常量则表示它的值不可以更改（我们将在后面的章节中介绍常量）。我们在第一个程序中创建了一个名为x的局

部变量：

```
static void Main()
{
  int x = 12 * 30;
  Console.WriteLine (x);
}
```

变量是表示存储位置的符号，它包含的值可能会不断变化。相反，常量总是表示同一个值（后面会更详细介绍）：

```
const int y = 360;
```

C#中所有值都是一种类型的实例。一个值或一个变量所包含的一组可能值均由其类型决定。

2.3.1 预定义类型示例

预定义类型是指那些由编译器特别支持的类型。int就是一种预定义类型，它代表从-2^{31}到$2^{31}-1$的32位整数集。我们能够按照下面的方式对int类型的实例执行数学运算等函数：

```
int x = 12 * 30;
```

C#中另一个预定义类型就是string。string类型表示由零个或多个字符组成的有限序列，例如“.NET”或*http://oreilly.com*。我们可以通过下面的方式调用函数来使用字符串：

```
string message = "Hello world";
string upperMessage = message.ToUpper();
Console.WriteLine (upperMessage); // HELLO WORLD

int x = 2010;
message = message + x.ToString();
Console.WriteLine (message); // Hello world2010
```

预定义类型bool只有两种值：true和false。bool类型通常与if语句一起用于条件分支。例如：

```
bool simpleVar = false;
if (simpleVar)
  Console.WriteLine ("This will not print");

int x = 5000;
bool lessThanAMile = x < 5280;
if (lessThanAMile)
  Console.WriteLine ("This will print");
```

提示：在C#中，预定义类型（也称为内建类型）被当作C#关键字。在.NET Framework中的System命名空间下包含了很多并不是预定义类型的重要类型（例如DateTime）。

2.3.2 自定义类型示例

正如我们能使用简单函数来构建复杂函数一样，也可以使用基本类型来构建复杂类型。在下面的例子中，将定义一个名为UnitConverter的自定义类型，这个类将作为单位转换的蓝图。

```
using System;
```

```
public class UnitConverter
{
  int ratio; // 字段
  public UnitConverter (int unitRatio) {ratio = unitRatio; } // 构造方法
  public int Convert (int unit) {return unit * ratio; } // 方法
}

class Test
{
  static void Main()
  {
    UnitConverter feetToInchesConverter = new UnitConverter (12);
    UnitConverter milesToFeetConverter = new UnitConverter (5280);

    Console.WriteLine (feetToInchesConverter.Convert(30)); // 360
    Console.WriteLine (feetToInchesConverter.Convert(100)); // 1200
    Console.WriteLine (feetToInchesConverter.Convert(
                         milesToFeetConverter.Convert(1))); // 63360
  }
}
```

1. 类型的成员

类型包含数据成员和函数成员。UnitConverter的数据成员是叫做ratio的字段。UnitConverter的函数成员是Convert方法和UnitConverter构造方法。

2. 预定义类型和自定义类型

C#的一个优点就是预定义类型和自定义类型只有很少的不同。预定义的int是整数的蓝图。它保存32位的数据，提供像ToString这种函数成员来使用这些数据。类似的是，我们自定义的UnitConverter类型也是单位转换的蓝图。它保存比率数据，还提供了函数成员来使用这些数据。

3. 构造方法和实例化

实例化某种类型即可创建数据。预定义类型可以简单地通过字面值进行实例化，例如12或"Hello World"。new运算符用于创建自定义类型的实例。我们用下面的语句创建并声明一个UnitConverter类型的实例。

```
UnitConverter feetToInchesConverter = new UnitConverter (12);
```

使用new运算符后会立刻实例化一个对象，对象的构造方法会在初始化时被调用。构造方法像方法一样被定义，不同的是方法名和返回类型简化成它所属的类型名。

```
public class UnitConverter
{
  ...
  public UnitConverter (int unitRatio) { ratio = unitRatio; }
  ...
}
```

4. 实例与静态成员

由类型的实例操作的数据成员和函数成员被称为实例成员。UnitConverter的Convert方法和int的ToString方法就是实例成员的例子。在默认情况下，成员就是实例成员。

那些不是由类型的实例操作而是由类型本身操作的数据成员和函数成员必须标记为static。Test.Main和Console.WriteLine就是静态方法。事实上，Console类就是一个静态类，它的所有成员都是静态的。永远也不能创建一个Console的实例，使它在整个应用程序中被共享使用。

对比静态成员和实例成员，在下面的代码中，实例字段Name属于特定的Panda实例，而静态字段Population属于所有的Panda实例：

```
public class Panda
{
  public string Name;                   // 实例字段
  public static int Population;         // 静态字段

  public Panda (string n)               // 构造方法
  {
    Name = n;                           // 为实例字段赋值
    Population = Population + 1;        // 增加静态字段Population的值
  }
}
```

下面的代码创建两个Panda实例，先打印它们的名字，再打印总数。

```
using System;

class Program
{
  static void Main()
  {
    Panda p1 = new Panda ("Pan Dee");
    Panda p2 = new Panda ("Pan Dah");

    Console.WriteLine (p1.Name);      // Pan Dee
    Console.WriteLine (p2.Name);      // Pan Dah

    Console.WriteLine (Panda.Population); // 2
  }
}
```

5. Public关键字

Public关键字将成员公开给其他类。在上例中，如果Panda类中的Name字段不是公有的（public），Test类就不能访问它。把成员标记为public就是在说："这就是我想让其他类型看到的，其他的都是我自己私有的"。用面向对象语言，我们称之为公有（public）成员封装了类中的私有（private）成员。

2.3.4 转换

在C#中，兼容类型的实例可以相互转换。转换始终会根据一个已经存在的值创建一个新的值。转换可以是隐式或显式。隐式转换自动发生，而显式转换需要*cast*关键字。在下面的例子中，我们把一个int类型隐式地转换成long类型（long的容量是int的两倍），把一个int显式地转换成short类型（short只有int的一半容量）。

```
int x = 12345;              // int是32位整型
long y = x;                 // 隐式转换成64位整型
short z = (short)x;         // 显式转换成16位整型
```

隐式转换只有在下列条件都满足时才被允许：

- 编译器能保证转换总是能成功。
- 没有信息在转换过程中丢失（注1）。

相对的，只有在满足下列条件时才需要显式转换：

- 编译器不能保证转换总是能成功。
- 信息在转换过程中有可能丢失。

（如果编译器可以确定某个转换一定会失败，那么这两种转换都会被禁止。包含泛型的转换在特定情况下也会失败——参见第3章的“类型参数的转换”）

提示：我们刚才看到的数值转换是C#中内置的。C#还支持引用转换，装箱转换（见第3章）和自定义转换（第4章的“运算符重载”）。对于自定义转换，编译器并没有强制遵守上面的规则，所以设计不好的类型有可能在转换时出现预想不到的结果。

2.3.5 值类型与引用类型

所有C#类型可以分成以下几类：

- 值类型
- 引用类型
- 泛型参数
- 指针类型

提示：在这节，我们将介绍值类型和引用类型。泛型参数将在第3章的“泛化”一节中介绍，指针类型将在第4章的“不安全代码和指针”一节中介绍。

值类型包含大多数内建类型（具体包括所有的数值类型、char类型和bool类型）以及自定义struct类型和enum类型。

引用类型包括所有的类、数据、委托和接口类型。

值类型和引用类型最根本的不同是它们在内存中的处理方式。

1. 值类型

值类型变量或常量的内容仅仅是一个值。例如，C#内建的值类型int的内容是32位数据。

可以通过struct关键字定义一个自定义值类型（参见图2-1）。

```
public struct Point { public int X, Y; }
```

对值类型实例的赋值操作总是会复制这些实例。例如：

注1：一个小警告，将一个非常大的long转换成double类型时，有可能造成精度丢失。

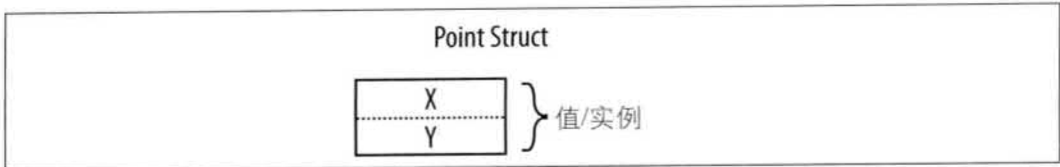


图2-1：内存中的值类型实例

```
static void Main()
{
  Point p1 = new Point();
  p1.X = 7;

  Point p2 = p1; // 赋值引起了复制

  Console.WriteLine (p1.X); // 7
  Console.WriteLine (p2.X); // 7

  p1.X = 9; // 改变p1.X的值

  Console.WriteLine (p1.X); // 9
  Console.WriteLine (p2.X); // 7
}
```

图2-2显示了p1和p2拥有不同的存储空间。

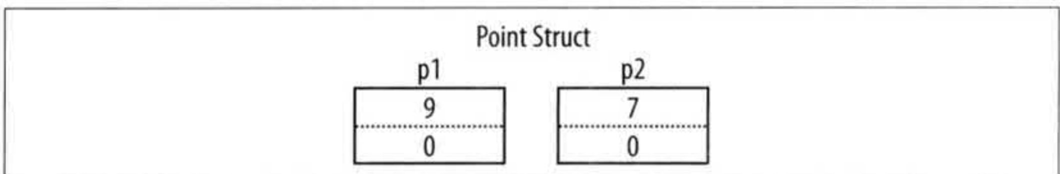


图2-2：赋值操作复制了值类型实例

2. 引用类型

引用类型比值类型复杂，它由两部分组成：对象和对象的引用。引用类型变量或常量的内容是对一个包含值的对象的引用。下面是我们之前例子中的Point类型，但我们重写了它，使它成为一个类而不是一个struct（见图2-3）。

```
public class Point { public int X, Y; }
```

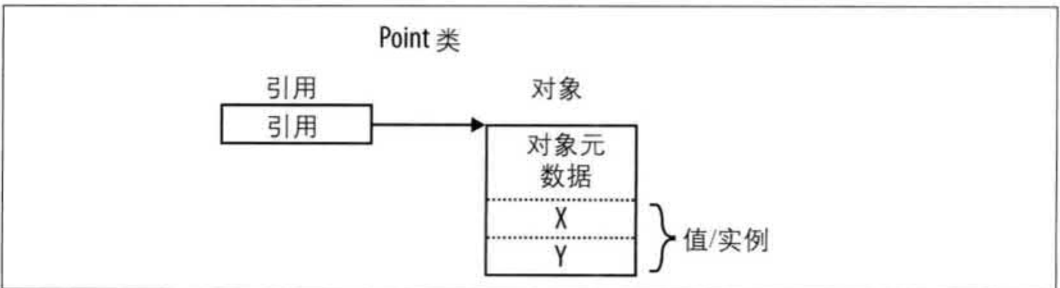


图2-3：内存中的引用类型实例

给引用类型赋值时只复制了引用，而不是对象实例。这允许不同变量指向同一个对象，而值类型通常不可能出现这种情况。如果现在Point是一个类，我们重复之前的例子，那么对X的操作就会影响Y的值了。

```
static void Main()
{
  Point p1 = new Point();
  p1.X = 7;

  Point p2 = p1; //  复制了p1的引用

  Console.WriteLine (p1.X); // 7
  Console.WriteLine (p2.X); // 7

  p1.X = 9; // 改变了p1.X的值

  Console.WriteLine (p1.X); // 9
  Console.WriteLine (p2.X); // 9
}
```

图2-4显示了p1和p2是指向同一对象的两个不同引用。

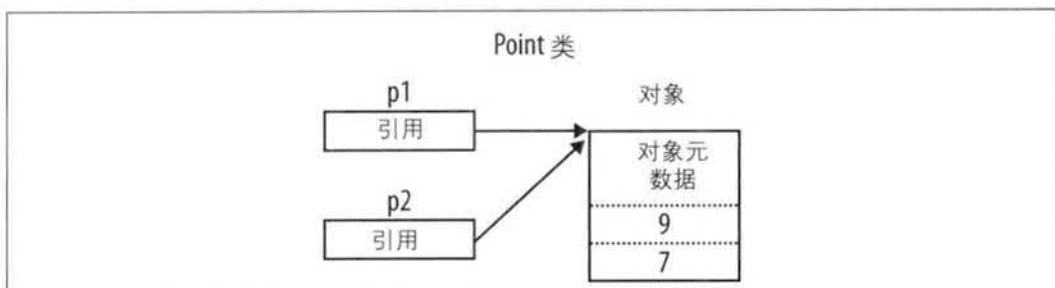


图2-4：赋值操作复制了引用

3. Null

一个引用可以赋值为字面值null，这表示它不指向任何对象：

```
class Point {...}
...

Point p = null;
Console.WriteLine (p == null); // True

// 下面一行会引起运行时错误
// (将抛出一个NullReferenceException异常):
Console.WriteLine (p.X);
```

相对的，值类型通常不能有null值：

```
struct Point {...}
...
Point p = null; // 编译时错误
int x = null; // 编译时错误
```

提示：C#中也有一种代表类型值为null的结构，叫做可空（nullable）类型（见第4章的“可空类型”一节）。

4. 存储开销

值类型实例正好占用需要存储其字段的内存。例如，Point要占用8字节的内存：

```
struct Point
{
  int x; // 4字节
  int y; // 4字节
}
```

提示：从技术上说，CLR用整数倍字段的大小（最大到8字节）来分配内存地址。因此，下面的结构体实际上占用了16字节的存储空间（第一个字段的7个字节被浪费了）：

```
struct A { byte b; long l; }
```

引用类型要求为引用和对象单独分配存储空间。对象占用了和字段一样的字节数，再加上额外的管理开销。对.NET运行时如何实现精确的开销是极为保密的，但是最小的开销是8个字节。它用于存储对象类型的键值，还有像多线程读写锁的状态和表明它是否被垃圾回收器选中等临时信息。每一个对象的引用都需要额外的4或8字节，这取决于.NET运行时是运行在32位平台还是64位平台上。

2.3.6 预定义类型分类

C#中的预定义类型有：

值类型

- 数值
  - 有符号整数（sbyte、short、int、long）
  - 无符号整数（byte、ushort、uint、ulong）
  - 实数（float、double、decimal）
- 逻辑值（bool）
- 字符（char）

引用类型

- 字符串（string）
- 对象（object）

C#中的预定义类型又称框架类型，它们都在System命名空间下。下面的两个语句仅在拼写上有所不同：

```
int i = 5;
System.Int32 i = 5;
```

在CLR中，除了decimal之外的一系列预定义值类型被认为是基本类型。之所以将其称为基本类型，是因为它们在编译过的代码中被指令直接支持。因此它们通常被翻译成底层处理器直接支持的指令。例如：

```
                    // 下面的是十六进制表示
int i = 7;          // 0x7
```

```
bool b = true;         // 0x1
char c = 'A';          // 0x41
float f = 0.5f;        // 用IEEE浮点型编码
```

System.IntPtr和System.UIntPtr类型也是基本类型（见第25章）。

2.4 数值类型

表2-1列出了C#中所有的预定义数值类型。

表2-1：C#中的预定义数值类型

C#类型	系统类型	后缀	容量	数值范围
整数——有符号				
Sbyte	SByte		8位	-2^{7}至 $2^{7}-1$
short	Int16		16位	-2^{15}至 $2^{15}-1$
int	Int32		32位	-2^{31}至$2^{31}-1$
long	Int64	L	64位	-2^{63}至$2^{63}-1$
整数——无符号				
byte	Byte		8位	0至$2^{8}-1$
ushort	UInt16		16位	0至$2^{16}-1$
uint	UInt32	U	32位	0至$2^{32}-1$
ulong	UInt64	UL	64位	0至$2^{64}-1$
实数				
float	Single	F	32位	±(~10^{-45}至10^{38})
doubl	Double	D	64位	±(~10^{-324}至10^{308})
decimal	Decimal	M	128位	±(~10^{-28}至10^{28})

在整数类型中，int和long是最基本的类型，C#和运行时都支持它们。其他的整数类型通常用于实现互操作性或存储空间使用效率非常重要的情况。

在实数类型中，float和double被称为浮点类型（注2），通常用于科学计算。decimal类型通常用于要求10位精度以上的数值计算和高精度的金融计算。

2.4.1 数值字面值

整型字面值可使用小数或十六进制小数标记，十六进制小数用0x前缀表示。例如：

```
int x = 127;
long y = 0x7F;
```

实数字面值可使用小数和指数标记。例如：

```
double d = 1.5;
```

注2：从技术上说，decimal也是一种浮点类型，但是在C#语言规范中通常不将其认为是浮点类型。

```
double million = 1E06;
```

1. 数值字面值类型接口

默认情况下，编译器认为数值字面值或者是double类型或者是整数类型。

- 如果这个字面值包含小数点或指数符号（E），那么它被认为是double。
- 否则，这个字面值的类型就是下列能满足这个字面值的第一个类型：int、uint、long和ulong。

例如：

```
Console.WriteLine (        1.0.GetType());  // Double  (double)
Console.WriteLine (       1E06.GetType());  // Double  (double)
Console.WriteLine (          1.GetType());  // Int32   (int)
Console.WriteLine ( 0xF0000000.GetType());  // UInt32  (uint)
```

2. 数值后缀

数值后缀显式地定义了一个字面值的类型。后缀可以是下列小写或大写字母：

种类	C#类型	例子
F	float	float f = 1.0F;
D	double	double d = 1D;
M	decimal	decimal d = 1.0M;
U	Uint	uint i = 1U;
L	long	long i = 1L;
UL	ulong	ulong i = 1UL;

后缀U和L很少需要，因为unit、long和ulong总是可以表示int或从int隐式转换过来的类型。

```
long i = 5;      // 从int无损地隐式转换成long
```

从技术上讲，后缀D是多余的，因为所有带小数点的字面值都被认为是double类型。总是可以给一个数字类型加上小数点。

```
double x = 4.0;
```

后缀F和M是最有用的，它在指定float或decimal字面值时使用。下面的语句在没有后缀F时无法被编译，因为4.5被认为是double，double是无法隐式转换成float的：

```
float f = 4.5F;
```

同样的规则也适用于decimal字面值：

```
decimal d = -1.23M;     // 缺少M后缀将无法编译
```

我们将在下一节中详细介绍数值转换的语义。

2.4.2 数值转换

1. 整数类型到整数类型的转换

整型转换在目标类型能表示源类型所有可能的值时是隐式转换，否则需要显式转换。例如：

```
int x = 12345;              // int是32位整数
long y = x;                 // 隐式转换成64位整数
short z = (short)x;         // 显式转换成16位整数
```

2. 浮点类型到浮点类型的转换

float能隐式转换成double，因为double能表示所有可能的float的值。反过来则必须是显式转换。

3. 浮点类型到整数类型的转换

所有的整数类型可以隐式转换成浮点数：

```
int i = 1;
float f = i;
```

反过来则必须是显式转换：

```
int i2 = (int)f;
```

提示：将浮点数转换成整数时，小数点后的数值将被截去，而不会四舍五入。静态类System.Convert提供了在不同值类型之间转换的四舍五入方法（见第6章）。

把一个大的整数类型隐式转换成浮点类型会保留整数部分，但是有时会丢失精度。这是因为浮点类型总是有比整数类型更大的数值，但是可能只有更少的精度。下面用一个大的数重写上例来证明：

```
int i1 = 100000001;
float f = i1;          // 保留了整数部分，却丢失了精度
int i2 = (int)f;       // 100000000
```

4. decimal类型转换

所有的整数类型都能隐式转换成decimal类型，因为小数类型能表示所有可能的整数值。其他所有的数值类型转换成小数类型或从小数类型转换到数值类型都必须是显式转换。

2.4.3 算术运算符

算术运算符（+、-、*、/、%）用于除了8位和16位的整数类型之外的所有数值类型：

```
+   加
-   减
*   乘
/   除
%   取余
```

2.4.4 自增和自减运算符

自增和自减运算符（++，--）给数值加1或减1。这两个运算符可以放在变量的前面或后面，这取决于你想让变量在计算表达式之前还是之后被更新。例如：

```
int x = 0,y=0;
Console.WriteLine (x++);//输出0；现在x的值是1
```

```
Console.WriteLine (++y);//输出1；现在y的值是1
```

2.4.5 特殊整数操作

1. 整数除法

整数类型的除法运算总是会截断余数。用一个值为0的变量做除数将产生一个运行时错误（DivisionByZeroException）。

```
int a = 2 / 3;      // 0

int b = 0;
int c = 5 / b;      // 抛出DivideByZeroException异常
```

用字面值0做除数将产生一个编译时错误。

2. 整数溢出

整数类型在运行算术运算时可能会溢出。默认情况下，溢出默默地发生而不会抛出任何异常。尽管C#规范不能预知溢出的结果，但是CLR（通用语言运行时）总是会造成溢出行为。例如，减少最小的整数值将产生最大的整数值：

```
int a = int.MinValue;
a--;
Console.WriteLine (a == int.MaxValue); // True
```

3. 整数运算溢出检查运算符

Checked运算符的作用是在运行时当整型表达式或语句达到这个类型的算术限制时，产生一个OverflowException异常而不是默默的失败。Checked运算符在有++、--、+、-（一元运算符和二元运算符）、*、/和整数类型间显式转换运算符的表达式中起作用。

提示：checked操作符对double和float数据类型没有作用（它们会溢出为特殊的“无限”值，后面将会介绍），对decimal类型也没有作用（这种类型总是受检的）。

Checked运算符能用于表达式或语句块的周围。例如：

```
int a = 1000000;
int b = 1000000;
int c = checked (a * b);      // 只检测这个表达式

checked                       // 检测语句块中所有的表达式
{
   ...
   c = a * b;
   ...
}
```

可以通过在编译时加上/checked+命令行开关（在Visual Studio中，可以在Advanced Build Settings中设置）来默认使程序中所有表达式都进行算术溢出检查。如果你只想禁用指定表达式或语句的溢出检查，可以用unchecked运算符。例如，下面的语句不会抛出异常——即使在编译时使用了/checked+也不会：

```
int x = int.MaxValue;
int y = unchecked (x + 1);
unchecked { int z = x + 1; }
```

4. 常量表达式的溢出检查

无论是否使用了/checked编译器开关，编译时的表达式计算总会检测溢出，除非应用了unchecked运算符。

```
int x = int.MaxValue + 1;               // 编译时错误
int y = unchecked (int.MaxValue + 1);   // 没有错误
```

5. 位运算符

C#支持如下的位运算符：

运算符	含义	示例表达式	结果
~	按位取反	~0xfU	0xfffffff0U
&	按位与	0xf0 & 0x33	0x30
|	按位或	0xf0 | 0x33	0xf3
^	按位异或	0xff00 ^xf0f0	0x0ff00
<<	按位左移	0x20 << 2	0x80
>>	按位右移	0x20 >> 1	0x10

2.4.6 8位和16位整数

8位和16位整数类型指的是byte、sbyte、short和ushort。这些类型缺少它们自己的算术运算符，所以C#隐式把它们转换成所需的大一些类型。当试图把运算结果赋给一个小整数类型时会产生编译时错误：

```
short x = 1, y = 1;
short z = x + y;          // 编译时错误
```

在上面这种情况下，x和y被隐式转换成int以便于进行加法运算。这表明运算结果也是int，它不能隐式转换回short（因为这将造成数据丢失）。我们必须添加一个显式转换使这条语句通过编译：

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

2.4.7 特殊的float和double值

不同于整数类型，浮点类型包含某些操作要特殊对待的值。这些特殊的值是NaN（Not a Number）、+∞、-∞和-0。Float和double类型包含用于NaN、+∞、-∞值（MaxValue、MinValue和Epsilon）的常量。例如：

```
Console.WriteLine (double.NegativeInfinity);   // 负无穷大
```

下面是代表double类型和float类型的特殊值：

特殊值	Double类型常量	Float类型常量
NaN	double.NaN	float.NaN
+∞	double.PositiveInfinity	float.PositiveInfinity
-∞	double.NegativeInfinity	float.NegativeInfinity
-0	-0.0	-0.0f

非零值除以零的结果是无穷大。例如：

```
Console.WriteLine ( 1.0 /  0.0);                    //正无穷大
Console.WriteLine (-1.0 /  0.0);                    //负无穷大
Console.WriteLine (1.0 / -0.0);                     //负无穷大
Console.WriteLine (-1.0 / -0.0);                    //正无穷大
```

零除以零或无穷大减去无穷大的结果是NaN。例如：

```
Console.WriteLine ( 0.0 /  0.0);                    //  NaN
Console.WriteLine ((1.0 /  0.0) -(1.0 / 0.0));      //  NaN
```

使用比较运算符（==）时，一个NaN的值永远也不等于其他的值，甚至不等于其他的NaN值：

```
Console.WriteLine (0.0 / 0.0 == double.NaN);    // False
```

必须使用`float.IsNaN`或`double.IsNaN`方法来判断一个值是不是NaN：

```
Console.WriteLine (double.IsNaN (0.0 / 0.0));   // True
```

无论何时使用`object.Equals`方法，两个NaN的值都是相等的：

```
Console.WriteLine (object.Equals (0.0 / 0.0, double.NaN));   // True
```

提示： NaN在表示特殊值时很有用。在WPF中，`double.NaN`表示值为“Automatic”。另一种表示方法是使用可空类型（nullable）（见第4章），还可以使用包含数值类型或者添加额外字段的自定义结构体（见第3章）。

`Float`和`double`遵循IEEE 754格式类型规范，原生支持几乎所有的处理器。可以在*http://www.ieee.org*上找到这些类型行为的详细信息。

2.4.8 double和decimal的对比

`double`类型在科学计算（例如计算空间坐标）时很有用。`decimal`类型在金融计算和计算那些“人为”的而非真实世界的值时很有用。下面是这两种类型的不同之处：

种类	double	decimal
内部表示	基数2	基数10
精度	15－16位小数	28－29位小数
范围	$\pm(\sim10^{-324}\sim10^{308})$	$(\sim10^{-28}\sim10^{28})$
特殊值	+0、-0、+∞、-∞和NaN	None
速度	处理器原生支持	非处理器原生支持（大约比double慢十倍）

2.4.9 实数四舍五入错误

Float和double在内部是基于2来表示数值的。因此只有基于2表示的数值才能被精确的表示。事实上，这意味着大多数有小数的字面值（它们基于10）将无法精确的表示。例如：

```
float tenth = 0.1f;// 并不是0.1
float one   = 1f;
Console.WriteLine (one - tenth * 10f);// -1.490116E-08
```

这就是为什么float和double不适合金融运算。相反的，decimal基于10。它能够精确地表示基于10的数值（也包括它的因子，基于2和基于5）。因为实型字面值是基于10的，所以decimal能精确地表示像0.1这样的数。然而，double和decimal都不能精确表示那些基于10的极小数：

```
decimal m = 1M / 6M;                // 0.1666666666666666666666666667M
double  d = 1.0 / 6.0;              // 0.16666666666666666
```

这将导致累积性的舍入错误：

```
decimal NotQuiteWholeM = m+m+m+m+m+m;  // 1.0000000000000000000000000002M
double  NotQuiteWholeD = d+d+d+d+d+d;  // 0.99999999999999989
```

这也将影响相等和比较操作：

```
Console.WriteLine (NotQuiteWholeM == 1M);   // False
Console.WriteLine (NotQuiteWholeD < 1.0);   // True
```

2.5 布尔类型和运算符

C#中的bool（System.Boolean类型的别名）能表示true和false的逻辑值。

尽管布尔类型值仅需要1位存储空间，但是运行时却使用1字节空间。这是因为字节是运行时和处理器能够有效使用的最小单位。为避免在使用数组时的空间浪费，.NET Framework提供了System.Collections命名空间下的BitArray类，它被设置成每个布尔值使用1位。

2.5.1 布尔类型转换

bool不能转换成数值类型，反之亦然。

2.5.2 相等和比较运算符

==和!=运算符用于判断任何类型相等还是不相等，总是返回一个bool值（注3）。值类型通常有很简单的相等含义：

```
int x = 1;
int y = 2;
int z = 1;
Console.WriteLine (x == y);         // False
Console.WriteLine (x == z);         // True
```

注3：可以通过重载这些运算符（见第4章）来返回一个非布尔类型，但是实际应用中很少使用。

对于引用类型，默认情况的相等是基于引用的，而不是底层对象的实际值（更多内容请见第6章）：

```
public class Dude
{
  public string Name;
  public Dude (string n) { Name = n; }
}
...
Dude d1 = new Dude ("John");
Dude d2 = new Dude ("John");
Console.WriteLine (d1 == d2);       // False
Dude d3 = d1;
Console.WriteLine (d1 == d3);       // True
```

相等和比较运算符==、!=、<、>、>=和<=用于所有的数值类型，但是用于实数时要特别注意（正如我们在实数“四舍五入错误”中见到的）。比较运算符也用于枚举（enum）类型成员，它比较枚举的潜在整数值，我们将在第3章的“枚举”中介绍。

我们将在第4章的“运算符重载”、“相等比较”和第6章的“顺序比较”中详细介绍相等和比较运算符。

2.5.3 条件运算符

&&和||运算符用于判断与和或条件。它们常常与代表非的!运算符一起使用。在下面的例子中，UseUmbrella方法在下雨或阳光充足（雨伞可以保护我们不会淋雨和晒到），以及没有风（因为雨伞在有风的时候不起作用）的时候返回true：

```
static bool UseUmbrella (bool rainy, bool sunny, bool windy)
{
  return !windy && (rainy || sunny);
}
```

&&和||运算符会在可能的情况下执行短路计算。在上面的例子中，如果刮风，rainy||sunny表达式将不会被计算。短路计算在允许某些表达式时是必要的，像下面的表达式不会抛出NullReferenceException异常：

```
if (sb != null && sb.Length > 0) ...
```

&和|运算符也用于判断与和或条件：

```
return !windy & (rainy | sunny);
```

不同之处是&和|运算符不支持短路计算。因此它们很少用于代替条件运算符。

提示：不同于C和C++，&和|运算符在用于布尔表达式时执行布尔比较（非短路计算）。&和|运算符只在用于数值运算时才执行位操作。

三元条件运算符（简称为条件运算符）使用q?a:b的形式，它在条件q为真时，计算a，否则计算b。例如：

```
static int Max (int a, int b)
```

```
{
  return (a > b) ? a : b;
}
```

条件表达式在LINQ语句中特别有用（见第8章）。

2.6 字符串和字符

C#中的char（System.Char类型的别名）表示一个Unicode字符，它占用2个字节。字符字面值在单引号（'）中指定：

```
char c = 'A';        // 简单字符
```

转义字符不能按照字面表示或解释。转义字符由反斜线（\）和一个表示特殊意思的字符组成。例如：

```
char newLine = '\n';
char backSlash = '\\';
```

表2-2中显示的是转义字符。

表2-2：转义字符

字符	含义	值
\'	单引号	0x0027
\"	双引号	0x0022
\\	斜线	0x005C
\0	空	0x0000
\a	警告	0x0007
\b	退格	0x0008
\f	走纸	0x000C
\n	换行	0x000A
\r	回车	0x000D
\t	水平制表符	0x0009
\v	垂直制表符	0x000B

\u（或\x）转义字符通过4位十六进制代码来指定任意Unicode字符：

```
char copyrightSymbol = '\u00A9';
char omegaSymbol     = '\u03A9';
char newLine         = '\u000A';
```

2.6.1 char转换

从字符类型到数值类型的隐式转换只在这个数值类型可以容纳无符号short类型时有效。对于其他的数值类型，则需要显式转换。

2.6.2 字符串类型

C#中的字符串类型（System.String的别名，我们将在第6章详细介绍）表示一些不变的、按顺序的Unicode字符。字符串字面值在双引号（"）中指定：

```
string a = "Heat";
```

提示：string类型是引用类型而不是值类型。但是它的相等运算符却遵守值类型的语义：

```
string a = "test";
string b = "test";
Console.Write (a == b);  // True
```

对char字面值有效的转义字符在字符串中也有效：

```
string a = "Here's a tab:\t";
```

这样就是当需要一个反斜杠时，必须要写两次：

```
string a1 = "\\\\server\\fileshare\\helloworld.cs";
```

为避免这种情况，C#允许逐字字符串字面值。逐字字符串字面值要加前缀@，它不支持转义字符。下面的逐字字符串和之前的是一样的：

```
string a2 = @ "\\server\fileshare\helloworld.cs";
```

逐字字符串字面值也可以贯穿多行：

```
string escaped  = "First Line\r\nSecond Line";
string verbatim = @"First Line
Second Line";

// 假设IDE使用CR-LF行分隔符：
Console.WriteLine (escaped == verbatim);  // True
```

可以通过在逐字字符串中写两次的方式包含双引号字符：

```
string xml = @"<customer id=""123""></customer>";
```

1. 连接字符串

+运算符连接两个字符串：

```
string s = "a" + "b";
```

右面的操作对象可以是非字符串类型的值，在这种情况下这个值的ToString方法将被调用。例如：

```
string s = "a" + 5;  // a5
```

既然字符串是不变的，那么重复地用+运算符来组成字符串是低效率的：一个更好的解决方案是用System.Text.StringBuilder类型（将在第6章介绍）。

2. 字符串比较

字符串类型并不支持<和>的比较。必须使用字符串类型的CompareTo方法，将在第6章介绍。

2.7 数组

数组代表固定数量的特定类型元素。为了高效率地读取，数组中的元素总是存储在连续的内存块中。

数组用元素类型后加方括号表示。例如：

```
char[] vowels = new char[5];     // 声明5个字符的数组
```

方括号也可以检索数组，通过位置读取特定元素：

```
vowels [0] = 'a';
vowels [1] = 'e';
vowels [2] = 'i';
vowels [3] = 'o';
vowels [4] = 'u';
Console.WriteLine (vowels [1]);      // e
```

因为数组索引从0开始，所以上面的语句打印“e”。我们可以使用for循环语句来遍历数组中的每个元素。下面例子中的for循环将把整型变量i从0到4循环一遍：

```
for (int i = 0; i < vowels.Length; i++)
  Console.Write (vowels [i]);            // aeiou
```

数组的Length属性返回数组中的元素数量。一旦数组被建立，它的长度将不能被更改。System.Collection命名空间和子命名空间提供了像可变数组和字典等高级数据结构。

数组初始化语句定义了数组中的每个元素。例如：

```
char[] vowels = new char[] {'a','e','i','o','u'};
```

或更简单的：

```
char[] vowels = {'a','e','i','o','u'};
```

所有的数组都继承自System.Array类，它提供了所有数组的通用服务。这些成员包括与数组类型无关的获取和定义元素的方法。我们将在第7章的“Array类”中详细介绍。

2.7.1 默认数组元素初始化

建立数组时总是用默认值初始化数组中的元素。类型的默认值是值为0的项。例如，定义一个整型数组。因为int是值类型，将在连续的内存块中分配1000个整数。每个元素的默认值都是0：

```
int[] a = new int[1000];
Console.Write (a[123]);            // 0
```

值类型和引用类型的对比

无论数组元素类型是值类型还是引用类型都有重要的性能影响。若元素类型是值类型，每个元素的值将作为数组的一部分进行分配。例如：

```
public struct Point { public int X, Y; }
...
Point[] a = new Point[1000];
int x = a[500].X;                    // 0
```

因为Point是一个类，所以建立数组时仅仅分配了1000个空引用。

```
public class Point { public int X, Y; }

...
Point[] a = new Point[1000];
int x = a[500].X;                    // 运行时错误，NullReferenceException异常
```

为避免这个错误，我们必须在实例化数组之后显式实例化1000个Point实例：

```
Point[] a = new Point[1000];
for (int i = 0; i < a.Length; i++) // 从0到999遍历i
   a[i] = new Point();             // 用新point实例定义数组元素i
```

无论是任何元素类型，数组本身总是引用类型对象。例如，下面的语句是合法的：

```
int[] a = null;
```

2.7.2 多维数组

多维数组分为两种类型：矩形数组和锯齿形数组。矩形数组代表n维的内存块，而锯齿形数组则是数组的数组。

1. 矩形数组

矩形数组声明时用逗号分隔每个维度。下面的语句声明了一个矩形二维数组，它的维度是3*3。

```
int [,] matrix = new int [3, 3];
```

数组的GetLength方法返回给定维度的长度（从0开始）：

```
for (int i = 0; i < matrix.GetLength(0); i++)
  for (int j = 0; j < matrix.GetLength(1); j++)
    matrix [i, j] = i * 3 + j;
```

矩形数组可以按照如下方式进行初始化（下面例子中的每个元素都被初始化成和之前例子相同的元素）：

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

2. 锯齿形数组

锯齿形数组在声明时用两个方括号表示每个维度。以下声明一个最外层维度是3的二维锯齿形数组的例子：

```
int [][] matrix = new int [3][];
```

提示：有意思的是，这里是new int[3][]，而不是new int[][3]。EricLippert写了一篇很好地说明这个问题的文章，参见：*http://albahari.com/jagged*。

内层维度在声明时可不指定。不同于矩形数组，锯齿形数组的每个内层数组都可以是任意长度；每个内层数组隐式初始化成空（null）而不是一个空数组；每个内层数组必须手工创建：


```
for (int i = 0; i < matrix.Length; i++)
{
  matrix[i] = new int [3];                    // 建立内层数组
  for (int j = 0; j < matrix[i].Length; j++)
    matrix[i][j] = i * 3 + j;
}
```

锯齿形数组可以按照如下方式进行初始化（下面例子中的每个元素都被初始化成和之前例子相同的元素）：

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

2.7.3 简化数组初始化表达式

有两种方式可以简化数组初始化表达式。第一种是省略new运算符和类型限制条件。

```
char[] vowels = {'a','e','i','o','u'};
int[,] rectangularMatrix =
{
  {0,1,2},
  {3,4,5},
  {6,7,8}
};

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

第二种是使用var关键字，使编译器隐式确定局部变量类型。

```
var i = 3;           // i被隐式确定成int
var s = "sausage";   // s被隐式确定成string
// 因此:
var rectMatrix = new int[,]    // rectMatrix隐式确定成 int[,]
{
  {0,1,2},
  {3,4,5},
```

```
  {6,7,8}
};

var jaggedMat = new int[][]     // jaggedMat隐式确定成int[][]
{
  new int[] {0,1,2},
  new int[] {3,4,5},
  new int[] {6,7,8}
};
```

隐式类型转换能进一步用于一维数组的这种情况。能在new关键字之后忽略类型限制符，而由编译器推断数组类型：

```
var vowels = new[] {'a','e','i','o','u'};    // 编译器推断成char[]
```

为了使隐式确定数组类型正常工作，所有的元素都必须可以隐式转换成同一种类型。例如：

```
var x = new[] {1,10000000000};    // 所有的元素都可转换成long
```

2.7.4 边界检查

运行时给所有的数组索引进行边界检查。如果使用了不合法的索引，就会抛出IndexOutOfRangeException异常：

```
int[] arr = new int[3];
arr[3] = 1;               // 抛出IndexOutOfRangeException异常
```

和Java一样，数组边界检查对类型安全和简化调试是很必要的。

提示： 通常来说，边界检查的性能消耗很小，即时编译器会进行优化。像在进入循环之前预先检查所有的索引是不安全的，以此来避免在每轮循环中都检查索引。另外，C#提供“unsafe”关键字来显式绕过边界检查（见第4章的“不安全代码和指针”小节）。

2.8 变量和参数

变量表示存储着可变值的存储空间。变量可以是*局部变量*、*参数*（*value*、*ref*或*out*）、*字段*（*instance*或*static*）或*数组元素*。

2.8.1 栈和堆

栈和堆是存储变量和常量的地方。它们每个都有不同的生存期语义。

1. 栈

栈是存储局部变量和参数的内存块。栈在进入和离开一个函数时逻辑增加和减少。考虑下面的方法（为避免干扰，省略了输入参数检查）：

```
static int Factorial (int x)
{
  if (x == 0) return 1;
```

```
    return x * Factorial (x-1);
  }
```

这个方法是递归的，也就是说它调用它自己。每次进入这个方法时，就在栈上分配一个新的int，每次离开这个方法时，就会释放一个int。

2. 堆

堆是指对象（例如引用类型实例）残留的内存块。每当一个新的对象被创建时，它就被分配进堆，同时返回这个对象的引用。当程序执行时，堆在新对象创建时开始填充。.NET运行时有垃圾回收器，它会定期从堆上释放对象，所以你的电脑不会内存不足。只要对象没有被引用，它就会被选中释放。

在下面的例子中，我们由创建一个引用StringBuilder对象的变量ref1开始，之后写出它的内容。之后StringBuilder对象立即被垃圾回收器选中，因为之后没有引用再使用它。

之后，我们创建另一个引用StringBuilder对象的变量ref2，再将它拷贝给ref3。尽管并不再使用ref2，ref3保持同一个StringBuilder对象存在以保证它不被垃圾回收器选中，直到我们不再使用它。

```
using System;
using System.Text;

class Test
{
  static void Main()
  {
    StringBuilder ref1 = new StringBuilder ("object1");
    Console.WriteLine (ref1);
    // 引用StringBuilder对象的ref1现在被垃圾回收器选中

    StringBuilder ref2 = new StringBuilder ("object2");
    StringBuilder ref3 = ref2;
    //引用StringBuilder对象的ref2现在没有被垃圾回收器选中

    Console.WriteLine (ref3);                   // object2
  }
}
```

无论变量在哪里声明，值类型实例以及对象引用一直存在。如果声明的实例作为对象中的字段或数组元素，那么实例存储于堆上。

提示：在C#中你无法显式删除对象，但在C++中可以。未引用的对象最终被垃圾回收器回收。

堆也存储静态字段和常量。不同于堆上被分配的对象（可以被垃圾回收器回收），静态字段和常量将一直存在直到应用程序域结束。

2.8.2 明确赋值

C#遵守明确赋值的规定。在实践中，这是指在没有unsafe上下文的情况下是不能访问未初始化内存的。明确赋值有三种含义：

- 局部变量在读取之前必须被赋值。
- 当调用方法时必须提供函数的参数。

• 其他的所有变量（像字段和数组元素）都自动在运行时被初始化。

例如，下面的代码将产生编译错误：

```
static void Main()
{
  int x;
  Console.WriteLine (x);        //编译错误
}
```

字段和数组元素都会用其类型的默认值自动初始化。下面的代码输出0，因为数组元素被隐式赋予默认值：

```
static void Main()
{
  int[] ints = new int[2];
  Console.WriteLine (ints[0]);    // 0
}
```

下面的代码输出0，因为字段被隐式赋予默认值：

```
class Test
{
  static int x;
  static void Main() { Console.WriteLine (x); }   // 0
}
```

2.8.3 默认值

所有类型实例都有默认值。预定义类型的默认值是值为0的项：

类型	默认值
所有引用类型	null
所有数值和枚举类型	0
字符类型	'\0'
布尔类型	False

能够对任何类型使用default关键字来获得其默认值（在实践中，这对于泛型非常有用。我们将在第3章介绍泛型）。

```
decimal d = default (decimal);
```

自定义值类型（例如结构体[struct]）中的默认值与自定义类型定义的每个字段的默认值相同。

2.8.4 参数

方法有一连串的参数。其中定义了一系列必须提供给方法的参数。在下面的例子中，Foo方法有一个名为p的int参数：

```
static void Foo (int p)
{
```

```
    p = p + 1;                  // p+1
    Console.WriteLine(p);       // 打印p的值
  }
  static void Main() { Foo (8); }
```

能通过ref和out修饰符来改变参数传递的方式：

参数修饰符	传递类型	必须明确赋值的参数
none	值类型	传入
ref	引用类型	传入
out	引用类型	传出

1. 值传递参数

通常，C#中参数默认是按值传递的。这意味着在将参数值传给方法时创建参数值的副本。

```
class Test
{
  static void Foo (int p)
  {
    p = p + 1;                  // p增加1
    Console.WriteLine (p);      // 在屏幕上打印p的值
  }

  static void Main()
  {
    int x = 8;
    Foo (x);                    // 创建x的副本
    Console.WriteLine (x);      // x的值还是8
  }
}
```

为p赋一个新值并没有改变x的值，因为p和x存储在不同的内存区域中。

值传递引用类型参数将赋值给引用而不是对象本身。在下面的例子中，Foo方法中的StringBuilder对象和Main方法中实例化的是同一个对象，但是他们有不同的引用。换句话说，变量sb和fooSB是引用同一个StringBuilder对象的不同变量。

```
class Test
{
  static void Foo (StringBuilder fooSB)
  {
    fooSB.Append ("test");
    fooSB = null;
  }

  static void Main()
  {
    StringBuilder sb = new StringBuilder();
    Foo (sb);
    Console.WriteLine (sb.ToString());    // test
  }
}
```

因为fooSB是引用的副本，把它赋值成null并没有把sb也赋值成null（然而，如果在声明和调用fooSB时使用ref修饰符，sb将会变成null）。

2. ref修饰符

如果按引用传递参数，C#使用ref参数修饰符。在下面的例子中，p和x指向同一块内存区域：

```
class Test
{
  static void Foo (ref int p)
  {
    p = p + 1;               // p增加1
    Console.WriteLine (p);   // 在屏幕上打印p的值
  }

  static void Main()
  {
    int x = 8;
    Foo (ref  x);            // 让Foo直接对x进行操作
    Console.WriteLine (x);   // 现在x的值是9
  }
}
```

现在给p赋新值将改变x的值。注意ref修饰符在声明和调用时都是必需的（注4）。这样就清楚地表明了将执行什么。

ref修饰符对于转换方法是必要的。（在第3章“泛型”中，我们将介绍如何编写适用于所有类型的转换方法）：

```
class Test
{
  static void Swap (ref string a, ref string b)
  {
    string temp = a;
    a = b;
    b = temp;
  }

  static void Main()
  {
    string x = "Penn";
    string y = "Teller";
    Swap (ref x, ref y);
    Console.WriteLine (x);   // Teller
    Console.WriteLine (y);   // Penn
  }
}
```

提示：无论参数是引用类型还是值类型，都可以实现值传递或引用传递。

---

3. out修饰符

out参数和ref参数类似，除了：

注4：当调用COM方法时有所不同，我们将在第25章中详细讨论。

- 不需要在传入函数之前赋值
- 必须在函数结束之前赋值

out修饰符通常用于获得方法的多个返回值。例如：

```
class Test
{
  static void Split (string name, out string firstNames,
                     out string lastName)
  {
     int i = name.LastIndexOf (' ');
     firstNames = name.Substring (0, i);
     lastName   = name.Substring (i + 1);
  }

  static void Main()
  {
    string a, b;
    Split ("Stevie Ray Vaughn", out a, out b);
    Console.WriteLine (a);                      // Stevie Ray
    Console.WriteLine (b);                      // Vaughn
  }
}
```

和ref参数一样，out参数是引用传递。

4. 引用传递的含义

当引用传递参数时，是为已存变量的存储空间起了个别名，而不是创建了新的存储空间。在下面的例子中，变量x和y代表相同的实例：

```
class Test
{
  static int x;

  static void Main() { Foo (out x); }

  static void Foo (out int y)
  {
    Console.WriteLine (x);                // x的值为0
    y = 1;                                // 改变y的值
    Console.WriteLine (x);                // x的值为1
  }
}
```

5. params修饰符

params参数修饰符在方法最后的参数中指定，它使方法接受任意数量的指定类型参数。参数类型必须声明为数组。例如：

```
class Test
{
  static int Sum (params int[] ints)
  {
    int sum = 0;
    for (int i = 0; i < ints.Length; i++)
```

```
      sum += ints[i];                          // 让sum和ints[i]相加
    return sum;
  }

  static void Main()
  {
    int total = Sum (1, 2, 3, 4);
    Console.WriteLine (total);                 // 10
  }
}
```

也可以将通常的数组提供给params参数。Main方法的第一行等同于：

```
int total = Sum (new int[] { 1, 2, 3, 4 } );
```

6. 可选参数

从C#4.0开始，方法、构造方法和索引器（见第3章）都可以被声明成可选参数。只要在声明时提供默认值，这个参数就是可选参数：

```
void Foo (int x = 23) { Console.WriteLine (x); }
```

可选参数在调用方法时可以被省略：

```
Foo();     // 23
```

默认参数23实际上被传递到可选参数x——编译器在调用端将值23传递到编译好的代码中。先前调用Foo的代码在语义上等同于：

```
Foo (23);
```

因为编译器在可选参数被用到的地方用默认值代替了可选参数。

警告：被其他程序集调用的public方法在添加可选参数时要求重新编译所有的程序集，因为参数是强制的。

可选参数的默认值必须由常量表达式或无参数的值类型构造方法指定。可选参数不能被标记为ref或out。

强制参数必须在可选参数方法声明和调用之前出现（params参数例外，它总是最后出现）。在下面的例子中，显式值1被传递给x，默认值0被传递给y：

```
void Foo (int x = 0, int y = 0) { Console.WriteLine (x + ", " + y); }

void Test()
{
  Foo(1);    // 1, 0
}
```

相反的（传递默认值给x，传递显式值给y），必须将命名参数和可选参数联合使用。

7. 命名参数

可以按名称而不是按参数的位置确定参数。例如：

```
void Foo (int x, int y) { Console.WriteLine (x + ", " + y); }

void Test()
{
  Foo (x:1, y:2);  // 1, 2
}
```

命名参数能按任意顺序出现。下面两种调用Foo方法在语义上是一样的：

```
Foo (x:1, y:2);
Foo (y:2, x:1);
```

提示：不同的是参数表达式按调用端参数出现的顺序计算。通常，这只对相互作用的局部有效表达式有所不同。像下面的代码，它将输出0,1：

```
int a = 0;
Foo (y: ++a, x: --a);  // ++a最先被计算
```

当然在实践中应该避免这样的代码。

命名参数和可选参数可以混合使用：

```
Foo (1, y:2);
```

然而，这里有规定：按位置的参数必须出现在命名参数之前。因此不能这样调用Foo方法：

```
Foo (x:1, 2);          // 编译时错误
```

命名参数在和可选参数混合使用时特别有用。例如，考虑下面的方法：

```
void Bar (int a = 0, int b = 0, int c = 0, int d = 0) { ... }
```

我们能像下面的代码一样只提供d的值并且调用它：

```
Bar (d:3)
```

这在调用COM API时非常有用，我们将在第25章详细讨论。

2.8.5 Var隐式类型局部变量

经常你想用一步声明和初始化变量。如果编译器能够从初始化表达式中推断出变量的类型，就能够使用var关键字（C#3.0中引入）来代替类型声明。例如：

```
var x = "hello";
var y = new System.Text.StringBuilder();
var z = (float)Math.PI;
```

它们完全等同于：

```
string x = "hello";
System.Text.StringBuilder y = new System.Text.StringBuilder();
float z = (float)Math.PI;
```

因为是直接等价，所以隐式类型变量是静态指定类型的。例如，下面的代码产生编译时错误：

```
var x = 5;
x = "hello";    // 编译时错误，x是int类型
```

提示：当无法直接从变量声明中推断出变量类型时，var关键字将降低代码的可读性。例如：

```
Random r = new Random();
var x = r.Next();
```

变量x的类型是什么呢？

在第4章的“匿名方法”中我们将介绍必须使用var的情况。

2.9 表达式和运算符

表达式本质上表示的是值。最简单的表达式是常量和变量。表达式能够用运算符进行转换和组合。运算符用一个或多个输入操作数来输出新的表达式。以下是常量表达式的例子：

```
12
```

可以用*运算符来组合两个操作数（字面值表达式12和30）：

```
12 * 30
```

由于操作数本身可以是表达式，所以可以创建更复杂的表达式。下例中的操作数是（12*30）：

```
1 + (12 * 30)
```

C#中的运算符分为一元运算符、二元运算符和三元运算符，这取决它们使用的操作数数量（1、2或3）。二元运算符总是使用中缀标记法，运算符在两个操作数中间。

2.9.1 基础表达式

基础表达式由C#语言内置的基础运算符表达式组成。例如：

```
Math.Log (1)
```

这个表达式由两个基础表达式组成。第一个表达式执行成员查找（用.运算符），第二个表达式执行方法调用（用()运算符）。

2.9.2 空表达式

空表达式是没有值的表达式。例如：

```
Console.WriteLine (1)
```

因为空表达式没有值，所以不能作为操作数来创建更复杂的表达式：

```
1 + Console.WriteLine (1)     // 编译时错误
```

2.9.3 赋值表达式

赋值表达式用=运算符将一个表达式的值赋给一个变量。例如：

```
x = x * 5
```

赋值表达式不是空表达式。实际上它包含了赋值操作的值，因此能再加上另一个表达式。在下面的例子中，表达式将2赋给x，将10赋给y：

```
y = 5 * (x = 2)
```

这种类型的表达式也能用于初始化多个值：

```
a = b = c = d = 0
```

复合赋值运算符是由其他运算符组合而成的简化运算符。例如：

```
x *= 2    // 等同于x = x * 2
x <<= 1   // 等同于 x = x << 1
```

（这条规则的例外是在第4章的“事件”小节中+=和-=运算符被特殊对待并关联到事件的add和remove访问器。）

2.9.4 运算符的优先级和结合性

当表达式包含多个运算符时，运算符的优先级和结合性决定了计算的顺序。优先级高的运算符先于优先级低的运算符执行。如果运算符的优先级相同，那么运算符的结合性决定计算的顺序。

1. 优先级

下面的表达式：

```
1 + 2 * 3
```

按下面的方式计算，因为*的优先级高于+：

```
1 + (2 * 3)
```

2. 左结合运算符

二元运算符（除了赋值运算符、lambda运算符、null合并运算符）是左结合运算符。换句话说，它们是从左往右计算。例如，下面的表达式：

```
8 / 4 / 2
```

由于左结合性而按以下方式计算：

```
( 8 / 4 ) / 2    // 1
```

插入括号可以改变实际的计算顺序：

```
8 / ( 4 / 2 )    // 4
```

3. 右结合运算符

赋值运算符、lambda运算符、null合并运算符和条件运算符是右结合运算符。换句话说，它们从右往左计算。右结合运算符允许多重赋值，例如：

```
x = y = 3;
```

首先将3赋给y，之后将表达式（3）的结果赋给x。

2.9.5 运算符表

表2-3按优先级列出了C#的运算符。同一类别的运算符的优先级相同。我们将在第4章的"运算符重载"中介绍用户可重载运算符。

表2-3：C#运算符（按优先级顺序分类）

类别	运算符符号	运算符名称	示例	用户是否可重载
基础	()	分组	while(x)	否
	.	成员访问	x.y	否
	->	结构体指针(不安全)	x->y	否
	()	函数调用	x()	否
	[]	数组/索引	a[x]	通过索引器
	++	后自增	x++	是
	--	后自减	x--	是
	new	创建实例	new Foo()	否
	stackalloc	不安全栈空间分配	stackal loc(10)	否
	typeof	从标识符获得类型	typeof(int)	否
	checked	检测整数溢出	checked(x)	否
	unchecked	不检测整数溢出	unchecked(x)	否
	Default	默认值	default(char)	否
	await	等待操作	await myTask	否
一元运算符	sizeof	获得结构体的大小	sizeof(int)	否
	+	加	+x	是
	-	减	-x	是
	!	非	!x	是
	~	按位求反	~x	是
	++	先自增	++x	是
	--	先自减	--x	是
	()	转换	(int)x	否
	* （不安全）	取地址的值	*x	否
	& （不安全）	取地址	&x	否
乘法	*	乘	x * y	是
	/	除	x / y	是
	%	取余	x % y	是

表2-3：C#运算符（按优先级顺序分类）（续）

类别	运算符符号	运算符名称	示例	用户是否可重载
加法	+	加	x + y	是
	-	减	x - y	是
位移	<<	左移	x >> 1	是
	>>	右移	x << 1	是
比较	<	小于	x < y	是
	>	大于	x > y	是
	<=	小于等于	x <= y	是
	>=	大于等于	x >= y	是
	is	类型是/是……的子类	x is y	否
	as	类型转换	x as y	否
相等	==	等于	x == y	是
	!=	不等于	x != y	是
逻辑与	&	与	x & y	是
逻辑异或	^	异或	x ^ y	是
逻辑或	|	或	x | y	是
条件与	&&	条件与	x && y	通过&
条件或	||	条件或	x || y	通过|
Null合并	??	Null条件	x ?? y	否
条件	?:	条件	(a>b)?a:b	否
赋值与Lambda	=	赋值	x = y	否
	*=	自身乘	x *= 2	通过*
	/=	自身除	x /= 2	通过/
	+=	自身加	x += 2	通过+
	-=	自身减	x -= 2	通过-
	<<=	自身左移	x <<= 2	通过<<
	>>=	自身右移	x >>= 2	通过>>
	&=	自身与	x &= 2	通过&
	^=	自身异或	x ^= 2	通过^
	|=	自身或	x |= 2	通过|
	=>	Lambda运算符	x => x + 1	否

2.10 语句

函数包含按出现的字面顺序执行的语句。语句块是大括号（{}）中出现的一系列语句。

2.10.1 声明语句

声明语句可以声明新变量，也可以用表达式初始化变量。声明语句以分号结束。可以用逗号分隔的列表声明多个同类型的变量。例如：

```
string someWord = "rosebud";
int someNumber = 42;
bool rich = true, famous = false;
```

常量的声明和变量声明类似，除了不能在声明之后改变它的值和必须在声明时初始化。

```
const double c = 2.99792458E08;
c += 10;                        // 编译时错误
```

局部变量

局部变量和常量的作用范围是在当前的语句块中。不能在当前的或嵌套的语句块中声明另一个同名的局部变量。例如：

```
static void Main()
{
  int x;
  {
    int y;
    int x;              // 错误，因为已经定义了x
  }
  {
    int y;              // 正确，因为这个范围里y没有被定义
  }
  Console.Write (y);   // 错误，因为y不在这个范围里
}
```

提示：变量的作用范围是它所在的整个代码段。也就是说，如果我们在例子中Main方法的末尾初始化声明x，也会得到相同的错误。

2.10.2 表达式语句

表达式语句是表达式也是合法的语句。表达式语句必须改变状态或调用某些改变的状态。改变的状态本质上是指改变一个变量。可能的表达式语句是：

- 赋值表达式（包括自增和自减表达式）
- 方法调用表达式（有返回值的和无返回值的）
- 对象实例化表达式

例如：

```
// 用声明语句声明变量：
string s;
int x, y;
System.Text.StringBuilder sb;
// 表达式语句
```

```
x = 1 + 2;                    // 赋值表达式
x++;                          // 自增表达式
y = Math.Max (x, 5);          // 赋值表达式
Console.WriteLine (y);        // 方法调用表达式
sb = new StringBuilder();     // 赋值表达式
new StringBuilder();          // 对象实例化表达式
```

当调用有返回值的构造函数或方法时，并不一定要使用返回值。除非构造函数或方法改变了某些状态，否则这些语句完全没有作用：

```
new StringBuilder();        //合法，但没有作用
new string ('c', 3);        //合法，但没有作用
x.Equals (y);               // 合法，但没有作用
```

C#语言基础

2.10.3 选择语句

C#有下面几种语句来有条件地控制程序的执行顺序：

- 选择语句（if，switch）
- 条件语句（?:）
- 循环语句（while、do-while、for、foreash）

本节介绍了两种最简单的结构：if-else语句和switch语句。

1. if语句

if语句是否执行代码体取决于布尔表达式是否为真。例如：

```
if (5 < 2 * 3)
  Console.WriteLine ("true");          // True
```

如果代码体是一条语句，可以省略大括号：

```
if (5 < 2 * 3)
{
  Console.WriteLine ("true");
  console.writeLine("Let's move on!");
}
```

2. else分句

if语句之后可以紧跟else分句：

```
if (2 + 2 == 5)
  Console.WriteLine ("Does Not compute");
else
  Console.WriteLine ("False");          // False
```

在else分句中，能嵌套另一个if语句：

```
if (2 + 2 == 5)
  Console.WriteLine ("Does Not compute");
else
  if (2 + 2 == 4)
   Console.WriteLine ("Computes");     // Computes
```

3. 用大括号改变执行流

else分句总是与其前语句块中紧邻的未配对的if语句结合。例如：

```
if (true)
  if (false)
    Console.WriteLine();
  else
    Console.WriteLine ("executes");
```

语义上等价于：

```
if (true)
{
  if (false)
    Console.WriteLine();
  else
    Console.WriteLine ("executes");
}
```

可以通过改变大括号的位置来改变执行顺序：

```
if (true)
{
  if (false)
    Console.WriteLine();
}
else
  Console.WriteLine ("does Not execute");
```

大括号可以明确地表明结构，这能提高嵌套if语句的可读性（即使编译器并不需要）。下面的模式是值得注意的：

```
static void TellMeWhatICanDo (int age)
{
  if (age >= 35)
    Console.WriteLine ("You can be president!");
  else if (age >= 21)
    Console.WriteLine ("You can drink!");
  else if (age >= 18)
    Console.WriteLine ("You can vote!");
  else
    Console.WriteLine ("You can wait!");
}
```

这里，我们参照其他语言的“elsif”结构来排序if和else语句（还有C#的#elif预处理程序指令）。Visual Studio自动安排这个模式并保持这种代码缩进。从语义上讲，紧跟着每一个if语句的else语句从功能上都是嵌套在else语句之中的。

4. The switch语句

switch语句可以根据变量可能值的选择来转移程序的执行。switch语句可以拥有比嵌套if语句更加简短的代码，因为switch语句只要求表达式计算一次。例如：

```
static void ShowCard(int cardNumber)
{
```

```
  switch (cardNumber)
  {
    case 13:
      Console.WriteLine ("King");
      break;
    case 12:
      Console.WriteLine ("Queen");
      break;
    case 11:
      Console.WriteLine ("Jack");
      break;
    case -1:                          // Joker 是-1
      goto case 12;                   // 在这个游戏里joker等同于queen
    default:                          //为其他cardNumber执行
      Console.WriteLine (cardNumber);
      break;
  }
}
```

只能在支持静态计算的类型表达式中使用switch语句，因此限制了它只适用于整数类型、字符串类型和枚举类型。

在每个case分句的结尾，必须用某种跳转语句明确说明下一步要执行的代码。这里有选项：

- Break（跳转到switch语句结尾）
- Goto case x（跳转到另一个case分句）
- Goto default（跳转到default分句）
- 任何其他的跳转语句——return、throw、continue或goto标签

当多于一个值要执行相同代码时，可以按顺序列出共同的case条件：

```
switch (cardNumber)
{
  case 13:
  case 12:
  case 11:
    Console.WriteLine ("Face card");
    break;
  default:
    Console.WriteLine ("Plain card");
    break;
}
```

switch语句的这种特性对于写出比嵌套if-else语句更清晰的代码来说很重要。

2.10.4 循环语句

C#能够用while、do-while、for和foreach语句重复执行一系列语句。

1. while和do-while循环

while循环在布尔表达式为真时重复执行一段代码。这个表达式在循环体被执行之前被检测。例如：

```
int i = 0;
while (i < 3)
{
  Console.WriteLine (i);
  i++;
}
输出:
0
1
2
```

do-while循环在功能上不同于while循环的是它在语句块执行之后检测表达式（保证语句块至少被执行一次）。以下是用do-while循环重写的上例：

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

2. for循环

for循环类似有特殊分句的while循环，这些特殊分句用于初始化和累积循环变量。for循环有下面的3个分句：

```
for (initialization-clause; condition-clause; iteration-clause)
  statement-or-statement-block
```

Initialization clause

在循环之前执行；用于初始化一个或多个循环变量。

Condition clause

是布尔表达式，当它为真时，将执行循环体。

Iteration clause

在每次循环语句体之后执行；通常用于更新循环变量。

例如，下面的代码将打印从0到2的数字：

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

下面的代码将打印前10个斐波那契数（每个数都是前面两个数的和）：

```
for (int i = 0, prevFib = 1, curFib = 1; i < 10; i++)
{
  Console.WriteLine (prevFib);
  int newFib = prevFib + curFib;
  prevFib = curFib; curFib = newFib;
}
```

for语句的3个部分都可以被省略。可以通过下面的代码来实现一个无限循环（也可以用while(true)代替）：

```
for (;;)
  Console.WriteLine ("interrupt me");
```

3. foreach循环

foreach语句遍历可枚举对象的每一个元素。大多数C#和.NET Framework中表示集合或元素列表的类型都是可枚举的。例如，数组和字符串都是可枚举的。以下是遍历一个字符串的每个字符的例子，从第一个字符到最后一个：

```
foreach (char c in "beer")   // c就是循环变量
  Console.WriteLine (c);
```

```
输出：
b
e
e
r
```

我们将在第4章的“枚举类型和迭代”中详细介绍。

2.10.5 跳转语句

C#的跳转语句有break、continue、goto、return和throw。

提示：跳转语句违背了try语句的可靠性规则（见第4章的“try语句和异常”）。这意味着：

- 跳转到try语句块之外的跳转总是在到达目的地之前执行try语句的finally语句块。
- 跳转语句不能从finally语句块内跳到块外。

---

1. break 语句

break语句用来结束循环体或switch语句体的执行：

```
int x = 0;
while (true)
{
  if (x++ > 5)
    break ;      // 结束循环
}
// 在循环结束之后继续执行
...
```

2. continue语句

continue语句放弃循环体中其后的语句，继续下一轮循环。下面的循环跳过了偶数：

```
for (int i = 0; i < 10; i++)
{
  if ((i % 2) == 0)        // 如果i是偶数
    continue;              // 继续下一轮循环

  Console.Write (i + " ");
}
```

```
输出: 1 3 5 7 9
```

3. goto语句

goto语句用于转移执行到语句块中的另一个标签处。格式如下：

```
goto statement-label;
```

或者用于switch语句内：

```
goto case case-constant;
```

标签语句仅仅是语句块中的占位符，用冒号后缀表示。下面的代码模拟for循环来遍历从1到5的数字：

```
int i = 1;
startLoop:
if (i <= 5)
{
  Console.Write (i + " ");
  i++;
  goto startLoop;
}
输出: 1 2 3 4 5
```

goto case *case-constant*用于转移执行到switch语句块中的另一个条件（见本章的“switch语句”）。

4. return语句

return语句退出方法，如果这个方法有返回值，同时必须返回方法指定返回类型的表达式。

```
static decimal AsPercentage (decimal d)
{
  decimal p = d * 100m;
  return p;             // 用某个值返回调用它的方法
}
```

return语句能出现在方法的任意位置。

5. throw语句

throw语句抛出异常来表示有错误发生（见第4章的“try语句和异常”）：

```
if (w == null)
  throw new ArgumentNullException (...);
```

2.10.6 其他语句

using语句用于调用在finally语句块中实现IDisposable接口的Dispose方法（见第4章的“try语句和异常”和第12章的“IDisposable接口、Dispose方法和Close方法”）。

提示： C#重载了using关键字，使它在不同上下文中有不同的含义。特别注意using指令不同于using语句。

Lock语句是调用Monitor类Enter方法和Exit方法的简化操作（见第14和23章）。

2.11 命名空间

命名空间是类型名称必须唯一的作用域。类型通常被组织到分层的命名空间里，这样既避免了命名冲突又使类型名更容易被找到。例如，处理公钥加密的RSA类型在下面的命名空间中被定义：

```
System.Security.Cryptography
```

命名空间组成了类型名的基本部分。下面的代码调用了RSA类型的Create方法：

```
System.Security.Cryptography.RSA rsa =
  System.Security.Cryptography.RSA.Create();
```

提示： 命名空间是独立于程序集的。程序集是像*.exe*或*.dll*一样的部署单元（见第18章）。

命名空间不影响成员的可见性——public、internal、private等。

namespace关键字为其中的类型定义了命名空间。例如：

```
namespace Outer.Middle.Inner
{
  class Class1 {}
  class Class2 {}
}
```

命名空间中的点（.）表明嵌套命名空间的层次结构。下面的代码在语义上等同于上例：

```
namespace Outer
{
  namespace Middle
  {
    namespace Inner
    {
      class Class1 {}
      class Class2 {}
    }
  }
}
```

可以用包含从外到内的所有命名空间的完全限定名来指代一种类型。例如，可以用Outer.Middle.Inner.Class1来指代上面例子中的Class1。

如果类型没有在任何命名空间中被定义，则说明它存在于全局命名空间内。全局命名空间也包含了顶级命名空间，像例子中的Outer命名空间。

2.11.1 using指令

using指令用于导入命名空间。这是不使用完全限定名来指代某种类型的便捷方法。这个例子在语义上等同于之前的程序：

```
using Outer.Middle.Inner;
```

```
class Test
{
  static void Main()
  {
    Class1 c; A
  }
}
//不需要使用全限定名称
```

提示： 在不同命名空间定义相同类型名称是合法的（而且通常是需要的）。然而，通常只有在开发者无法同时导入两个命名空间时才需要这样做。在.NET框架中，有一个很好的例子是TextBox类，System.Windows.Controls（WPF）和System.Web.UI.WebControls（ASP.NET）均定义了这个名称的类。

2.11.2 命名空间中的规定

1. 名称范围

外层命名空间中声明的名称能够直接在内层命名空间中使用。这里的名称Middle和Class1都被隐式导入到Inner中：

```
namespace Outer
{
  namespace Middle
  {
    class Class1 {}
    namespace Inner
    {
      class Class2 : Class1  {}
    }
  }
}
```

如果想使用同一命名空间分层结构的不同分支中的类型，你就要使用部分限定名。在下面的例子中，SalesReport类继承Common.ReportBase类：

```
namespace MyTradingCompany
{
  namespace Common
  {
    class ReportBase {}
  }
  namespace ManagementReporting
  {
    class SalesReport : Common.ReportBase  {}
  }
}
```

2. 名称隐藏

如果相同的类型名出现在内层和外层命名空间中，内层的类型优先。如果要使用外层命名空间中的类型，必须使用它的完全限定名：

```
namespace Outer
{
```

```
    class Foo { }
    namespace Inner
    {
      class Foo { }
      class Test
      {
        Foo f1;         // = Outer.Inner.Foo
        Outer.Foo f2;   // = Outer.Foo
      }
    }
  }
```


提示：所有的类型名在编译时都被转换成完全限定名。中间语言（IL）代码不包含非限定名和部分限定名。

3. 重复的命名空间

可以重复声明同一个命名空间，只要它里面的类型名不冲突：

```
namespace Outer.Middle.Inner
{
  class Class1 {}
}

namespace Outer.Middle.Inner
{
  class Class2 {}
}
```

我们也可以将这个例子分成两个不同的源文件，将每个类都编译到不同的程序集中。

源文件1：

```
namespace Outer.Middle.Inner
{
  class Class1 {}
}
```

源文件2：

```
namespace Outer.Middle.Inner
{
  class Class2 {}
}
```

4. 嵌套的using指令

我们能在命名空间中使用嵌套using指令。可以在命名空间声明中指定using指令的范围。在下面的例子中，Class1在一个命名空间中可见，但是在另一个中不可见：

```
namespace N1
{
  class Class1 {}
}

namespace N2
```

```
{
  using N1;

  class Class2 : Class1 {}
}
namespace N2
{
  class Class3 : Class1 {}   // 编译时错误
}
```

2.11.3 为类型和命名空间创建别名

引入命名空间有可能引起类型名的冲突。因此可以只引入需要的类型而不是整个命名空间，为每个类型创建别名。例如：

```
using PropertyInfo2 = System.Reflection.PropertyInfo;
class Program { PropertyInfo2 p; }
```

下面代码为整个命名空间创建别名：

```
using R = System.Reflection;
class Program { R.PropertyInfo p; }
```

2.11.4 高级命名空间特性

1. 外部

外部别名允许引用两个完全限定名相同的类型（例如，命名空间和类型名都相同）。这种特殊情况只发生在两种类型来自不同的程序集。思考下面的例子：

程序库1：

```
// csc target:library /out:Widgets1.dll widgetsv1.cs

namespace Widgets
{
  public class Widget {}
}
```

程序库2：

```
// csc target:library /out:Widgets2.dll widgetsv2.cs

namespace Widgets
{
  public class Widget {}
}
```

应用程序：

```
// csc /r:Widgets1.dll /r:Widgets2.dll application.cs

using Widgets;

class Test
```

```
{
  static void Main()
  {
    Widget w = new Widget();
  }
}
```

这个应用程序并不能被编译，因为Widget类是不明确的。外部别名可以消除应用程序的不明确性：

```
// csc /r:W1=Widgets1.dll /r:W2=Widgets2.dll application.cs

extern alias W1;
extern alias W2;

class Test
{
  static void Main()
  {
    W1.Widgets.Widget w1 = new W1.Widgets.Widget();
    W2.Widgets.Widget w2 = new W2.Widgets.Widget();
  }
}
```

2. 命名空间别名限定符

之前提到，内层命名空间中的名称隐藏了外层命名空间中的名称。但是，有时候即使使用类型的完全限定名也无法解决冲突。思考下面的例子：

```
namespace N
{
  class A
  {
    public class B {}                    //嵌套类型
    static void Main() { new A.B(); }    // 实例化类B
  }
}

namespace A
{
  class B {}
}
```

Main方法将会实例化嵌套类B或命名空间A中的类B。编译器总是给当前命名空间中的标识符更高的优先级；在这种情况下，将会实例化嵌套类B。

要解决这样的冲突，可以限定命名空间名，涉及到下面的方法：

- 全局命名空间——所有命名空间的根命名空间（由上下文关键字global指定）
- 一系列的外部别名

::用于限定命名空间别名。下面的例子中，我们使用全局命名空间（这通常出现在自动生成的代码中，以避免名称冲突）：

```
namespace N
{
  class A
```

```
    {
      static void Main()
      {
        System.Console.WriteLine (new A.B());
        System.Console.WriteLine (new global::A.B());
      }

      public class B {}
    }
  }

  namespace A
  {
    class B {}
  }
```

这里是使用别名限定符的例子（修改自“外部”中的例子）：

```
extern alias W1;
extern alias W2;

class Test
{
  static void Main()
  {
    W1::Widgets.Widget w1 = new W1::Widgets.Widget();
    W2::Widgets.Widget w2 = new W2::Widgets.Widget();
  }
}
```

第3章 在C#中创建类

在这一章中，我们将深入讨论类和类成员。

3.1 类

类是最常见的一种引用类型，最简单的类的定义如下：

```
Class YourClassName
{
}
```

复杂的类可能包含以下内容：

类属性	*类属性及类修饰符*。非嵌套的类修饰符有：public、internal、abstract、sealed、static、unsafe、partial
类名	*各种类型参数，唯一基类，多个接口*
花括号内	*类成员（方法、成员属性、索引器、事件、字段、构造方法、运算符函数、嵌套类型和终止器）*

本章涵盖除了类属性、运算符方法和unsafe关键字外的所有上述内容，unsafe关键字将在第4章介绍。以下将逐一介绍各个类成员。

3.1.1 字段

字段是类或结构体中的变量。例如：

```
class Octopus
{
  string name;
  public int Age = 10;
}
```

以下修饰符可以用来修饰字段：

静态修饰符	static
访问权限修饰符	public internal private protected
继承修饰符	new
不安全代码修饰符	unsafe
只读修饰符	readonly
跨线程访问修饰符	volatile

1. 只读修饰符

只读修饰符防止字段值在构造后被更改。只读字段只能在声明时或在其所属的类构造方法中被赋值。

2. 初始化字段

字段不一定要初始化。没有被初始化的字段系统会赋一个默认值（0、\0、null、false）。字段初始化语句在构造方法之前执行：

```
Public int Age =10;
```

3. 同时声明多个字段

为了简便，可以用逗号分隔的列表声明一组同类型的字段，这是声明具有共同属性和修饰符的一组字段的简洁写法。例如：

```
static readonly int legs = 8,
                    eyes = 1;
```

3.1.2 方法

方法是用一组语句实现某个行为。方法能从调用语句的特定类型的传入参数中接收输入数据，并把输出数据以特定的返回值类型返回给调用语句。方法也可以返回void类型，表明这个方法不向调用方返回任何值。此外，方法还可以通过ref/out参数向调用方返回值。

方法签名在整个类中必须是唯一的。方法签名包括方法名、参数类型（但不包括参数名及返回值类型）。

方法可以用以下的修饰符：

静态修饰符	static
访问权修饰符	public internal private protected
继承修饰符	new virtual abstract override sealed
部分方法修饰符	partial
非托管代码修饰符	unsafe extern

1. 重载方法

只要确保方法签名不同，可以在类中重载方法（多个方法共用同一个方法名）。例如，下面的一组方法允许同时出现在同一个类中：

```
void Foo (int x){...}
void Foo (double x){...}
void Foo (int x, float y){...}
void Foo (float x, int y){...}
```

但是，下面的两对方法则不能同时出现在一个类中，因为它们的返回值类型和参数修饰符不属于方法签名的一部分：

```
void Foo (int x){...}
float Foo (int x){...}              // 编译时错误

void Goo (int[] x){...}
void Goo (params int[] x){...}      // 编译时错误
```

在C#中创建类

2. 值传递和引用传递

参数是按值传递还是按引用传递，也是方法签名的一部分。例如，Foo(int)和Foo(ref int)或Foo(out int)可以同时出现在一个类中。但Foo(ref int)和Foo(out int)不能同时出现在一个类中：

```
void Foo(int x){...}
void Foo(ref int x){...} // 到此处编译正确
void Foo(out int x){...} // 编译时错误
```

3.1.3 类实例构造方法

构造方法执行类或结构体的初始化代码。构造方法的定义和方法的定义类似，区别仅在于构造方法名和返回值只能和封装它的类相同：

```
public class Panda
{
  String name;                    // 定义字段
  public Panda (string n);        // 定义构造方法
  {
    name = n;                     // 初始化代码（给字段赋值）
  }
}
...
Panda p = new Panda ("Petey"); // 调用构造方法
```

构造方法支持以下修饰符：

访问权限修饰符	public internal private protected
非托管代码修饰符	unsafe extern

1. 重载构造方法

类或结构体可以重载构造方法。为了避免重复编码，一个构造方法可以用this关键字调用另一个构造方法：

```
using System;

public class Wine
```

```
{
  public decimail Price;
  public int Year;
  public Wine (decimal price) { Price = price; }
  public Wine (decimal price, int year) : this(price) {Year = year; }
}
```

当一个构造方法调用另一个时，被调用的构造方法先执行。

可以向另一个构造方法传递表达式：

```
public Wine (decimal price, DateTime year) : this(price,year.Year) { }
```

表达式内不能再使用this引用，例如，不能调用实例方法。（这是强制性的，因为这个对象当前没有通过构造函数进行实例化，所以调用任何方法都可能失败。）但表达式可以调用静态方法。

2. 隐式无参数构造方法

C#编译器自动为没有显式定义构造方法的类生成构造方法。但是，一旦显式定义了构造方法，系统将不再生成无参数构造方法。

对于结构体来说，无参数构造方法是结构体所固有的，因此，不能自己定义。结构体的隐式构造方法的作用是用默认值初始化每个字段。

3. 构造方法和字段的初始化顺序

首先，在声明字段的时候赋予初始值：

```
class Player
{
  int shields = 50;// 第一个被初始化
  int health = 100;// 第二个被初始化
}
```

字段初始化按声明的先后顺序，在构造方法之前执行。

4. 非公有构造方法

构造方法不一定都是公有的。通常，定义非公有的构造方法的原因是为了在一个静态方法中控制类实例的创建。静态方法可以用于从池中返回类对象，而不必创建一个新对象实例，或用来根据不同的输入属性返回不同的子类。这种使用方式的模板如下：

```
public class Class1
{
  Class1() {}//私有构造方法
  public static Class1 Create(...)
  {
    //在这里定义自己的逻辑，返回Class1的实例
    ...
  }
}
```

3.1.4 对象初始化器

为了简化类对象的初始化，可以在调用构造方法的语句中直接初始化对象的可访问字段或属性。例如下面的类：

```
public class Bunny
{
  public string Name;
  public bool LikesCarrots;
  public bool LikesHumans;

  public Bunny() {}
  public Bunny(string n) {Name = n; }
}
```

用初始化器初始化Bunny对象的方法如下：

```
//注意无参数构造方法可以省略空括号
Bunny b1 = new Bunny {Name ="Bo",LikesCarrots = true,LikesHumans = false };
Bunny b2 = new Bunny("Bo")      { LikesCarrots = true, LikesHumans = false };
```

确切地说，构造b1和b2的代码等同于：

```
Bunny temp1 = new  Bunny();// temp1是编译器生成的名字
temp1.Name =  Bo ;
temp1.LikesCarrots = true;
temp1.LikesHumans = false;
Bunny b1 = temp1;

Bunny temp2 = new Bunny("Bo");
temp2.LikesCarrots = true;
temp2.LikesHumans = false;
Bunny b2 = temp2;
```

使用临时变量是为了确保在初始化过程中如果抛出异常，不会得到一个初始化未完成的对象。

对象初始化器是C#3.0引入的新概念。

对象初始化器与可选参数

如果不用对象初始化器，也可以让Bunny类的构造方法接受可选参数：

```
public Bunny ( string name,bool likesCarrots = false,bool likesHumans = false)
{
  Name = name;
  LikesCarrots = likesCarros;
  LikesHumans = LikesHumans;
}
```

这个构造方法可以用如下的语句构造Bunny对象：

```
Bunny b1 = new Bunny(name: "Bo",likesCarrots:true);
```

这样做的优点是我们可以设置Bunny的字段或属性（后面会讲解）为只读。如果在对象的生命周期内，不需要改变字段值或属性值，将字段或属性设为只读是非常有用的。

缺点是所有的可选参数都在调用方处理，换句话说，C#将我们的构造方法调用翻译成：

```
Bunny b1= new Bunny ("Bo", true, false);
```

这使得如果在另一个程序集中实例化Bunny类，而当Bunny类再加一个可选参数（如likesCats）时可能出错。除非引用该类的程序集也重新编译，否则它还将继续调用三个参数的构造方法（现在已经不存在了），并出现运行时错误。（还有一种难以发现的错误是，如果我们修改了某个可选参数的默认值，另一个程序集中的调用方在重新编译前，还会继续使用旧的可选值。）

因此，如果想使程序在不同版本的程序集中保持二进制兼容，最好避免在公有方法中使用可选参数。

3.1.5 this引用

this引用指的是引用类实例自身。在下面的示例中，方法Marry将Partner的mate字段设定为this。

```
public class Panda
{
  public Panda Mate;

  public void Marry(Panda partner)
  {
    Mate = partner;
    partner.Mate = this;
  }
}
```

this引用也用来避免类字段和局部变量或属性相混淆。例如：

```
public class Test
{
  string name;
  public Test(string name) { this.name = name; }
}
```

this引用仅对类或结构体的非静态成员有效。

3.1.6 属性

从外表看属性和字段很类似，但属性内部像方法一样包含逻辑。例如，从下面的代码不能判断CurrentPrice到底是字段还是属性：

```
Stock msft = new Stock();
msft.CurrentPrice = 30;
msft.CurrentPrice -= 3;
Console.WriteLine (msft.CurrentPrice);
```

属性和字段的声明很类似，但属性比字段多了一个get/set块。下面是CurrentPrice作为属性的实现方法：

```
public class Stock
{
```

```
    decimal currentPrice;// “背后”的私有字段
    public decimal CurrentPrice// 公有属性
    {
      get { return currentPrice; } set {currentPrice = value; }
    }
  }
```

get和set提供属性的访问器。读取属性值时会运行get访问器，它必须返回属性类型的值。给属性赋值时，运行set访问器，它有一个命名为value的隐含参数，类型和属性类型相同，值直接被指定给私有字段（本例中是currentPrice）。

尽管访问属性和字段的方法相同，但不同之处在于，属性在获取和设置值时，给实现者提供了完全的控制能力。这种控制能力使得实现者可以选择所需的任何的内部通信机制，而无需将属性的内部细节暴露给用户。在本例中，set方法可以在value超出有效范围值时抛出异常。

提示：本书中广泛使用私有字段，以免干扰读者注意力。在实际应用中，为了提高封装性，可能更多地在公有字段上应用公有属性。

属性可以用下面的修饰符：

静态修饰符	static
访问权限修饰符	public internal private protected
继承修饰符	new virtual abstract override sealed
非托管代码修饰符	unsafe extern

1. 只读和计算属性

如果只定义了get访问器，属性就是只读的；如果只定义了set访问器，属性就是只写的，但很少用到只写属性。

通常属性会用一个简短的后台字段来存储其所代表的数据，但属性也可以从其他数据计算出来。例如：

```
Decimal currentPrice, shareOwned;
public decimal Worth
{
  get{ return currentPrice * sharesOwned; }
}
```

2. 自动属性

属性最常见的实现方法是get访问器和set访问器，对一个同类型的私有字段进行简单的读写操作。自动属性的声明表明由编译器提供上述实现方法。可以把本节的第一个示例重新定义为：

```
public class Stock
{
  ...
  public decimal CurrentPrice { get; set;}
}
```

编译器会自动产生一个后台的私有字段，该字段名由编译器生成，且不能被引用。如果希望属性对外暴露成只读属性，set访问器可以标注为private的。在C# 3.0中引入了自动属性。

3. get和set的访问权限

get和set访问器可以有不同的访问级别。典型的用法是，将一个public的属性中的set访问器设置成internal或private的：

```
public class Foo
{
  private decimal x;
  public decimal x
  {
    get { return x;}
    private set { x = Math.Round (value, 2); }
  }
}
```

注意，属性本身被声明具有较高的访问权限（本例中是public），然后在需要较低级别的访问器上添加较低级别的访问权限修饰符。

4. CLR属性的实现

C#属性访问器在系统内部被编译成名为get_*XXX*和set_*XXX*的方法：

```
public int get_CurrentPrice { ... }
public void set_CurrentPrice (decimal value) { ... }
```

简单的非虚拟属性访问器被JIT（即时）编译器编译成内联的，消除了属性和字段访问方法的性能差别。内联是一种优化方法，它用方法的函数体替代方法调用。

通过WinRT的属性，编译器就可以假定是put_*XXX*命名转换，而不是set_*XXX*。

3.1.7 索引器

索引器为访问类或结构体中封装的列表或字典型数据元素提供了自然的访问接口。索引器和属性很相似，但索引器通过索引值而非属性名访问数据元素。string类具有索引器，可以通过int索引访问其中的每一个char值：

```
string s = "hello";
Console.WriteLine(s[0]);// 'h'
Console.WriteLine(s[3]);// 'l'
```

当索引是整型时，使用索引器的方法类似于使用数组。

提示：索引器和属性具有相同的修饰符（详见“属性”一节中的介绍）。

---

1. 实现索引器

要编写一个索引器，首先定义一个名为this的属性，将参数定义放在一对方括号中。例如：

```
class Sentence
{
  sting[] words ="The quick brown fox".Split();

  public string this [int wordNum]   // 索引器
  {
    get { return words [wordNum]; }
    set { words [wordNum] = value; }
  }
}
```

下面是使用索引器的方法：

```
Sentence s= new Sentence();
Console.WriteLine (s[3]);//fox
S[3] = "kangaroo";
Console.WriteLine (s[3]);//Kangaroo
```

一个类可以定义多个参数类型不同的索引器，索引器也可以有多个参数：

```
public string this [int arg1, string arg2]
{
  get { ... } set { ... }
}
```

如果省略set访问器，索引器就变成只读的。

2. CLR索引器的实现

索引器在系统内部被编译成名为get_Item和set_Item的方法，如下所示：

```
public string get_Item (int wordNum) { ... }
public void set_Item (int wordNum, string value) { ... }
```

3.1.8 常量

常量是值永远不会改变的字段。常量在编译时静态赋值，并且在使用时，编译器直接替换该值，类似于C++中的宏。常量可以是内置的数据类型：bool、char、string或枚举类型。

常量用关键字const定义，并且必须以特定值初始化。例如：

```
public class Test
{
  public const string Message = "Hello World";
}
```

常量在使用时比静态只读字段有更多限制——不仅能使用的类型有限，而且初始化字段的语句含义也不同。常量和静态只读变量的不同之处还有，常量是在编译时赋值的。例如：

```
public static double Circumference (double radius)
{
  return 2 * System.Math.PI * radius;
}
```

编译成：

```
public static double Circumference (double radius)
{
  return 6.2831853071795862 * radius;
}
```

这样做是合理的，因为PI是常量，它的值永远不变。相反，静态只读字段可以在每个应用中有不同的值。

提示： 静态只读字段的好处还有，当提供给其他程序集时，可以更新数值。例如：假定程序集X提供一个如下常量：

```
public const int MaximumThreads = 20;
```

如果程序集Y引用X并使用该常量，值20在编译时就固定在Y程序集中了。这样，若X后来将常量设成50并重新编译，Y如果不重新编译，还会使用旧值20。静态只读字段可以避免这一问题。

从另一角度看，将来可能发生变化的任意值都不受其定义约束，所以不应该表示为一个常量。

常量也可以在方法内声明，例如：

```
static void Main()
{
  Const double twoPI = 2 * System.Math.PI;
  ...
}
```

常量可以使用以下修饰符：

访问权限修饰符	public internal private protected
继承修饰符	new

3.1.9 静态构造方法

静态构造方法是每个类执行一次，而不是每个类实例执行一次。一个类只能定义一个静态构造方法，并且必须没有参数，必须和类同名：

```
class Test
{
  static Test() { Console.WriteLine("Type Initialized"); }
}
```

运行时在使用类之前自动调用静态构造方法，下面两种行为可以触发静态构造方法：

- 实例化类
- 访问类的静态成员

静态构造方法只有两个修饰符：unsafe和extern。

警告： 如果静态构造方法抛出一个未处理异常（第4章），类在整个应用程序的生命周期内都是不可用的。

静态构造方法和字段初始化顺序

静态字段在调用静态构造方法之前执行初始化。如果一个类没有静态构造方法，字段在类被使用前初始化或在运行时随机选一个更早的时间执行初始化（这说明静态构造方法的存在可能使字段初始化比正常时间晚执行）。

静态字段按字段声明的先后顺序初始化。下例中X初始化为0，Y初始化为3。

```
class Foo
{
  public static int X = Y;//0
  public static int Y = 3; //3
}
```

如果我们调换两个字段的初始化顺序，两个字段都将被初始化为3。下例中先打印0后打印3，因为字段初始化器在X被初始化为3前实例化Foo：

```
class program
{
  staic void Main() { Console.WriteLine (Foo.X); } // 3
}

class Foo
{
  public static Foo Instance = new Foo();
  public static int X=3;

  Foo() { Console.WriteLine(X); } // 0
}
```

如果交换两行粗体语句的顺序，上例输出两个3。

3.1.10 静态类

类可以标记为static，表明它必须仅由静态成员组成，并且不能产生子类。System.Console和System.Math类就是静态类的最好示例。

3.1.11 终止器

终止器是只能在类中使用的方法，它在垃圾收集器回收没有被应用的对象前执行。终止器的语法是类名加前缀~：

```
class Class1
{
  ~Class1()
  {
    ...
  }
}
```

实际上，这是重载对象的Finalize方法的C#语法，编译器将其扩展成如下的方法声明：

```
protected override void Finalize()
{
  ...
```

```
    base.Finalize();
  }
```

我们在第12章详细讨论垃圾回收。

终止器允许使用以下修饰符：

非托管代码修饰符	unsafe

3.1.12 局部类和方法

局部类允许一个类分开定义，典型的用法是分开在多个文件中。从其他源文件自动生成的类（如：XSD文件）需要和自定义的方法交互时，通常使用partail类。例如：

```
// PaymentFormGen.cs -自动生成的
partial class PaymentForm { ... }

// PaymentForm.cs -自定义的
partial class PaymentForm { }
```

每个部分必须由partial声明，下面的写法是不合法的：

```
partial class PaymentForm { }
class PaymentForm { }
```

局部类的各组成部分不能有冲突的成员。例如具有相同参数的构造方法，不能重复出现。局部类完全由编译器处理，也就是说，各组成部分在编译时必须可用，并必须编译在同一个程序集中。

有两个方法为partial类定义基类：

- 在每个部分定义同一个基类。例如：

  ```
  partial class PaymentForm : ModalForm { }
  partial class PaymentForm : ModalForm { }
  ```

- 仅在其中一部分定义基类。例如：

  ```
  partial class PaymentForm : ModalForm {}
  partial class PaymentForm {}
  ```

另外，每个部分都可以独立定义并实现接口。我们将在后面“继承”和“接口”中详细介绍基类和接口。

局部方法

局部类可以包含局部方法，这些方法使自动生成的局部类可以为自定义方法提供自定义钩子（hook）。例如：

```
partial class PaymentForm// 自动生成的类文件中
{
  ...
  partial void ValidatePayment (decimal amount);
}

partial class PaymentForm// 自定义的文件中
{
```

```
  ...
  partial void ValidatePayment (decimal amount)
  {
    if(amount > 100)
      ...
  }
}
```

局部方法由两部分组成：定义和实现。定义一般由代码生成器产生，而实现多为手工编写。如果没有提供方法的实现，方法的定义会被编译器清除。这使得自动代码生成可以自由提供钩子（hook），而不用担心代码过于臃肿。局部方法必须是void型，并且默认是private的。

局部方法在C# 3.0中引入。

在C#中创建类

3.2 继承

为了扩展或自定义原类，类可以继承另一个类。继承类让你可以重用另一个类的方法，而无需重新构建。一个类只能继承自唯一的类，但可以被多个类继承，从而形成类的层次。在本例中，我们定义一个名为Asset的类：

```
public class Asset
{
  public string Name;
}
```

然后定义一个Stock类和一个House类，他们都继承自Asset类。他们具有Asset类的所有特征，而各自又有增加的定义：

```
public class Stock : Asset// 从Asset继承
{
  public long SharesOwned;
}

Public class House : Asset// 从Asset继承
{
  public decimal Mortgage;
}
```

下面是两个类的使用方法：

```
Stock msft = new Stock { Name = "MSFT", SharesOwned = 1000 };
Console.WriteLine (msft.Name);                  // MSFT
Console.WriteLine (msft.SharesOwned);           // 1000

House mansion = new House { Name = "Mansion",Mortgage = 250000 };
console.WriteLine (mansion.Name);               // Mansion
Console.WriteLine (mansion.Mortage);            // 250000
```

子类Stock和House都从基类Asset继承了Name属性。

提示： 子类也被称为派生类；基类也被称为超类。

3.2.1 多态

引用是多态的。意味着x类型的变量可以指向x子类的对象。例如：

```
public static void Display (Asset asset)
{
  System.Console.WriteLine (asset.Name);
}
```

这个方法可以用来显示Stock和House类，因为这两个类都继承自Asset类。

```
Stock msft = new Stock ...;
House mansion = new House ...;

Display (msft);
Display (mansion);
```

多态性之所以能实现，是因为子类（Stock和House）具有基类（Asset）的全部特征。反过来，则不正确。如果Display改成接收House类，就不能把Asset类传给它：

```
static void Main() { Display (new Asset()); }// 编译时错误

public satic void Display (House house)       // 不能接收Asset类
{
  System.Console.WriteLine (house.Mortgage);
}
```

3.2.2 类型转换和引用转换

对象引用可以被：

- 隐式向上转换成基类的引用
- 显式向下转换为子类的引用

在可兼容的类型引用之间向上类型转换或向下类型转换即为引用转换：生成一个新的引用指向同一个对象。向上转换总是能成功，而向下转换只有在对象的类型符合要求时才能成功。

1. 向上类型转换

向上类型转换创建一个基类指向子类的引用。例如：

```
Stock msft = new Stock();
Asset a = msft;      // 向上转换
```

向上转换以后，变量a还是指向msft指向的Stock对象。被引用的对象本身不会被替换或改变：

```
Console.WriteLine (a == msft);        // 正确
```

虽然a和msft指向同一个对象，但a指向对象时有更严格的要求：

```
Console.WriteLine (a == msft);        // 正确
Console.WriteLine (a.ShareOwned);     // 错误：SharesOwned未定义
```

上面最后一行产生一个编译错误，因为变量a是Asset类型，虽然它指向的是Stock类型的对象。如果要访问ShareOwned字段，你必须先把Asset类型向下转换成Stock类型。

2. 向下类型转换

向下类型转换创建一个子类指向基类的引用。例如：

```
Stock msft = new Stock();
Asset a = msft;                      // 向上转换
Stock s = (Stock)a;                  // 向下转换
Console.WriteLine (s.ShareOwned);    // 没有错误
Console.WriteLine (s == a);          // 输出True
Console.WriteLine (s == msft)        // 输出True
```

对于向上转换，只影响了引用，被引用的对象没有变化。而向下转换必须是显式转换，因为它可能导致运行时错误：

```
House h = new House();
Asset a = h;            // 向上转换永远会成功
Stock s = (Stock)a;     // 向下转换出错：a不是Stock类型
```

如果向下转换出错，会抛出InvalidCastException。这是一个运行时类型检查的例子（我们后面还会在“静态和运行时类型检查”中详细介绍。

3. as运算符

as运算符在向下类型转换出错时为变量赋值null（而不是抛出异常）：

```
Asset a = new Asset();
Stock s = a as Stock//s是null，不抛出异常
```

这个操作相当有用，接下来只需判断结果是否为null。

```
if(s!= null) Console.WriteLine(s.SharesOwned);
```

提示： 如果不用判断结果是否为null，使用cast更好，因为如果发生错误，cast会抛出描述更清楚的异常。我们可以通过比较下面两行代码看出：

```
int shares = ((Stock)a).ShareOwned;    //方法#1
int shares = (a as Stock).ShareOwned;  //方法#2
```

如果a不是Stock类型，第一行代码抛出InvlidCastException，很清楚地描述了错误。第二行代码抛出NullReferenceException，这就比较模糊，到底是因为a不是Stock类型还是因为a为null呢？

从另一个角度看，使用cast操作符的意思是告诉编译器：“已确定这个值的类型；如果判断错误，那么代码可能有bug，所以要抛出一个异常！”而如果使用as操作符，则表示不确定其类型，需要根据运行时输出结果来确定其类型。

as运算符不能用来实现自定义转换（见第4章“运算符重载”），也不能用于数值型转换：

```
long x = 3 as long;     //编译时错误
```

提示： as和cast运算符也可以用来实现向上类型转换，但不常用，因为隐式转换就可以实现。

4. is运算符

is运算符用于检查引用的转换能否成功，换句话说，它是检查一个对象是否是从某个特定类派生

（或是实现某个接口），经常在向下类型转换前使用。

```
if(a is Stock)
  Console.WriteLine (((Stock)a).SharesOwned);
```

is运算符不能用于自定义类型转换和数值型类型转换，但它可以用于拆箱机制的类型转换（详见“object类型”）。

3.2.3 虚函数成员

标识为virtual的函数可以被提供特定实现的子类重载。方法、属性、索引器和事件都可以被声明为virtual：

```
public class Asset
{
   public string Name;
   public virtual decimal Liability { get { return 0; } }
}
```

子类通过override修饰符重载虚方法：

```
public class Stock : Asset
{
  public long SharesOwned;
}

public class House : Asset
{
  public decimal Mortgage;
  public override decimal Liability { get { return Mortgage; } }
}
```

Asset类的Liability默认值是0。Stock类不用限定这一行为，而House类限定让Liability属性返回Mortgage的值：

```
House mansion = new House { Name = McMansion , Mortgage = 250000 };
Asset a = mansion;
Console.WriteLine (mansion.Liability); // 250000
Console.WriteLine (a.Liability); // 250000
```

虚方法和重载方法的标识、返回值以及访问权限必须完全一致。重载方法可以通过base关键字调用其基类的实现（我们在“base关键字”中详细介绍）。

警告：从构造方法调用虚方法可能很危险，因为编写子类的人在重写方法时不可能知道正在操作一个未完全实例化的对象。换而言之，重写方法最终会访问到一些依赖于未被构造方法初始化的域的方法或属性。

3.2.4 抽象类和抽象成员

被声明为*abstract*的抽象类不能被实例化。只有抽象类的具体实现子类才能被实例化。

抽象类中可以定义抽象成员，抽象成员和虚成员相似，但抽象成员不提供默认的实现。实现必须由子类提供，除非子类也被声明为抽象类：

```
public abstract class Asset
{
  // 注意实现为空
  public abstract decimal NetValue { get; }
}

public class Stock : Asset
{
  public long ShareOwned;
  public decimal CurrentPrice;

  // 像虚方法一样重载
  public override decimal NetValue
  {
    get { return CurrentPrice * ShareOwned; }
  }
}
```


3.2.5 隐藏继承成员

基类和子类可能定义相同的成员，例如：

```
public class A{ public int Counter = 1; }
public class B : A { public int Counter = 2; }
```

类B中的字段Counter隐藏了类A中的字段Counter。通常，这种情况的产生是由于编程时，在定义子类的成员之后又把相同的成员加到基类中。因此，编译器会产生一个警告，可以用下面的方法避免二义性：

- A类的引用（在编译时）绑定到A.Counter
- B类的引用（在编译时）绑定到B.Counter

有时需要故意隐藏一个成员，这种情况下，可以在子类中使用new修饰符。new修饰符的作用仅为防止编译器发出警告。写法如下：

```
public class A     { public     int Counter = 1; }
public class B : A { public new int Counter = 2; }
```

修饰符new把你的意图传达给编译器以及其他编程人员，即重复的成员不是无意的。

提示： C#在不同的上下文环境中使用new关键字表达完全不同的含义。特别要注意new运算符和new成员修饰符的不同。

---

new和virtual的比较

思考下面的类层级：

```
public class BaseClass
{
  public virtual void Foo() { Console.WriteLine ("BaseClass.Foo"); }
}

public class Overrider : BaseClass
{
```

```
    public override void Foo() { Console.WriteLine ("Overrider.Foo"); }
  }
  public class Hider : BaseClass
  {
    Public new void Foo() { Console.WriteLine ("Hider.Foo"); }
  }
```

下面的代码说明了Overrider和Hider类的不同：

```
Overrider over = new Overrider ();
BaseClass b1 = over;
over.Foo();           // Overrider.Foo
b1.Foo();             // Overrider.Foo

Hider h = new Hider();
BaseClass b2 = h;
h.Foo();              // Hider.Foo
b2.Foo();             // BaseClass.Foo
```

3.2.6 密封方法和类

重载的方法成员可用sealed关键字密封它的实现，以防止该方法被它的更深层次的子类再次重载。在前面的虚方法成员示例中，我们可以密封House类的Liability实现，以防止继承自House的子类重载Liability。写法如下：

```
public sealed override decimal Liability { get { return Mortgage; } }
```

也可以在类中使用sealed修饰符来密封整个类，含义是密封类中所有的虚方法。密封类比密封方法成员更常见。

3.2.7 base关键字

关键字base和关键字this很类似。它有两个重要目的：

- 从子类访问重载的基类方法成员
- 调用基类的构造方法（见下节）

本例中，House类用关键字base访问Asset类对Liability的实现：

```
public class House : Asset
{
  ...
  public override decimal Liability
  {
    get { return base.Liability + Mortgage; }
  }
}
```

通过关键字base，我们非显式地访问了Asset类的Liablility非虚属性。这表明，不管实例的运行时类型如何，都将访问Asset类的该属性。

如果Liability是隐藏属性而非重载的属性，该方法也同样有效。（也可以在调用方法前，转换成基类，以访问隐藏的成员。）

3.2.8 构造和继承

子类必须声明自己的构造方法。例如，如果定义如下子类：

```
public class Baseclass
{
  public int X;
  public Baseclass() {}
  public Baseclass(int x) { this.X = x; }
}
public class Subclass : Bassclass { }
```

下面的语句是不合法的：

```
Subclass s = new Subclass (123);
```

子类必须重新定义它想对外公开的任何构造方法。不过，定义子类的构造方法，也可以通过使用关键字base调用基类的某个构造方法实现：

```
public class Subclass : Baseclass
{
  public Subclass (int x) : base (x) { }
}
```

关键字base和this用法类似，但base关键字调用的是基类中的构造方法。

基类的构造方法总是先执行，这保证了*base*的初始化发生在作为子类的特例初始化之前。

1. 隐式调用基类无参数的构造方法

如果子类中的构造方法省略base关键字，那么基类的无参数构造方法将被隐式调用：

```
public class BaseClass
{
  public int X;
  public BaseClass () { X = 1; }
}
public class Subclass : BaseClass
{
  public Subclass() { Console.WriteLine (x); } // 1
}
```

如果基类没有无参数的构造方法，子类的构造函数中就必须使用base关键字。

2. 构造方法和字段初始化的顺序

当对象被实例化时，初始化按以下顺序进行：

（1）从子类到基类：

a. 初始化字段

b. 指定被调用基类的构造方法中的变量

（2）从基类到子类：

a. 构造方法体执行

代码如下：

```
public class B
{
  int x = 0; // 第三个执行
  public B (int x)
  {
    ...// 第四个执行
  }
}
public class D : B
{
  int y = 0; // 第一个执行
  public D(int x)
    : base (x+1)//第二个执行
  {
    ...// 第五个执行
  }
}
```

3.2.9 重载和解析

继承对方法的重载有特殊的影响。看下面的两个重载：

```
static void Foo (Asset a) { }
static void Foo (House h) { }
```

当重载被调用时，类型最明确的优先匹配：

```
House h = new House( );
Foo(h);                //调用Foo(House)
```

具体调用哪个重载是静态决定的（编译时）而不是在运行时决定。下面的代码调用Foo(Asset)，尽管a在运行时是House类型的：

```
Asset a = new House ( );
Foo(a);                //调用Foo(Asset)
```

提示：如果把Asset类转换成动态的（dynamic）（见第4章），会在运行时决定哪个重载被调用，这样就会基于对象的实际类型进行选择：

```
Asset a = new House(...);
Foo ((dynamic)a);    // 调用Foo(House)
```

3.3 object类型

object类（System.Object）是所有类型的最终基类。任何类型都可以向上转换成object类型。

为了说明这个类型的重要性，首先介绍通用栈。栈是一种遵循LIFO（Last-In First-Out，后进先出法）

的数据结构。栈有两种操作：*push*表示一个元素进栈和*pop*表示一个元素出栈。下面是能容纳10个对象的栈的简单实现：

```
public class Stack
{
  int position;
  object[] data = new object[10];
  public void Push (object obj)       { data[position++] = obj;}
  public object Pop()                 { return data[ -- position]; }
}
```

因为栈操作的对象是object类，所以可以实现任意类型的对象实例的进栈和出栈：

```
Stack stack = new Stack();
stack.Push("sausage");
string s = (string) stack.Pop();      // 向下类型转换，需要显式转换

Console.WriteLine(s);                 // sausage
```

承载了类的优点，object是引用类型。尽管如此，int等数值类型也可以和object类型相互转换，并加入栈中。C#的这个特性称为类型一致化，代码演示如下：

```
stack.Push(3);
int three = (int) stack.Pop();
```

当数值类型和object类型之间相互转换时，公共语言运行时（CLR）必须作一些特定的工作，实现数值类型和引用类型的转换这个过程被称为装箱和拆箱。

提示： 下面的“泛化”章节，我们会讲述如何改进Stack类，使之能更好地处理同类型数据。

3.3.1 装箱和拆箱

装箱是将数值类型实例转换成引用类型实例的行为。引用类型可以是object类或接口（本章后面将介绍接口）（注1）。本例中，我们将int类型装箱成一个object对象。

```
int x = 9;
object obj = x;       // 把int类型装箱
```

拆箱操作正好相反，它把object类型转换成原始的数值类型：

```
int y = (int)obj;     // 把int类型拆箱
```

拆箱需要显式进行。运行时检查提供的值类型是否与真正的对象类型相匹配，并在检查出错误时，抛出InvalidCastException。例如，下面的例子抛出异常，因为long类型和int类型不匹配：

```
object obj = 9;           // 9被自动识别为int类型
long x = (long) obj;      // 抛出InvalidCastException
```

下面的语句正确：

```
object obj = 9;
```

注1：引用类型也可以是System.ValueType或System.Enum（见第6章）。

```
long x = (int)obj;
```

以下语句也正确：

```
object obj = 3.5;                  // 3.5被自动识别为double类型
int x = (int) (double) obj;        // x 现在是3
```

最后一行代码中，（double）是拆箱操作，（int）是数值转换操作。

提示： 装箱转换对系统提供一致的数据类型至关重要。但这个体系并不是完美的：在“泛化”章节会介绍，数组和泛型的变量只能支持引用转换，不能支持装箱转换：

```
object[] a1 = new string [3]; //合法
object[] a2 = new int[3]      //出错
```

装箱和拆箱的实质是复制

装箱是把数值类型的实例复制到新对象中，而拆箱是把对象的内容复制回数值类型的实例中。下面的示例修改了i的值，但并不会改变它先前被装箱时拷贝的值：

```
int i = 3;
object boxed = i;
i = 5;
Console.WriteLine(boxed);    // 3
```

3.3.2 静态和运行时类型检查

C#在静态（编译时）和运行时都会进行类型检查。

静态类型检查使编译器能在程序没有运行的情况下检查正确性。下面代码会出错，因为编译器强制进行静态类型检查：

```
int x = "5";
```

在引用或拆箱操作的向下类型转换时，由CLR执行运行时类型检查。例如：

```
object y = "5";
int z = (int) y;             //运行时错误，向下类型转换失败
```

可以进行运行时类型检查，是因为堆栈中的每个对象都在内部存储了类型标识，这个标识可以通过调用object类的GetType方法读取。

3.3.3 GetType方法和typeof运算符

所有C#的类型在运行时都会维护System.Type类的实例。有两个基本方法可以获得System.Type对象：

- 在类实例上调用GetType方法
- 在类名上使用typeof运算符

GetType在运行时赋值；typeof在编译时静态赋值（如果使用泛型类型，那么它将由即时编译器解析）。

System.Type有针对类型名、程序集、基类等的属性。例如：

```
using System;

public class Point { public int X,Y; }

class Test
{
  static void Main()
  {
    Point p = new Point();
    Console.WriteLine(p.GetType().Name);                    // Point
    Console.WriteLine(typeof(Point).Name);                  // Point
    Console.WriteLine(p.GetType() == typeof(Point));        // True
    Console.WriteLine(p.X.GetType().Name);                  // Int32
    Console.WriteLine(p.Y.GetType().FullName);              // System.Int32
  }
}
```

同时System.Type还有作为运行时反射模式的访问器。在第19章介绍该内容。

3.3.4 ToString方法

ToString方法返回类实例的默认文本表述。这个方法被所有内置类型重载。下面是对int类使用ToString方法的示例：

```
int x = 1;
string s = x.ToString();// s 是"1"
```

可以用下面的方式重载自定义类的ToString方法：

```
public class Pansda
{
  public string Name;
  public override string ToString() { return Name; }
}
...

Panda p = new Panda { Name = "Petey"};
Console.WriteLine(p);// Petey
```

如果不重写ToString，那么这个方法会返回类型名称。

提示：当直接在数值型对象上调用像ToString这样的重载的object成员时，不会发生装箱。只有进行类型转换时，才会执行装箱操作：

```
Int x = 1;
String s1 = x.ToString();      // 调用没有装箱的值
Object box = x;
String s2 = box.ToString();    // 调用装箱后的值
```

3.3.5 列出object成员

列出object的所有成员：

```
Public class Object
```

```
{
  public Object();

  public extern Type GetType();

  public virtual bool Equals (object obj);
  public static bool Equals (object objA, object objB);
  public static bool ReferenceEquals (object objA, object objB);

  public virtual int GetHashCode();

  public virtual string ToString();
  protected override void Finalize();
  protected extern object MemberwiseClone();
}
```

Equals，ReferenceEquals和GetHashCode方法将在第6章的“等值比较”一节讲述。

3.4 结构体

结构体和类相似，不同之处在于：

- 结构体是值类型，而类是引用类型。
- 结构体不支持继承（除了隐式派生自object类的，更精确些说，是派生自System.ValueType）。

除了以下三项内容，结构体可以包含类的所有成员：

- 无参数的构造方法
- 终止器
- 虚成员

当表示值类型时使用结构体更理想而不用类。数值类就是很好的例子，此时，指派一个值的副本比指派一个引用更自然。因为结构体是值类型，每个实例不需要在堆栈上实例化，创建一个类型的多个实例时可以节约空间。例如，创建一个数值类型的数组，只需要分配一个堆栈空间。

3.4.1 结构体构造语义

结构体的构造语义如下：

- 隐含存在一个无法重载的无参数构造方法，将字段按位置零。
- 定义结构体的构造方法时，必须显式指定每个字段。
- 不能在结构体内初始化字段。

下面是一个声明和调用结构体构造方法的示例：

```
public struct Point
{
  int x, y;
  public Point(int x, int y) { this.x = x; this.y = y; }
```

```
}
...
Point p1 = new Point();              // p1.x和p1.y都是0
Point p2 = new Point(1,1);           // p2.x和p2.y都是1
```

以下的示例包含三个编译时错误：

```
public struct Point
{
  int x = 1;                              // 不合法：不能初始化字段
  int y;
  public Point() {}                       // 不合法：不能有无参数的构造方法

  public Point(int x) { this.x = x; }// 不合法：必须指定y值
}
```

如果把struct替换成class，上面的写法都合法。

3.5 访问权限修饰符

为了提高封装性，类或类成员会在声明中添加五个访问权限修饰符之一，来限制其他类和其他程序集对它的访问权限：

`public`
: 完全访问权限；枚举类型成员或接口隐含的访问权限。

`internal`
: 仅可访问程序集和友元程序集；非嵌套类型的默认访问权限。

`private`
: 仅在包含类型可见；类和结构体成员的默认访问权限。

`protected`
: 仅在包含类型和子类中可见。

`protected internal`
: protected和internal的访问权限并集Eric Lippert是这样解释的：默认情况下尽可能将所有成员定义为私有，然后每一个修饰符都会提高其访问级别。所以用protected internal修饰的成员在两个方面的访问级别都提高了。

提示： CLR有对protected和internal访问权限交集的定义，但C#并不支持。

3.5.1 举例

Class2在本程序集外可访问，而Class1不可以：

```
class Class1 {}                 // Class1是internal访问权限的（默认）
public class Class2 {}
```

ClassB有对本程序集的其他类提供的字段x；ClassA没有：

```
class ClassA { int x;          } // x是private访问权限的（默认）
class ClassB { internal int x; }
```

Subclass里的函数可以调用Bar但不能调用Foo：

```
class BaseClass
{
  void Foo(){}                   // Foo是private访问权限的（默认）
  protected void Bar() {}
}

class Subclass: BaseClass
{
  void Test1() { Foo(); }        // 出错：不能访问Foo
  void Test2() { Bar(); }        // 正确
}
```

3.5.2 友元程序集

在高级语义应用中，加上System.Runtime.CompilerServices.InternalsVisibleTo属性，就可以把internal成员提供给其他的友元程序集，用如下方法指定友元程序集：

```
[assembly: InernalsVisibleTo("Friend")]
```

如果友元程序集有强命名（见第18章），必须指定其完整的160字节公共键值：

```
[assembly: InternalsVisibleTo( "StrongFriend, PublicKey = 0024f000048c...")]
```

可以用LINQ查询从强命名的程序集中提取完整的公共键值（第8章详细介绍LINQ）：

```
String key = string.Join("",
Assembly.GetExecutingAssembly().GetName().GetPublicKey()
.Select(b=>b.ToString("x2"))
.ToArray());
```

提示： 在LINQPad中和本例相辅相成的一个例子，是浏览程序集后复制程序集的完整公共键值到剪贴板。

3.5.3 程序集的权限封顶

类权限是它内部声明的成员访问权限的封顶。关于权限封顶最常用的示例是internal类中的public成员。例如：

```
class C { public void Foo() {}}
```

类C的（默认）访问权限是internal，它作为Foo的最高访问权限，把Foo的权限改为internal。Foo指定为public的原因一般是，如果类C的访问权限要改成public的，重构会更容易。

3.5.4 访问权限修饰符的限制

当重载基类的函数时，重载函数的访问权限必须一致。例如：

```
class BaseClass { protected virtual void Foo() {} }
```

```
class Subclass1 : BaseClass { protected override void Foo() {} } // 正确
class Subclass2 : BaseClass { public override void Foo() {} }    // 错误
```

（一个例外情况是，如果在另一个程序集中重写一个protected internal方法，那么重写方法必须修饰为protected。）

编译器会阻止使用任何不一致的访问权限修饰符。例如，子类可以比基类访问权限低，但不能比基类访问权限高：

```
internal class A {}
public class B: A {}          //错误
```

3.6 接口

接口和类相似，但接口只为成员提供定义而不提供实现。接口和类的不同之处有：

- 接口的成员都是隐含抽象的。相反，类可以包含抽象成员和有具体实现的成员。
- 一个类（或结构体）可以实现多个接口。相反，类只能继承一个类，而结构体完全不支持继承（只能从System.ValueType派生）。

接口声明和类声明很类似，但接口不提供其成员的实现，因为它的所有成员都是隐式定义为抽象的，这些成员将由实现接口的类或结构体实现。接口只能包含方法、属性、事件、索引器，这些正是类中可以定义为抽象的成员。

下面是System.Collections命名空间中IEnumerator接口的定义：

```
public interface IEnumerator
{
  pool MoveNext();
  object Current { get; }
  void Reset();
}
```

接口成员总是隐式地定义成public的，并且不能用访问权限修饰符声明。实现接口意味着为其所有成员提供public的实现：

```
internal class Countdown : IEnumerator
{
  int count = 11;
  public bool MoveNext() { return count -->0;                 }
  public object Current  { get { return count; }              }
  public void Reset()    { throw new NotSupportedException(); }
}
```

可以把对象隐式转换为它实现的任意一个接口。例如：

```
IEnumerator e = new Countdown();
While(e.MoveNext())
  Console.Write(e.Current);       // 10987654210
```

提示：尽管Countdown是internal权限的类，通过把Countdown实例转换成IEnumerator，其内部实现IEnumerator接口的成员可以作为public成员访问。例如，如果同程序集中的一个公有类定义了如下方法：

```
Public static class Util
{
  Public static object GetCountDown()
  {
    Return new Countdown();
  }
}
```

另一个程序集的调用者可以执行：

```
IEnumerator e = (IEnumerator) Util.GetCountDown();
e.MoveNext();
```

如果IEnumerator被定义成internal，上面的方法则不正确。

3.6.1 扩展接口

接口可以从其他接口派生。例如：

```
public interface IUndoable              { void Undo(); }
public interface IRedoable : IUndoable  { void Redo(); }
```

IRedoable继承所有IUndoable的成员。换而言之，实现IRedoable的类型还必须实现IUndoable的成员。

3.6.2 显式接口实现

当实现多个接口时，有时成员标识符会有冲突。显式实现接口成员可以解决冲突。看下面的例子：

```
interface I1 { void Foo(); }
interface I2 { int Foo(); }

public class Widget: I1,I2
{
   public void Foo()
  {
    Console.WriteLine("Widget's implementation of I1.Foo");
  }
  int I2.Foo()
  {
    Console.WriteLine("Widget's implementation of I2.Foo");
    Return 42;
  }
}
```

因为I1和I2都有Foo标识符，Widget显式实现了I2的Foo方法。这是两个同名方法在同一个类中同时存在。调用显式实现成员的唯一方法是先转换为相应的接口：

```
Widget w = new Widget();
w.Foo();                     // Widget对I1.Foo的实现
((I1)w).Foo();               // Widget对I1.Foo的实现
((I2)w).Foo();               // Widget对I2.Foo的实现
```

另一个使用显式实现接口成员的原因是，隐藏那些和类的正常用法差异很大或有严重干扰性的成员。例如：实现ISerializable接口的类型，通常需要避免强调它的ISerializable成员，除非显式转换成这个接口。

3.6.3 虚方法实现接口成员

默认情况下，接口成员的实现是隐式定义为sealed。为了能重载，必须在基类中标识为virtual或者abstract。例如：

```
public interface IUndoable { void Undo(); }

public class TextBox: IUndoable
{
  Public virtual void Undo()
  {
    Console.WriteLine("TextBox.Undo");
  }
}

public class RichTextBox : TextBox
{
  public override void Undo()
  {
    Console.WriteLine("RichTextBox.Undo");
  }
}
```

不管从基类还是接口中调用接口成员，调用的都是子类的实现：

```
RichTextBox r = new RichTextBox();
r.Undo();                          //RichTextBox.Undo
((IUndoable)r).Undo();             // RichTextBox.Undo
((TextBox)r).Undo();               // RichTextBox.Undo
```

显式实现的接口成员不能标识为virtual的，也不能实现通常意义的重载。但是它可以被重新实现。

3.6.4 在子类中重新实现接口

子类可以重新实现基类中已经被实现的任意一个接口。不管基类中该成员是不是virtual的，当通过接口调用时，重新实现都能够屏蔽成员的实现。它不管接口成员是隐式还是显式实现都有效，但后者效果更好。

下面的例子中，TextBox显式实现IUndoable.Undo，所以不能标识为virtual。为了实现重载，RichTextBox必须重新实现IUndoable的Undo方法：

```
public interface IUndoable { void Undo(); }

public class TextBox : IUndoable
{
  void IUndoable.Undo() { Console.WriteLine("TextBox.Undo"); }
}

public class RichTextBox : TextBox, IUndoable
{
```

```
  public new void Undo() { Console.WriteLine ("RichTextBox.Undo"); }
}
```

从接口调用成员的重新实现时，调用的是子类的实现：

```
RichTextBox r = new RichTextBox();
r.Undo();                 // RichTextBox.Undo   例1
((IUndoable)r).Undo();    // RichTextBox.Undo   例2
```

假定RichTextBox定义不变，如果TextBox隐式实现Undo：

```
public class TextBox : IUndoable
{
  public void Undo() { Console.WriteLine( TextBox.Undo ); }
}
```

这样，为我们提供了另一种调用Undo的方法，如例3所示，它将"中断"系统。

```
RichTextBox r = new RichTextBox();
r.Undo();                    // RichTextBox.Undo      例1
((IUndoable)r).Undo();       // RichTextBox.Undo      例2
((TextBox)r).Undo();         // TextBox.Undo          例3
```

例3显示出，重新实现屏蔽仅当通过接口调用成员时有效，从基类调用时无效。这个特性通常不尽人意，因为它有二义性。重新实现主要是为了重载显式实现的接口成员。

接口重新实现的替代方法

即使在显式成员实现中，接口重新实现还是容易出问题，原因如下：

- 子类无法调用基类的方法
- 定义基类时不能预测方法是否会被重新实现，可能不允许这个潜在的功能

重新实现是未知子类时最不理想的方法，更好的选择是，在定义基类时不允许使用重新实现。有两种方法可以做到：

- 当隐式实现成员时，如果合适，标为virtual
- 当显式实现成员时，如果能预测子类可能要重载某些逻辑，用下面的模式：

```
public class TextBox: IUndoable
{
  void IUndoable.Undo(){ Undo(); }              // 调用下面的方法
  protected virtual void Undo() { Console.WriteLine("TextBox.Undo"); }
}

public class RichTextBox : TextBox
{
  protected override void Undo() { Console.WriteLine("RichTextBox.Undo"); }
}
```

如果不添加子类，可以把类标注成sealed，以占用接口的重新实现。

3.6.5 接口和装箱

将结构体转换成接口会引发装箱机制。调用结构体的隐式实现接口成员不会引发装箱。

```
Interface I { void Foo(); }
struct S : I { public void Foo() {} }

...
S s = new S();
s.Foo();// 没有装箱

I i = s;// 当转换为接口时引发装箱
i.Foo();
```

写类和写接口的对比

指导原则：

- 当能自然地共享实现时，使用类和子类
- 当实现是独立的，为类定义接口

观察下面的类：

```
abstract class Animal {}
abstract class Bird : Animal {}
abstract class Insect : Animal {}
abstract class FlyingCreature : Animal {}
abstract class Carnivore : Animal {}

// 具体实现类：

class Osrich : Bird {}
class Eagle : Bird, FlyingCreature, Carnivore {}  // 不合法
class Bee : Insect, FlyingCreature {}// 不合法
class Flea : Insect, Carnivore {}// 不合法
```

Eagle类、Bee类和Flea类无法编译，因为它们继承自多个类，这是非法的。为了解决这个问题，我们必须把其中的某些类转换成接口。问题是转换哪个类呢？遵照一般准则，我们看出所有昆虫类共享实现，所有鸟类共享实现，所以Insect和Bird还保持为类。相反，"能飞的生物"的"能飞"是独立的机制，"食肉动物"的"食肉"是独立的机制，所以我们把FlyingCreature和Carnivore转换成接口：

```
interface IFlyingCreature {}
interface ICarnivore {}
```

在特定的语义中，Bird和Insect可以对应Windows控件和Web控件，FlyingCreature和Carnivore对应IPrintable和IUndoable。

3.7 枚举类型

枚举类型是一种特殊的数值类型，可以在枚举类型中定义一组命名的数值常量。例如：

```
public enum BorderSide { Left, Right, Top, Bottom }
```

使用枚举类型的方法如下：

```
BorderSide topside = BorderSide.Top;
bool isTop = (topside == BorderSide.Top); // true
```

每个枚举成员都对应一个整型数，默认情况下：

- 对应的数值是int型的
- 按枚举成员的声明顺序，自动指定的常量为0、1、2……

可以指定其他的整数类型代替默认类型，例如：

```
public enum BorderSide : byte { Left, Right, Top, Bottom }
```

也可以显式指定每个枚举成员对应的值：

```
public enum BorderSide : byte { Left = 1, Right = 2, Top = 10, Bottom = 11 }
```

提示： 编译器还支持显式指定部分枚举成员。没有指定的枚举成员，在最后一个显式指定的值的基础上递增。上例等价于：

```
public enum BorderSide : byte
 { Left = 1, Right, Top = 10, Bottom }
```

3.7.1 枚举类型转换

枚举类型的实例可以和它对应的整型值互相显式转换：

```
int i = (int) BorderSide.Left;
BorderSide side = (BorderSide) i;
bool leftorRight = (int) side <= 2;
```

也可以显式地将一个枚举类型转换成另一个。假设HorizontalAlignment定义如下：

```
public enum HorizontalAlignment
{
  Left = BorderSide.Left,
  Right = BorderSide.Right,
  Center
}
```

两个枚举类型之间的转换通过对应的数值进行：

```
HorizontalAlignment h = (HorizontalAlignment) BorderSide.Right;
// 等价于：
HorizontalAlignment h = (HorizontalAlignment) (int)BorderSide.Right;
```

在枚举表达式中，编译器对数值0进行特别处理，不需要显式转换：

```
BorderSide b = 0;    // 不用转换
if(b == 0) ...
```

对0进行特别管理原因有两个：

- 第一个枚举成员经常被用作“默认”值
- 在合并枚举类型中，0表示不标识类型

3.7.2 标志枚举类型

枚举类型成员可以合并。为了避免混淆，合并枚举类型的成员要显式指定值，典型的增量为2。例如：

```
[Flags]
public enum BorderSides { Left = 0, Right = 2, Top = 4, Bottom = 8 }
```

使用位运算符操作合并枚举类型的值，例如|和&，它们作用在对应的整型数值上。

```
BorderSides leftRight = BorderSides.Left | BorderSides.Right;

if((leftRight & BorderSides.Left) != 0)
  Console.WriteLine ("Includes Left");        // Includes Left

string formatted = leftRight.ToString();      // "Left, Right"
BorderSides s = BorderSides.Left;
s|=BorderSides.Right
Console.WriteLine(s==leftRight);      // True

s ^=BorderSides.Right;                // 切换BorderSides.Right
Console.WriteLine(s);                 // Left
```

依照惯例，当枚举类型元素被合并时，一定要应用Flags属性。如果声明了一个没有标注Flags属性的枚举类型，枚举类型的成员仍然可以合并，但是当在该枚举实例上调用ToString方法时，输出一个数值而非一组名字。

一般来讲，合并枚举类型通常用复数名而不用单数名。

为了方便起见，可以把合并的成员直接放在枚举的声明内：

```
[Flags]
public enum BorderSides
{
    None=0,
    Left = 1, Right = 2, Top = 4, Bottom = 8
    LeftRight = Left | right,
    TopBottom = Top | Bottom,
    All = LeftRight | TopBottom
}
```

3.7.3 枚举运算符

枚举类型可以使用的运算符有：

```
=   ==   !   =   <   >   <=   >=   +   -   ^   &   |   ~
+=   -=   ++   --   sizeof
```

位运算符、算术运算符和比较运算符都返回对应整型值的运算结果。枚举类型和整型之间可以做加法，但两个枚举类型之间不能做加法。

3.7.4 类型安全问题

请看下面的枚举类型：

```
public enum BorderSide { Left, Right, Top, Bottom }
```

因为枚举类型可以和它对应的整型值相互转换，枚举的真实值可能超出枚举类型成员的数值范围。例如：

```
BorderSide b = (BorderSide) 12345;
Console.WriteLine (b);//12345
```

位操作和算术操作也会产生非法值：

```
BorderSide b = BorderSide.Bottom;
b++;// 不会报错
```

不合法的BorderSide枚举值会破坏如下程序：

```
void Draw(BorderSide side)
{
  if(side == BorderSide.Left)        {...}
  else if (side == BorderSide.Right) {...}
  else if (side == BorderSide.Top)   {...}
  else                               {...} // 这里被当成BorderSide.Bottom
}
```

其中一个解决方案是再加一个else子句：

```
...
else if (side == BorderSide.Bottom)...
else throw new ArgumentException ("Invalid BorderSide:" + side, "side");
```

另一个解决方案是，先检查枚举值的合法性。静态方法Enum.IsDefined有此功能：

```
BorderSide side = (BorderSide) 12345;
Console.WriteLine (Enum.IsDefined (typeof (BorderSide), side));   // False
```

但是，Enum.IsDefined对标志枚举类型不起作用，然而下面的方法（使用Enum.ToString()），可以在标志枚举类型合法时返回true。

```
static bool IsFlagDefined (Enum e)
{
  decimal d;
  return !decimal.TryParse(e.ToString(), out d);
}

[Flags]
public enum BorderSides { Left =1, Right = 2, Top =4, Bottom = 8 }

static void Main()
{
  for(int i = 0; i<=16; i++)
  {
    BorderSide side = (BorderSide)i;
    Console.WriteLine(IsFlagDefined(side) + "  " + side);
  }
}
```

3.8 嵌套类型

嵌套类型是声明在另一个类型内部的类型。例如：

```
public class TopLevel
{
  public class Nested { }                    // 嵌套类型
  public enum Color { Red, Blue, Tan }       // 嵌套枚举类型
}
```

嵌套类型有如下特征：

- 可以访问包含它的外层类中的私有成员，以及外层类所能访问的所有内容。
- 可以使用所有的访问权限修饰符修饰，而不仅限于public和internal。
- 嵌套类型的默认访问权限是private而不是internal。
- 从外层类以外访问嵌套类型，需要用外层类名称限定（就像访问静态成员一样）。

例如，为了从TopLevel类以外访问Color.Red，我们必须：

```
TopLevel.Color color = TopLevel.Color.Red;
```

所有类型都可以被嵌套，但只有类和结构体才能嵌套其他类型。

下面是从嵌套类型访问外层私有成员的例子：

```
public class TopLevel
{
  static int x;
  class Nested
  {
    static void Foo() { Console.WriteLine(TopLevel.x); }
  }
}
```

下面是对嵌套类型使用protected访问权限修饰符的例子：

```
public class TopLevel
{
  protected class Nested { }
}

public class SubTopLevel : TopLevel
{
  static void Foo() { new TopLevel.Nested(); }
}
```

下面是从外层类以外引用嵌套类的例子：

```
public class TopLevel
{
  Public class Nested { }
}

Class Test
```

在C#中创建类

```
{
  TopLevel.Nested n;
}
```

嵌套类型在编译器中的应用也很普遍，如编译器用于生成捕获迭代和匿名方法结构状态的私有类。

提示： 如果使用嵌套类型的主要原因，是避免一个命名空间中类型定义杂乱无章，那么可以考虑使用嵌套命名空间。使用嵌套类型的原因，应该是利用它较强的访问控制能力，或者是因为嵌套类必须访问其外层类的私有成员。

3.9 泛化

C#对书写能跨类型复用的代码，有两个不同的支持机制：继承和泛化。但继承的复用性来自基类，而泛化的复用性是通过带有“占位符”类的“模板”。和继承相比，泛化能提高类型的安全性以及减少类型的转换和装箱。

提示： C#的泛化和C++的模板是相似的概念，但它们的工作方法不同。我们将在“C#泛化和C++模板的比较”章节讲解。

3.9.1 泛型

泛型中声明类型参数——占位符类型，由泛型的使用者填充，它支持类型变量。下面是一个泛型Stack<T>用于在栈中存放T类型的实例。Stack<T>声明了单个类型参数T：

```
public class Stack<T>
{
  int position;
  T[] data = new T[100];
  public void Push (T obj)    { data[position++] = obj; }
  public T pop()              { return data[--position]; }
}
```

使用Stack<T>的方法如下：

```
Stack<int> stack = new Stack<int>();
stack.Push(5);
stack.Push(10);
int x = stack.Pop();        // x是10
int y = stack.Pop();        // y是5
```

Stack<int>用类型参数int填充T，运行时隐式创建类型。Stack<int>具有如下的定义（为了防止混淆类名用#代替，替换类用黑体显示）：

```
public class ###
{
  int position;
  int[] data;
  public void Push(int obj)   { data [position++] = obj; }
  public int Pop()            { return data[--position]; }
}
```

技术上，我们称Stack<T>是开放类型，称Stack<int>是关闭类型。在运行时，所有泛型的实例都是关闭的——占位符类型填充。下面的语句是不合法的：

```
var stack = new Stack<T>();   //不合法：T是什么类型?
```

只有在类或方法的内部，T才可以被定义为类型参数：

```
public class Stack<T>
{
  ...
  public Stack<T> Clone()
  {
    Stack<T> clone = new Stack<T>();    // 合法
    ...
  }
}
```


3.9.2 为什么存在泛化

泛化是为了代码能跨类型复用而设计的。假定我们需要一个类型完整的栈，如果没有泛型，解决方法之一是为每个需要的元素类型硬编码类的不同版本（如IntStack、StringStack等）。显然，这将导致产生大量的重复代码。另一个解决方法是写一个用object作为元素类型的栈：

```
public class ObjectStack
{
  int position;
  object[] data = new object[10];
  public void Push(object obj)  { data[position++] = obj; }
  public object Pop()           { return data[--position]; }
}
```

但是ObjectStack类不会像硬编码的IntStack类一样只处理整型元素。而且，ObjectStack需要用到装箱和向下类型转换，这些都不能在编译时检查：

```
// 假设规定只存储整型元素
ObjectStack stack = new ObjectStack();

stack.Push("s");// 错误类型，但不会报错!
int i = (int)stack.Pop();// 向下类型转换－运行时错误
```

我们需要的栈是既能对各种不同元素类型都支持，又要有一种方法能很容易地限定栈的元素为特定类型，以提高类的安全性和减少类型转换和装箱。泛化恰好通过允许参数化元素类型，提供了这些功能。Stack<T>具有ObjectStack和IntStack的全部优点。与ObjectStack的共同点是，Stack<T>只要写一次，就可以在所有类型上都工作。和IntStack的共同点是，Stack<T>的元素是特定的某个类型，它的独特之处在于操作的类是T，我们可以在编程时任意替换。

提示： ObjectStack在功能上等价于Stack<object>。

3.9.3 泛化方法

泛化方法指在方法的标识符内声明类参数。

使用泛化方法，许多基本数学函数可以用一个通用的方法实现。下面是交换两个任意类型值的泛化方法：

```
Static void Swap<T> (ref T a, ref T b)
{
  T temp = a;
  a = b;
  b = temp;
}
```

Swap<T>的使用方法如下：

```
int x = 5;
int y = 10;
Swap(ref x, ref y);
```

通常不需要提供参数的类型给泛化方法，因为编译器可以在后台推断出类型。如果有歧义，可以用下面的方法调用泛化方法：

```
Swap<int> (ref x, ref y);
```

在泛型中，只有新引入类型参数的方法才被归为泛化方法（用尖括号标出）。泛化类Stack中的Pop方法仅仅使用了类中已经存在的类型参数T，所以不属于泛化方法。

唯有方法和类可以引入类型参数。属性、索引器、事件、字段、构造方法、运算符等都不能声明类型参数，虽然它们可以参与使用所在的类中已经声明的类型参数。例如，可以写一个索引器返回一个泛化项：

```
public T this [int index] { get { return data[index]; } }
```

类似的，构造方法可以参与使用已存在的类型参数，但不能引入新的类型参数：

```
public Stack<T>() { }    // 不合法
```

3.9.4 声明类型参数

可以在声明类、结构体、接口、委托（见第4章）和方法时引入类型参数。其他的结构，如属性，不能引入类型参数，但可以使用类型参数。例如属性Value使用T：

```
public struct Nullable<T>
{
  public T Value { get; set; }
}
```

泛型或方法可以有多个参数。例如：

```
class Dictionary<TKey,TValue> {...}
```

实例化方法：

```
Dicionary<int,string> myDic = new Dictionary<int,string>();
```

或者：

```
var myDic = new Dictionary<int,string>();
```

泛型名和泛化方法名可以被重载，只要类型参数的数量不同即可。例如，下面的两个类名不会冲突：

```
class A<T> { }
class A<T1,T2> { }
```

提示：习惯上，泛型和泛化方法如果只有一个类型参数，只要参数的含义明确，一般把这个类型参数命名为T。当使用多个类型参数时，每个类型参数都使用T作为前缀，后面跟一个更具描述性的名称。

3.9.5 typeof和无绑定泛型

在运行时不存在开放的泛型：开放泛型被汇编成程序的一部分而关闭。但运行时可能存在无绑定（unbound）泛型，只用作类对象。C#中唯一指定无绑定泛型的方法是使用typeof运算符：

```
class A<T> { }
class A<T1,T2> { }
...

Type a1 = typeof(A<>);    // 无绑定类型（注意没有指定类参数）
Type a2 = typeof(A<,>);   // 用逗号表明有多个类参数
```

开放泛型类型一般与反射API（第19章）一起使用。

也可以用typeof运算符指定关闭的类：

```
Type a3 = typeof(A<int,int>);
```

或在运行时关闭的开放类：

```
class B<T> { void X() { Type t = typeof(T); } }
```

3.9.6 泛化的默认值

可以用default关键字获取赋给泛型类参数的默认值。引用类型的默认值是null，数值类型的默认值是将类的所有字段按位置0：

```
static void Zap<T> (T[] array)
{
  For(int i = 0; i<array.Length; i++)
    array[i] = default(T);
}
```

3.9.7 泛化的约束

默认情况下，类型参数可以被任何类型替换。在类型参数上应用约束，可以定义类型参数为指定类型。下面是可用的约束：

```
where T : base-class  //基类约束
where T : interface   //接口约束
where T : class       //引用类型约束
```

```
where T : struct        //数值类型约束（排除可空类型）
where T : new()         //无参数构造方法约束
where U : T             //裸类型约束
```

下面的示例GenericClass<T,U>中，要求T派生自SomeClass且实现接口Interface1，要求U提供无参数构造方法：

```
class     SomeClass { }
interface Interface1 { }

class GenericClass<T,U> where T : SomeClass, Interface1
                        where U : new()
{ }
```

约束可以应用在方法和类的任何类型参数的定义中。

基类约束或接口约束规定类型参数必须是某个类的子类或实现特定类或接口。这允许参数类可以被隐式转换成特定类或接口。例如，假定写一个泛化方法Max，返回两个值中的较大者。我们可以利用框架中定义的泛化接口IComparable<T>：

```
public interface IComparable<T>    // 接口的简化写法
{
  int CompareTo(T other);
}
```

如果other比this大，CompareTo方法返回正值。用此接口作为约束，我们可以写如下的Max方法（为了避免分散注意力，省略了null值检查）：

```
static T Max<T> (T a, T b) where T : IComparable<T>
{
  return a.CompareTo(b) > 0 ? a : b;
}
```

Max方法可以接收任意类型的参数，实现IComparable<T>接口（该接口包含大部分内置的类，如int和string）：

```
int z = Max(5,10);// 10
string last = Max("ant","zoo");// zoo
```

类约束和结构体约束规定T必须是引用类型或数值类型（不能为空）。结构体约束的一个很好的示例是System.Nullable<T>结构体（我们在第4章“可空类型”一节详细介绍）：

```
struct Nullable<T> where T: struct{...}
```

无参数构造方法约束要求T有一个公有的无参数构造方法。如果定义了这个约束，就可以在T中调用new()：

```
static void Initialize<T> (T[] array) where T : new()
{
  for (int i = 0; i< array.Length; i++)
    array[i] = new T();
}
```

裸类型约束要求一个类型参数从另一个类型参数派生。本例中，方法FilteredStack返回另一个Stack，返回的Stack只包含原类中满足U条件的部分元素。

```
class Stack<T>
{
  Stack<U> FilterStack<U>() where U:T {...}
}
```

3.9.8 子类泛型

泛型类和非泛型的类一样，都可以作为子类。子类可以让基类中的类型参数保持开放，如下所示：

```
class Stack<T>                   {...}
class SpecialStack<T> : Stack<T> {...}
```

子类也可以用具体类型关闭泛型参数：

```
class IntStack : Stack <int> {...}
```

子类还可以引入新的类型变量：

```
class List<T>                     {...}
class KeyedList<T,Tkey> : List<T> {...}
```

提示：技术上，子类型中所有类型参数都是新的：可以说子类型关闭后又重新开放了基类的类型参数。这表明子类可以为其重新打开的类型参数使用更有意义的新名称：

```
class List<T> {...}
class KeyedList<TElement, TKey> : List<TElement> {...}
```

3.9.9 自引用泛化声明

当关闭类型参数时，类可以用自己作为实体类：

```
public interface IEqutable<T> { bool Equals (T obj); }

public class Ballon : IEquatable<Balloon>
{
  public string Color { get; set; }
  public int CC { get; set; }

  public bool Equals(Balloon b)
  {
    If(b == null) return false;
    Return b.Color == Color && b.CC = CC;
  }
}
```

下面的写法也是合法的：

```
class Foo<T> where T : IComparable<T> {...}
class Bar<T> where T : Bar<T> {...}
```

3.9.10 静态数据

对每个封装的类来说，静态数据是全局唯一的：

```
class Bob<T> { public static int Count; }

class Test
{
  static void Main()
  {
    Console.WriteLine (++Bob<int>.Count);     // 1
    Console.WriteLine (++Bob<int>.Count);     // 2
    Console.WriteLine (++Bob<string>.Count);  // 1
    Console.WriteLine (++Bob<object>.Count);  // 1
  }
}
```

3.9.11 类型参数的转换

C#的类型转换运算符可以进行多种转换，包括：

- 数值型转换
- 引用型转换
- 装箱/拆箱转换
- 自定义转换（通过运算符重载，见第4章）

根据原数据的类型，在编译时决定转换成何种类型，并实现转换。因为编译时还不知道原数据的确切类型，使得泛型参数具有有趣的语义。如果导致二义性，编译器会产生一个错误。

最常见的应用是实现一个引用的类型转换：

```
StringBuilder Foo<T> (T arg)
{
  if(arg is StringBuilder)
    return (StringBuilder) arg;   // 编译不通过
   ...
}
```

因为不知道T的确切类型，编译器认为是自定义转换。最简单的解决方法就是改用as运算符，因为它不能进行自定义类型转换，所以不会产生二义性：

```
StringBuilder Foo<T> (T arg)
{
  StringBuilder sb = arg as StringBuilder;
  if(sb!= null) return sb;
  ...
}
```

一般的做法是，先转换成object类型。这种方法能够实现，是因为转换自或转换到object类型，被假定为不是自定义转换而是引用或装箱/拆箱转换。下例中，StringBuilder是引用类型，所以必须是引用转换：

```
Return (StringBuilder)(object) arg;
```

拆箱转换也会产生二义性。下面可能是拆箱转换、数值转换或自定义转换：

```
int Foo<T> (T x) { return (int)x; }   // 编译时错误
```

下面的方法先转换成object再转换成int（很明显这是一个拆箱转换）：

```
int Foo<T> (T x) { return (int)(object)x; }
```

3.9.12 协变

假定S是B的子类，如果X<S>允许引用转换成X<B>，那么称X为协变类。

提示：由于C#符号的共变性（和逆变性），所以“可改变”表示可以通过隐式引用转换进行改变——如A是B的子类，或者A实现B。数字转换、装箱转换和自定义转换都不包含在内。

换句话说，如果下面的语句合法，那么类IFoo<T>是协变类：

```
IFoo<string> b = ... ;
IFoo<object> s = b;
```

C#4.0中，泛化接口支持协变（泛化委托也支持，见第4章），但泛化类不支持。数组也支持协变（如果S是B的子类，S[]可以转换成B[]），这里作一些讨论和比较。

提示：协变性和逆变性（或简称变性）是高级概念。在C#中引入和强化协变的背后动机在于，允许泛化接口和泛型（尤其是框架中定义的那些，例如IEnumerable<T>）像人们所期待的那样，做更多的工作。编程人员可以利用协变性和逆变性这些概念的优点，而无需掌握它们背后的细节。

---

1. 类

为了保证静态类的安全性，泛化类不是协变的。看下面的例子：

```
class Animal { }
class Bear : Animal { }
class Camel: Animal { }

public class Stack<T>   //简单的栈实现
{
  int position;
  T[] data = new T[100];
  public void Push(T obj)     { data[position++] = obj; }
  public T Pop()              { return data[- - position]; }
}
```

下面的语句编译通不过：

```
Stack<Bear> bears = new Stack<Bear>();
Stack<Animal> animals = bears;          // 编译时错误
```

这个限制避免了下面代码发生运行时错误：

```
animal.Push(new Camel());       // 试图把Camel类加入Bear栈
```

但是，协变性的缺失可能妨碍复用性。假定下例中，我们想写一个Wash方法来操作整个Animal栈：

```
public class ZooCleaner
```

```
{
  public static void Wash ( Stack<Animal> animals) {...}
}
```

如果调用Wash方法操作Bear栈，将会导致编译时错误。解决方法是，重新定义一个带有约束的Wash方法：

```
class ZooCleaner
{
  public static void Wash<T> (Stack<T> animals) where T : Animal {...}
}
```

这样，我们就可以用如下方法调用Wash了：

```
Stack<Bear> bears = new Stack<Bear>();
ZooCleaner.Wash(bears);
```

另一种解决方法是让Stack<T>实现一个协变的泛化接口，后面会举出示例。

2. 数组

由于历史原因，数组array类型具有协变性。这表明如果B是A的子类（A和B都是引用类型），那么B[]可以转换成A[]。例如：

```
Bear[] bears = new Bear[3];
Animal[] animals = bears;          // 正确
```

这种可复用性的缺点在于，指定元素值可能导致运行时错误：

```
animal[0] = new Camel();       // 运行时错误
```

3. 接口

在C#4.0中，泛化接口对用out修饰符标注的类型参数支持协变。和数组不同，out修饰符保证了协变性的接口是完全类型安全的。为了阐述这一点，假定Stack类实现了下面的接口：

```
public interface IPoppable<out T> { T Pop(); }
```

T前的out修饰符是C#4.0的新特性，表明T只用在输出的位置（例如：方法的返回值）。out修饰符将接口标识为具有协变性的并允许以下操作：

```
var bears = new Stack<Bear>();
bears.Push (new Bear());
// Bears实现IPoppable<Bear>。我们可以把它转换成IPoppable<Animal>：
IPoppable<Animal> animals = bears;    // 合法
Animal a = animal.Pop();
```

基于接口有协变性的优点，将bears转换成animals是编译器允许的。它是类型安全的，因为编译器试图避免发生Camel类进栈，因为T仅能在输出的位置出现，所以不可能将Camel类输入接口。

提示： 接口中的协变和逆变的典型应用是使用接口：很少需要向协变性接口写入。确切地说，由于CLR的限制，为了协变性将方法参数标注为out是不合法的。

如前所述，我们可以利用类型转换的协变性解决复用性问题：

```
public class ZooCleaner
{
  public static void Wash(IPoppable<Animal> animals) {...}
}
```

提示：第7章所讲的IEnumerator<T>和IEnumerable<T>接口在Framwork4.0中都标识为协变的。这样，如果需要，可以将IEnumerable<string>转换成IEnumerable<object>。

如果在输入的位置使用协变类型参数（例如：方法的参数或可写属性），编译器会产生错误。

提示：不管泛型还是数组，协变（逆变）仅对引用转换的元素有效而对装箱转换无效。因此，如果写了一个接收IPoppable<object>类为参数的方法，可以用IPoppable<string>作为参数调用，而不能用IPoppable<int>。

3.9.13 逆变

前面我们已经知道，如果S是B的子类且X(S)允许引用类型转换到X(B)，则类X是协变的。那么能反方向转换的类是逆变的——从X(B)到X(S)。泛化接口支持逆变当泛型参数只出现在输入的位置，且被指定了in修饰符时。扩展我们前面的示例，如果Stack<T>类实现了如下接口：

```
public interface IPushble<in T> { void Push(T obj); }
```

下面的语句是合法的：

```
IPushable<Animal> animals = new Stack<Animal>();
IPushable<Bear> bears = animals;        // 合法
bears.Push (new Bear());
```

IPushable中没有成员输出类型为T，所以不会出现将animals转换成bears的问题（但是通过这个接口不能实现Pop方法）。

提示：Stack<T>类既可以实现IPushable<T>接口也可以实现IPoppable<T>接口——尽管T在两个接口中有不同的协变注解。它可以实现的原因是，只能从一个接口中激活协变，因此在协变转换之前，必须先选定IPoppable或IPushable。选定的接口会限制在合适的协变规则下进行合法的操作。

这也说明了为什么将类（如Stack<T>）定义为协变是没有意义的。

再看一个例子，下面的接口是.NET框架中的定义：

```
Public interface IComparer<in T>
{
  // 返回值为a和b的相对顺序
  int Compare(T a, T b);
}
```

因为这个接口是逆变的，我们可以用IComparer<object>比较两个字符串：

```
var objectComparer = Comparer<object>.Default;
// objectComparer实现了IComparer<object>接口
IComparer<string> stringComparer = objectComparer;
int result = stringComparer.Compare("Brett", "Jemaine");
```

与之相对的协变，如果将逆变的参数用在输出的位置（例如，作为返回值或在可读属性中），编译器将会报错。

3.9.14 C#泛化与C++模板的比较

C#的泛化与C++的模板在应用上很相似，但它们的工作原理却大不相同。两者都发生了生产者和消费者的语义关联，其中生产者的占位符类型由消费者填充。但是C#的泛化中，生产者类（例如开放类型List<T>）可以被编译到程序库中（如*mscorlib.dll*），因为在生产者和产生关闭类型的消费者间的语义交换直到运行时才实际发生。而C++模板中，这一语义交换是在编译时进行的。这表明我们不能在C++中以*.dll*形式部署模板库，因为生产者类仅存在于源代码中。这使得动态语法检查很难实现，更不用说联机创建或参数化类了。

为了深究这一情形的产生原因，我们重新观察C#的Max方法：

```
static T Max<T> (T a, T b) where T : IComparable<T>
{
  return a.CompareTo(b) > 0 ? a:b;
}
```

为什么我们不能像下面这样做呢？

```
static T Max<T> (T a, T b)
{
  return a > b ? a:b;                    //编译时错误
}
```

原因是，Max需要编译一次后对所有可能的T值都能工作。上例编译不能通过，因为对于任意类型T，“>”没有统一的含义。实际上，并不是所有的类型都支持“>”运算符。与之对应，下面的代码是用C++模板写的同样Max方法。该代码会为每个T值分别编译，对特定T呈现“>”不同语义，如果某个T不支持“>”运算符，则编译通不过：

```
Template <class T> T Max (T a, T b)
{
  return a > b ? a : b;
}
```

第4章

C#高级特性

本章内容是在前面章节概念的基础上探讨C#的高级特性。前4节必须按顺序阅读，后面各节可按任意顺序阅读。

4.1 委托

委托将方法调用者和目标方法动态关联起来。

*代理类型*定义了代理实例可调用的方法。具体地，它定义了方法的返回类型及其参数类型。下面的语句定义了一个代理类型Transformer：

```
delegate int Transformer (int x);
```

Transformer兼容任何返回类型为int且有一个int类型参数的方法，如：

```
static int Square (int x) { return x * x; }
```

将一个方法赋值给一个代理变量，可以创建一个代理实例：

```
Transformer t = Square;
```

它可以像调用方法一样调用：

```
int answer = t(3); // answer is 9
```

下面是一个完整例子：

```
delegate int Transformer (int x);
class Test
{
  static void Main()
  {
    Transformer t = Square;          // 创建委托实例
    Int result = t(3);               // 调用委托
    Console.WriteLine(result);       // 9
  }
  Static int Square (int x) { return x*x; }
}
```

委托实例实际上是调用者的代表：调用者先调用委托，然后委托调用目标方法。这种间接调用方式可以将调用者和目标方法分开。

调用委托和调用方法类似（因为委托的目的仅仅是提供一定程序的间接性）：

```
t(3);
```

语句：

```
Transformer t = Square;
```

是下面语句的简写：

```
Transformer t = new Transformer (Square);
```

提示： 技术上，当引用没有括号和参数的Square方法时，我们指定的是一组方法。如果该方法被重载，C#会根据选中委托的签名取出正确的重载方法。

语句：

```
t(3);
```

是下面语句的简写：

```
t.Invoke(3);
```

提示： 委托和回调相似，是捕获C函数指针等结构体的一般方法。

4.1.1 用委托写插入式方法

委托变量动态指定调用的方法。这个特性对于编写插入式方法非常有用。本例中有一个名为Transform的公共方法，它对整型数组的每个元素进行变换。为了指定插入式变换，Transform方法定义了一个委托参数。

```
public delegate int Transformer (int x);

class Util
{
  public static void Transform (int[] value, Transformer t)
  {
    for (int i = 0; i< values.Length; i++)
      values[i] = t(values[i]);
  }
}

class Test
{
  static void Main()
  {
    int[] values = { 1, 2, 3 };
```

```
    Util.Transform (values, Square);          // 动态挂接到Square函数
    foreach (int i in values)
      Console.Write (i + "  ");               // 1 4 9
  }

  static int Square (int x) { return x * x; }
}
```

4.1.2 多播委托

所有的委托实例都有多播能力。意思是一个委托实例不仅可以引用一个目标方法，而且可以引用一组目标方法。用运算符+和+=联合多个委托实例。例如：

```
SomeDelegate d = SomeMethod1;
d += SomeMethod2;
```

后一句等同于：

```
d = d + SomeMethod2;
```

现在调用d不仅会调用SomeMethod1，而且会调用SomeMethod2。委托按照添加的顺序依次被触发。

运算符-和-=从左边的委托操作数中移除右边的委托操作数。例如：

```
d-= SomeMethod1;
```

现在调用d只能触发SomeMethod2。

可以在委托变量上+或+=null值，等价于为变量指定一个新值：

```
SomeDelegate d = null;
d += SomeMethod1;            // 当d是null值时等价于d = SomeMethod1;
```

同样，在只有唯一目标方法的委托上调用-=等价于为该变量指定null值。

提示：委托是不可变的，因此调用+=或-=的实质是创建一个新的委托实例，并把它赋值给已有变量。

如果多播委托有非void的返回类型，调用者从最后一个触发的方法接收返回值。前面的方法仍然被调用，但返回值都被丢弃了。大部分情况下调用的多播委托都返回void类型，所以这个细小的差别就没有了。

提示：所有委托类型都是从System.MulticastDelegate派生的，System.MulticastDelegate继承自System.Delegate。C#将委托中使用的+、-、+=和-=都编译成System.Delegate的静态Combine和Remove方法。

多播委托的实例

假如写了一个需要运行很长时间的例程，该例程通过调用委托向它的调用者报告进程执行情况。在本例中，例程HardWork有一个ProgressReporter委托参数，例程调用该参数指明进程运行情况：

```
public delegate void ProgressReporter (int percentComplete)

public class Util
{
  Public static void HardWork (ProgressReporter p)
  {
    for (int i=0;i<10;i++)
    {
      p(i*10);                                // 调用委托
      System.Threading.Thread.Sleep(100);     // 模拟长时间运行
    }
  }
}
```

为了监视执行进度，在Main方法中创建一个多播委托实例p，这样通过两个独立的方法监视执行进度：

```
class Test
{
  static void Main()
  {
    ProgressReporter p= WritePRogressToConsole;
    p+=WriteProgressToFile;
    Util.HardWork(p);
  }

  static void WriteProgressToConsole (int percentComplete)
  {
    Console.WriteLine (percentComplete);
  }

  static void WriteProgressToFile (int percentComplete)
  {
    System.IO.File.WriteAllText ("progress.txt",
                                 percentComplete.ToString());
  }
}
```

4.1.3 实例方法和静态方法的Target属性

当委托对象指向一个实例方法时，委托对象不仅需维护到方法的引用，而且需维护到方法所属类实例的引用。System.Delegate类的Target属性表示这个类实例（当委托引用静态方法时为null）。例如：

```
public delegate void ProgressReporter (int percentComplete);

class Test
{
  static void Main()
  {
    X x = new X();
    ProgressReporter p = x.InstanceProgress;
    P(99);                                  // 99
    Console.WriteLine (p.Target ==x);       // True
    Console.WriteLine (p.Method);           // Void InstanceProgress(Int32)
  }
}
```

```
class X
{
  public void InstanceProgress (int percentComplete)
  {
    Console.WriteLine(percentComplete);
  }
}
```

4.1.4 委托中的泛型

委托类可以包含泛型参数。例如：

```
public delegate T Transformer<T> (T arg);
```

根据上面的定义，可以写一个泛化的实体类方法Transform，让它对任何类型都有效：

```
public class Util
{
  public static void Transform<T>(T[] values, Transformer<T> t)
  {
    for(int i=0;i<values.Length; i++)
      values[i] = t(values[i]);
  }
}

class Test
{
  static void Main()
  {
    int[] values = {1,2,3};
    Util.Transform(values,Square);        // Square函数中的动态钩子
    foreach (int i in values)
      Console.Write(i+"  ");              // 1 4 9
  }
  static int Square (int x) { return x * x; }
}
```

4.1.5 Func和Action委托

有了泛化委托，我们就可以写非常泛化的小型委托类，它们可以为具有任意返回类型和任意多参数的方法服务。它们是Func和Action委托，在System命名空间中定义（in和out用来标志变量，我们后面将会说明）：

```
delegate TResult Func <out TResult>                 ();
delegate TResult Func <in T, out TResutlt>          (T arg);
delegate TResult Func <in T1, in T2, out TResult>   (T1 arg1, T2 arg2);
……以此类推，最多到T16

delegate void Action                 ();
delegate void Action <int T>         (T arg);
delegate void Action <in T1, in T2>  (T1 arg1, T2 arg2);
……以此类推，最多到T16
```

这些委托都完全泛化，前面举例的Transformer委托，可以用一个带T类型参数并返回T类型值的Func委托代替：

```
public static void Transform<T> (T[] values, Func<T,T> transformer)
{
  for (int i = 0; i< values.Length; i++)
    values[i] = transformer (values[i]);
}
```

只有ref/out和指针参数在这些委托中没有涉及到。

提示：在Framework 2.0之前，并不存在Func和Action代理（因为那时还不存在泛型）。由于有这个历史问题，所以Framework的许多代码都使用自定义代理类型，而不使用Func和Action。

4.1.6 委托和接口

能用委托解决的问题，都可以用接口解决。例如，下面的ITransformer接口可以解决上面的“过滤”问题：

```
public interface ITransformer
{
  int Transform (int x);
}

public class Util
{
  public static void TransformAll (int[] values, ITransformer t)
  {
    for (int i = 0; i< values.Length; i ++)
      values[i] = t.Transform (values[i]);
  }
}

class Squarer : ITransformer
{
  public int Transform (int x) { return x * x; }
}
...

static void Main()
{
  int[] values = {1,2,3};
  Util.TransformAll (values, new Squarer());
  foreach(int i in values)
    Console.WriteLine (i);
}
```

在下面的情形中，委托可能是比接口更好的选择：

- 接口内只定义一个方法
- 需要多播能力
- 订阅者需要多次实现接口

虽然在ITransformer实例中我们不需要多播。但接口内只定义了一个方法。而且订阅者有可能为了支持不同的转换（如平方或立方转换），需要多次实现ITransformer接口。如果使用接口，由于Test类

只能实现一次ITransformer，必须为每种转换分别写一个类。这样做会很麻烦：

```
class Squarer: ITransformer
{
  public int Transform (int x) { return x * x; }
}
class Cuber: ITransformer
{
  public int Transform(int x) { return x * x * x; }
}
...
static void Main()
{
  int[] values = {1,2,3};
  Util.TransformAll(values, new Cuber());
  foreach (int i in values)
    Console.WriteLine(i);
}
```

4.1.7 委托的兼容性

1. 类型的兼容性

即使签名相似，委托类也互不兼容：

```
delegate void D1();
delegate void D2();
...
D1 d1= Method1;
D2 d2 = d1;                  // 编译时错误
```

提示： 但是允许下面的写法：

```
D2 d2= new D2(d1);
```

如果委托实例指向相同的目标方法，则认为它们是等价的：

```
delegate void D();
...
D d1 = Method1;
D d2 = Method1;
Console.WriteLine(d1==d2);   // True
```

如果多播委托按照相同的顺序引用相同的方法，则认为它们是等价的。

2. 参数的兼容性

当调用一个方法时，可以给方法的参数提供大于其指定类型的变量。这是正常的多态行为。基于同样的原因，委托也可以有大于它目标方法参数类型的参数，这称为逆变。

下面举例说明：

```
delegate void StringAction(string s);

class Test
{
  Static void Main()
  {
    StringAction sa = new StringAction (ActOnobject);
    sa("hello");
  }

  Static void ActOnobject(object o)
  {
    Console.WriteLine(o);   // hello
   }
}
```

（和类型参数变体一样，代理变体只适用于*引用转换*。）

委托很少替其他人调用方法。本例中，在调用StringAction时，参数类型是string。当该参数接下来作为目标方法时，参数隐式向上转换为object。

提示： 标准事件模式的设计宗旨是在其使用公共基类EventArgs时应用逆变。例如，可以用两个不同的委托调用同一个方法，一个传递MouseEventArgs，另一个传递KeyEventArgs。

---

3. 返回类型的兼容性

如果调用一个方法，得到的返回值类型可能大于请求的类型，这是正常的多态性行为。基于同样的原因，委托的返回类型可以小于它的目标方法的返回值类型，这被为协变。例如：

```
delegate object ObjectRetriever();

class Test
{
  static void Main()
  {
    ObjectRetriever o = new ObjectRetriever (RetriveString);
    object result = o();
    Console.WriteLine(result);// hello
  }
  static string RetriveString() { return "hello"; }
}
```

ObjectRetriever期望返回一个object，但object子类还会委托返回类型协变。

4. 泛化委托类型参数协变（C# 5.0）

第3章我们介绍了泛化接口怎样支持协变和逆变类型参数。委托也具有同样的功能。

如果要定义一个泛化委托类型，最好按照如下准则：

- 将只用在返回值的类型参数标注为协变（out）。
- 将只用在参数的类型参数标注为逆变（in）。

这样可以使协变自然地在类型之间继承关系。

下面的委托（在System命名空间中定义）支持协变：

```
delegate TResult Func<out TResult>();
```

允许：

```
Func<string> x = …;
Func<object> y = x;
```

下面的委托（在System命名空间中定义）支持逆变：

```
delegate void Action<in T> (T arg);
```

允许：

```
Action<object> x = ...;
Action<string> y = x;
```

4.2 事件

当使用委托时，一般会出现两种角色：广播者和订阅者。

广播者是包含委托字段的类，它决定何时调用委托广播。

订阅者是方法目标的接收者，通过在广播者的委托上调用+=和-=，决定何时开始和结束监听。一个订阅者不知道也不干涉其他的订阅者。

事件是使这一模式正式化的语言形态。事件是只显示委托中广播/订阅需要的子特性的结构。使用事件的主要目的在于：保护订阅互不影响。

声明事件最简单的方法是，在委托成员的前面加上event关键字：

```
// Delegate definition
public delegate void PriceChangedHandler (decimal oldPrice,
                                          decimal newPrice);
public class Broadcaster
{
  //Event declaration
  public event PriceChangedHandler PriceChanged;
}
```

Broadcaster类中的代码对Process有完全访问权限，并把它看作一个委托。Broadcaster类外面的代码只能在Progress事件上执行+=和-=操作。

事件内部是怎样工作的？

用下面的语句声明事件，编译器做了以下三件事情：

```
public class Broadcaster
```

```
{
  public event ProgressReporter   Progress;
}
```

首先，编译器把事件的声明编译成类似于下面的形式：

```
EventHandler _priceChanged;   // 私有委托
public event EventHandler PriceChanged
{
     add    { _priceChanged += value; }
     remove { _priceChanged -= value; }
}
```

add和remove关键字表示了明确的事件访问器，类似于属性访问器的行为。后面讲述怎么编写访问器。

其次，编译器在Broadcaster类里面除了找到+=和-=外还有对PriceChanged的引用，并把它们重定向到带下划线的委托字段_priceChanged。

再次，编译器将事件上的+=和-=编译成调用事件的add和remove访问器。有意思的是，当应用于事件时，它使得+=和-=行为是唯一的：和其他应用方式不同，这里+=和-=不是简单的+和-后跟赋值运算符的简写。

观察下面的实例。Stock类中每当Price值发生变化时，引发PriceChanged事件：

```
public delegate void PriceChangedHandler (decimal oldPrice,
                                          decimal newPrice);

public class Stock
{
  string symbol;
  decimal price;

  public Stock (string symbol) { this.symbol = symbol; }

  public event PriceChangedHandler PriceChanged;

  public decimal Price
  {
    get { return price; }
    set
    {
      if(price==value) return;       // 如果没有变化则退出
      decimal oldPrice=price;
      price = value;
      if(PriceChanged != null)       // 如果调用列表不为空，引发事件
        PriceChanged(price, value);
    }
  }
}
```

本例中，如果把event关键字去掉，PriceChanged就变成了普通的委托字段，运行结果不变。但是，Stock类就没有原先健壮了，这时订阅者可以通过以下方式互相影响：

- 通过重新指派PriceChanged替换其他的订阅者（不用+=运算符）。
- 清除所有的订阅者（将PriceChanged设置为null）。

- 通过调用其他委托，广播到其他订阅者。

提示：WinRT事件具有稍微不同的语义，它附加到一个事件会返回一个令牌，且又必须从事件分离。编译器自身会解决这个问题（通过保存一个内部令牌字典），所以可以把WinRT事件当作普通CLR事件使用。

4.2.1 标准事件模式

.NET框架为事件定义了一个标准模式。它的目的是保持框架和用户代码之间的一致性。标准事件模式的核心是System.EventArgs——预定义的没有成员的框架类（不同于静态Empty属性）。EventArgs是用于为事件传递信息的基类。在Stock实例中，我们定义EventArgs的子类，用来在事件PriceChanged被引发时，传递新旧Price值：

```
public class PriceChangedEventArgs : System.EventArgs
{
  public readonly decimal LastPrice;
  public readonly decimal NewPrice;

  public PriceChangedEventArgs (decimal lastPrice, decimal newPrice)
  {
    LastPrice = lastPrice;
    NewPrice = newPrice;
  }
}
```

考虑到复用性，EventArgs子类根据它包含的内容命名（而非根据将被使用的事件命名）。它一般以属性或只读字段将数据。

定义了EventArgs的子类，下一步是选择或定义事件的委托，需遵循三条原则：

- 委托必须以void作为返回值。
- 委托必须接受两个参数：第一个是object类，第二个是EventArgs的子类。第一个参数表明事件的广播者，第二个参数包含需要传递的额外信息。
- 委托的名称必须以*EventHandler*结尾。

框架定义一个名为System.EventHandler<>的泛化委托，该委托满足如下条件：

```
public delegate void EventHandler<TEventArgs>
  (object source, TEventArgs e) where TEventArgs : EventArgs;
```

提示：在泛化出现之前（C#2.0以前），我们只能以如下方式自定义委托：

```
public delegate void PriceChangedHandler
(object sender, PriceChangedEventArgs e);
```

由于历史原因，框架中的大部分事件使用的委托是这样定义的。

再下一步是定义选定委托类型的事件，这里使用泛化委托EventHandler：

```
public class Stock
{
  ...
```

```
  public event EventHandler<PriceChangedEventArgs> PriceChanged;
}
```

最后，该模式要求写一个受保护的（protected）虚方法引发事件。方法名必须和事件名一致，以*On*作前缀，并接受唯一的EventArgs参数：

```
public class Stock
{
  ...

  public event EventHandler<PriceChangedEventArgs> PriceChanged;

  protected virtual void OnPriceChanged (PriceChangedEventArgs e)
  {
    if(priceChanged != null) PriceChanged (this, e);
  }
}
```

提示： 在多线程情形下（见第14章），为了保证线程安全，在测试和调用委托前，需要将它指定到一个临时变量上：

```
var temp = PriceChanged;
if(temp!=null) temp(this,e);
```

此处提供了一个子类可以调用或重载事件的中心点（假定不是密封类）。下面是完整的例子：

```
using System;

public class PriceChangedEventArgs : EventArgs
{
  public readonly decimal LastPrice;
  public readonly decimal NewPrice;

  public PriceChangedEventArgs(decimal lastPrice, decimal newPrice)
  {
    LastPrice = lastPrice; NewPrice = newPrice;
  }
}

public class Stock
{
  string symbol;
  decimal price;

  public Stock(string symbol) { this.symbol = symbol; }

  public event EventHandler<PriceChangedEventArgs> PriceChanged;

  protected virtual void OnPriceChanged (PriceChangedEventArgs e)
  {
    if(PriceChanged != null) PriceChanged (this,e);
  }

  public decimal Price
  {
    get { return price; }
```

```
      set
      {
        if(price == value) return;
        OnPriceChanged (new PriceChangedEventArgs (price,value));
        price = value;
        OnPriceChanged (new PriceChangedEventArgs (oldPrice, price));
      }
    }
  }

  class Test
  {
    static void Main()
    {
      Stock stock = new Stock("THPW");
      stock.Price = 27.10M;
      // 注册PriceChanged事件
      stock.PriceChanged += stock_PriceChanged;
      stock.Price = 31.59M;
    }

    static void stock_PriceChanged(object sender, PriceChangedEventArgs e)
    {
      if((e.NewPrice  -e.LastPRice)/ e.LastPrice > 0.1M)
        Console.WriteLine("Alert, 10% stock price increase!");
    }
  }
```


如果事件不传递额外的信息，可以使用预定义的非泛化委托EventHandler。本例中，我们重写Stock类，使Price属性发生变化时，引发PriceChanged事件，事件除了传达已发生的信息外没有必须传递的信息。为了避免对EventArgs不必要的初始化，这里使用EventArgs.Empty属性。

```
  public class Stock
  {
    string symbol;
    decimal price;

    public Stock (string symbol) { this.symbol = symbol; }

    public event EventHandler PriceChanged;

    protected virtual void OnPriceChanged(EventArgs e)
    {
      if(PriceChanged != null) PriceChanged (this, e);
    }

    public decimal Price
    {
      get { return price; }
      set
      {
        if(price == value) return;
        price = value;
        OnPriceChanged(EventArgs.Empty);
      }
    }
  }
```

4.2.2 事件访问器

事件访问器是对+=和-=功能的实现。默认情况下，访问器由编译器隐式实现。例如下面的事件声明：

```
public event EventHandler PriceChanged;
```

编译器把它转换为：

- 一个私有的委托字段
- 一对公有的事件访问器函数，它们实现私有委托字段的+=、-=运算

可以通过明确定义事件访问器来代替这个过程。下面是手动实现前面实例中的PriceChanged事件：

```
private EventHandler _priceChange;            // 声明一个私有委托

public event EventHandler PriceChanged
{
  add { _priceChanged += value; }
  remove { _priceChanged -= value; }
}
```

本例和C#的默认访问器实现了同样的功能（但C#还能保证更新委托时的线程安全，见第21章）。通过自定义事件访问器，指示C#不要产生默认的字段和访问器逻辑。

显式定义的事件访问器，可以在委托的存储和访问上进行更复杂的操作。有以下三种常用情形：

- 当事件访问器仅为广播该事件的另一个类作交接。
- 当类定义了大量事件，而大部分时间有很少订阅者，例如Windows控件。这种情况下，最好在字典中存储订阅者的委托实例，因为字典比大量的空委托字段的引用需要更少的存储开销。
- 当显式实现声明事件的接口时。

下面示例阐明了第三种情形：

```
public interface IFoo { event EventHandler Ev; }

class Foo: IFoo
{
  private EventHander ev;

  event EventHandler IFoo.Ev
  {
    add { ev += value; }
    remove { ev -= value; }
  }
}
```

提示： 事件的add和remove部分被编译成add_*XXX*和remove_*XXX*方法。

4.2.3 事件的修饰符

和方法相似，事件可以是虚拟的（virtual）、重载的（overridden）、抽象的（abstract）或密封的

（sealed）。事件还可以是静态的（static），例如：

```
public class Foo
{
  public static event EventHandler<EventArgs> StaticEvent;
  public virtual event EventHandler<EventArgs> VirtualEvent;
}
```

4.3 Lambda表达式

Lambda表达式是写在委托实例上的匿名方法。编译器立即将lambda表达式转换成下面两种情形其中的一种：

- 委托实例
- `Expression<Tdelegate>`类型的表达式树，该表达式树将lambda表达式内的代码显示为可遍历的对象模式。这使得对lambda表达式的解释可以延迟到运行时。

下面的委托类型：

```
delegate int Transformer (int i);
```

可以指定和调用下面的`x=>x*x` lambda表达式：

```
Transformer sqr = x => x * x;
Console.WriteLine(sqr(3));    // 9
```

提示：编译器在内部将这种lambda表达式编译成一个私有方法，并把表达式代码移到该方法中。

Lambda表达式有以下形式：

（*参数*）=> *表达式或语句块*

为了方便，在只有一个可推测类型的参数时，可以省略小括号。

本例中，只有一个参数x，表达式是`x*x`：

```
x => x * x;
```

Lambda表达式使每个参数和委托的参数一致，表达式的类型（可以为`void`）和委托的返回值类型一致。

本例中，`x`和参数`i`一致，表达式`x*x`和返回值类型`int`一致，因此它和`Transformer`委托相兼容：

```
delegate int Transformer (int i);
```

Lambda表达式代码除了可以是表达式还可以是语句块。我们可以把上例改写成：

```
x => { return x * x; };
```

Lambda表达式通常和Func或Action委托一起使用，因此可以将前面的表达式写成下面的形式：

```
Func<int,int> sqr = x => x * x;
```

下面是一个带两个参数的表达式实例：

```
Func<string, string, int> totalLength = (s1,s2) => s1.Length + s2.Length;
int total = totalLength ("hello", "world");   // total的值为10
```

Lambda表达式是C#3.0中引入的概念。

4.3.1 明确指定Lambda参数类型

编译器通常可以根据上下文推断出lambda参数的类型。但当不能推断时，必须明确指定每个参数的类型。例如下面的表达式：

```
Func<int,int> sqr = x => x * x;
```

编译器可以推断出x是int型。也可以显式指定x的类型：

```
Func<int,int> sqr = (int x) => x * x;
```

4.3.2 捕获外部变量

Lambda表达式可以引用方法内的内部变量和参数（外部变量）。例如：

```
static void Main()
{
  int factor = 2;
  Func<int,int> multiplier = n => n * factor;
  Console.WriteLine(multiplier(3));           // 6
}
```

Lambda表达式引用的外部变量称为捕获变量。捕获变量的表达式称为一个闭包。

捕获的变量在真正调用委托时被赋值，而不是在捕获时赋值：

```
Int factor = 2;
Fanc<int,int> multiplier = n => n * factor;
factor =10;
Console.WriteLine(multiplier(3));             // 30
```

Lambda表达式可以自动更新捕获变量：

```
int seed = 0;
Func<int> natural = () => seed ++;
Console.WriteLine(natural());                 // 0
Console.WriteLine(natural());                 // 1
Console.WriteLine(seed);                      // 2
```

捕获变量的生命周期可以延伸到和委托的生命周期相同。下面的例子中，局部变量seed本应在Natural执行完后消失，但因为seed被捕获了，它的生命周期延长到和捕获它的委托natural的生命周期相同：

```
static Func<int> Natural()
{
  int seed =0;
  return () => seed ++;                       // 返回一个闭包
}
```

```
static void Main()
{
  Func<int> natural = Natural();
  Console.WriteLine (natural());     // 0
  Console.WriteLine (natural());     // 1
}
```

在lambda表达式内实例化的局部变量，在每次调用委托实例期间是唯一的。如果我们把上例改成在lambda表达式内部实例化seed，程序的结果（不是本例预期的结果）将与前面不同：

```
static Func<int> Natural()
{
  return() => { int seed = 0; return seed ++; };
}

static void Main()
{
  Func<int> natural = Natural();
  Console.WriteLine (natural());     // 0
  Console.WriteLine (natural());     // 0
}
```

C#高级特性

提示：在内部捕获是通过把被捕获的变量“提升”到私有类的字段实现的。当方法被调用时，实例化该类，并将其生命周期绑定在委托的实例上。

捕获循环变量

当捕获for或foreach语句中的循环变量时，C#把这些循环变量看作是声明在循环外部的。这表明每个循环捕获的是相同的变量。下面的程序输出333而不是012：

```
Action[] actions = new Action[3];

for(int i = 0; i<3; i++)
  action [i] = () => Console.Write(i);

foreach(Action a in actions) a();    // 333
```

每个闭包（加粗部分）捕获相同的变量i，（如果变量i在循环中保持不变，则非常有用；甚至可以根据需要在循环体中显式修改i的值。）每个委托在调用时才看到i值，此时，i值为3。把for循环展开更容易理解：

```
Action[] actions = new Action[3];
int i = 0;
action[0] = () => Console.Write(i);
i = 1;
action[1] = () => Console.Write(i);
i = 2;
action[2] = () => Console.Write(i);
i = 3;
foreach (Action a in actions) a();    // 333
```

结论是，如果希望输出012，需要把循环变量指定到循环内部的局部变量中。

```
Action[] actions = new Action[3];
for(int i = 0; i<3; i++)
{
  int loopScopedi = i;
  action[i] = () => Console.Write(loopScopedi);
}
foreach(Action a in actions) a();    // 012
```

这将导致闭包在每次循环中捕获不同的变量。

提示：在C# 5.0之前，foreach 循环以同样的方式工作：

```
Action[] actions = new Action[3];
int i = 0;

foreach (char c in "abc")
  actions [i++] = () => Console.Write (c);
foreach (Action a in actions) a(); // ccc in C# 4.0
```

这会引起很大的困惑：与for循环不同，foreach循环中的迭代变量是不可变的，所以人们可以将它当作循环体的局部变量。C# 5.0修复了这个问题，上面的例子会输出abc。

警告：在技术上，这是一个破坏性改变，因为在C# 5.0中重新编译C# 4.0可能会产生不同的结果。通常，C#团队都会尽力避免出现破坏性改变；然而，这里的“破坏”几乎肯定可以指出C# 4.0程序的一个未发现bug，而不是故意信任旧行为。

4.4 匿名方法

匿名方法是C#2.0引入的特性，并通过C#3.0的lambda表达式得到大大扩展。匿名方法类似于lambda表达式，但没有下面的特性：

- 确定类型的参数
- 表达式语法（匿名方法必须是语句块）
- 在指定到Expression<T>时，编译成表达式树的功能

写匿名方法的方法是：delegate关键字后面跟参数声明（可选），然后是方法体。以下面的委托为例：

```
delegate int Transformer (int i);
```

如下是实现和调用匿名方法：

```
Transformer sqr = delegate(int x) { return x * x; };
Console.WriteLine(sqr(3));                        // 9
```

第一行代码语义上等同于下面的lambda表达式：

```
Transformer sqr =      (int x) => { return x * x; };
```

或简写为：

```
Transformer sqr =          x => x * x;
```

提示：完全省略参数声明是匿名方法独有的特性——即使委托需要这些参数声明。在声明带默认空句柄的事件时很有用，例如：

```
public event EventHandler Clicked = delegate {} ;
```

这样，在引发事件时就避免了检查是否为null。下面的写法也是合法的：

```
Clicked += delegate { Console.WriteLine("clicked"); };        // 无参数
```

匿名方法和lambda表达式使用同样的方法捕获外部变量。

4.5 try语句和异常

try语句是为了处理错误或清理代码而定义的语句块。try块后面必须跟有catch块或finally块或两个块都有。当try块执行发生错误时，执行catch块；当结束try块时（如果当前是catch块，则当结束catch块时），不管有没有发生错误，都执行finally块来清理代码。

catch块可以访问Exception对象，该对象包含错误信息。catch中可以弥补错误也可以再次抛出异常。当仅仅是记录错误或要抛出更高层次的错误时，我们选择再次抛出异常。

finally块在程序中起决定作用，因为任何情况下它都被执行，通常用于清除任务，例如关闭网络连接等。

try语句举例如下：

```
try
{
  ... // 在执行本代码块时，可以抛出异常
}
catch (ExceptionA ex)
{
  ... // 处理ExceptionA类型的异常
}
catch(ExceptionB ex)
{
  ... // 处理ExceptionB类型的异常
}
finally
{
  ... // 清除代码
}
```

思考下面的程序：

```
class Test
{
  static int Calc (int x) { return 10/x; }

  static void Main()
  {
    int y = Calc(0);
```

C#高级特性

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

因为x是0，运行时抛出DivideByZeroException异常，程序终止。可以通过catch捕获异常来防止程序异常终止：

```
class Test
{
  static int Calc (int x) { return 10/x; }

  static void Main()
  {
    try
    {
      int y = Calc(0);
      Console.WriteLine(y);
    }
    catch(DivideByZeroException ex)
    {
      Console.WriteLine("x cannot be zero");
    }
    Console.WriteLine("program completed");
  }
}

输出：
x cannot be zero
program completed
```

提示： 上面是为了阐述异常处理列举的简单实例。在实际工作中，更好的处理方法是在调用Calc前显式检查除数是否为0。

异常处理需要几百个时钟周期，代价相对较高。

当抛出异常时，公共语言运行时CLR询问：当前是否在能捕获异常的try语句块中运行？

- 如果是，执行转到相应的catch块，如果catch块成功地运行结束，执行转到try下面的语句（如果存在，finally块优先执行）。
- 如果否，执行跳转到调用函数，重复上述询问（在执行finally块之后）。

如果没有用于处理异常的函数，用户将看到一个错误提示框，并且程序终止。

4.5.1 catch子句

catch子句定义捕获哪些类型的异常。这些异常应该是System.Exception或System.Exception的子类。

捕获System.Exception表示捕获所有可能的异常。用于以下情况：

- 不管哪种特定类型的异常，程序都可以修复
- 希望重新抛出该异常（可以在记入日志后）
- 程序终止前的最后一个错误处理

更常见的做法是，为了避免处理程序没有被定义的情况，只捕获特定类型的异常（例如OutOfMemoryException异常）。

可以在多个catch子句中处理各种异常类型（下例也可以通过显式属性检查实现）：

```
class Test
{
  static void Main (string[] args)
  {
    try
    {
      byte b = byte.Parse(args[0]);
      Console.WriteLine(b);
    }
    catch(IndexOutOfRangeException ex)
    {
      Console.WriteLine ("Please provid at least one argument");
    }
    catch (FormatException ex)
    {
      Console.WriteLine("That's Not a number!");
    }
    catch (OverflowException ex)
    {
      Console.WriteLine("You've given me more than a byte");
    }
  }
}
```

对于每一种给定的异常，只有一个catch子句执行。如果想要建立捕获更普遍的异常（如System.Exception）的安全网，必须把处理特定异常的语句放在前面。

如果不需要使用变量值，不指定变量也可以捕获异常：

```
catch(StackOverflowException)        // 没有变量
{
  ...
}
```

甚至，变量和类型可以都省略，表示指捕获所有异常：

```
catch { ... }
```

提示：除C#外的其他语言中，可以抛出不是派生自Exception类的对象（但不推荐）。CLR自动把此对象封装在RuntimeWrappedException类中（该类派生自Exception）。

4.5.2 finally程序块

无论是否抛出异常，也不管try程序块是否完全执行，finally程序块总是被执行。通常用finally程序块来清除代码。

在以下情况下执行finally程序块：

- catch块执行完成
- 由于跳转语句（如return或goto）离开try块
- try块结束

finally块为程序添加了决定性内容，在下面实例中，无论是否符合以下条件，打开的文件总能被关闭：

- try块正常结束
- 因为是空文件，提前返回EndOfStream
- 读取文件时抛出IOException异常

```
static void ReadFile()
{
  StreamReader reader = null;      // System.IO命名空间
  try
  {
    reader = File.OpenText("file.txt");
    if(reader.EndOfStream) return;
    Console.WriteLine(reader.ReadToEnd());
  }
  finally
  {
    if(reader!= null) reader.Dispose();
  }
}
```

本例通过在StreamReader上调用Dispose方法来关闭文件。在finally块中调用对象的Dispose方法是贯穿.NET框架的标准约定，且在C#的using语句中也明确支持。

using语句

许多类内部封装了非托管资源，例如文件管理、图像管理、数据库连接等。这些类实现System.IDisposable接口，这个接口定义了一个名为Dispose的无参数方法，用于清除这些非托管资源。using语句提供了一种在finally块中调用IDisposable接口对象的Dispose方法的优雅方法。

下面语句：

```
using (StreamReader reader = File.OpenText("file.txt"))
{
  ...
}
```

完全等同于：

```
StreamReader reader = File.OpenText("file.txt");
try
{
  ...
}
finally
{
  If(reader!=null)
    ((IDisposable)reader).Dispose();
}
```

我们在第12章详细讲解销毁模式。

4.5.3 抛出异常

可以在运行时或用户代码中抛出异常。下面的例子中Display方法抛出System.ArgumentNullException异常：

```
class Test
{
  static void Display (string name)
  {
    if(name == null)
      throw new ArguementNullException("name");

    Console.WriteLine(name);
  }

  static void Main()
  {
    try { Display(null); }
    catch ( ArgumentNullExcepion ex)
    {
      Console.WriteLine("Caught the exception");
    }
  }
}
```

重新抛出异常

可以捕获异常后再重新抛出，如下：

```
try { ... }
catch (Exception ex)
{
  // 记录错误
  ...
  throw;                  // 重新抛出同一个异常
}
```

提示：如果将throw替换为throw ex，那么这个例子仍然有效，但是新产生异常的StackTrace属性不再反映原始的错误。

此处的重新抛出异常，使记录错误后，异常不会被删除。它还可以用于取消对本应在外部处理的异常的内部处理：

```
using System.Net;          // （见第16章）
...

string s = null;
using (WebClient wc = new WebClient())
  try{ s= wc.DownloadString("http://www.albahari.com/nutshell/");}
  catch(WebException ex)
  {
```

C#高级特性

```
    if(ex.Status == WebExceptionStatus.NameResolutionFailure)
      Console.WriteLine ("Bad domain name");
    else
      throw;        // 不能处理其他种类的WebException，所以重新抛出
}
```

另一种常见情形是，重新抛出某个明确的异常类型，例如：

```
try
{
  ... // 从XML元数据中解析出DateTime值
}
catch(FormatException ex)
{
  throw new XmlException ("Invalid DateTime",ex);
}
```

重新抛出异常不会影响异常的StackTrace属性（见下一节）。当重新抛出一个不同类型的异常时，可以设置InnerException属性为原始的异常，这样有利于调试。几乎所有类型的异常都可以实现这一目的。

4.5.4 System.Exception的关键属性

System.Exception类的最重要的属性有下面几个：

StackTrace
: 表示从异常的起源到catch块的所有方法的字符串。

Message
: 描述异常的字符串。

InnerException
: 导致外部异常的内部异常（如果有的话）。它本身还可能有另一个InnerException。

提示： 所有的C#异常都是运行时异常，没有和Java对等的编译时检查异常。

4.5.5 常用的异常类型

下面的异常类型在CLR和.NET框架中广泛使用，可以在程序中自主抛出这些异常或者将它们作为基类来派生自定义异常类：

System.ArgumentException
: 当使用不恰当的参数调用函数时抛出，这通常表明程序有bug。

System.ArgumentNullException
: ArgumentException的子类，当函数参数为null（意料外的）时抛出。

System.ArgumentOutOfRangeException
: ArgumentException的子类，当属性值太大或太小时抛出（通常是数值型）。例如：当给只接收正值的函数传递负数时。

`System.InvalidOperationException`

不管是哪种特定的属性值，当对象的状态不符合方法正确执行的要求时抛出。例如读取未打开的文件或访问列表的下一个元素，而与之相关的列表在循环中已被部分修改。

`System.NotSupportedException`

该异常抛出表示不支持特定功能。例如：在只读的集合上应用Add方法。

`System.NotImplementedException`

该异常抛出表明某个方法还没有具体实现。

`System.ObjectDisposedException`

当函数调用的对象已被释放时抛出。

另一个常见的异常类型是NullReferenceException。当一个对象的值为null并访问它的成员时，CLR就会抛出这个异常（表示代码有bug）。下面的语句会直接抛出一个NullReferenceException异常（仅用于测试目的）：

```
throw null;
```


4.5.6 TryXXX方法的模板

当方法出错时，可以选择返回某种类型的错误代码或抛出异常。一般情况下，如果错误发生在正常的工作流之外或者希望方法的直接调用者不进行错误处理时，抛出异常。但有些情况下最好给调用者提供两种选择。以int类型为例，它为Parse方法定义了两个版本：

```
public int Parse    (string input);
public bool TryParse(string input, out int returnValue);
```

如果类型解析失败，Parse方法抛出异常，TryParse方法返回false。

可以用*XXX*方法调用*TryXXX*方法来实现该模板：

```
public return-type XXX (input-type input)
{
  return-type returnValue;
  if(!TryXXX(input, out returnValue))
    throw new YYYException(…);
  return returnValue;
}
```

4.5.7 异常的替换方法

在int.TryParse中，函数可以通过返回值或参数向调用函数返回错误代码。尽管对于简单的可预见的错误，这是个可行的方法，但对于所有的错误类型，它就显得捉襟见肘了，不仅会使方法的签名晦涩，而且增加了不必要的复杂性，使码混乱。它也不能推广到如运算符（例如：除法运算符）、属性等不是方法的函数上。此时可替换的方法是，把错误放在一个公共的地方，使其对调用堆栈中所有的函数都可见（例如：每个线程中存储当前错误的静态方法）。但是，这要求每个函数都参与错误传播模式，这本身就是冗长且容易出错的。

4.6 枚举类型和迭代

4.6.1 枚举类型

*Enumerator*是只读的，且游标只能在顺序值上向前移，实现下面对象之一：

- `System.Collections.IEnumerator`
- `System.Collections.Generic.IEnumerator<T>`

提示： 从技术上讲，任何具有MoveNext方法和Current属性的对象，都被看作是enumerator类型的。这个宽泛定义的存在是为了在C#1.0中，不经过装箱/拆箱操作就可以支持数值类型的枚举。这个优点在泛化概念出现后就不存在了，而且实际上C#2的动态绑定已经不支持了。

---

foreach语句用来在可枚举的对象上执行迭代操作。可枚举对象是顺序表的逻辑表示，它本身不是一个游标，但对象自身产生游标。可枚举对象可以是：

- IEnumerable或IEnumerable<T>的实现
- 具有名为GetEnumerator的方法返回一个*enumerator*

提示： IEnumerator和IEnumerable在System.Collections命名空间中定义。IEnumerator<T>和IEnumerable<T>在System.Collection.Generic命名空间中定义。

枚举类型的模板如下：

```
class Enumerator       // 通常实现IEnumerator或IEnumerator<T>
{
  public IteratorVariableType Current { get { … } }
  public bool MoveNext() { … }
}
class Enumerable       // 通常实现IEnumerable或IEnumerable<T>
{
  public Enumerator GetEnumerator() { … }
}
```

下面是用高级方法即foreach语句，遍历单词*beer*的每个字母：

```
foreach(char c in "beer")
  Console.WriteLine(c);
```

下面是用低级方法即不用foreach语句，遍历单词*beer*的每个字母：

```
using(var enumerator = "beer".GetEnumerator())
  while(enumerator.MoveNext())
  {
    var element = enumerator.Current;
    Console.WriteLine(element);
}
```

如果enumerator实现了IDisposable，那么foreach语句也起到using语句的作用，像前面讲过的例子

那样隐式释放枚举对象。

第7章详细介绍enumeration的接口。

4.6.2 集合初始化方法

可以通过一个简单的步骤实例化和填充可枚举的对象。例如：

```
using System.Collections.Generic;
...
List<int> list = new List<int> { 1,2,3 };
```

编译器将上面语句翻译成：

```
using System.Collections.Generic;
...
List<int> list = new List<int>();
list.Add(1);
list.Add(2);
list.Add(3);
```

它要求可枚举对象实现System.Collections.IEnumerable接口，并且有可调用的带适当个数参数的Add方法。

C#高级特性

4.6.3 迭代器

和foreach语句是枚举对象的使用者相对，迭代器是枚举对象的生产者。本例中，我们用迭代器返回Fibonacci数列表（Fibonacci数列表的每个数字是前两个数之和）：

```
using System;
using System.Collections.Generic;

class Test
{
  static void Main()
  {
    foreach(int fib in Fib(6))
      Console.Write(fib + " ");
  }

  static IEnumerable<int> Fibs(int fibCount)
  {
    for(int i = 0; prevFib =1, curFib =1; i< fibCount; i++)
    {
      yield return prevFib;
      int newFib = prevFib+curFib;
      prevFib = curFib;
      curFib = newFib;
    }
  }
}
```

输出：1 1 2 3 5 8

return语句表示该方法返回的值，而yield return语句表示从本枚举器产生的下一个元素。在每条yield语句中，元素返回给调用者，但要维护被调用者的状态，这样方法才能在调用者索取下一个元素时继续执行。被调用者状态的生命周期绑定在枚举器上，这样当调用者结束枚举时，状态可以被释放。

提示：编译器将迭代方法转换成实现IEnumerable<T>和/或IEnumerator<T>的私有类，迭代器的程序块逻辑在编译器写的枚举类中，被“反转”并连接MoveNext方法和Current属性。这表明当调用迭代方法时，所做的只是实例化编译器写的类，编写的代码并不真正运行。编写的代码只有在开始遍历结果序列时才真正运行，典型的如foreach语句。

4.6.4 迭代器语义

迭代器是包含一个或多个yield语句的方法、属性或索引器，迭代器必须返回以下四个接口之一（否则，编译器会报错）：

```
// Enumerable接口
System.Collections.IEnumerable
System.Collections.Generic.IEnumerable<T>

// Enumerator接口
System.Collections.IEnumerator
System.Collections.Generic.IEnumerator<T>
```

返回*enumerable*接口和返回*enumerator*接口的迭代器具有不同的语义，我们将在第7章介绍。

允许使用多个*yield*语句，例如：

```
class Test
{
  static void Main()
  {
    foreach (string s in Foo())
      Console.WriteLine(s);          // 打印 "One", "Two", "Three"
  }

  static IEnumerable<string> Foo()
  {
    yield return "One";
    yield return "Two";
    yield return "Three";
  }
}
```

1. yield break语句

yield break语句表明迭代器不返回后面的元素而是提前结束。我们可以把Foo修改成下面的实例：

```
static IEnumerable<string> Foo(bool breakEarly)
{
  yield return "One";
  yield return "Two";

  If(breakEarly)
```

```
    yield break;

  yield return "Three";
}
```

提示： 迭代器块中使用return语句是不合法的，必须使用yield break语句来代替。

2. 迭代器和try/catch/finally语句块

yield return语句不能出现在带catch子句的try语句块中：

```
IEnumerable<string> Foo()
{
  try { yield return "One"; }        // 不合法
  catch { … }
}
```

yield return语句也不能出现在catch或finally语句块中。出现这些限制的原因是编译器必须将迭代器转换为带有MoveNext、Current和Dispose成员的普通类，而且转换异常处理语句块可能会大大增加代码复杂性。

但是，可以在只带finally语句块的try块中使用yield语句：

```
IEnumerable<string> Foo()
{
  try { yield return "One"}  // 合法
  finally { … }
}
```

当枚举器到达队列末尾或者处理完毕时，finally语句块就可以执行了。如果提前使用break中断循环，那么foreach语句会显式地结束枚举器，这是一种正确使用枚举器的方法。在以显式方式处理枚举器时，我们可以使用一个标记来提前放弃枚举，而不需要结束它，从而绕过finally语句块。可以将显式的枚举封装到一个using语句中，从而规避这个风险。

```
string firstElement = null;
var sequence = Foo();
using (var enumerator = sequence.GetEnumerator())
  if (enumerator.MoveNext())
    firstElement = enumerator.Current;
```

4.6.5 组合序列

迭代器具有高度可组合性。我们可以扩展实例，使之只输出偶数斐波纳契数列：

```
using System;
using System.Collections.Generic;

class Test
{
  static void Main()
  {
    foreach (int fib in EvenNumbersOnly (Fibs(6)))
      Console.WriteLine (fib);
```

C#高级特性

```
    }
    static IEnumerable<int> Fibs (int fibCount)
    {
      for (int i = 0, prevFib = 1, curFib = 1; i < fibCount; i++)
      {
         yield return prevFib;
         int newFib = prevFib+curFib;
         prevFib = curFib;
         curFib = newFib;
      }
    }
    static IEnumerable<int> EvenNumbersOnly (IEnumerable<int> sequence)
    {
      foreach (int x in sequence)
        if ((x % 2) == 0)
          yield return x;
    }
  }
```

当执行MoveNext()操作时，每一个元素都只有到了最后才会进行计算。图4-1显示了随时间变化的数据请求和数据输出。

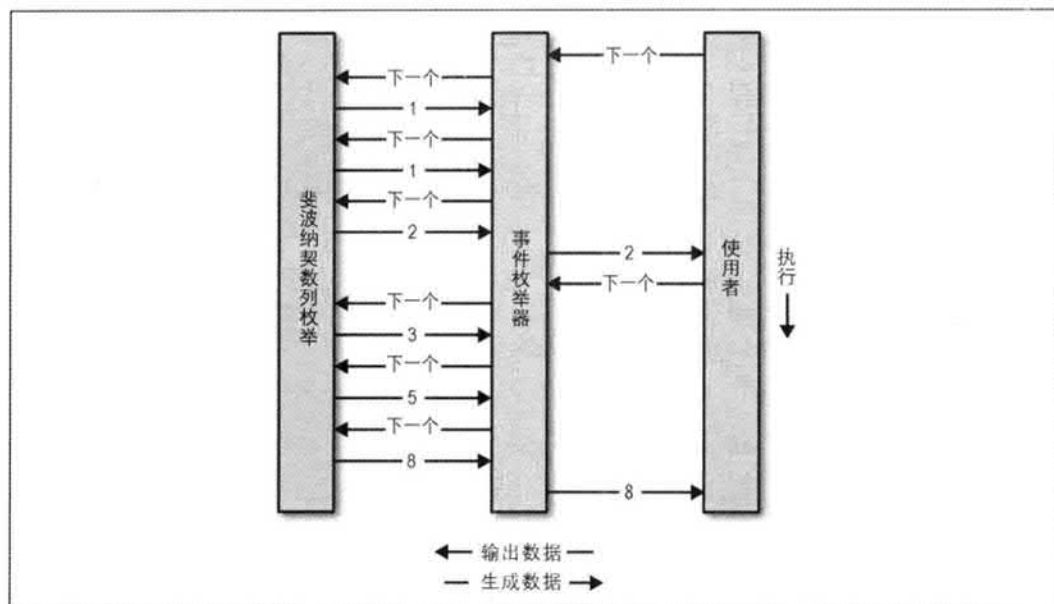


图4-1：组合序列

迭代器模式的组合性在LINQ中是非常有用的，我们将在第8章再进行讨论。

4.7 可空类型

引用类型可以表示一个不存在的值，即空引用。然而，数值类型一般不能用来表示空值。例如：

```
string s = null;      // 正确，是引用类型
int i = null;         // 编译错误，数值类型不能为空
```

若要在数值类型中表示空值，必须使用特殊的结构即可空类型（Nullable）。可空类型是由数据类型后加一个“?”表示的：

```
int? i = null;                      // 正确，可空类型
Console.WriteLine (i == null);      // 结果为真
```

4.7.1 Nullable<T>结构体

T?转换成System.Nullable<T>。而Nullable<T>是一个轻量的不变结构，它只有两个域，分别是Value和HasValue。System.Nullable<T>实质上是很简单的：

```
public struct Nullable<T> where T : struct
{
  public T Value {get;}
  public bool HasValue {get;}
  public T GetValueOrDefault();
  public T GetValueOrDefault (T defaultValue);
  ...
}
```

代码：

```
int? i = null;
Console.WriteLine (i == null);              // 结果为真
```

转换成：

```
Nullable<int> i = new Nullable<int>();
Console.WriteLine (! i.HasValue);     // 结果为真
```

当HasValue为假时尝试获取Value，程序会抛出一个InvalidOperationException异常。当HasValue为真时，GetValueOrDefault()会返回Value，否则返回new T()或者一个特定的自定义默认值。

T?的默认值是null。

4.7.2 隐式与显式的可空对象转换

从T到T?的转换是隐式的，而从T?到T的转换则必须是显式的。例如：

```
int? x = 5;         // 隐式
int y = (int)x;     // 显式
```

显式强制转换与直接调用可空对象的Value属性实际上是等价的。因此，当HasValue为假时，程序会抛出一个InvalidOperationException异常。

4.7.3 装箱和拆箱可空值

如果T?是装箱的，那么堆中的装箱值包含的是T，而不是T?。这种优化方式是可以实现的，因为装箱值是一个可能已经赋值为空的引用类型。

C#允许通过as运算符对一个可空类型进行拆箱。如果强制转换出错，那么结果为null：

```
object o = "string";
int? x = o as int?;
Console.WriteLine (x.HasValue);      // 结果为假
```

4.7.4 运算符提升

Nullable<T>结构体并没有定义诸如<、>或者==的运算符。尽管如此，下面的代码仍然能够正常编译和执行：

```
int? x = 5;
int? y = 10;
bool b = x < y;              // 结果为真
```

这是因为编译器会从实际值类型盗取或“提升”小于运算符。在语义上，它会将前面的比较表达式转换成以下语句：

```
bool b = (x.HasValue && y.HasValue) ? (x.Value < y.Value) : false;
```

换句话说，如果x和y都有值，那么它会通过int的小于运算符比较；否则，它会返回false。

运算符提升表示可以隐式地使用T的运算符来处理T?。可以定义专门针对T?处理的运算符来实现特殊用途的空值操作，但是在大多数情况中，最好通过编译器来自动地应用系统的空值逻辑。下面是一些应用示例：

```
int? x = 5;
int? y = null;

// 等于运算符示例
Console.WriteLine (x == y);    // 结果为假
Console.WriteLine (x == null);// 结果为假
Console.WriteLine (x == 5);    // 结果为真
Console.WriteLine (y == null);// 结果为真
Console.WriteLine (y == 5);    // 结果为假
Console.WriteLine (y != 5);    // 结果为真

// 关系运算符示例
Console.WriteLine (x < 6);     // 结果为真
Console.WriteLine (y < 6);     // 结果为假
Console.WriteLine (y > 6);     // 结果为假

// 其他运算符示例
Console.WriteLine (x + 5);     // 10
Console.WriteLine (x + y);     // null（指向空行）
```

编译器会基于运算符类型来执行空值逻辑。下一节将详细介绍这些不同的规则。

1. 等于运算符（==和!=）

提升等于运算符处理空值的方式与引用类型相似。这意味着两个空值是相等的：

```
Console.WriteLine (       null ==        null);      // 结果为真
Console.WriteLine ((bool?)null == (bool?)null);      // 结果为真
```

而且：

- 如果只有一个操作数为空，那么结果不相等。
- 如果两个操作数都不为空，那么比较它们的Value。

2. 关系运算符（<、<=、>=、>）

关系运算符的运算原则表明空值操作数的比较是无意义的。这意味着比较两个空值或比较一个空值与一个非空值的结果都是false。

```
bool b = x < y; // 转换:
bool b = (x == null || y == null) ? false : (x.Value < y.Value);
// b为假（假设x为5而y为null）
```

3. 其他运算符（+、-、*、/、%、&、|、^、<<、>>、+、++、--、!、~）

如果任意一个操作数为空，那么这些运算符都会返回空值。这类似于SQL采用的模式。

```
int? c = x + y;    // 转换:
int? c = (x == null || y == null)
         ? null
         : (int?) (x.Value + y.Value);
// c为null（假设x为5而y为null）
```

唯一例外的是计算bool?的&和|运算符，我们稍后将对这两个运算符进行介绍。

4. 混合使用可空和不可空运算符

可以混合使用可空和不可空类型，这是因为T与T?之间存在隐式转换机制：

```
int? a = null;
int b = 2;
int? c = a + b;       // c为null - 相当于 a + (int?)b
```

4.7.5 使用&和|运算符计算bool?

如果操作数的类型是bool?，那么&和|运算符会将null作为一个未知值看待。所以，null|true的结果为真，因为：

- 如果未知值为假，那么结果为真。
- 如果未知值为真，那么结果为真。

类似地，null & false的结果为假。这个操作与SQL相似。下面的例子说明其他一些组合用法：

```
bool? n = null;
bool? f = false;
bool? t = true;
Console.WriteLine (n | n);   // (null)
Console.WriteLine (n | f);   // (null)
Console.WriteLine (n | t);   // 结果为真
Console.WriteLine (n & n);   // (null)
Console.WriteLine (n & f);   // 结果为假
Console.WriteLine (n & t);   // (null)
```

4.7.6 空值合并运算符

??运算符是空值合并运算符，它既可用来计算可空值类型，也可用来计算引用类型。也就是说，如果操作数不为空，直接计算；否则，计算其默认值。例如：

```
int? x = null;
int y = x ?? 5; // y为5

int? a = null, b = 1, c = 2;
Console.WriteLine (a ?? b ?? c); // 1（第一个非空值）
```

??运算符的结果等同于使用一个显式默认值调用GetValueOrDefault，除非当变量不为空时传递给GetValueOrDefault的表达式从未求值。

4.7.7 可空值类型应用场景

可空类型常用来表示未知值。这种表示方法经常用于数据编程，其中类通过可空字段映射到数据表。如果这些字符是字符串类型（例如，Customer表的EmailAddress字段），因为字符串是CLR的一种引用类型，那么它是可以为空的。然而，大多数其他的SQL字段类型都有对应的CLR结构体类型，可空类型在将SQL映射到CLR时是非常有用的。例如：

```
// 映射到数据库中的Customer表
public class Customer
{
  ...
  public decimal? AccountBalance;
}
```

可空类型还可用于表示所谓环境属性的后备字段。如果环境属性为空，那么返回其父类的值。例如：

```
public class Row
{
  ...
  Grid parent;
  Color? color;

  public Color Color
  {
    get { return color ?? parent.Color; }
    set { color = Color == parent.Color ? (Color?)null : value; }
  }
}
```

4.7.8 可空类型的替代方法

在可空类型成为C#语言的一部分之前（例如C# 2.0以前版本），有许多方法可用来处理可空值类型，由于历史原因，这些方法的使用实例现在仍然存在于.NET Framework中。其中一个就是将一个特定的非空值指定为“空值”，使用实例就是字符串和数组类。String.IndexOf在找不到字符时会返回特殊值-1。

```
int i = "Pink".IndexOf ('b');
Console.WriteLine (s);         // -1
```

然而，Array.IndexOf只有在索引小于0时才会返回-1。最常见的规则是IndexOf返回比数据下限小1的值。在下一个例子中，IndexOf在未找到某个元素时返回0。

```
// 创建一个下限为1（而不是0）的数组：

Array a = Array.CreateInstance (typeof (string),new int[] {2}, new int[] {1});
a.SetValue ("a", 1);
a.SetValue ("b", 2);
Console.WriteLine (Array.IndexOf (a, "c"));   // 0
```

以下原因决定了指定个“魔法值”是有问题的：

- 使用它意味着每个数值类型有不同的空值表示方式。相反，可空值只用一种通用模式来处理所有的数值类型。
- 可能会出现指定不合理的值。在上一个例子中，可能总会使用-1。更早前表示一个未知账号余额和未知温度值的例子也采用相同的方法。
- 不对魔法值进行测试可能会导致出现错误值，而且这个错误值会直到运行时才被发现，它会产生一个意外的值。然而，没有测试空值的HasValue会马上抛出一个InvalidOperationException异常。
- 类型不支持获得空值。类型可以传达程序的意图，允许编译器检查其正确性，实现编译器的保持一致的规则集。

C#高级特性

4.8 运算符重载

运算符可以经过重载实现更自然的自定义类型语法。运算符重载非常适合用来表示最普通的基本数据类型的自定义结构体。例如，自定义数字类型很适合实现运算符重载。

下面的符号运算符都是可以重载的：

+(一元的)	-(一元的)	!	~	++
--	+	-	*	/
%	&	|	^	<<
>>	==	!=	>	<
>=	<=			

下面的运算符也可以重载：

- 隐式和显式转换（使用implicit和explicit关键字实现）
- 常量true和false

下面的运算符可以间接进行重载：

- 复合赋值运算符（例如+=、/=）可以通过重载非复合运算符（例如+、/）进行隐式重载。
- 条件运算符&&和||可以通过重载位运算符&和|进行隐式重载。

4.8.1 运算符函数

运算符是通过声明一个运算符函数进行重载的。运算符函数具有以下规则：

- 函数名是通过operator关键字及其后的运算符指定的。
- 运算符函数必须标记为static和public。
- 运算符函数的参数表示的是操作数。
- 运算符函数的返回类型表示的是表达式的结果。
- 运算符函数所声明的类型至少有一个操作数。

在下面的例子中，我们定义了一个名为Note的结构体，它表示一个音符，然后再重载运算符+：

```
public struct note
{
  int value;
  public Note (int semitonesFromA) { value = semitonesFromA; }
  public static Note operator + (Note x, int semitones)
  {
    return new Note (x.value + semitones);
  }
}
```

这个重载使我们能够给一个Note加上一个int：

```
Note B = new Note (2);
Note CSharp = B + 2;
```

重载一个赋值运算符会自动支持相应的复合赋值运算符。在上例中，因为我们重载了+号，所以也就能够使用+=了：

```
CSharp += 2;
```

4.8.2 重载等号和比较运算符

有些时候，我们在编写结构体或类时会需要重载等号和比较运算符。重载等号和比较运算符有一些特殊的规则和要求，我们将在第6章进行详细介绍。这些规则可以总结如下：

成对重载

C#编译器要求逻辑上成对的运算符必须同时定义。这些运算符包括（== !=）、（< >）和（<= >=）。

Equals和GetHashCode

在大多数情况中，如果重载了（==）和（!=），那么通常也需要重载对象中定义的Equals和GetHashCode方法，使之具有合理的行为。如果没有按要求重载，那么C#编译器将会发出警告。（详细介绍请参考第6章的“等式比较”。）

IComparable和IComparable<T>

如果重载了（< >）和（<= >=），那么还应该实现IComparable和IComparable<T>。

4.8.3 自定义隐式和显式转换

隐式和显式转换也是可重载的运算符。这些转换经过重载后一般能使强关联类型（如数字类型）之间的转换变得更加简明和自然。

如果要在弱关联类型之间进行转换，那么更适合采用以下方式：

- 编写一个具有该转换类型的参数的构造函数。
- 编写ToXXX和（静态）FromXXX方法进行类型转换。

正如关于类型的内容所介绍的，隐式转换的原理是它们能够保证转换成功，而且转换过程不会丢失信息。相反，当运行时环境要决定转换是否会成功，或者信息在转换过程中是否会丢失时，必须使用显式转换。

在这个例子中，我们在音符类型Note和双精度浮点数类型（表示以赫兹为单位的音符频率）之间定义了转换规则：

```
...
// 转换为赫兹
public static implicit operator double (Note x)
{
  return 440 * Math.Pow (2, (double) x.value / 12 );
}
// 转换自赫兹（精确到最接近的音节）
public static explicit operator Note (double x)
{
  return new Note ((int) (0.5 + 12 * (Math.Log (x/440) / Math.Log(2) ) ));
}
...
Note n = (Note)554.37;       // 显式转换
double x = n;                // 隐式转换
```

提示： 按照我们的方法，这个例子使用ToFrequency方法（与静态的FromFrequency方法）比隐式和显式运算符更容易实现。

警告： 自定义转换会被as和is运算符忽略：

```
Console.WriteLine (554.37 is Note);   // False
Note n = 554.37 as Note;              // 错误
```

4.8.4 重载true和false

虽然true和false运算符会在极少用的类型（即布尔型）上重载，但是它们不能转换为bool。例如，一个类型通过重载true和false实现了三个逻辑状态，这样的类型可以无缝地与条件语句和运算符一起使用，即if、do、while、for、&&、||和?:。结构体System.Data.SqlTypes.SqlBoolean支持这个转换功能。

例如：

```
SqlBoolean a = SqlBoolean.Null;
if (a)
  Console.WriteLine ("True");
else if (!a)
  Console.WriteLine ("False");
else
  Console.WriteLine ("Null");
```

输出：

```
Null
```

下面代码重新实现了SqlBoolean中可以验证true和false运算符的一部分代码：

```
public struct SqlBoolean
{
  public static bool operator true (SqlBoolean x)
  {
    return x.m_value == True.m_value;
  }

  public static bool operator false (SqlBoolean x)
  {
    return x.m_value == False.m_value;
  }

  public static SqlBoolean operator ! (SqlBoolean x)
  {
   if (x.m_value == Null.m_value) return Null;
   if (x.m_value == False.m_value) return True;
   return False;
  }

  public static readonly SqlBoolean Null = new SqlBoolean(0);
  public static readonly SqlBoolean False = new SqlBoolean(1);
  public static readonly SqlBoolean True = new SqlBoolean(2);

  private SqlBoolean (byte value) { m_value = value; }
  private byte m_value;
}
```

4.9 扩展方法

扩展方法允许一个现有类型扩展新的方法而不需要修改原始类型的定义。扩展方法是静态类的静态方法，其中第一个参数需要使用this修饰符，类型就是扩展的类型。例如：

```
public static class StringHelper
{
  public static bool IsCapitalized (this string s)
  {
    if (string.IsNullOrEmpty(s)) return false;
    return char.IsUpper (s[0]);
  }
}
```

虽然IsCapitalized扩展方法是字符串的实例方法，但是可以被调用，如下所示：

```
Console.WriteLine ("Perth".IsCapitalized());
```

在编译后，扩展方法调用会被转换回一个普通的静态方法调用：

```
Console.WriteLine (StringHelper.IsCapitalized ("Perth"));
```

这个转换过程如下：

```
arg0.Method (arg1, arg2, ...);              // 扩展方法调用
StaticClass.Method (arg0, arg1, arg2, ...); // 静态方法调用
```

接口也一样可以扩展：

```
public static T First<T> (this IEnumerable<T> sequence)
{
  foreach (T element in sequence)
    return element;

  throw new InvalidOperationException ("No elements!");
}
...
Console.WriteLine ("Seattle".First()); // S
```

扩展方法是C# 3.0后增加的特性。

4.9.1 扩展方法链

扩展方法类似于实例方法，也支持一种链接函数的方法。假设有下面两个函数：

```
public static class StringHelper
{
  public static string Pluralize (this string s) {...}
  public static string Capitalize (this string s) {...}
}
```

x和y是相同的，并且两者都与"sausage"进行比较，但是x使用扩展方法，而y使用静态方法。

```
string x = "sausage".Pluralize().Capitalize();
string y = StringHelper.Capitalize (StringHelper.Pluralize ("sausage"));
```

4.9.2 多义性与解析

1. 命名空间

只有命名空间在定义域内，我们才能够访问扩展方法。例如，下面的扩展方法IsCapitalized：

```
using System;

namespace Utils
{
  public static class StringHelper
  {
    public static bool IsCapitalized (this string s)
  {
      if (string.IsNullOrEmpty(s)) return false;
```

```
            return char.IsUpper (s[0]);
        }
    }
}
```

如果要使用IsCapitalized，那么下面的应用程序必须导入Utils，否则会出现编译错误：

```
namespace MyApp
{
  using Utils;

  class Test
  {
    static void Main()
    {
      Console.WriteLine ("Perth".IsCapitalized());
    }
  }
}
```

2. 扩展方法与实例方法

任何兼容的实例方法总是优先于扩展方法。在下面的例子中，即使被调用的参数x的类型为int，总是优先调用Test的Foo方法：

```
class Test
{
  public void Foo (object x) { }    // Foo方法总是被调用
}

static class Extensions
{
  public static void Foo (this Test t, int x) { }
}
```

在这个例子中，唯一调用扩展方法的方法是通过普通的静态调用语法；即，Extensions.Foo(...)。

3. 扩展方法与扩展方法

如果两个扩展方法名称相同，那么扩展方法必须作为一个普通的静态方法调用，才能够区分所调用的方法。然而，如果其中一个扩展方法具有更具体的参数，那么有更具体参数的方法优先级更高。

例如，下面这两个类：

```
static class StringHelper
{
  public static bool IsCapitalized (this string s) {...}
}
static class ObjectHelper
{
  public static bool IsCapitalized (this object s) {...}
}
```

下面的代码调用StringHelper的IsCapitalized方法：

```
bool test1 = "Perth".IsCapitalized();
```

如果要调用ObjectHelper的IsCapitalized方法，必须明确地定义：

```
bool test2 = (ObjectHelper.IsCapitalized ("Perth"));
```

类型一般比接口更具体一些。

4.10 匿名类型

匿名类型是一个由编译器临时创建来存储一组值的简单类。如果要创建一个匿名类型，我们可以使用new关键字，后面加上对象初始化语句，在其中指定该类型包含的属性和值。例如：

```
var dude = new { Name = "Bob", Age = 1 };
```

编译器会将这个语句（近似）转换为以下形式：

```
internal class AnonymousGeneratedTypeName
{
  private string name;       // 实际的字段名称是无关的
  private int    age;        // 实际的字段名称是无关的

  public AnonymousGeneratedTypeName (string name, int age)
  {
    this.name = name; this.age = age;
  }

  public string Name { get { return name; } }
  public int Age     { get { return age; } }

  // 重载了Equals和GetHashCode方法（见第6章）。
  // 也重载了ToString方法。
}
...

var dude = new AnonymousGeneratedTypeName ("Bob", 23);
```

必须使用var关键字来引用一个匿名类型，因为类型的名称是编译器产生的。

匿名类型的属性名可以从本身是一个标识符或以标识符结尾的表达式得到。例如：

```
int Age = 23;
var dude = new { Name = "Bob", Age, Age.ToString().Length };
```

等同于：

```
var dude = new { Name = "Bob", Age = Age, Length = Age.ToString().Length };
```

如果这两个匿名类型实例的元素是相同类型的，并且它们在相同的程序集中声明，那么它们在内部是相同的类型。

```
var a1 = new { X = 2, Y = 4 };
var a2 = new { X = 2, Y = 4 };
Console.WriteLine (a1.GetType() == a2.GetType());    // 结果为真
```

此外，匿名类型的Equals方法也被重载了，从而能够执行正确的等于比较运算：

```
Console.WriteLine (a1 == a2);                // 结果为假
```

```
Console.WriteLine (a1.Equals (a2));  // 结果为真
```

下面的代码可以创建多个匿名类型数组：

```
var dudes = new[]
{
  new { Name = "Bob", Age = 30 };
  new { Name = "Tom", Age = 40 }
};
```

匿名类型主要是在编写LINQ查询时使用（见第8章），并且是C# 3.0后才出现的特性。

4.11 动态绑定

动态绑定是将绑定（解析类型、成员和操作的过程）从编译时延迟到运行时。在编译时，如果程序员知道某个特定函数、成员或操作的存在，而编译器还不知道，那么动态绑定是很有用的。这种情况通常出现在操作动态语言（如IronPython）和COM时，而且如果不使用动态绑定，就只能使用反射机制。

动态类型是通过上下文关键字dynamic声明的：

```
dynamic d = GetSomeObject();
d.Quack();
```

动态绑定类型会告诉编译器“不要紧张”。我们认为d的运行时类型具有一个Quack方法，但是我们无法静态地证明这一点。由于d是动态的，所以编译器推迟到运行时才将Quack绑定给d。为了真正理解这个概念，我们需要先区分静态绑定和动态绑定。

4.11.1 静态绑定与动态绑定

典型的绑定例子就是在编译表达式时将一个名称映射到一个具体的函数上。如果要编译以下表达式，那么编译器需要找到Quack方法的实现：

```
d.Quack();
```

假设d的静态类型为Duck：

```
Duck d = ...
d.Quack();
```

最简单的情况是，编译器查找到Duck中无参数的Quack方法进行绑定。如果绑定失败，编译器会将搜索范围扩大到具有可选参数的方法、Duck的基类的方法和将Duck作为其第一个参数的扩展方法。如果还是没有找到匹配的方法，那么程序将出现一个编译错误。无论绑定的是什么样的方法，其底线是已知绑定是由编译器实现的，而且绑定是完全依赖于之前已经知道的操作数类型（在这里是d）。这就是所谓的静态绑定。

现在将d的静态类型改为object：

```
object d = ...
d.Quack();
```

调用Quack时，我们会遇到一个编译错误，因为虽然存储在d中的值包含了一个名为Quack的方法，但

是只有所包含信息为变量类型时编译器才知道方法的存在，而这里的类型是object。现在，将d的静态类型改为dynamic：

```
dynamic d = ...
d.Quack();
```

动态类型类似于object，同样不表现为一种类型。其区别是能够在编译时在不知道它存在的情况下使用它。动态对象是基于其运行时类型进行绑定的，而不是基于编译时类型。当编译器遇到一个动态绑定表达式时（通常是一个包含任意动态类型值的表达式），它仅仅对表达式进行打包，而绑定则在后面的运行时执行。

在运行时，如果一个动态对象实现了IDynamicMetaObjectProvider，那么这个接口将用来执行绑定。否则，绑定的发生方式就几乎像是编译器已经事先知道动态对象的运行时类型一样。我们将这两种方式称为自定义绑定和语言绑定。

提示： COM可认为是第三种绑定方式（见第25章）。

4.11.2 自定义绑定

自定义绑定是通过实现了IDynamicMetaObjectProvider（IDMOP）而实现的。虽然可以在用C#编写的类型上实现IDMOP，而且这也是很有用的，但是更常用的方法是从一种在DLR上用.NET实现的动态语言中获得一个IDMOP对象，如IronPython或IronRuby。这些对象已经隐式地实现了IDMOP，这是直接控制它们所执行的操作含义的一种方法。

我们将在第20章对自定义的绑定器进行深入讨论，下面编写一个简单的绑定器来演示这个特性：

```
using System;
using System.Dynamic;

public class Test
{
  static void Main()
  {
    dynamic d = new Duck();
    d.Quack();                  // 调用Quack方法
    d.Waddle();                 // 调用Waddle方法
  }
}

public class Duck : DynamicObject
{
  public override bool TryInvokeMember (
    InvokeMemberBinder binder, object[] args, out object result)
  {
    Console.WriteLine (binder.Name + " method was called");
    result = null;
    return true;
  }
}
```

Duck类实际上并没有Quack方法。相反，它使用自定义绑定拦截并解释所有的方法调用。

4.11.3 语言绑定

语言绑定是在一个动态对象未实现IDynamicMetaObjectProvider时出现的。语言绑定在处理设计不当的类型或.NET类型系统的内在限制时是非常有用的（我们将在第19章详细介绍）。使用数字类型的一个常见问题是它们没有通用接口。我们已经知道方法是可以动态绑定的，运算符也一样可以动态绑定：

```
static dynamic Mean (dynamic x, dynamic y)
{
  return (x + y) / 2;
}

static void Main()
{
  int x = 3, y = 4;
  Console.WriteLine (Mean (x, y));
}
```

其明显的好处是，不需要重复地处理每种数字类型。然而，会因此影响静态类型安全性以及可能会遇到运行时异常，而不是编译时错误。

提示：动态绑定会损坏静态类型安全性，但不会影响运行时类型安全性。与反射机制（第19章）不同，不能通过动态绑定绕过成员访问规则。

通过特殊设计，语言运行时绑定可以无限接近静态绑定的效果，使动态对象的运行时类型能够在编译时确定。在前一个例子中，如果我们在Mean代码中直接处理int类型，那么程序的行为是相同的。静态和动态绑定之间最显著的差异在于扩展方法，我们将在本章的“不可调用函数”中对进行介绍。

提示：动态绑定也会对性能产生影响。然而，由于DLR的缓存机制对同一个动态表达式的重复调用进行了优化，允许在一个循环中高效地调用动态表达式。这个优化机制能够使一个简单的动态表达式的处理负载对硬件的性能影响控制在100ns以内。

4.11.4 RuntimeBinderException

如果一个成员绑定失败，那么程序会抛出RuntimeBinderException异常。可以将它看作是一个运行时的编译错误：

```
dynamic d = 5;
d.Hello();              // 抛出RuntimeBinderException
```

抛出异常的原因是int类型没有Hello方法。

4.11.5 动态类型的运行时表现

dynamic和object类型之间可以执行一个深度等值比较。在运行时，下面这个表达式的结果为true：

```
typeof (dynamic) == typeof (object)
```

我们可以将这个原理扩展到结构类型和数组类型：

```
typeof (List<dynamic>) == typeof (List<object>)
typeof (dynamic[]) == typeof (object[])
```

与对象引用相似，动态引用也可以指向除指针类型以外的任意类型的对象：

```
dynamic x = "hello";
Console.WriteLine (x.GetType().Name); // String

x = 123; // 正确（尽管是同一个变量）
Console.WriteLine (x.GetType().Name); // Int32
```

在结构上，对象引用和动态引用之间没有任何区别。动态引用可以直接在它所指的对象上执行动态操作。可以从object转换到dynamic，以便执行任何一个希望在object上执行的动态操作：

```
object o = new System.Text.StringBuilder();
dynamic d = o;
d.Append ("hello");
Console.WriteLine (o); // hello
```

提示：在一个提供公开的dynamic成员的类型上的反射表明这些成员是为了注释objects。例如：

```
public class Test
{
  public dynamic Foo;
}
```

等价于：

```
public class Test
{
  [System.Runtime.CompilerServices.DynamicAttribute]
  public object Foo;
}
```

这使该类型的使用者知道Foo应该作为动态类型使用，同时使不支持动态绑定的语言能够回退到对象的使用上。

4.11.6 动态转换

动态类型会对其他所有类型进行隐式转换：

```
int i = 7;
dynamic d = i;
int j = d;            // 不需要强制转换（隐式转换）
```

如果要成功进行转换，动态对象的运行时类型必须能够隐式地转换到目标的静态类型上。前一个例子之所以能够成功执行，是因为int类型能够隐式地转换为long类型。

下一个例子会抛出RuntimeBinderException异常，因为int类型不能够隐式地转换为short类型：

```
int i = 7;
dynamic d = i;
short j = d;                // 抛出 RuntimeBinderException
```

4.11.7 var与dynamic

var和dynamic类型表面上是相似的，但是它们实际上是有区别的：

var由编译器确定类型。

dynamic由运行时确定类型。

```
dynamic x = "hello";        // 静态类型是dynamic，运行时类型是string
var y = "hello";            // 静态类型是string，运行时类型是string
int i = x;                  // 运行时错误
int j = y;                  // 编译时错误
```

一个由var声明的变量的静态类型可以是dynamic：

```
dynamic x = "hello";
var y = x;              // y的静态类型是dynamic
int z = y;              // 运行时错误
```

4.11.8 动态表达式

域、属性、方法、事件、构造函数、索引器、运算符和转换都是可以动态调用的。

不允许尝试获取返回类型为void的动态表达式的结果的操作，这与具有静态类型的表达式相似。两者的区别在于错误发生在运行时：

```
dynamic list = new List<int>();
var result = list.Add (5);          // 抛出RuntimeBinderException
```

包含动态操作数的表达式一般情况下本身就是动态的，因为缺少类型信息的结果是向下传递：

```
dynamic x = 2;
var y = x * 3;        // y的静态类型是动态的
```

这个规则有一些例外情况。首先，将一个动态表达式转换为静态类型会产生一个静态表达式：

```
dynamic x = 2;
var y = (int)2;       // y的静态类型是int
```

其次，构造函数调用总是产生静态的表达式，即使调用时使用的是动态参数。在这个例子中，x被设置为静态类型StringBuilder：

```
dynamic capacity = 10;
var x = new System.Text.StringBuilder (capacity);
```

此外，在少数情况下，包含动态参数的表达式也是静态的，包括传递索引到数组和委托创建表达式。

4.11.9 无动态接收者的动态调用

dynamic的标准用例是包含一个动态接收者。这意味着某个动态对象是动态函数调用的接收者：

```
dynamic x = ...;
x.Foo();              // x是接收者
```

然而，还可以使用动态参数调用已知的静态函数。这种调用受到动态重载解析的影响，并且可能包括：

- 静态方法
- 实例构造函数
- 已知静态类型的接收者的实例方法

在下面的例子中，动态获取边界的特殊Foo取决于动态参数的运行时类型：

```
class Program
{
  static void Foo (int x) { Console.WriteLine ("1"); }
  static void Foo (string x) { Console.WriteLine ("2"); }

  static void Main()
  {
    dynamic x = 5;
    dynamic y = "watermelon";

    Foo (x);                // 1
    Foo (y);                // 2
  }
}
```

因为其中不包含动态接收者，所以编译器能够静态地执行一个基本检查来确定该动态调用是否能够成功执行。它会检查是否存在名称和参数数量已确定的函数。如果没有发现候选函数，那么会得到一个编译时错误。例如：

```
class Program
{
  static void Foo (int x)    { Console.WriteLine ("1"); }
  static void Foo (string x) { Console.WriteLine ("2"); }

  static void Main()
  {
    dynamic x = 5;
    Foo (x, x);             // 编译错误——参数个数不符
    Fook (x);               // 编译错误——无此方法名
  }
}
```

4.11.10 动态表达式的静态类型

显然，动态类型用在动态绑定中。但是，静态类型在可能的情况下也用在动态绑定中。例如：

```
class Program
{
  static void Foo (object x, object y) { Console.WriteLine ("oo"); }
  static void Foo (object x, string y) { Console.WriteLine ("os"); }
  static void Foo (string x, object y) { Console.WriteLine ("so"); }
  static void Foo (string x, string y) { Console.WriteLine ("ss"); }

  static void Main()
  {
    object o = "hello";
```

```
        dynamic d = "goodbye";
        Foo (o, d);                    // os
      }
    }
```

Foo(o,d)的调用是动态绑定的，因为它其中的一个参数d的类型是dynamic。但是，由于o的类型是静态的，所以即使这个绑定是动态的，也会使用这个静态参数。在这个例子中，过载解析将选择Foo的第二种实现方法，这是由o的静态类型和d的动态类型决定的。换句话说，编译器会尽其所能地静态化。

4.11.11 不可调用的函数

有一些函数是不能够动态调用的。如下：

- 扩展方法（通过扩展方法语法）
- 接口的所有成员
- 子类隐藏的基类成员

理解这其中的原因对于理解动态绑定是非常有用的。

动态绑定包含两部分信息：调用的函数名和调用该函数的对象。然而，在这三种不可调用的情况中，还涉及到一个附加类型，它只在编译时可见。在C# 5.0中，我们是无法动态指定这类附加类型的。

在调用扩展方法时，它的附加类型是隐式的，这是扩展方法定义所在的静态类。编译器会根据源代码中的using指令来搜索这个类。这使得扩展方法成为只适用于编译时的概念，因为using指令在编译后会消失（当它们在绑定过程中完成了将简单的名称映射到完整命名空间名称的任务之后）。

当通过接口调用成员时，需要通过一个隐式或显式的转换来指定这个附加类型。有两种情况需要执行这个操作：调用显式实现的接口成员和调用另一程序集内部类型中实现的接口成员。我们可以通过下面两种类型来演示前一种情况：

```
interface IFoo { void Test(); }
class Foo : IFoo { void IFoo.Test() {} }
```

如果要调用Test方法，我们必须将它转换为IFoo接口。这种情况通过静态方式实现是最简单的：

```
IFoo f = new Foo();  // 隐式转换为接口
f.Test();
```

下面是动态类型转换的例子：

```
IFoo f = new Foo();
dynamic d = f;
d.Test();               // 抛出异常
```

粗体字显示的隐式转换是告诉编译器将f的后续成员调用绑定到IFoo上，而不是绑定到Foo上——换句话说，要通过IFoo接口的视角来查看该对象。然而，这个视角在运行时会消失，所以DLR无法完成这个绑定过程。整个消失过程如下所示：

```
Console.WriteLine (f.GetType().Name);         // Foo
```

类似的情况也出现在调用隐藏的基类成员上：必须通过强制转换或base关键字来指定一个附加类型，否则附加类型会在运行时丢失。

4.12 属性

前面已经介绍了在程序中使用修饰符给代码元素指定属性，如virtual或ref。这些结构是语言内置的。属性是添加自定义信息到代码元素（程序集、类型、成员、返回值和参数）的扩展机制。这种可扩展性对于C#语言中那些深度整合到类型系统且不需要特殊关键字或结构的服务是非常有用的。

属性的一个常见例子是序列化，就是将任意对象转换为一个特定格式或从特定格式生成一个对象的过程。在这种情况中，某个字段的属性可以指定该字段的C#表示方式和该字段的表示方式之间的转换。

4.12.1 属性类

属性是通过直接或间接地继承抽象类System.Attribute的方式定义的。如果要将一个属性附加到一个代码元素中，那么就需要在该代码元素之前用方括号指定属性的类型名称。例如，下面的例子将ObsoleteAttribute附加到Foo类上：

```
[ObsoleteAttribute]
public class Foo {...}
```

编译器能够识别这个属性，如果某个标记为弃用的类型或成员被引用时，编译器会发出警告。按照惯例，所有属性类型都以Attribute结尾。C#能够识别这个后缀，也可以在附加一个属性时省略这个后缀。

```
[Obsolete]
public class Foo {...}
```

ObsoleteAttribute是在System命名空间中声明的一种类型，如下所示（省略部分代码）：

```
public sealed class ObsoleteAttribute : Attribute {...}
```

C#语言和.NET Framework包含了大量的预定义属性。我们将在第17章介绍如何自定义属性。

4.12.2 命名与位置属性参数

属性可能具有一些参数。在下面的例子中，我们将XmlElementAttribute应用到一个类中。这个属性告诉XML串行器（位于System.Xml.Serialization）一个对象是如何以XML格式表示以及如何接受属性参数的。下面的属性将CustomerEntity类映射到一个名为Customer的XML元素上，这个元素属于命名空间*http://oreilly.com*：

```
[XmlElement ("Customer", Namespace="http://oreilly.com")]
public class CustomerEntity { ... }
```

属性参数分为两类：位置和命名。在前一个例子中，第一个参数是位置参数；第二个参数是命名参数。位置参数对应于属性类型的公开构造函数的参数；命名参数则对应于该属性类型的公开字段或公开属性。

当指定一个属性时，必须包含对应于其中一个属性构造函数的位置参数。命名参数则是可选的。

在第19章中，我们将介绍一些有效的参数类型及其求值规则。

4.12.3 属性目标

不需要显式指定，属性目标就是它后面紧跟的代码元素而且一般是一个类型或类型成员。然而，也可以给程序集附加一些属性。这要求显式地指定属性的目标。下面的例子使用CLSCompliant属性为整个程序集指定CLS的合法性：

```
[assembly:CLSCompliant(true)]
```

4.12.4 指定多个属性

一个代码元素可以指定多个属性。每一个属性可以列在同一对方括号中（用逗号分隔）或者在多对方括号中或者结合两种方式。下面三个例子在语义上是相同的：

```
[Serializable, Obsolete, CLSCompliant(false)]
public class Bar {...}

[Serializable] [Obsolete] [CLSCompliant(false)]
public class Bar {...}

[Serializable, Obsolete]
[CLSCompliant(false)]
public class Bar {...}
```

4.13 调用者信息属性（C# 5）

从C# 5开始，可以给可选参数添加3个调用者信息属性中的一个，它们可以让编译器从调用者源代码获取参数的默认值：

- [CallerMemberName]表示调用者的成员名称
- [CallerFilePath]表示调用者的源代码文件路径
- [CallerLineNumber]表示调用者源代码文件的行号

下面代码的Foo方法演示了这3个属性：

```
using System;
using System.Runtime.CompilerServices;

class Program
{
  static void Main()
  {
      Foo();
  }

  static void Foo (
    [CallerMemberName] string memberName = null,
    [CallerFilePath] string filePath = null,
    [CallerLineNumber] int lineNumber = 0)
  {
    Console.WriteLine (memberName);
    Console.WriteLine (filePath);
    Console.WriteLine (lineNumber);
  }
}
```

假设我们的程序位于c:\source\test\Program.cs，那么输出结果是：

```
Main
c:\source\test\Program.cs
8
```

和标准的可选参数一样，替代操作在调用位置完成。因此，方法的语法是这样：

```
static void Main()
{
  Foo ("Main", @"c:\source\test\Program.cs", 8);
}
```

调用者信息属性很适合用于记录日志以及实现一些模式，如当一个对象的某个属性发生变化时，触发一个变化通知事件。事实上，.NET框架有一个专门实现这个效果的标准接口INotifyPropertyChanged（位于System.ComponentModel）：

```
public interface INotifyPropertyChanged
{
  event PropertyChangedEventHandler PropertyChanged;
}

public delegate void PropertyChangedEventHandler
  (object sender, PropertyChangedEventArgs e);

public class PropertyChangedEventArgs : EventArgs
{
  public PropertyChangedEventArgs (string propertyName);
  public virtual string PropertyName { get; }
}
```

注意，PropertyChangedEventArgs方法需要接收发生变化的属性的名称。然而，通过使用[CallerMemberName]属性，就可以实现这个接口，并且不需要指定属性名称即可调用这个事件：

```
public class Foo : INotifyPropertyChanged
{
  public event PropertyChanged = delegate { }
  void RaisePropertyChanged ([CallerMemberName] string propertyName = null)
  {
    PropertyChanged (this, new PropertyChangedEventArgs (propertyName));
  }

  string customerName;
  public string CustomerName
  {
    get { return customerName; }
    set
    {
      if (value == customerName) return;
      customerName = value;
      RaisePropertyChanged();
      // 编译器会将上面一行代码转换为：
      // RaisePropertyChanged ("CustomerName");
    }
  }
}
```

4.14 不安全代码和指针

C#支持通过标记为不安全和使用/unsafe编译器选项编译的代码块中的指针直接进行内存操作。指针类型主要用来与C语言API进行互操作，但是也可用来访问托管堆以外的内存，或者分析严重影响性能的热点。

4.14.1 指针基础

对于每一种数值类型或指针类型*V*，它们都有对应的指针类型*V**。指针实例保存了值的地址。这可以认为是类型*V*，但是指针类型可以（不安全地）转换到其他任意指针类型。主要的指针运算符有：

运算符	作用
&	取址运算符返回指向某个值的地址的指针
*	引用运算符返回指针地址所指的值
->	指向成员的指针运算符是一个快捷语法，其中x->y等同于(*x).y

4.14.2 不安全代码

使用unsafe关键字标记一个类型、类型成员或语句块，就可以在该范围内使用指针类型和对内存执行C++中的指针操作。下面的例子使用指针实现一个位图的快速处理：

```
unsafe void BlueFilter (int[,] bitmap)
{
  int length = bitmap.Length;
  fixed (int* b = bitmap)
  {
    int* p = b;
    for (int i = 0; i < length; i++)
      *p++ &= 0xFF;
  }
}
```

不安全代码与对应的安全实现相比运行速度更快。在这个例子中，代码需要使用一个遍历数组索引和检查边界的嵌套循环。一个不安全的C#方法的执行速度可能比调用外部C函数更快一些，因为它不会出现由于离开托管的执行环境而造成负载。

4.14.3 fixed语句

fixed语句是用来锁定托管对象的，如前面例子中的位图。在程序的执行过程中，许多对象都是从堆中分配和回收的。为了避免内存浪费或碎片，垃圾回收器会移动这些对象。因此，如果一个对象的地址在引用时发生变化，那么指向该对象的指针是无效的，所以fixed语句会告诉垃圾回收器“锁定”这个对象，而且不要移动它。由于这可能对运行时效率产生一定的影响，所以fixed代码块只能短暂使用，而且堆分配应该避免出现在fixed代码块中。

在一个fixed语句中，可以获得一个指向任意数值类型、数值类型数组或字符串的指针。对于数组和字符串，这个指针实际上是指向第一个元素，它是一种数值类型。

在引用类型中以内联方式声明的数值类型是要求锁定该引用类型的，如下所示：

```
class Test
{
  int x;
  static void Main()
  {
    Test test = new Test();
    unsafe
    {
       fixed (int* p = &test.x)      // 锁定测试
       {
         *p = 9;
       }
       System.Console.WriteLine (test.x);
    }
  }
}
```

我们将在第25章的“将结构体映射到未托管内存”中进一步介绍fixed语句。

4.14.4 指向成员的指针运算符

除了&和*运算符，C#还支持C++中的->运算符，可以在结构体中使用：

```
struct Test
{
  int x;
  unsafe static void Main()
  {
    Test test = new Test();
    Test* p = &test;
    p->x = 9;
    System.Console.WriteLine (test.x);
  }
}
```


4.14.5 数组

1. stackalloc关键字

我们可以在代码中显式地通过stackalloc关键字分配栈中的内存。由于这部分内存是从栈上分配的，所以其生命周期仅限于方法的执行时间，这点与其他的局部变量相同。这个代码块可以使用[]运算符实现内存索引：

```
int* a = stackalloc int [10];
for (int i = 0; i < 10; ++i)
   Console.WriteLine (a[i]);          // 打印原始内存
```

2. 固定大小的缓冲区

我们也可以使用fixed关键字在一个结构体代码块中分配内存。

```
unsafe struct UnsafeUnicodeString
{
  public short Length;
```

```
    public fixed byte Buffer[30];
  }
  unsafe class UnsafeClass
  {
    UnsafeUnicodeString uus;
    public UnsafeClass (string s)
    {
      uus.Length = (short)s.Length;
      fixed (byte* p = uus.Buffer)
       for (int i = 0; i < s.Length; i++)
         p[i] = (byte) s[i];
     }
  }
  class Test
  {
    static void Main() { new UnsafeClass ("Christian Troy"); }
  }
```

上例是UnsafeClass的实例，使用fixed关键字将对象锁定到包含该缓冲区的堆上。因此，fixed表示两个不同的方面：大小固定和位置固定。这两者通常一起使用，即固定大小的缓冲区必须在固定的位置上使用。

4.14.6 void*

空指针（void*）不给出假定底层数据的具体类型，它对于处理原始内存的函数是非常有用的。任意指针类型都可以隐式地转换为void*。void*不可以被解除引用，算术运算符不能通过void指针执行，例如：

```
  class Test
  {
    unsafe static void Main()
    {
      short[ ] a = {1,1,2,3,5,8,13,21,34,55};
        fixed (short* p = a)
        {
          //sizeof返回值——类型的大小，单位为字节
          Zap (p, a.Length * sizeof (short));
        }
      foreach (short x in a)
        System.Console.WriteLine (x);  // 打印所有零
    }
    unsafe static void Zap (void* memory, int byteCount)
    {
      byte* b = (byte*) memory;
        for (int i = 0; i < byteCount; i++)
      *b++ = 0;
    }
  }
```

4.14.7 指向未托管代码的指针

指针也很适于访问位于托管堆之外的数据（如与C DLL或COM交互时），以及处理不在主存中的数

据（如图形化内存或嵌入式设备的存储介质）。

4.15 预处理指令

预处理指令向编译器提供关于代码范围的额外信息。最常用的预处理指令是条件指令，它提供了一种将某些代码加入或排除出编译范围的方法。例如：

```
#define DEBUG
class MyClass
{
  int x;
  void Foo()
  {
    # if DEBUG
    Console.WriteLine ("Testing: x = {0}", x);
    # endif
  }
  ...
}
```


在这个类中，Foo语句是根据DEBUG符号的出现而有选择地进行编译。如果删除DEBUG符号，那么这条语句就无法编译。预处理符号可以定义在源代码中（如上例），然后它们会通过命令行选项/define:*symbol*传递给编译器。

通过#if和#elif指令，可以使用||、&&和!运算符在多个符号上执行或、与、非操作。下面的指令会指示编译器当定义了TESTMODE符号而未定义DEBUG符号时执行后面的代码：

```
#if TESTMODE && !DEBUG
...
```

然而，一定要记住你不是在开发普通的C#表达式，并且你所操作的符号与变量是毫无关系的——不管是静态或是动态的。

#error和#warning符号会要求编译器在遇到一些不符合要求的编译符号时产生一条警告或错误信息，从而防止出现条件指令的偶然误用。表4-1列出了预处理指令。

表4-1：预处理指令

预处理指令	操作
#define 符号	定义一个符号
#undef 符号	取消一个符号定义
#if 符号 [运算符 符号2]...	判断符号；运算符可以是==、!=、&&和||，后面跟#else、#elif和#endif
#else	执行到下一个#endif之前的代码
#elif 符号 [运算符 符号2]	组合#else分支和#if判断
#endif	结束条件指令
#warning文本	显示在编译器输出中的警告文本
#error文本	显示在编译器输出中的错误文本
#pragma warning [disable | restore]	Disables/restores compiler warning(s)

表4-1：预处理指令（续）

预处理指令	操作
#line [数字 ["*file*"] | hidden]	数字指定源代码的行号；*file*是计算机输出的文件名hidden指示调试器忽略从这里开始到下一个#line指令的代码
#region名称	标记大纲开始位置
#end区域	结束一个大纲区域

4.15.1 Conditional属性

使用Conditional修饰的属性只有在出现指定的预处理符号时才编译。例如：

```
// file1.cs
#define DEBUG
using System;
using System.Diagnostics;
[Conditional("DEBUG")]
public class TestAttribute : Attribute {}

// file2.cs
#define DEBUG
[Test]
class Foo
{
  [Test]
  string s;
}
```

如果DEBUG符号位于*file2.cs*的范围内，那么编译器不会加入[Test]属性。

4.15.2 编译指令警告

当编译器发现代码中一些无意的错误时，会产生警告信息。与错误不同，警告一般不会中止应用程序的编译过程。

编译器产生的警告信息在排查代码bug时是非常有用的。然而，如果得到的是虚假的警告，那么它们的作用就是负面的。在大型应用程序中，保持良好的信噪比对于发现“真正的”警告是非常重要的。

为了达到这一目标，编译器允许通过#pragma warning指令有选择性地避免一些警告。在这个例子中，我们指示编译器在域Message未被使用时不产生警告：

```
public class Foo
{
  static void Main() { }

  #pragma warning disable 414
  static string Message = "Hello";
  #pragma warning restore 414
}
```

省略#pragma warning指令中的数字可以禁用或恢复所有的警告代码。

如果你非常了解这个指令的使用方法，可以使用/warnaserror开关进行编译，它能够指示编译器将所有其他警告显示为错误。

4.16 XML文档

文档注释是一种嵌入的、记录类型或成员的XML。文档注释位于类型或成员声明之前，以三个斜线开头：

```
/// <summary> Cancel运行一个查询.</summary>
public void Cancel() { ... }
```

多行注释的方法如下：

```
/// <summary>
/// Cancel运行一个查询
/// </summary>
public void Cancel() { ... }
```

也可以采用以下方法（注意开头有两个星号）：

```
/**
    <summary> Cancel运行一个查询. </summary>
*/
public void Cancel() { ... }
```

如果使用/doc指令进行编译，那么编译器会将文档注释存储到一个XML文件中，并进行校对。这个特性主要有两种作用：

- 如果与编译的程序集位于同一个文件夹，那么Visual Studio会自动读取这个XML文件，使用这些信息向同名程序集的使用者提供IntelliSense成员清单。
- 第三方工具（如Sandcastle和NDoc）可以将XML文件转换成HTML帮助文件。

4.16.1 标准的XML文档标签

下面是Visual Studio和文档生成器支持的标准的XML标签：

<summary>

```
<summary>...</summary>
```

指定IntelliSense为类型或成员显示的工具提示信息；一般为一个单词或句子。

<remarks>

```
<remarks>...</remarks>
```

标签间的文本可用于描述类型或成员。文档生成器会获取这些信息，然后将它合并到类型或成员的描述信息中。

<param>

```
<param name="name">...</param>
```

解释方法的参数。

<returns>

```
<returns>...</returns>
```

解释方法的返回值。

<exception>

```
<exception [cref="type"]>...</exception>
```

列出方法可能抛出的异常（cref指的是异常类型）。

<permission>

```
<permission [cref="type"]>...</permission>
```

指定文档记录的类型或成员所要求的IPermission类型。

<example>

```
<example>...</example>
```

给出一个示例供文档生成器使用，通常包含描述信息和源代码（源代码一般位于<c>或<code>标签内）。

<c>

```
<c>...</c>
```

指定一个内联代码片断。这个标签通常用在<example>代码块内。

<code>

```
<code>...</code>
```

指定一个多行代码示例。这个标签通常用在<example>代码块内。

<see>

```
<see cref="member">...</see>
```

将一个内联交叉引用插入到另一个类型或成员中。HTML文档生成器一般会将它转换为一个超链接。如果该类型或成员名是无效的，那么编译器会产生一条警告信息。如果要引用泛型类型，那么要使用花括号，例如cref="Foo{T,U}"。

<seealso>

```
<seealso cref="member">...</seealso>
```

交叉引用另一个类型或成员。文档生成器一般会将它写入页面底部一个单独的“See Also”小节中。

<paramref>

```
<paramref name="name"/>
```

引用<summary>或<remarks>标签内的一个参数。

<list>

```
<list type=[ bullet | number | table ]>
  <listheader>
    <term>...</term>
    <description>...</description>
  </listheader>
  <item>
    <term>...</term>
    <description>...</description>
   </item>
</list>
```

指示文档生成器生成一个项目符号、编号或表格式的列表。

<para>

```
<para>...</para>
```

指示文档生成器将指定内容单独作为一个段落。

<include>

合并包含文档的一个外部XML文件。path属性表示一个用于查询该文件中某个特定元素的XPath查询。

4.16.2 用户定义标签

C#编译器所能识别的预定义XML标签没有突出的特点，但是可以定义自己的特殊标签。编译器唯一进行特殊处理的有<param>标签（通过这个标签验证参数名称及方法记录的所有参数）和cref属性（通过这个属性验证属性所指是一个真实的类型或成员，然后将它扩展为完整的类型或成员ID）。也可以在自定义标签中使用cref属性，而且它的验证方式与预定义的<exception>、<permission>、<see>和<seealso>标签相同。

4.16.3 类型或成员交叉引用

类型名称和类型或成员交叉引用会被转换为唯一指定这个类型或成员的ID。这些名称是由一个定义ID所表示内容的前缀和一个类型或成员的签名组成的。这样的成员前缀有：

XML类型前缀	ID前缀的应用目标
N	命名空间
T	类型（类、结构体、枚举、接口、委托）
F	字段
P	属性（包括索引）
M	方法（包括特殊方法）
E	事件
!	错误

这些规则描述了如何正确记录所生成的签名，但是是非常复杂的。

下面是所生成的一个类型和ID示例：

```
// 命名空间没有独立的签名
namespace NS
{
  /// T:NS.MyClass
  class MyClass
  {
    /// F:NS.MyClass.aField
    string aField;

    /// P:NS.MyClass.aProperty
    short aProperty {get {...} set {...}}

    /// T:NS.MyClass.NestedType
```

```
  class NestedType {...};

  /// M:NS.MyClass.X()
  void X() {...}
  /// M:NS.MyClass.Y(System.Int32,System.Double@,System.Decimal@)
  void Y(int p1, ref double p2, out decimal p3) {...}

  /// M:NS.MyClass.Z(System.Char[ ],System.Single[0:,0:])
  void Z(char[ ] 1, float[,] p2) {...}

  /// M:NS.MyClass.op_Addition(NS.MyClass,NS.MyClass)
  public static MyClass operator+(MyClass c1, MyClass c2) {...}

  /// M:NS.MyClass.op_Implicit(NS.MyClass)_System.Int32
  public static implicit operator int(MyClass c) {...}

  /// M:NS.MyClass.#ctor
  MyClass() {...}

  /// M:NS.MyClass.Finalize
  ~MyClass() {...}

  /// M:NS.MyClass.#cctor
  static MyClass() {...}
}
```

第5章

框架概述

.NET Framework中几乎所有的功能都是通过大量的托管类型提供的。这些类型被组织成有层次的命名空间，并且被打包成一套程序集，与CLR一起构成.NET平台。

有些.NET类型是由CLR直接使用的，并且对于托管的宿主环境而言是必不可少的。这些类型位于一个名为*mscorlib.dll*的程序集中，包括C#的内置类型，以及基本的集合类、流处理类型、序列化、反射、多线程和原生互操作性。

除此之外是一些附加类型，它们充实了CLR层面的功能，提供了其他一些特性，如XML、网络和LINQ等。这些类型位于*System.dll*、*System.Xml.dll*和*System.Core.dll*中，并且与*mscorlib*一起提供丰富的编程环境供Framework的其他部分使用。这个核心框架在很大程度上定义了本书其余章节的内容范围。

.NET Framework的其余部分是由一些实用API组成的，主要包括以下三个方面的功能：

- 用户接口技术
- 后台技术
- 分布式系统技术

表5-1列出了各个版本C#、CLR和.NET框架的兼容性历史。C# 5.0对应CLR 4.5，这个版本比较特殊，因为它属于CLR 4.0的补丁版本。这意味着安装CLR 4.5之后，目标平台是CLR 4.0的应用实际上运行在CLR 4.5上；因此，微软特别注意向后兼容性。

表5-1：C#、CLR和.NET Framework版本

C#版本	CLR版本	Framework版本
1.0	1.0	1.0
1.2	1.1	1.1
2.0	2.0	2.0、3.0
3.0	2.0（SP1）	3.5
4.0	4.0	4.0
5.0	4.5（Patched CLR4.0）	4.5

本章将介绍.NET Framework的所有主要方面，首先介绍本文涉及的核心类型，最后概括介绍一些实用技术。

提示： 程序集和命名空间在.NET Framework中是相互交叉的。其中最极端的情况是*System.Core.dll*和*mscorlib.dll*，这两个库都定义了许多命名空间，但是其中没有以*mscorlib*或*System.Core*为前缀的。然而，有一些交叉不是很明显，反而更容易混淆，如*System.Security.Cryptography*中的类型。除了少数类型位于*System.Security.dll*，大多数类型都位于*System.dll*。

.NET Framework 4.5新特性

Framework 4.5的新特性包括：

- 通过返回Task的方法广泛支持异步编程
- 支持zip压缩协议（第15章）
- 通过新增HttpClient类改进HTTP支持（第16章）
- 改进垃圾收集器和程序集资源回收的性能
- 支持WinRT互操作性和开发Metro风格平板应用的API

此外，还有一个新的TypeInfo类（第19章）、以及可以指定与正则表达工超时时间相匹配的超时时间（第26章）。

在并行计算领域，还有一个全新库Dataflow，可用于开发生产者/消费者风格的网络。

此外，WPF、WCF和WF（工作流基础）库也有一些改进。

许多核心类型定义在以下程序集中：*mscorlib.dll*、*System.dll*和*System.Core.dll*。第一个程序集*mscorlib.dll*包括运行时环境本身所需要的类型；*System.dll*和*System.Core.dll*包含程序员所需要的其他核心类型。后两个程序集单独分开是历史原因造成的：当微软团队推出Framework 3.5时，他们采用附加开发模式，所以它运行在现有的CLR 2.0上（而Framework 4.5则附加在4.0之上）。因此，几乎所有新的核心类型（如支持LINQ的类）都会进入一个新程序集，微软将其命名为*System.Core.dll*。

.NET Framework 4.0新特性

Framework 4.0增加了以下新特性：

- 新的核心类型：BigInteger（大数字）、Complex（复数）和元组（第6章）
- 新的SortedSet集合（第7章）
- 代码协定，使方法能够通过共同的义务和责任实现更可靠的交互（第13章）
- 直接支持内存映射文件（第15章）
- 延迟的文件和目录I/O方法，它们返回IEnumerable<T>而不是数组（第15章）

- *动态语言运行时*（DLR）成为.NET Framework的一部分（第20章）
- *安全透明*，简化了保证部分可信环境中程序库安全性的方法（第21章）
- 新的多线程结构，包括更强大的Monitor.Enter重载、新的信号发送类（Barrier和CountdownEvent）和延迟初始化原语（第22章）
- 支持多核处理的*并行计算*API，包括Parallel LINQ (PLINQ)、命令式数据与任务并行性结构、支持并发的集合和低延迟同步机制与spinning原语（第23章）
- 用于监控应用程序域资源的方法（第24章）

Framework 4.0还包含了一些ASP.NET的改进，包括MVC框架和Dynamic Data，以及Entity Framework、WPF、WCF和Workflow等方面的改进。此外，它还包含了新的*Managed Extensibility Framework*库，以帮助运行时环境实现组合、发现和依赖注入。

5.1 CLR和核心框架

5.1.1 系统类型

大多数的基础类型都直接位于System命名空间。其中包括C#的内置类型、Exception基类、Enum、Array和Delegate基类，以及Nullable、Type、DateTime、TimeSpan和Guid。System命名空间也包含执行数字计算功能（Math）、生成随机数（Random）和各种数据类型转换（Convert和BitConverter）的类型。

第6章将介绍这些类型以及定义.NET Framework中用来执行各种任务标准协议的接口，如格式化（IFormattable）和顺序比较（IComparable）。

System命名空间还定义了IDisposable接口和与垃圾回收器交互的GC类。这些问题将在第12章进行介绍。

5.1.2 文本处理

在System.Text命名空间中有一个StringBuilder类（可编辑或可修改的同类字符串），以及处理文本编码的类型，如UTF-8（Encoding及其子类型）。我们将在第6章对此进行介绍。

在System.Text.RegularExpressions命名空间中有一些执行基于模式的搜索和替换操作的高级类型，这些类型将在第26章进行介绍。

5.1.3 集合

.NET Framework提供了各种处理集合项目的类。其中包括基于链表和基于字典的结构，以及一组统一它们常用特性的标准接口。所有集合类型都定义在下面的命名空间中，将在第7章对此进行介绍。

```
System.Collections                    // 非泛型框架
System.Collections.Generic            // 泛型框架
System.Collections.Specialized        // 强类型框架
```

```
System.Collections.ObjectModel        // 自定义框架基类
System.Collections.Concurrent         // 线程安全框架（第23章）
```

5.1.4 查询

Framework 3.5增加了语言集成查询（Language Integrated Query，LINQ）。LINQ允许对本地和远程集合执行类型安全查询（例如SQL Server表），将在第8～10章对此进行介绍。LINQ的最大优势是提供了一种跨多个域的统一查询API。解析LINQ查询的类型位于以下命名空间中：

```
System.Linq                  // 将LINQ转换为对象和PLINQ
System.Linq.Expressions      // 用于手工创建表达式
System.Xml.Linq              // 将LINQ转换为XML
```

完整的.NET模板还包括下面命名空间：

```
System.Data.Linq      // 将LINQ转换为SQL
System.Data.Entity    // 将LINQ转换为实体(Entity Framework)
```

（Metro模板不包含整个`System.Data.*`命名空间。）

LINQ to SQL和Entity Framework API使用了`System.Data`命名空间的ADO.NET底层类型。

5.1.5 XML

XML在.NET Framework中被广泛使用，同时也得到广泛支持。第10章主要介绍LINQ to XML——一个可通过LINQ构建和查询的轻量XML文档对象模型。第11章介绍旧版的W3C DOM，以及高性能的底层读/写类和Framework对XML模式、样式表与XPath的支持。这些XML命名空间有：

```
System.Xml                    // XmlReader, XmlWriter + 旧W3C DOM
System.Xml.Linq               // LINQ to XML DOM
System.Xml.Schema             // XSD支持
System.Xml.Serialization      // .NET类型的声明式XML序列化
```

下面的命名空间位于标准的.NET模板中（但不在Metro模板中）：

```
System.Xml.XPath              // XPath查询语言
System.Xml.Xsl                // 样式表支持
```

5.1.6 诊断与代码协定

在第13章中，我们将介绍.NET的日志和断言功能，以及Framework 4.0新的代码协议系统。同时将介绍如何与其他过程交互、记录Windows事件日志以及使用性能计数器进行监控，这些类型是在`System.Diagnostics`命名空间中定义的。

5.1.7 并发与异步

大多数现代应用都需要同时处理多个任务。C# 5.0和Framework 4.5通过异步函数和高级结构（如任务和任务组合器）对这个方面进行了简化。在先介绍多线程的基础概念之后，第14章详细介绍了这方面内容。操作线程和异步操作的类型位于`System.Threading`和`System.Threading.Tasks`命名空间。

5.1.8 流与I/O

Framework提供了基于流的模型进行底层输入/输出操作。流一般用于文件和网络连接的直接读写操作，它们可以被链接和封装到装饰流中，从而实现压缩或加密功能。第14章将介绍.NET流架构，以及对文件与目录、压缩、隔离存储、管道和内存映射文件的特殊支持。Stream和I/O类型是在System.IO命名空间中定义的。

5.1.9 网络

可以通过System.Net中的类型直接访问标准的网络协议，如HTTP、FTP、TCP/IP和SMTP。在第15章中，我们将逐一介绍如何使用这些协议进行通信，首先介绍一些简单的任务，如下载，最后介绍使用TCP/IP直接接收POP3电子邮件。以下是我们定义的命名空间：

```
System.Net
System.Net.Http              // HttpClient
System.Net.Mail              // 用于通过SMTP发送邮件
System.Net.Sockets           // TCP、UDP和IP
```

Metro应用不能使用后两个命名空间，相反必须使用WinRT类型才能实现这些功能。

5.1.10 序列化

Framework提供了几个可以将对象保存为二进制或文本方式的系统。这些系统是分布式应用程序技术所必需的，如WCF、Web Services和Remoting，它们也可用于将对象保存到文件和从文件恢复对象。在第17章中，我们将介绍三种序列化引擎：数据协定序列化、二进制序列化和XML序列化。序列化的类型位于以下命名空间：

```
System.Runtime.Serialization
System.Xml.Serialization
```

Metro模板不包含二进制序列化引擎。

5.1.11 程序集、反射和属性

C#程序编译产生的程序集包含可执行指令（存储为中间语言或IL）和元数据，它描述了程序的类型、成员和属性。通过反射机制，可以在运行时检查元数据或者执行某些操作，如动态调用方法。通过Reflection.Emit，可以随时创建新代码。

在第18章中，我们将介绍程序集的构成，以及程序集签名、全局程序集缓存和资源及文件引用解析的方法。在第18章中，我们将介绍反射和属性——说明如何检查元数据、动态调用函数、编写自定义属性、创建新类型和解析原始IL。使用反向和操作程序集的类型位于以下命名空间中：

```
System
System.Reflection
System.Reflection.Emit
```

5.1.12 动态编程

在第20章中，我们将介绍动态编程和使用动态语言运行时（Dynamic Language Runtime，DLR）的模式，动态语言运行时已经成为CLR的一部分。我们将介绍如何实现访问者模式、编写自定义动态对象

和实现IronPython的互操作。动态编程的类型位于System.Dynamic中。

5.1.13 安全性

.NET Framework具有自己的安全层，从而能够将程序集装入沙箱，甚至将自己装入沙箱。在第21章中，我们将首先介绍代码访问、角色和身份安全性以及CLR 4.0的新透明模型。然后介绍Framework的加密方法，包括加密、散列法和数据保护。这些类型定义在以下命名空间中：

```
System.Security
System.Security.Permissions
System.Security.Policy
System.Security.Cryptography
```

Metro模板只包含System.Security；加密操作则在WinRT中处理。

5.1.14 高级线程

C# 5的异步函数可以显著简化并发编程，因为它们减少了底层技术的使用。然而，开发者有时候仍然需要使用信号发送结构、线程内存储、读/写锁等。第22章详细介绍这方面的内容。线程类型位于System.Threading命名空间。

5.1.15 并行编程

第23章将详细介绍与多核处理器使用相关的库和类型，其中包括任务并行、命令式数据并行和函数并行（PLINQ）等API。

5.1.16 应用程序域

CLR支持在一个进程中增加额外的隔离级别，即应用程序域。在第24章中，我们将介绍可以操作的应用程序域属性，并说明如何在同一个进程中创建和使用额外的应用程序域执行其他操作，如单元测试。我们还将介绍如何使用*Remoting*与这些应用程序域进行交互。AppDomain类型定义在System命名空间中。

5.1.17 原生互操作性与COM互操作性

您可以与原生代码和COM代码实现互操作。原生互操作性使您能够调用未托管DLL中的函数、注册回调函数、映射数据结构和操作原生数据类型。COM互操作性使您能够调用COM类型和将.NET类型传递给COM。这些类型支持System.Runtime.InteropServices中定义的函数，我们将在第25章进行介绍。

5.2 应用技术

5.2.1 用户界面技术

.NET Framework提供了4种支持基于用户界面的应用程序的API。

ASP.NET (System.Web.UI)

编写运行在标准网页浏览器上的瘦客户端应用程序。

Silverlight

在网页浏览器上实现富用户界面。

Windows Presentation Foundation (`System.Windows`)

编写富客户端应用程序。

Windows Forms (`System.Windows.Forms`)

支持遗留富客户端应用程序。

一般而言，瘦客户端应用程序指的是网站；而富客户端应用程序则是最终用户必须下载或安装在客户端计算机上的程序。

1. ASP.NET

用ASP.NET编写的应用程序运行在Windows IIS上，并且几乎可以用所有的网页浏览器访问。以下是ASP.NET相对于富客户端技术的优点：

- 客户端不需要任何部署
- 客户端可以运行在非Windows平台上
- 很容易部署更新

而且，由于在ASP.NET应用程序中编写的功能都运行在服务器上，所以可以将数据访问层设计在同一个应用程序域中，而且不损害安全性或可扩展性。相反，实现相同功能的富客户端一般不具备相应的安全性或可扩展性（富客户端的方法是在客户端和数据库之间插入一个中间层。中间层运行在一台远程应用程序服务器上[通常与数据库服务器一起]，并通过WCF、Web Services或Remoting与富客户端通信）。

在编写网页时，可以选择传统的Web Forms或者新的MVC（模型－视图－控制器）API。这两种方法都基于ASP.NET基础架构。从一开始，Framework就支持Web Forms；MVC则是在后来Ruby on Rails和MonoRail流行之后才出现的。虽然在Framework 4.0中首次出现，但MVC框架在公开测试界面方面已经很成熟。一般而言，它具有比Web Forms更好的编程抽象，也支持更多的HTML生成控制，唯一缺失的是Web Forms的设计器。这使Web Forms仍然适合用来编写主要包含静态内容的网页。

ASP.NET的缺点很大程度源于瘦客户端系统的常见缺点：

- 网页浏览器界面很大程度上限制了可以执行的操作。
- 保持客户端状态或替代客户端是很难实现的。

然而，可以通过一些客户端脚本或其他技术来改进交互性和响应性，如AJAX（*http://ajax.asp.net*有很好的AJAX资源）。AJAX的使用可以通过诸如jQuery等库进行简化。

编写ASP.NET应用程序的类型位于`System.Web.UI`命名空间及其子命名空间中，并且属于*System.Web.dll*程序集。

2. Silverlight

Silverlight在技术上并不属于.NET Framework的主框架：它是一个独立的框架，包含了一部分的Framework核心特性，增加了作为网页浏览器插件运行的功能。它的图形化模型实际上是WPF的子集，这使您能够利用已有知识来开发Silverlight应用程序。Silverlight可以作为网页浏览器的小型跨平

台插件下载，此特性与Macromedia的Flash很像。

Flash是一个更大型的软件库，所以在这个领域中占据领先位置。因此，Silverlight主要用于一些边缘场景，例如企业内部网。

3. Metro

Windows Metro库同样不属于.NET框架，它只用于在Windows 8中开发平板电脑界面（参见第1章的“C#与Windows Runtime”）。Metro API源于WPF的启发，并且使用XAML实现布局。其命名空间包括`Windows.UI`和`Windows.UI.Xaml`。

4. Windows Presentation Foundation (WPF)

WPF是在Framework 3.0时引入的，用来编写富客户端应用程序。WPF的优点源于它的预处理器Windows Forms，具体包括：

- 支持复杂的图形功能，如任意角转换、3D渲染和全真透明。
- 主要计量单位不是像素，所以应用程序可以正确显示在任何DPI（每英寸点数）设置上。
- 支持大量的动态布局，因此可以随意设置应用程序布局而不会出现元素重叠。
- 使用DirectX实现快速渲染，可以很好地利用图形硬件加速功能。
- 用户界面可以在XAML文件中通过声明方式定义，这些文件是独立维护的“代码后”文件，这有利于将外观与功能隔离。

然而，WPF的规模和复杂性使学习周期比较长。

编写WPF应用程序的类型位于`System.Windows`命名空间以及除`System.Windows.Forms`之外的所有子命名空间中。

5. Windows Forms

Windows Forms是一个与.NET Framework同时期出现的富客户端API。与WPF相比，Windows Forms相对简单，它支持编写一般Windows应用程序时所需要使用的大多数特性，也能够良好地兼容遗留应用程序。但是与WPF相比，它有许多缺点：

- 位置与尺寸的控制是基于像素的，当客户端DPI设置与开发者设置不同时，用它编写的应用程序很容易出错。
- 绘制非标准控制的API是GDI+，虽然它相对灵活，但是渲染大区域的速度会慢一些（并且缺少双重缓冲、抖动很严重）。
- 缺少全真透明控制。
- 很难进行正确的动态布局。

最后一点是使用WPF替代Windows Forms最主要的理由——即使编写的业务应用程序只需要一个普通的用户界面，而不需要考虑“用户体验”。WPF中的布局元素，如`Grid`，简化了标签和文本框的使用，因为即使修改的本地语言后它们总是对齐的，并不需要增加处理逻辑，也不会出现抖动。而且，不需要处理屏幕分辨率，因为WPF布局元素最初设计为自动调整分辨率大小。

另一方面，Windows Forms的学习过程相对简单，并有丰富的第三方控件支持。

Windows Forms类型位于命名空间System.Windows.Forms（在*System.Windows.Forms.dll*中）和System.Drawing（在*System.Drawing.dll*中）中。其中后者包含了绘制自定义控件的GDI+类型。

5.2.2 后台技术

1. ADO.NET

ADO.NET是托管的数据访问API。虽然它的名称源于20世纪90年代的ADO（ActiveX Data Objects），但是这两种技术是完全不同的。ADO.NET包含两个主要的底层组件：

提供者层

提供者模型定义了数据库提供者底层访问的通用类和接口。这些接口包括连接、命令、适配器和读取器（数据库的只向前的只读游标）。Framework包含对Microsoft SQL Server和Oracle的原生支持，具有OLE-DB和ODBC提供者。

DataSet模型

一个DataSet是一个数据的结构化缓存。它类似于一个常驻内存的原始数据库，其中定义了SQL结构，如表、记录行、字段、关系、约束和视图。通过对数据缓存的编程，可以减少数据库的交互数量、增加服务器可扩展性以及加快富客户端用户界面的响应速度。DataSet是可序列化的，它支持通过客户端和服务器应用程序之间的线路传输。

提供者层只有两个API，它们提供了通过LINQ查询数据库的功能：

- LINQ to SQL（从Framework 3.5开始引入）
- Entity Framework（从Framework 3.5 SP1开始引入）

这两种技术都包含对象/关系映射器（ORM），意味着它们会自动将对象（基于定义的类）映射到数据库的记录行。这允许用户通过LINQ查询这些对象，而不需要编写SQL select语句并且不需要手动编写SQL insert/delete/update语句进行对象更新。因此可以明显减少应用程序的数据访问层代码行（特别是"重"代码），并实现较强的静态类型安全性。虽然DataSet仍然是唯一能够存储和序列化状态变化的技术（这在多层应用程序中是非常有用的），但是这些技术也可以使我们不需要使用DataSet作为数据容器。虽然这个过程不太灵活，而且DataSet本身不具灵活性，但是可以将LINQ to SQL或Entity Framework与DataSet结合。换言之，现在还没有现成的便捷方法可以使用Microsoft的ORM来编写n层应用程序。

LINQ to SQL比Entity Framework更简单、更快速，并且一般会产生更好的SQL。Entity Framework则更具灵活性，可以在数据库和查询的类之间创建复杂的映射。除了SQL Server，Entity Framework还支持一些第三方数据库。

2. Windows Workflow

Windows Workflow是一个对可能长期运行的业务过程进行建模和管理的框架。Workflow目标是成为一个标准的提供一致性和互操作性的运行时库。Workflow有助于减少动态控制的决策树的编码量。

Windows Workflow严格意义上并不是一种后台技术，可以在任何地方使用它（例如，UI的页面流）。

Workflow是从.NET Framework 3.0开始出现的，它的类型定义在System.WorkFlow命名空间中。实际上Workflow在Framework 4.0中进行了修改，增加的新类型位于System.Activities命名空间。

3. COM+和MSMQ

Framework允许通过System.EnterpriseServices命名空间中的类型与COM+进行互操作，以实现诸如分布式事务等服务。它也支持通过System.Messaging中的类型使用MSMQ（Microsoft Message Queuing），微软消息队列实现异步的单向消息传递。

5.2.3 分布式系统技术

1. Windows Communication Foundation（WCF）

WCF是Framework 3.0引入的一个复杂的通信基础架构。WCF非常灵活且可配置，这使它的两个预处理器——Remoting和（.ASMX）Web Services，大多是冗余的。

WCF、Remoting和Web Services很相似的方面就是它们都实现以下允许客户端和服务器应用程序进行通信的基本模型：

- 在服务器端，可以指定希望远程客户端应用程序能够调用的方法。
- 在客户端，可以指定或推断将要调用的服务器方法的签名。
- 在服务器和客户端，都可以选择一种传输和通信协议（在WCF中，这是通过一个绑定完成的）。
- 客户端建立一个服务器连接。
- 客户端调用远程方法，并在服务器上透明地执行。

WCF会通过服务协定和数据协定进一步对客户端和服务器进行解耦。概念上，客户端会发送一条（XML或二进制）消息给远程服务的终端，而非直接调用一个远程方法。这种解耦方式的好处是客户端不会依赖于.NET平台或任意私有的通信协议。

WCF是高度可配置的，它支持广泛的标准化消息协议，包括WS-*。这使您能够与运行不同软件（可能是不同平台）的组件进行通信，同时支持一些高级特性，如加密。WCF的另一个好处是可以直接修改协议，而不需要修改客户端或服务器应用程序的其他内容。

与WCF通信的类型位于System.ServiceModel命名空间中。

2. Remoting和Web Services

Remoting和.ASMX Web Services是WCF的预处理器，虽然Remoting仍然适合在相同进程中的应用程序域之间进行通信，但是它们在WCF中几乎是冗余的（见第24章）。

Remoting的功能针对一些紧密耦合的应用程序。一个典型的例子是，当客户端和服务器都是由同一个公司（或共享相同程序集的公司）编写的.NET应用程序时，它们之间的通信一般会包含可能很复杂的自定义.NET对象的交换，而Remoting基础架构会在不需要干预的情况下对它们进行序列化和反序列化。

Web Services针对一些低耦合或SOA类型应用程序。一个典型的例子是，服务器可以设计成用来接收源自（在各种平台上）运行各种软件的客户端的基于SOAP的简单消息。Web Services只能使用HTTP或SOAP作为传输和格式化协议，而应用程序一般是运行在IIS上。互操作性的好处在于性能成本方面——Web Services应用程序一般在执行和开发时间上的速度都比精心设计的Remoting应用程序慢。

Remoting的类型位于System.Runtime.Remoting命名空间中；而Web Services的类型则位于System.

Web.Services中。

3. CardSpace

CardSpace是一种基于令牌的验证和身份管理协议，是专门设计的用来简化最终用户的密码管理的。这个技术并未受到很大的关注，因为在不同主机上移动令牌是很困难的（*OpenID*是一个可避免这个问题的流行替代方法）。

CardSpace基于开放XML标准创建，并且各部分可以独立于Microsoft。用户可以具有多个身份，它们是由第三方（身份提供者）维护的。当用户希望访问站点X的资源时，用户会向身份提供者请求认证，然后身份提供者会发布一个令牌给站点X。这避免了直接将密码提供给站点X，减少了用户需要维护的身份个数。

通过一个安全的HTTP通道进行连接时，WCF允许通过System.IdentityModel.Claims和System.IdentityModel.Policy命名空间中的类型指定一个CardSpace身份。

第6章

框架基础

编程所需要的许多核心工具都不是由C#语言提供的，而是由.NET Framework中的类型提供的。在本章中，我们将介绍Framework在基础编程任务中的作用，如虚拟等值比较、顺序比较和类型转换。我们还将介绍基本的Framework类型，如String、DateTime和Enum。

本节的类型位于System命名空间中，除了以下类型：

- StringBuilder定义在System.Text中，其中还包括文本编码的类型。
- CultureInfo与相关类型定义在System.Globalization中。
- XmlConvert定义在System.Xml中。

6.1 字符串与文本处理

6.1.1 字符（char）

一个C#的char表示一个Unicode字符，它是System.Char结构体的别名。在第2章中，我们介绍了如何表示char字面值。例如：

```
char c = 'A';
char newLine = '\n';
```

System.Char定义了许多处理字符的静态方法，如ToUpper、ToLower和IsWhiteSpace。可以通过System.Char类型或它的别名char调用这些方法：

```
Console.WriteLine (System.Char.ToUpper ('c'));      // C
Console.WriteLine (char.IsWhiteSpace ('\t'));       // True
```

ToUpper和ToLower会受到最终用户的语言环境的影响，这可能会导致出现细微的缺陷。下面的表达式在土耳其语言环境中会得到false值：

```
char.ToUpper ('i') == 'I'
```

因为在土耳其语中，char.ToUpper('i')的结果是'İ'（注意字符上面的点）。为了避免出现这个问

题，System.Char、System.String还提供了针对语言变化的ToUpper和ToLower，它们加上后缀*Invariant*。这些命名总是采用英语语言规则：

```
Console.WriteLine (char.ToUpperInvariant ('i'));      // I
```

快捷形式是：

```
Console.WriteLine (char.ToUpper ('i', CultureInfo.InvariantCulture))
```

更多关于语言环境和文化的介绍，请参考本章的“格式化和解析”。

char保留的大多数静态方法都与字符分类有关，如表6-1所示。

表6-1：字符分类静态方法

静态方法	包含的字符	包含的Unicode分类
IsLetter	A～Z、a～z和其他字母字符	UpperCaseLetter LowerCaseLetter TitleCaseLetter ModifierLetter OtherLetter
IsUpper	将字母转换成大写	UpperCaseLetter
IsLower	将字母转换成小写	LowerCaseLetter
IsDigit	各种字母表的0～9数字	DecimalDigitNumber
IsLetterOrDigit	字母与数字	IsLetter与IsDigit
IsNumber	所有数字及Unicode分数和罗马数字	DecimalDigitNumber LetterNumber OtherNumber
IsSeparator	空格及所有Unicode分隔符	LineSeparator ParagraphSeparator
IsWhiteSpace	所有分隔符及\n、\r、\t、\f和\v	LineSeparator ParagraphSeparator
IsPunctuation	西方和其他字母表的标点符号	DashPunctuation ConnectorPunctuation InitialQuotePunctuation FinalQuotePunctuation
IsSymbol	大部分其他可打印符号	MathSymbol ModifierSymbol OtherSymbol
IsControl	小于0x20的不可打印的“控制”字符，如\r、\n、\t、\0和0x7F与0x9A之间的字符	（无）

框架基础

对于更细的分类，char提供了一个名为GetUnicodeCategory的静态方法：它返回一个UnicodeCategory枚举值，它的成员即表6-1最右列所显示的值。

提示： 通过显式转换一个整数，可以产生一个位于Unicode集之外的char。要检测字符的有效性，我们可以调用char.GetUnicodeCategory：如果结果是UnicodeCategory.OthernotAssigned，那么这个字符就是无效的。

一个char占用16个二进制位——这足以表示*Basic Multilingual Plane*中的所有Unicode字符。如果超过这个范围，必须使用替代组：我们将在本章的“文本编码与Unicode”中介绍它的实现方法。

6.1.2 字符串

C#的string（== System.String）是一个不可变的（不可修改的）字符序列。在第2章中，我们介绍了如何表示一个字符串字面值、执行等值比较和连接两个字符串。本节将介绍其余的字符串处理函数，它们是System.String类的静态和实例成员函数。

1. 创建字符串

创建字符串的最简单方法就是给变量定义一个字面值，这和我们在第2章的做法一样：

```
string s1 = "Hello";
string s2 = "First Line\r\nSecond Line";
string s3 = @"\\server\fileshare\helloworld.cs";
```

要创建一个重复的字符序列，可以使用string的构造方法：

```
Console.Write (new string ('*', 10));        // **********
```

还可以从char数组创建字符串。而ToCharArray方法则是执行相反操作：

```
char[] ca = "Hello".ToCharArray();
string s = new string (ca);          // s = "Hello"
```

我们还可以重载string的构造方法来接受各种（不安全的）指针类型，以便创建其他类型字符串，如char*。

2. Null和空字符串

空字符串是长度为0的字符串。如果要创建空字符串，可以使用一个字母值或静态的string.Empty字段；如果要测试空字符串，可以执行一个等值比较或测试它的Length属性：

```
string empty = "";
Console.WriteLine (empty == "");              // True
Console.WriteLine (empty == string.Empty);    // True
Console.WriteLine (empty.Length == 0);        // True
```

由于字符串是引用类型，它们也可能是null：

```
string nullString = null;
Console.WriteLine (nullString == null);          // True
Console.WriteLine (nullString == "");            // False
Console.WriteLine (nullString.Length == 0);      // NullReferenceException
```

静态的string.IsNullOrEmpty方法是测试一个给定字符串是null还是空白的快捷方法。

3. 访问字符串中的字符

字符串的索引器可以返回一个指定索引位置的字符。与所有操作字符串的方法相似，它是从0开始计数的索引：

```
string str = "abcde";
char letter = str[1];         // letter == 'b'
```

string还实现了IEnumerable<char>，所以可以用foreach遍历它的字符：

```
foreach (char c in "123") Console.Write (c + ","); // 1,2,3,
```

4. 字符串内搜索

在字符串内搜索的最简单方法是Contains、StartsWith和EndsWith。所有这些方法都返回true或false：

```
Console.WriteLine ("quick brown fox".Contains ("brown"));    // True
Console.WriteLine ("quick brown fox".EndsWith ("fox"));      // True
```

Contains方法并没有提供这种重载的便利方法，但是可以使用IndexOf方法实现相同的效果。

IndexOf方法更强大：它会返回指定字符或子字符串的首次出现位置（-1表示该子字符串不存在）：

```
Console.WriteLine ("abcde".IndexOf ("cd"));    // 2
```

StartsWith、EndsWith和IndexOf都有重载方法，我们可以指定一个StringComparison枚举变量或CultureInfo对象，控制大小写和文字顺序（参见第6章的“顺序与文化比较”）。默认为使用当前文化规则执行区分大小写的匹配。下面的代码则使用不变文化规则执行不区分大小定的搜索：

```
"abcdef".StartsWith ("abc", StringComparison.InvariantCultureIgnoreCase)
```

重载的IndexOf还可以接受startPosition（搜索开始索引器）和StringComparison枚举值。

```
Console.WriteLine ("abcde".IndexOf ("CD",
                   StringComparison.CurrentCultureIgnoreCase));     // 2
```

LastIndexOf与IndexOf类似，但是它是从后向前开始搜索的。

IndexOfAny则返回任意一系列字符的首次匹配位置：

```
Console.Write ("ab,cd ef".IndexOfAny (new char[] {' ', ','} ));       // 2
Console.Write ("pas5w0rd".IndexOfAny ("0123456789".ToCharArray() )); // 3
```

LastIndexOfAny则在相反方向执行相同的操作。

5. 字符串处理

由于String是不可变的，所有“处理”字符串的方法都会返回一个新的字符串，而原始字符串则不受影响（其效果与重新赋值一个字符变量一样）。

Substring是取字符串的一部分：

```
string left3 = "12345".Substring (0, 3);      // left3 = "123";
string mid3 = "12345".Substring (1, 3);       // mid3 = "234";
```

如果省略长度，那么会得到剩余的字符串：

```
string end3 = "12345".Substring (2);          // end3 = "345";
```

Insert和Remove会从一个指定位置插入或删除一些字符：

```
string s1 = "helloworld".Insert (5, ", ");   // s1 = "hello, world"
string s2 = s1.Remove (5, 2);                // s2 = "helloworld";
```

PadLeft和PadRight会用特定字符将字符串（如果未指定，则使用空格）填充成指定的长度：

```
Console.WriteLine ("12345".PadLeft (9, '*'));       // ****12345
Console.WriteLine ("12345".PadLeft (9));            //     12345
```

如果输入字符串长度大于填充长度，那么返回不发生变化的原始字符串。

TrimStart和TrimEnd会从字符串的开始或结尾删除指定的字符；Trim则用两个方法执行删除操作。默认情况下，这些函数会删除空白字符（包括空格、制表符、换行符和这些字符的Unicode变体）：

```
Console.WriteLine (" abc \t\r\n ".Trim().Length);    // 3
```

Replace会替换字符中出现的特定字符或子字符串：

```
Console.WriteLine ("to be done".Replace (" ", " | ") );      // to | be | done
Console.WriteLine ("to be done".Replace (" ", "")    );      // tobedone
```

ToUpper和ToLower会返回输入字符串相应的大写和小写字符。默认情况下，它们会受用户的当前语言设置的影响；ToUpperInvariant和ToLowerInvariant总是采用英语字母表规则。

6. 分割和连接字符串

Split接受一个句子，返回一个单词数组：

```
string[] words = "The quick brown fox".Split();

foreach (string word in words)
  Console.Write (word + "|");                  // The|quick|brown|fox|
```

默认情况下，Split使用空白字符作为分隔符；经过重载后也可以接受包含char或string分隔符的params数组。Split还可以选择接受一个StringSplitOptions枚举值，它支持删除一些空项：这在一行单词由多种分隔符分隔时很有用。

静态的Join方法执行与Split相反的操作。它需要一个分隔符和字符串数组：

```
string[] words = "The quick brown fox".Split();
string together = string.Join (" ", words);         // The quick brown fox
```

静态的Concat方法与Join类似，但是它只接受字符串数组参数，并且没有分隔符。Concat与+操作符效果完全相同（实际上，编译器会将+转换成Concat）：

```
string sentence     = string.Concat ("The", " quick", " brown", " fox");
string sameSentence = "The" + " quick" + " brown" + " fox";
```

7. String.Format与组合格式字符串

静态的Format方法提供了创建嵌入变量字符串的便利方法。嵌入的变量可以是任意类型；而Format会直接调用它们的ToString。

包含嵌入变量的主字符串称为*组合格式字符串*。调用String.Format时，需要提供一个组合格式字符串，后面紧跟每一个嵌入式变量。例如：

```
string composite = "It's {0} degrees in {1} on this {2} morning";
string s = string.Format (composite, 35, "Perth", DateTime.Now.DayOfWeek);

// s == "It's 35 degrees in Perth on this Friday morning"
```

（而这就是摄氏度！）

花括号里面的每一个数字称为格式项。这些数字对应参数位置，后面可以跟：

- 逗号与应用的*最小宽度*
- 冒号与*格式字符串*

最小宽度用于对齐各个列。如果这个值为负数，那么数据就是左对齐的；否则，数据就是右对齐的。例如：

```
string composite = "Name={0,-20} Credit Limit={1,15:C}";

Console.WriteLine (string.Format (composite, "Mary", 500));
Console.WriteLine (string.Format (composite, "Elizabeth", 20000));
```

运行结果是：

```
Name=Mary            Credit Limit=         $500.00
Name=Elizabeth       Credit Limit=      $20,000.00
```

而不使用string.Format的写法是：

```
string s = "Name=" + "Mary".PadRight (20) +
           " Credit Limit=" + 500.ToString ("C").PadLeft (15);
```

信用额度是通过“C”格式字符串格式化为货币值。我们将在本章的“格式化和解析”详细介绍格式化字符串。

提示： 组合格式字符串的缺点是它很容易出现一些只有在运行时才能发现的错误，如格式项的个数比值的个数多或者少。如果格式项和值一起出现，那么这种错误是很难出现的。

6.1.3 字符串比较

进行两个值比较时，.NET Framework有两个不同的概念：*等值比较*和*顺序比较*。等值比较会判断两个实例在语义上是否是相同的；而顺序比较则将两个（如果有）实例按照升序或降序排列，然后判断哪一个首先出现。

提示：等值比较并不是顺序比较的一个子集；这两种方法有各自不同的用途。例如，两个不同的值在相同顺序的位置是允许的。我们将在6.10节中进行介绍。

对于字符串等值比较，可以使用==操作符或者其中一个字符串的Equals方法。后者功能更强一些，因为它们允许指定一些选项，如区分大小写。

警告：另一个不同点是，如果变量被转换成object类型，那么==就不一定是按字符串处理。我们将在本章的“等值比较”中说明其原因。

对于字符串顺序比较，可以使用CompareTo实例方法或静态的Compare和CompareOrdinal方法：这些方法会返回一个正数、负数或0，这取决于第一个值是在第二个值之后、之前还是同时出现。

在详细介绍每一个方法之前，我们需要了解.NET的基础字符串比较算法。

1. 顺序比较与文化比较

字符串比较有两种基本的算法：按顺序的和区分文化的。顺序比较会直接将字符解析为数字（根据它们的Unicode数值）；文化比较则参照特定的字母表来解析字符。特殊的文化有两种：“当前文化”，这是基于计算机控制面板的设置；“不变文化”，这在任何计算机上都是相同的。

对于等值比较，顺序和特定文化的算法都是很有用的。然而，在排序时，人们通常选择词义相关的比较：对字符串按字母表排序时，需要一个字母顺序表。顺序比较则使用Unicode数字位置值，这可能会使英语字符按字母顺序排序——但是即使这样也可能不满足你的期望。例如，在区分大小写时，字符串“Atom”、“atom”和“Zamia”的比较结果是什么？不变文化会将它们按以下顺序排列：

```
"Atom", "atom", "Zamia"
```

但是顺序比较则会将它们排列为：

```
"Atom", "Zamia", "atom"
```

这是因为不变文化封装了一个字母表，它认为大写字符与其对应的小写字符是相邻的（AaBbCcDd…）。然而，顺序算法将所有大写字母排列在前面，然后才是全部小写字符（A...Z, a...z）。这实际上是回归到20世纪60年代发明的ASCII字符集。

2. 字符串等值比较

尽管顺序比较有一些局限性，但是字符串的==操作符总是执行区分大小写的顺序比较。当不带参数调用时，string.Equals的实例版本也是一样的；这定义了string类型的“默认”等值比较行为。

提示：字符串的==和Equals函数选择顺序算法的原因是它既高效又具有确定性。字符串等值比较被认为是基础操作，并且远比顺序比较的使用更频繁。

等式的“严格”概念也与常见的==操作符用途保持一致。

下面的方法允许执行区分文化和大小写的比较：

```
public bool Equals(string value, StringComparison comparisonType);
```

```
public static bool Equals (string a, string b,StringComparison comparisonType);
```

静态方法会更适合一些，因为即使其中一个或两个字符串为null它也一样有效。StringComparison是一个按如下方式定义的enum：

```
public enum StringComparison
{
CurrentCulture,              // 区分大小写
CurrentCultureIgnoreCase,
InvariantCulture,            // 区分大小写
InvariantCultureIgnoreCase,
Ordinal,                     // 区分大小写
OrdinalIgnoreCase
}
```

例如：

```
Console.WriteLine (string.Equals ("foo", "FOO",
                   StringComparison.OrdinalIgnoreCase));      // True

Console.WriteLine ("ṻ" == "ǖ");                              // False

Console.WriteLine (string.Equals ("ṻ", "ǖ",StringComparison.CurrentCulture)); // ?
```

（最终的比较结果是由计算机当前语言设置决定的。）

3. 字符串顺序比较

String的CompareTo实例方法执行区分文化和区分大小写的顺序比较。与==操作符不同，CompareTo不使用顺序比较：对于顺序比较，区分文化的算法更有效。

下面是方法的定义：

```
public int CompareTo (string strB);
```

提示： CompareTo实例方法实现了IComparable泛型接口，这是在整个.NET Framework中使用的标准比较协议。这意味着字符串的CompareTo定义了默认的顺序行为字符串，例如，在那些作为排序集合的应用程序中。更多关于IComparable的内容，见本章的“顺序比较”。

对于其他类型的比较，可以调用静态的Compare和CompareOrdinal方法：

```
public static int Compare (string strA, string strB,StringComparison comparisonType);
public static int Compare (string strA, string strB, bool ignoreCase, CultureInfo
culture);
public static int Compare (string strA, string strB, bool ignoreCase);

public static int CompareOrdinal (string strA, string strB);
```

后面两个方法只是前面两个方法的快捷调用方式。所有顺序比较的方法都会返回正数、负数或0，这取决于第一个值是在第二个值之后、之前还是相同位置：

```
Console.WriteLine ("Boston".CompareTo ("Austin"));   // 1
Console.WriteLine ("Boston".CompareTo ("Boston"));   // 0
Console.WriteLine ("Boston".CompareTo ("Chicago"));  // -1
```

```
Console.WriteLine ("ü".CompareTo ("ü"));          // 0
Console.WriteLine ("foo".CompareTo ("FOO"));      // -1
```

下面的语句使用当前文化执行区分大小写的比较：

```
Console.WriteLine (string.Compare ("foo", "FOO", true)); // 0
```

通过CultureInfo对象，可以插入任意的字母表：

```
// CultureInfo定义在System.Globalization命名空间下
CultureInfo german = CultureInfo.GetCultureInfo ("de-DE");
int i = string.Compare ("M ller", "Muller", false, german);
```

6.1.4 StringBuilder

StringBuilder类（System.Text命名空间）表示一个可变（可编辑）的字符串。使用StringBuilder，可以Append、Insert、Remove和Replace子字符串，而不需要替换整个StringBuilder。

StringBuilder的构建函数可以选择接受一个初始字符串值，以及其内部容量的初始值（默认是16个字符）。如果需要更大的容量，那么StringBuilder会自动调整它的内部结构，以容纳（会有一些性能开销）最大的容量（默认为int.MaxValue）。

StringBuilder的一个普遍使用方法是通过重复调用Append来创建一个长字符串。这个方法比复杂连接普通字符串类型要高效得多：

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 50; i++) sb.Append (i + ",");
```

要获得最终结果，我们可以调用ToString()：

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

0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,
27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,
```

提示： 在上面的例子中，表达式i+","表示我们在重复连接字符串。然而，这只会带来极小的性能开销，因为在这里我们使用的字符串很小，并且不会随循环迭代而增长。然而，为了实现最大的性能，我们可以将循环体修改为：

```
{ sb.Append (i.ToString()); sb.Append (","); }
```

AppendLine执行新添加一行字符串（在Windows中是"\r\n"）的Append操作。AppendFormat接受一个组合格式字符串，与String.Format类似。

除了Insert、Remove和Replace方法（Replace函数类似于字符串的Replace），StringBuilder定义了一个Length属性和一个可写的索引器，可用来获取/设置每个字符串。

如果要清除StringBuilder的内容，我们可以创建一个新的StringBuilder或者将它的Length设为0。

> 警告：将StringBuilder的Length设置为0不会减小它的*内部*容量。所以，如果StringBuilder之前包含1百万个字符，那么在将它的长度设置为0后，它仍然会占用约2MB的内存。如果释放内存，那么必须创建一个新的StringBuilder，然后允许将旧的对象清除出引用范围（可以被垃圾回收器回收）。

6.1.5 文本编码与Unicode

字符集是一种字符配置，每一对配置包含一个数字码或代码点。通常使用的字符集有两种：Unicode和ASCII。Unicode具有约1百万个字符的地址空间，目前已分配的大约有100,000个。Unicode包含世界上使用最广泛的语言，以及一些历史语言和特殊符号。ASCII字符集只是Unicode字符集的前128个字符，它包括U.S.风格键盘上的大多数字符。ASCII的出现比Unicode早30年，有时仍然以其简单性和高效性而得到应用：每一个字符是用一个字节表示的。

.NET类型系统的设计使用的是Unicode字符集。但是，ASCII是隐含支持的，因为它是Unicode的子集。

文本编码是将字符从数字代码点映射到二进制表示的方法。在.NET中，文本编码主要用于处理文本文件或流。当将一个文本文件读取到字符串时，*文本编码器*会将文件数据从二进制转换为使用char和string类型的内部Unicode表示法。文本编码能够限制所能表示的字符以及可能影响存储效率的字符。

.NET的文本编码有两类：

- 一类是将Unicode字符映射到其他字符集
- 一类是使用标准的Unicode编码模式

第一类包含遗留编码方式，如IBM的EBCDIC和包含前128位区域扩展字符的8位字符集，它在Unicode之前已经很普遍（由代码页确定）。ASCII编码也属于这一类：它将对前128个字符编码，然后去掉其他字符。这个分类也包含*非遗留*的GB18030，这是从2000年以来在中国或销售往中国的应用程序强制要求的标准。

第二类是UTF-8、UTF-16和UTF-32（和废弃的UTF-7）。每一种编码方式在空间效率上都有所差别。UTF-8对于大多数文本而言是最具空间效率的：它使用*1～4*个字节来表示每个字符。前128个字符只需要1个字节，这样它就能够兼容ASCII。UTF-8是最普遍的文本文件和流的编码方式（特别是在互联网上），它是.NET中默认的流I/O编码方式（事实上，它几乎是所有语言隐含的默认编码方式）。

UTF-16使用一个或两个16位字来表示一个字符，它是.NET内部用来表示字符和字符串的方式。有一些程序也使用UTF-16写文件。

UTF-32是空间效率最低的：每一个代码点直接对应一个32位数，所以每个字符都会占用4个字节。因此，UTF-32很少使用。然而，它可以简化随机访问，因为每个字符都对应相同的字节数。

1. 获取一个Encoding对象

System.Text中的Encoding类是封装文本编码类的通用基本类型。它有一些子类，它们的作用是封装各种编码方式的相似特性。初始化一个正确配置类的最简单方法是用一个标准的IANA名称调用Encoding.GetEncoding：

```
Encoding utf8 = Encoding.GetEncoding ("utf-8");
Encoding chinese = Encoding.GetEncoding ("GB18030");
```

最常用的编码也可以通过专用的Encoding静态属性获取：

编码名称	Encoding的静态属性
UTF-8	Encoding.UTF-8
UTF-16	Encoding.Unicode（非UTF-16）
UTF-32	Encoding.UTF-32
ASCII	Encoding.ASCII

静态的GetEncodings方法会返回所有支持的编码方式清单以及它们的标准IANA名称：

```
foreach (EncodingInfo info in Encoding.GetEncodings())
  Console.WriteLine (info.Name);
```

获取编码方式的另一个方法是直接初始化Encoding类。这样做能够通过构建函数参数设置不同的选项，包括：

- 在解码时，如果遇到一个无效字节，是否抛出异常。默认值是false。
- 是否使用最高有效字节（*大字节存储次序*）或最低有效字节（*小字节存储次序*）。默认值是小字节存储次序，这是Windows操作系统的标准。
- 是否使用字节顺序标记（表示字节顺序的前缀）。

2. 文件与流I/O编码

Encoding对象最常见的应用是控制文件或流的文本读写操作。例如，下面的代码以UTF-16的编码方式将"Testing..."写入一个名为*data.txt*的文件中：

```
System.IO.File.WriteAllText ("data.txt", "Testing...", Encoding.Unicode);
```

如果省略最后一个参数，WriteAllText会使用最普遍的UTF-8编码方式。

提示： UTF-8是所有文件和流I/O的默认文本编码方式。

我们将在第15章"流适配器"中再对这个问题进行阐述。

3. 将Encoding转换为字节数组

Encoding对象和字节数组之间也可以进行互相转换。GetBytes方法将使用指定的编码方式将string转换为byte[]；而GetString则将byte[]转换为string。

```
byte[] utf8Bytes  = System.Text.Encoding.UTF8.GetBytes    ("0123456789");
byte[] utf16Bytes = System.Text.Encoding.Unicode.GetBytes ("0123456789");
byte[] utf32Bytes = System.Text.Encoding.UTF32.GetBytes   ("0123456789");

Console.WriteLine (utf8Bytes.Length);        // 10
Console.WriteLine (utf16Bytes.Length);       // 20
Console.WriteLine (utf32Bytes.Length);       // 40

string original1 = System.Text.Encoding.UTF8.GetString    (utf8Bytes);
string original2 = System.Text.Encoding.Unicode.GetString (utf16Bytes);
string original3 = System.Text.Encoding.UTF32.GetString   (utf32Bytes);

Console.WriteLine (original1);               // 0123456789
```

```
Console.WriteLine (original2);                // 0123456789
Console.WriteLine (original3);                // 0123456789
```

4. UTF-16和替代配对

.NET将字符和字符串存储为UTF-16格式。因为UTF-16的每一个字符都需要一个或两个16位字，而一个字符只有16位长度，有一些Unicode字符需要用两个字符来表示，这会造成两个结果：

- 字符串的Length属性值可能大于它的实际字符数。
- 一个字符不总能够完整表示一个Unicode字符。

大多数应用程序会忽略这一点，因为几乎所有常用的字符集都对应于Unicode中一个名为*Basic Multilingual Plane*（BMP）的部分，这一部分只要求一个UTF-16的16位字。BMP包括了许多世界各地的语言，并且包含30,000多个汉字字符。只有一些古代语言、乐谱符号和生僻汉字字符不包含在内。

如果需要支持双字字符，下面的char静态方法可以将一个32位代码点转换为一个包含两个字符的字符串，也可以执行反向转换：

```
string ConvertFromUtf32 (int utf32)
int    ConvertToUtf32   (char highSurrogate, char lowSurrogate)
```

我们将双字字符称为替代字符（surrogates）。它们很容易辩认，因为每一个字都是从0xD800到0xDFFF。可以使用下面的char静态方法来判断：

```
bool IsSurrogate     (char c)
bool IsHighSurrogate (char c)
bool IsLowSurrogate  (char c)
bool IsSurrogatePair (char highSurrogate, char lowSurrogate)
```

System.Globalization命名空间中的StringInfo类也提供了一组处理双字字符的方法和属性。

BMP以外的字符一般需要特殊的字体，并且只有少量的操作系统支持。

6.2 日期和时间

在System命名空间中有三个不可变结构可用来表示日期和时间：DateTime、DateTimeOffset和TimeSpan。而C#没有定义与这些类型相对应的关键字。

6.2.1 TimeSpan

TimeSpan表示一段时间间隔或者是一天内的时间。对于后者，它就是一个“时钟”时间（不包括日期），它等同于从半夜12点开始到现在的时间（假设没有夏时制）。TimeSpan的最小单位为100纳秒，最大值为1千万天，可以为正数或负数。

创建TimeSpan的方法有三种：

- 通过其中一个构造方法
- 通过调用其中一个静态的From...方法
- 通过两个DateTime相减得到

以下是构造方法：

```
public TimeSpan (int hours, int minutes, int seconds);
public TimeSpan (int days, int hours, int minutes, int seconds);
public TimeSpan (int days, int hours, int minutes, int seconds,int milliseconds);
public TimeSpan (long ticks); // Eachtick = 100ns
```

如果希望指定一个单位的时间间隔，如分钟、小时等，那么静态的From...方法更方便：

```
public static TimeSpan FromDays (double value);
public static TimeSpan FromHours (double value);
public static TimeSpan FromMinutes (double value);
public static TimeSpan FromSeconds (double value);
public static TimeSpan FromMilliseconds (double value);
```

例如：

```
Console.WriteLine (new TimeSpan (2, 30, 0));          // 02:30:00
Console.WriteLine (TimeSpan.FromHours (2.5));         // 02:30:00
Console.WriteLine (TimeSpan.FromHours (-2.5));        // -02:30:00
```

TimeSpan重载了<、>、+和-操作符。以下表达式可以得到一个表示2.5小时的TimeSpan：

```
TimeSpan.FromHours(2) + TimeSpan.FromMinutes(30);
```

以下表达式则表示10天减去1秒的时间：

```
TimeSpan.FromDays(10) - TimeSpan.FromSeconds(1);      // 9.23:59:59
```

使用这个表达式，我们就能够表示整数的Days、Hours、Minutes、Seconds和Milliseconds：

```
TimeSpan nearlyTenDays = TimeSpan.FromDays(10) - TimeSpan.FromSeconds(1);
Console.WriteLine (nearlyTenDays.Days);               // 9
Console.WriteLine (nearlyTenDays.Hours);              // 23
Console.WriteLine (nearlyTenDays.Minutes);            // 59
Console.WriteLine (nearlyTenDays.Seconds);            // 59
Console.WriteLine (nearlyTenDays.Milliseconds);       // 0
```

相反，Total...属性则返回表示整个时间跨度的double类型值：

```
Console.WriteLine (nearlyTenDays.TotalDays);          // 9.99998842592593
Console.WriteLine (nearlyTenDays.TotalHours);         // 239.999722222222
Console.WriteLine (nearlyTenDays.TotalMinutes);       // 14399.9833333333
Console.WriteLine (nearlyTenDays.TotalSeconds);       // 863999
Console.WriteLine (nearlyTenDays.TotalMilliseconds);  // 863999000
```

静态的Parse方法则执行与ToString相反的操作，它能将一个字符串转换为一个TimeSpan。TryParse执行与Tostring相同的操作，但是当转换失败时，它会返回false，而不是抛出异常。XmlConvert类也提供了符合标准XML格式化协议的TimeSpan/字符串转换方法。

TimeSpan的默认值是TimeSpan.Zero。

TimeSpan也可用于表示一天内时间（从半夜12点开始经过的时间）。要获得当前的时间，我们可以调用DateTime.Now.TimeOfDay。

6.2.2 DateTime和DateTimeOffset

DateTime和DateTimeOffset表示日期或者时间的不可变结构。它们的最小单位为100纳秒，值的范围从0001年到9999年。

DateTimeOffset是从Framework 3.5开始引入的，在功能上类似于DateTime。它的主要特性是能够存储UTC偏移值，这允许我们在比较不同时区的时间值时得到更有意义的结果。

提示：另一篇关于引入DateTimeOffset的基本原理的文章在MSDN BCL博客上。其标题为“A Brief History of DateTime”，作者是Anthony Moore。

---

1. DateTime和DateTimeOffset的选择

DateTime和DateTimeOffset在处理时区的方式上是不同的。DateTime具有三个状态标记，可表示DateTime是否与下列因素相关：

- 当前计算机的本地时间
- UTC（相当于现代的格林威治时间）
- 不确定

DateTimeOffset更加特殊——它将UTC的偏移量存储为一个TimeSpan：

```
July 01 2007 03:00:00 -06:00
```

这会影响等值比较结果，而且是在DateTime和DateTimeOffset之间进行选择的主要依据。特别是：

- DateTime会忽略三个比较状态标记，并且当两个值的年、月、日、时、分等相同时就认为它们是相等的。
- 如果两个值引用相同的时间点，那么DateTimeOffset就认为它们是相等的。

提示：夏时制会使这个结果差别很大，即使应用程序不需要处理多个地理时区。

所以，DateTime会认为下面两个值是不相同的，而DateTimeOffset则认为是相同的：

```
July 01 2007 09:00:00 +00:00 (GMT)
July 01 2007 03:00:00 -06:00 (local time, Central America)
```

在大多数情况中，DateTimeOffset的等值比较逻辑会更好一些。例如，在计算哪两个国际事件发生时间较晚时，DateTimeOffset就能够隐含地获得正确答案。类似地，组织分布式拒绝服务攻击的黑客也可能利用DateTimeOffset！实现与DataTime相同的操作，DataTime需要在应用程序中标准化一个时区（一般是UTC）。

出现这个问题的原因主要有两个：

- 为了给最终用户良好的体验，UTC DateTimes要求在格式化之前显式地转换为本地时间。
- 人们很容易忘记使用本地DateTime。

框架基础

如果在运行时指定与本地计算机相关的值，使用DateTime会更好。例如，如果希望在本地时间的下周日凌晨3点（这是活动最少的时刻）在每一个跨国办公室上调度一次存档。这时，使用DateTime可能更适合一些，因为它与每个站点的本地时间相对应。

提示： 在内部，DateTimeOffset使用一个短整数来存储以分钟为单位的UTC偏移值。例如，它不会存储任何与区域相关的信息，所以其中没有任何内容表示+08:00偏移值指的是新加坡时间还是珀斯时间。

我们将在本章的“日期和时区”中再次阐述时区和等值比较操作。

提示： SQL Server 2008通过一个新的同名数据类型增加了对DateTimeOffset的直接支持。

2. 创建一个DateTime

DateTime定义了能够接受年、月和日以及可选的时、分、秒和毫秒的构造方法。

```
public DateTime (int year, int month, int day);

public DateTime (int year, int month, int day,
                 int hour, int minute, int second, int millisecond);
```

如果只指定日期，那么时间会被隐含地设置为半夜时间（0:00）。

DateTime构造方法也允许指定一个DateTimeKind——这是一个具有以下值的枚举值：

```
Unspecified, Local, Utc
```

这三个值与前一节所介绍的三个状态标记相对应。Unspecified是默认值，它表示DateTime是未指定时区的。Local表示与当前计算机的本地时区相关。本地DateTime不包含它引用了哪一个特定的时区，而且与DateTimeOffset不同的是，它也不包含UTC偏移值。

DateTime的Kind属性返回它的DateTimeKind。

DateTime的构造方法也经过重载从而可以接受Calendar对象——允许使用System.Globalization中所定义的日历子类指定一个时间。例如：

```
DateTime d = new DateTime (5767, 1, 1, new System.Globalization.HebrewCalendar());
Console.WriteLine (d);    // 12/12/2006 12:00:00 AM
```

（这个例子的日期格式化取决于计算机控制面板中的设置。）DateTime总是使用默认的公历，在这个例子中，创建过程中发生了一次转换。如果要使用另一个日历进行计算，那么必须使用Calendar子类自己的方法。

也可以使用long类型的计数值（ticks）来创建DateTime，其中计数值是从午夜开始算起的100纳秒数。

在互操作性上，DateTime提供了静态的FromFileTime和FromFileTimeUtc方法来转换一个Windows文件时间（由long指定），并且提供了FromOADate来转换一个OLE自动日期/时间（由double指定）。

要从字符串创建DateTime，我们必须调用静态的Parse或ParseExact方法。这两个方法都接受可选标记和格式提供者；ParseExact还接受格式字符串。我们在本章的“格式化和解析”中将更详细地讨论时间的解析。

3. 创建一个DateTimeOffset

DateTimeOffset具有类似的构造方法。其区别是还需要指定一个TimeSpan类型的UTC偏移值：

```
public DateTimeOffset (int year, int month, int day,
                       int hour, int minute, int second,
                       TimeSpan offset);

public DateTimeOffset (int year, int month, int day,
                       int hour, int minute, int second, int millisecond,
                       TimeSpan offset);
```

TimeSpan必须刚好是整数分钟，否则函数会抛出一个异常。

DateTimeOffset也有一些接受Calendar对象、long计数值的构造方法，以及接受字符串的静态的Parse和ParseExact方法。

还可以通过下面一种构造方法从现有的DateTime创建DateTimeOffset：

```
public DateTimeOffset (DateTime dateTime);
public DateTimeOffset (DateTime dateTime, TimeSpan offset);
```

也可以通过隐式转换创建：

```
DateTimeOffset dt = new DateTime (2000, 2, 3);
```

提示：从DateTime隐式转换到DateTimeOffset是很简单的，因为大多数的.NET Framework类型都支持DateTime——而不是DateTimeOffset。

如果没有指定偏移量，那么可以使用以下规则从DateTime值推断出偏移值：

- 如果DateTime具有一个UTC的DateTimeKind，那么其偏移量为0。
- 如果DateTime具有一个Local或Unspecified（默认）的DateTimeKind，那么偏移量从当前的本地时区计算得到。

为了在其他方法中进行转换，DateTimeOffset提供了三个属性，它们返回DateTime类型的值：

- UtcDateTime属性会返回一个UTC时间表示的DateTime。
- LocalDateTime属性返回一个以当前本地时区（在需要时进行转换）表示的DateTime。
- DateTime属性返回一个以任意指定的时区表示的DateTime，以及一个Unspecified的Kind（例如，它返回一个UTC时间及偏移值）。

4. 当前的DateTime/DateTimeOffset

DateTime和DateTimeOffset都具有一个静态的Now属性，它会返回当前的日期和时间：

```
Console.WriteLine (DateTime.Now);          // 11/11/2007 1:23:45 PM
Console.WriteLine (DateTimeOffset.Now);    // 11/11/2007 1:23:45 PM -06:00
```

DateTime也具有Today属性，它返回日期部分：

```
Console.WriteLine (DateTime.Today);        // 11/11/2007 12:00:00 AM
```

静态的Utcnow属性会返回以UTC表示的当前日期和时间：

```
Console.WriteLine (DateTime.Utcnow);          // 11/11/2007 7:23:45 AM
Console.WriteLine (DateTimeOffset.Utcnow);    // 11/11/2007 7:23:45 AM +00:00
```

所有这些方法的精度取决于操作系统，并且一般是在10～20毫秒内。

5. 处理日期和时间

DateTime和DateTimeOffset提供了返回各种日期/时间的类似实例属性：

```
DateTime dt = new DateTime (2000, 2, 3, 10, 20, 30);
Console.WriteLine (dt.Year);          // 2000
Console.WriteLine (dt.Month);         // 2
Console.WriteLine (dt.Day);           // 3
Console.WriteLine (dt.DayOfWeek);     // Thursday
Console.WriteLine (dt.DayOfYear);     // 34

Console.WriteLine (dt.Hour);          // 10
Console.WriteLine (dt.Minute);        // 20
Console.WriteLine (dt.Second);        // 30
Console.WriteLine (dt.Millisecond);   // 0
Console.WriteLine (dt.Ticks);         // 630851700300000000
Console.WriteLine (dt.TimeOfDay);     // 10:20:30 返回一个TimeSpan值
```

DateTimeOffset也有一个类型为TimeSpan的Offset属性。

这两种类型都提供以下执行计算的实例方法（大多数都接受类型为double或int的参数）：

```
AddYears AddMonths AddDays
AddHours AddMinutes AddSeconds AddMilliseconds AddTicks
```

这些方法都返回一个新的DateTime或DateTimeOffset，并且考虑了闰年的问题。可以通过输入一个负值来执行减法运算。

Add方法会将一个TimeSpan值与一个DateTime或DateTimeOffset值相加。操作符+已经重载，从而可执行相同的运算：

```
TimeSpan ts = TimeSpan.FromMinutes (90);
Console.WriteLine (dt.Add (ts));
Console.WriteLine (dt + ts);
```

可以用一个TimeSpan值减去一个DateTime/DateTimeOffset值，以及用两个DateTime/DateTimeOffset值进行相减。后一种运算会得到：

```
DateTime thisYear = new DateTime (2007, 1, 1);
DateTime nextYear = thisYear.AddYears (1);
TimeSpan oneYear = nextYear - thisYear;
```

6. 格式化和解析

调用DateTime的ToString会将结果格式化为一个短日期（全部是数字），后跟一个长时间（包括秒）。例如：

```
13/02/2000 11:50:30 AM
```

默认情况下，操作系统的控制面板决定日、月或年是否在前，是否使用前导零，以及是使用12小时还是24小时时间格式。

调用DateTimeOffset的ToString效果是一样的，只是它同时返回偏移值：

```
3/02/2000 11:50:30 AM -06:00
```

ToShortDateString和ToLongDateString方法只返回日期部分。长日期格式也是由控制面板决定的，例如“Saturday, 17 February 2007”。ToShortTimeString和ToLongTimeString只返回时间部分，如17:10:10（前者不包括秒）。

刚刚介绍的这四个方法实际上是四个不同的*格式字符串*的快捷方式。ToString重载后可以接受一个格式字符串和提供者，这允许指定大量的选项，并且控制区域设置的应用方式。

警告： 如果文化设置与格式化发生时差别很大，那么DateTimes和DateTimeOffsets可能会解析出错。可以通过使用ToString及一个忽略文化设置的格式字符串来避免这个问题（例如“o”）：

```
DateTime dt1 = DateTime.Now;
string cannotBeMisparsed = dt1.ToString ("o");
DateTime dt2 = DateTime.Parse (cannotBeMisparsed);
```

静态的Parse和ParseExact方法执行与ToString相反的操作，它们将一个字符串转换成一个DateTime或DateTimeOffset。Parse方法重载后也可以接受格式提供者。

7. DateTime和DateTimeOffset空值

因为DateTime和DateTimeOffset是结构体，它们是不可为空的。当需要将它们设置为空时，可以使用以下两种方法：

- 使用一个Nullable类型值（例如，DateTime?或DateTimeOffset?）。
- 使用静态域DateTime.MinValue或DateTimeOffset.MinValue（这些类型的默认值）。

使用一个可空值通常是最佳方法，因为编译器会防止出现错误。DateTime.MinValue对于兼容C# 2.0（引入了可空类型）之前编写的代码是很有用的。

警告： 在一个DateTime.MinValue值上调用ToUniversalTime或ToLocalTime可以使它不再是DateTime.MinValue（这取决于在GMT的哪一边）。如果是在GMT右边（英格兰时区，夏令时以外），那么完全没有问题，因为本地时间和UTF时间是相同的。这是对经受英格兰冬天之苦的补偿！

6.3 日期与时区

在本节中，我们将更详细地介绍时区如何影响DateTime和DateTimeOffset，也将介绍提供关于时间偏移和夏令时信息的TimeZone和TimeZoneInfo类型。

6.3.1 DateTime与时区

DateTime处理时区的方法很简单。在内部，它使用两部分信息来存储DateTime：

- 一个62位数，表示从1/1/0001开始的时间计数
- 一个2位枚举数，表示DateTimeKind（Unspecified、Local或Utc）

当比较两个DateTime实例时，只有它们的计数值是可以比较的，它们的DateTimeKinds是被忽略的：

```
DateTime dt1 = new DateTime (2000, 1, 1, 10, 20, 30, DateTimeKind.Local);
DateTime dt2 = new DateTime (2000, 1, 1, 10, 20, 30, DateTimeKind.Utc);
Console.WriteLine (dt1 == dt2);          // True
DateTime local = DateTime.Now;
DateTime utc = local.ToUniversalTime();
Console.WriteLine (local == utc);        // False
```

实例方法ToUniversalTime/ToLocalTime会转换为普通/本地时间。这些方法会使用计算机的当前时区设置，返回一个包含Utc或Local的DateTimeKind的新DateTime。如果调用ToUniversalTime处理一个已经是Utc的DateTime，或者调用ToLocalTime处理一个已经是Local的DateTime，那么是不会发生任何转换的。然而，如果调用ToUniversalTime或ToLocalTime处理一个Unspecified的DateTime，那么就会发生转换。

可以使用静态的DateTime.SpecifyKind方法创建与另一个同类DateTime不同的DateTime：

```
DateTime d = new DateTime (2000, 12, 12);    // Unspecified
DateTime utc = DateTime.SpecifyKind (d, DateTimeKind.Utc);
Console.WriteLine (utc);                     // 12/12/2000 12:00:00 AM
```

6.3.2 DateTimeOffset与时区

在内部，DateTimeOffset包括一个值总是UTC的DateTime域和一个以分钟表示UTC偏移值的16位整型域。这些比较只检查（UTC）DateTime；而Offset主要用来实现格式化。

ToUniversalTime/ToLocalTime方法会返回一个表示相同时间点的DateTimeOffset，但是它包含一个UTC或Local偏移量。与DateTime不同的是，这些方法不影响底层的日期/时间值，而只影响偏移值。

```
DateTimeOffset local = DateTimeOffset.Now;
DateTimeOffset utc   = local.ToUniversalTime();

Console.WriteLine (local.Offset);          // -06:00:00 (在中美洲)
Console.WriteLine (utc.Offset);            // 00:00:00

Console.WriteLine (local == utc);          // True
```

要在比较中包含Offset，必须使用EqualsExact方法：

```
Console.WriteLine (local.EqualsExact (utc)); // False
```

6.3.3 TimeZone与TimeZoneInfo

TimeZone和TimeZoneInfo类提供了关于时区名称、UTC偏移量和夏令时规则等信息。TimeZoneInfo在两者中较为强大，并且是Framework 3.5的新增特性。

这两种类型的最大区别是TimeZone只能访问当前的本地时区，而TimeZoneInfo则能够访问全世界的时区。而且，TimeZoneInfo具有更丰富的（虽然有时不宜使用）基于规则的夏令时描述模型。

1. TimeZone

静态的TimeZone.CurrentTimeZone方法会基于当前的本地设置返回一个TimeZone对象。下面的语句说明在Western Australia运行的结果：

```
TimeZone zone = TimeZone.CurrentTimeZone;
Console.WriteLine (zone.StandardName);      // W. Australia Standard Time
Console.WriteLine (zone.DaylightName);      // W. Australia Daylight Time
```

IsDaylightSavingTime和GetUtcOffset方法则按以下方式工作：

```
DateTime dt1 = new DateTime (2008, 1, 1);
DateTime dt2 = new DateTime (2008, 6, 1);
Console.WriteLine (zone.IsDaylightSavingTime (dt1)); // True
Console.WriteLine (zone.IsDaylightSavingTime (dt2)); // False
Console.WriteLine (zone.GetUtcOffset (dt1));         // 09:00:00
Console.WriteLine (zone.GetUtcOffset (dt2));         // 08:00:00
```

GetDaylightChanges方法会返回一个指定年份的夏令时信息：

```
DaylightTime day = zone.GetDaylightChanges (2008);
Console.WriteLine (day.Start);    // 26/10/2008 2:00:00 AM (注意是日/月/年)
Console.WriteLine (day.End);      // 30/03/2008 3:00:00 AM
Console.WriteLine (day.Delta);    // 01:00:00
```

2. TimeZoneInfo

TimeZoneInfo类采用类似的处理方式。TimeZoneInfo.Local返回当前的本地时区：

```
TimeZoneInfo zone = TimeZoneInfo.Local;
Console.WriteLine (zone.StandardName);      // W. Australia Standard Time
Console.WriteLine (zone.DaylightName);      // W. Australia Daylight Time
```

TimeZoneInfo也有IsDaylightSavingTime和GetUtcOffset方法，区别是它们可以接受DateTime或DateTimeOffset。

通过时区ID调用FindSystemTimeZoneById，就可以获得世界上任何一个时区的TimeZoneInfo。从这一方面可以证明，只有TimeZoneInfo才有这种功能。我们将坚持使用Western Australia，其中的原因很快便会明朗。

```
TimeZoneInfo wa = TimeZoneInfo.FindSystemTimeZoneById
                  ("W. Australia Standard Time");

Console.WriteLine (wa.Id);                           // W. Australia Standard Time
Console.WriteLine (wa.DisplayName);                  // (GMT+08:00) Perth
Console.WriteLine (wa.BaseUtcOffset);                // 08:00:00
Console.WriteLine (wa.SupportsDaylightSavingTime);   // True
```

属性Id对应于传递给FindSystemTimeZoneById的值。静态的GetSystemTimeZones方法则返回全世界所有的时区；因此，可以列出如下所示的全部有效的时区ID字符串：

```
foreach (TimeZoneInfo z in TimeZoneInfo.GetSystemTimeZones())
  Console.WriteLine (z.Id);
```

提示：还可以通过调用TimeZoneInfo.CreateCustomTimeZone创建一个自定义的时区。因为TimeZoneInfo是不可变的，所以必须输入方法参数需要的所有相关数据。

可以通过调用ToSerializedString将一个预定义或自定义的时间序列化为具有（半）可读性的字符串，还可以调用TimeZoneInfo.FromSerializedString对其进行反序列化。

静态的ConvertTime方法可以将一个DateTime或DateTimeOffset从一个时区转换到另一个时区。可以只包含一个目标TimeZoneInfo，或者同时包含源和目标TimeZoneInfo对象。还可以使用ConvertTimeFromUtc和ConvertTimeToUtc方法从UTC直接转换。

对于夏令时处理，TimeZoneInfo还提供了以下方法：

- 如果一个DateTime位于时钟前移时跳过的小时（或时差），那么IsInvalidTime会返回true。
- 如果DateTime或DateTimeOffset位于时钟回调时重复的小时（或时差），那么IsAmbiguousTime会返回true。
- GetAmbiguousTimeOffsets会返回一个表示任意DateTime或DateTimeOffset的有效偏移选择的TimeSpans数组。

与TimeZone不同，可以从DateZoneInfo获取一个表示夏令时起止时间的简单日期。相反，必须调用GetAdjustmentRules，它返回关于所有年份的所有夏令时规则的声明总结。每条规则都有表示规则有效日期范围的DateStart和DateEnd。

```
foreach (TimeZoneInfo.AdjustmentRule rule in wa.GetAdjustmentRules())
  Console.WriteLine ("Rule: applies from " + rule.DateStart +
                                        " to " + rule.DateEnd);
```

Western Australia是在2006年中期首次引入*夏令时*的（然后在2009年废除）。要求为第一年设置一条特殊规则，因此，它有两条规则：

```
Rule: applies from 1/01/2006 12:00:00 AM to 31/12/2006 12:00:00 AM
Rule: applies from 1/01/2007 12:00:00 AM to 31/12/2009 12:00:00 AM
```

每一个AdjustmentRule都有一个类型为TimeSpan的DaylightDelta属性（几乎在每一种情况下都是一个小时），以及一些名为DaylightTransitionStart和DaylightTransitionEnd的属性。后面两个属性是TimeZoneInfo.TransitionTime类型的，具有以下属性:：

```
public bool IsFixedDateRule { get; }
public DayOfWeek DayOfWeek { get; }
public int Week { get; }
public int Day { get; }
public int Month { get; }
public DateTime TimeOfDay { get; }
```

转换时间有一些复杂，因为它需要表示固定日期和浮动日期。浮动日期的一个例子就是三月的最后一天。解析一个转时间的规则有以下几种：

1. 对于结束转换，如果IsFixedDateRule是true，Day是1，Month是1，而TimeOfDay是DateTime.MinValue，那么这个年份里没有夏令时的结束时间（这只可能出现在南半球刚刚引入夏令时的某个地区）。

2. 否则，如果IsFixedDateRule是true，那么Month、Day和TimeOfDay属性决定了调整规则的起止时间。

3. 否则，如果IsFixedDateRule是false，那么Month、DayOfWeek、Week和TimeOfDay属性决定了调整规则的起止时间。

在最后一种情况中，Week指的是月份的周数，“5”表示最后一周。我们可以通过列举wa时区来说明这一点：

```
foreach (TimeZoneInfo.AdjustmentRule rule in wa.GetAdjustmentRules())
{
  Console.WriteLine ("Rule: applies from " + rule.DateStart +
                                      " to " + rule.DateEnd);

  Console.WriteLine ("   Delta: " + rule.DaylightDelta);

  Console.WriteLine ("   Start: " + FormatTransitionTime
                                     (rule.DaylightTransitionStart, false));

  Console.WriteLine ("    End: " + FormatTransitionTime
                                     (rule.DaylightTransitionEnd, true));

  Console.WriteLine();
}
```

在FormatTransitionTime中，我们使用以上介绍的规则：

```
static string FormatTransitionTime (TimeZoneInfo.TransitionTime tt,
                                    bool endTime)
{
  if (endTime && tt.IsFixedDateRule
              && tt.Day == 1 && tt.Month == 1
              && tt.TimeOfDay == DateTime.MinValue)
    return "-";

  string s;
  if (tt.IsFixedDateRule)
    s = tt.Day.ToString();
  else
    s = "The " +
        "first second third fourth last".Split() [tt.Week - 1] +
        "" + tt.DayOfWeek + " in";

  return s + " " + DateTimeFormatInfo.CurrentInfo.MonthNames [tt.Month-1]
           + " at " + tt.TimeOfDay.TimeOfDay;
}
```

Western Australia的结果是很有趣的，因为它同时演示了固定和浮动的日期规则以及没有结束日期的情况：

```
Rule: applies from 1/01/2006 12:00:00 AM to 31/12/2006 12:00:00 AM
   Delta: 01:00:00
   Start: 3 December at 02:00:00
   End:   -

Rule: applies from 1/01/2007 12:00:00 AM to 31/12/2009 12:00:00 AM
   Delta: 01:00:00
   Start: The last Sunday in October at 02:00:00
   End:   The last Sunday in March at 03:00:00
```

提示： Western Australia实际只在这一点上是不同的。下面说明我们是如何发现它的：

```
from zone in TimeZoneInfo.GetSystemTimeZones()
let rules = zone.GetAdjustmentRules()
where
  rules.Any
   (r => r.DaylightTransitionEnd.IsFixedDateRule) &&
  rules.Any
   (r => !r.DaylightTransitionEnd.IsFixedDateRule)
select zone
```

6.3.4 夏令时与DateTime

如果使用DateTimeOffset或UTCDateTime进行等值比较是不受夏令时影响的。但是对于本地DateTime，夏令时可能会带来影响。

这些规则可以总结为：

- 夏令时只影响本地时间，而不影响UTC时间。
- 当时钟回调时，如果（且仅当）使用本地DateTime时，依赖于时间前移的比较可能会出错。
- 即使时钟回调了，总是可以在UTC和本地时间之间来回调整（在同一台计算机上）。

IsDaylightSavingTime可以指示给定的本地DateTime是否与夏令时有关。UTC时间总是会返回false：

```
Console.Write (DateTime.Now.IsDaylightSavingTime());      // True或False
Console.Write (DateTime.Utcnow.IsDaylightSavingTime());   // 总是为False
```

假设dto是一个DateTimeOffset，下面的表达式得到同样的效果：

```
dto.LocalDateTime.IsDaylightSavingTime
```

夏令时的结束会给使用本地时间的算法带来非常复杂的问题。当时钟回调时，同一个小时（或者更准确说是Delta）会重复。我们可以通过用计算机的“模糊时区”实例化一个DateTime来说明这一点，然后减去Delta（这个例子需要测试夏令时有一定的耐心！）：

```
DaylightTime changes = TimeZone.CurrentTimeZone.GetDaylightChanges (2010);
TimeSpan halfDelta = new TimeSpan (changes.Delta.Ticks / 2);
DateTime utc1 = changes.End.ToUniversalTime() - halfDelta;
DateTime utc2 = utc1 - changes.Delta;
```

如果代码依赖于时钟前移，那么将这些变量转换为本地时间可以说明为什么应该使用UTC，而不是本地时间：

```
DateTime loc1 = utc1.ToLocalTime();  // （太平洋标准时区）
DateTime loc2 = utc2.ToLocalTime();
Console.WriteLine (loc1);            // 2/11/2010 1:30:00 AM
Console.WriteLine (loc2);            // 2/11/2010 1:30:00 AM
Console.WriteLine (loc1 == loc2);    // True
```

尽管loc1和loc2显示的结果是相同的，但是它们内部是不同的。DateTime保留了一个特殊位来表示模糊时区的哪一边是任意的本地时间！正如我们看到的，这个位在比较时会被忽略。但是当明确地格式化DateTime时，它就会起作用：

```
Console.Write (loc1.ToString ("o")); // 2010-11-02T02:30:00.0000000-08:00
Console.Write (loc2.ToString ("o")); // 2010-11-02T02:30:00.0000000-07:00
```

当将时间转换回UTC时，这个位就是可读的，它可以保证本地和UTC时间的双向转换：

```
Console.WriteLine (loc1.ToUniversalTime() == utc1); // True
Console.WriteLine (loc2.ToUniversalTime() == utc2); // True
```

提示： 可以通过先调用ToUniversalTime处理两个DateTime而对它们进行比较。如果（且仅当）其中一个是Unspecified的DateTimeKind时，这个方法才会失效。这里可能发生的错误是选用DateTimeOffset的另一个原因。

6.3.5 格式化与解析

格式化表示将对象转换为一个字符串；而解析表示将一个字符串转换为某种对象。在编程时，有很多情况要求我们使用格式化或解析。因此，.NET Framework提供了许多转换机制：

`ToString`和`Parse`
: 这些方法提供了许多类型的默认功能。

格式提供者
: 这些实际上是额外的`ToString`（和`Parse`）方法，它们可以接受*格式字符串*和*格式提供者*。格式提供者是非常灵活和与文化相关的。.NET Framework包含数字类型和`DateTime`/`DateTimeOffset`的格式提供者。

`XmlConvert`
: 这是一个静态类，它包含格式和解析方法，同时支持XML标准。当你需要处理文化或者希望避免解析错误时，`XmlConvert`也适用于一般用途的转换。`XmlConvert`支持数字类型、布尔型、`DateTime`、`DateTimeOffset`、`TimeSpan`和`Guid`。

类型转换器
: 这是面向设计器和XAML解析器的。

在这一节，我们将讨论前两个机制，主要是格式提供者。在后续章节中，我们将介绍`XmlConvert`、类型转换器以及其他的转换机制。

6.3.6 ToString和Parse

最简单的格式化机制是`ToString`方法。它能够为所有简单的值类型（布尔型、`DateTime`、`DateTimeOffset`、`TimeSpan`、`Guid`和所有数字类型）产生有意义的输出。对于反向转换，这些类型都定义了静态的Parse方法。例如：

```
string s = true.ToString();         // s = "True"
bool b = bool.Parse (s);            // b = true
```

如果解析失败，它会抛出一个`FormatException`。许多类型还定义了`TryParse`方法，如果转换失败，它会返回`false`，而不是抛出一个异常：

```
int i;
```

```
bool failure = int.TryParse ("qwerty", out i);
bool success = int.TryParse ("123", out i);
```

如果遇到错误，在异常处理代码块中调用TryParse是更快速且更好的处理方式。

DateTime(Offset)的Parse、TryParse方法和数字类型会使用本地文化设置；可以通过指定一个CultureInfo对象修改这个行为。指定一个不变的文化通常是很好的做法。例如，将"1.234"解析为一个double型数值，在德国会得到1234：

```
Console.WriteLine (double.Parse ("1.234")); // 1234（德国的表示方式）
```

这是因为在德国，"."表示千位分隔符，而不是小数点。而指定一个不变文化可以解决这个问题：

```
double x = double.Parse ("1.234", CultureInfo.InvariantCulture);
```

同样的方法也适用于ToString()：

```
string x = 1.234.ToString (CultureInfo.InvariantCulture);
```

6.3.7 格式提供者

有时需要对格式化和解析进行更多的控制。例如，格式化一个DateTime(Offset)的方法有很多。格式提供者支持大量的格式化和解析控制，并且支持数字类型和日期/时间类型。格式提供者也可以用于用户界面的格式化和解析控制。

使用格式提供者的方法是IFormattable。所有数字类型和DateTime(Offset)都实现了这个接口：

```
public interface IFormattable
{
  string ToString (string format, IFormatProvider formatProvider);
}
```

第一个参数是*格式字符串*，第二个参数是*格式提供者*。格式字符串提供一些指令；而格式提供者则决定了这些指令是如何转换的。例如：

```
NumberFormatInfo f = new NumberFormatInfo();
f.CurrencySymbol = "$$";
Console.WriteLine (3.ToString ("C", f));     // $$ 3.00
```

在这里，"C"是一个格式字符串，它表示货币；而NumberFormatInfo对象是一个格式提供者，它决定了如何表示货币和其他数字表示。这个机制支持全球化。

提示： 所有数字和日期的格式字符串都列在本章的"标准格式字符串和解析标记"中。

如果指定一个null格式字符串或提供者，那么它会使用默认值。默认的格式提供者是CultureInfo.CurrentCulture，除非重新赋值，否则它对应的是计算机的运行时控制面板设置。例如：

```
Console.WriteLine (10.3.ToString ("C", null)); // $10.30
```

为了方便起见，大多数类型都重载了ToString方法，可以省略null提供者：

```
Console.WriteLine (10.3.ToString ("C"));     // $10.30
```

```
Console.WriteLine (10.3.ToString ("F4"));     // 10.3000（精确到小数点后4位）
```

调用ToString处理一个DateTime(Offset)或不带参数的数字类型相当于使用一个默认的格式提供者，即格式字符串为空。

.NET Framework定义了以下三种格式提供者（它们都实现了IFormatProvider）：

NumberFormatInfo、DateTimeFormatInfo、CultureInfo。

提示：所有enum类型都可以格式化，但是它们没有具体的IFormatProvider类。

1. 格式提供者和CultureInfo

在格式提供者的上下文中，CultureInfo作为其他两个格式提供者的间接机制，返回一个适合文化区域设置的NumberFormatInfo或DateTimeFormatInfo。

在下面的例子中，我们请求了一个特殊文化（英国英语）：

```
CultureInfo uk = CultureInfo.GetCultureInfo ("en-GB");
Console.WriteLine (3.ToString ("C", uk));     // £3.00
```

使用适合en-GB文化的默认NumberFormatInfo对象来执行。

下例是使用不变文化设置来格式化一个DateTime。不变文化总是保持同一个设置，而与计算机的设置无关：

```
DateTime dt = new DateTime (2000, 1, 2);
CultureInfo iv = CultureInfo.InvariantCulture;
Console.WriteLine (dt.ToString (iv));            // 01/02/2000 00:00:00
Console.WriteLine (dt.ToString ("d", iv));       // 01/02/2000
```

提示：不变文化是基于美国文化的，它有以下的不同点：

- 货币符号是¤而不是$。
- 日期和时间格式化为带前导零的（虽然月份仍然在前）。
- 时间是使用24小时格式，而不是AM/PM标识符。

2. 使用NumberFormatInfo或DateTimeFormatInfo

在下例中，我们实例化一个NumberFormatInfo，并将组分隔符从逗号改为空格。然后我们使用它将一个数字保留小数点后三位：

```
NumberFormatInfo f = new NumberFormatInfo ();
f.NumberGroupSeparator = " ";
Console.WriteLine (12345.6789.ToString ("N3", f));   // 12 345.679
```

NumberFormatInfo或DateTimeFormatInfo的初始设置是基于不变文化的。然而，有时候选择不同的起点会更有用。为了做到这一点，可以克隆一个现有的格式提供者：

```
NumberFormatInfo f = (NumberFormatInfo)
                     CultureInfo.CurrentCulture.NumberFormat.Clone();
```

虽然原始格式提供者是只读的，但克隆的格式提供者总是可写的。

3. 组合格式化

组合格式字符串可以包含组合变量替代符和格式字符串。静态的string.Format方法可以接受一个组合格式字符串，我们在本章的“字符串与文本处理”已经介绍过，例如：

```
string composite = "Credit={0:C}";
Console.WriteLine (string.Format (composite, 500));   // Credit=$500.00
```

Console类本身重载了它的Write和WriteLine方法，以接受一个组合格式字符串，这个例子可以简化为：

```
Console.WriteLine ("Credit={0:C}", 500); // Credit=$500.00
```

可以将组合格式字符串附加到StringBuilder上（通过AppendFormat），以及进行I/O的TextWriter上（见第15章）：

string.Format接受可选的格式提供者。它的一个简单应用是调用任意对象的ToString，同时传递格式提供者。例如：

```
string s = string.Format (CultureInfo.InvariantCulture, "{0}", someObject);
```

等同于下面的代码：

```
string s;
if (someObject is IFormattable)
  s = ((IFormattable)someObject).ToString (null,
                                        CultureInfo.InvariantCulture);
else if (someObject == null)
  s = "";
else
  s = someObject.ToString();
```

4. 通过格式提供者进行解析

通过格式提供者进行解析是没有标准接口的。相反，每一个参与的类型都会重载它静态的Parse（和TryParse）方法来接受一个格式提供者，以及一个可选的NumberStyles或DateTimeStyles枚举类型。

NumberStyles和DateTimeStyles决定解析的工作方式：它们指定一些设置，如是否允许括号或货币符号出现在输入字符串中（默认情况下，这两个设置都是否）。例如：

```
int error = int.Parse ("(2)");         // 抛出异常

int minusTwo = int.Parse ("(2)", NumberStyles.Integer |
                                 NumberStyles.AllowParentheses);   // 正常

decimal fivePointTwo = decimal.Parse ("£5.20", NumberStyles.Currency,
                       CultureInfo.GetCultureInfo ("en-GB"));
```

下一节列出了所有的NumberStyles和DateTimeStyles成员以及每一个类型的默认解析规则。

5. IFormatProvider和ICustomFormatter

所有格式提供者都实现了IFormatProvider接口：

```
public interface IFormatProvider { object GetFormat (Type formatType); }
```

这个方法的目的是提供间接方法允许CultureInfo按照一个正确的NumberFormatInfo或DateTimeInfo对象来完成这个操作。

通过实现IFormatProvider以及ICustomFormatter，也可以编写能够处理现有类型的自定义格式提供者。ICustomFormatter只定义了下面这个方法：

```
string Format (string format, object arg, IFormatProvider formatProvider);
```

下面的自定义格式提供者能够将数字转换为单词：

```
// 程序可以从这里下载：http://www.albahari.com/nutshell/

public class WordyFormatProvider : IFormatProvider, ICustomFormatter
{
  static readonly string[] _numberWords =
   "zero one two three four five six seven eight nine minus point".Split();

  IFormatProvider _parent;   // 允许客户链接到格式提供者

  public WordyFormatProvider () : this (CultureInfo.CurrentCulture) { }
  public WordyFormatProvider (IFormatProvider parent)
  {
    _parent = parent;
  }

  public object GetFormat (Type formatType)
  {
    if (formatType == typeof (ICustomFormatter)) return this;
    return null;
  }

  public string Format (string format, object arg, IFormatProvider prov)
  {
    // 如果它不是所要的格式字符串，那么使用上一级提供者：
    if (arg == null || format != "W")
      return string.Format (_parent, "{0:" + format + "}", arg);

    StringBuilder result = new StringBuilder();
    string digitList = string.Format (CultureInfo.InvariantCulture,
                                      "{0}", arg);
    foreach (char digit in digitList)
    {
      int i = "0123456789-.".IndexOf (digit);
      if (i == -1) continue;
      if (result.Length > 0) result.Append (' ');
      result.Append (_numberWords[i]);
    }
    return result.ToString();
  }
}
```

注意在Format方法中，我们通过InvariantCulture使用string.Format将输入数字转换成字符串。它比只通过调用ToString()来处理arg简单得多，但是它会使用CurrentCulture。需要不变文化的原因体现在下面几行代码中：

```
int i = "0123456789-.".IndexOf (digit);
```

这里的关键是数字字符串只由0123456789-.等字符构成，并且不包含任何国际化字符。

下面的例子使用的是WordyFormatProvider：

```
double n = -123.45;
IFormatProvider fp = new WordyFormatProvider();
Console.WriteLine (string.Format (fp, "{0:C} in words is {0:W}", n));

// -$123.45用单词表示是one two three point four five
```

自定义格式提供者只能够用在组合格式字符串中。

6.4 标准格式字符串与解析标记

标准格式字符串决定数字类型或DateTime/DateTimeOff集是如何转换为字符串的。格式字符串有两种：

标准格式字符串
　　可以使用标准格式字符串实现基本的控制。标准格式字符串是由一个字母及其后面一个可选的数字（它的作用由前面的字母决定）组成。例如，"C"或"F2"。

自定义格式字符串
　　可以使用自定义格式字符串作为模板对每一个字符进行精细控制。例如，"0:#.000E+00"。

自定义格式字符串与自定义格式提供者无关。

6.4.1 数字格式字符串

表6-2列出了所有的标准数字格式字符串。

表6-2：标准数字格式字符串

字母	意义	示例输入	结果	说明
G或g	“一般”	1.2345, "G"	1.2345	转换为小或大数字的指数符号。
		0.00001, "G"	1E-05	
		0.00001, "g"	1e-05	G3表示将精度限制为总共3个数字（小数点前后相加）
		1.2345, "G3"	1.23	
		12345, "G3"	1.23E04	
F	固定小数位数	2345.678, "F2"	2345.68	F2表示将数字四舍五入为小数点后两位
		2345.6, "F2"	2345.60	
N	带组分隔符的固定小数位数（“数值”）	2345.678, "N2"	2,345.68	同上，带分组分隔符（如1,000）（详细用法见格式提供者）
		2345.6, "N2"	2,345.60	

表6-2：标准数字格式字符串（续）

字母	意义	示例输入	结果	说明
D	数字前填充0	123, "D5"	00123	只适用于整数。
		123, "D1"	123	D5表示将数字填充为5位数，但不把大于5位的数字截短
E或e	强制使用指数符号	56789, "E"	5.678900E+004	默认精确到小数点后6位
		56789, "e"	5.678900e+004	
		56789, "E2"	5.68E+004	
C	货币	1.2, "C"	$1.20	不带数字的C使用格式提供者的默认D.P.数
		1.2, "C4"	$1.2000	
P	百分比	.503, "P"	50.30%	使用格式提供者的符号和布局
		.503, "P0"	50%	小数位可以选择性地重写
X或x	十六进制数	47, "X"	2F	X指大写十六进制数；x指小写十六进制数
		47, "x"	2f	
		47, "X4"	002F	只适用于整数
R	圆整	1f / 3f, "R"	0.333333343	针对float和double类型，R会压缩所有数字保证精确圆整

如果不提供数字格式字符串或者使用null或空字符串，那么相当于使用不带数字的"G"标准格式化字符串。这包括以下两种形式：

- 小于10^{-4}或大于该类型精度的数字表示为指数（科学计数）形式。
- 受float或double精度限制的两位小数是经过圆整的，避免了从二进制形式转换过来的内在不精确性。

提示：这里描述的自动圆整通常是有好处的，但是不太引人注意。然而，如果需要圆整一个数字，那么它可能会产生问题；换言之，将它转换为字符串，再转换回数字（可能反复多次），同时保持值不变。因此，我们需要使用"R"格式字符串来处理这种隐式转换。

表6-3列出了一些自定义数字格式字符串。

表6-3：自定义数字格式字符串

字母	意义	示例输入	结果	说明
#	数字占位符	12.345, ".##"	12.35	限定D.P.后小数位数
		12.345, ".####"	12.345	
0	0占位符	12.345, ".00"	12.35	与#相同，但是用0补齐前后不足的位数
		12.345, ".0000"	12.3500	
		99, "000.00"	099.00	

表6-3：自定义数字格式字符串

字母	意义	示例输入	结果	说明
.	小数点			表示D.P. 符号来自NumberFormatInfo
,	分组符号	1234, "#,###,###" 1234, "0,000,000"	1,234 0,001,234	符号来自NumberFormatInfo
, （同上）	倍增符号	1000000, "#," 1000000, "#,,	1000 1	如果逗号在D.P.末尾或前面，那么它就是一个倍增符——将结果除以1,000、1,000,000等
	百分号	0.6, "00%"	60%	先乘以100，然后增加从NumberFormatInfo获得的百
分号				
E0、e0、E +0、e+0 E-0、e-0	指数符号	1234, "0E0" 1234, "0E+0" 1234, "0.00E00" 1234, "0.00e00"	1E0 1E+3 1.25E03 1.25e03	
\	转义字符	50, @"\#0"	#50	与字符串前缀@一起使用——或者使用\\
'xx''xx'	逐字引用	50, "0 '...'"	50 ...	
;	分段符	15, "#;(#);zero" -5, "#;(#);zero" 0, "#;(#);zero"	15 -5 0	（正数） （负数） (0)
其他字符	逐字	35.2, "$0 . 00c"	$35 . 00c	

6.4.2 NumberStyles

每一种数字类型都定义了一个静态的Parse方法，它接受NumberStyles参数。NumberStyles是一个标记枚举值，可以判断如何读取转换为数字类型的字符串。它具有以下可组合成员变量：

```
AllowLeadingWhite    AllowTrailingWhite
AllowLeadingSign     AllowTrailingSign
AllowParentheses     AllowDecimalPoint
AllowThousands       AllowExponent
AllowCurrencySymbol  AllowHexSpecifier
```

NumberStyles也定义了以下复合成员变量：

```
None Integer Float Number HexNumber Currency Any
```

除了None，所有复合值都包含AllowLeadingWhite和AllowTrailingWhite。其余标记如图6-1所示，其中最常使用的有三种。

	AllowLeadingSign	AllowTrailingSign	AllowParenthesis	AllowDecimalPoint	AllowThousands	AllowExponent	AllowCurrencySymbol	AllowHexSpecifier
Integer	✓							
Float	✓			✓		✓		
Number	✓	✓		✓	✓			
HexNumber								✓
Currency	✓	✓	✓	✓	✓		✓	
Any	✓	✓	✓	✓	✓	✓	✓	

图6-1：复合的NumberStyles

当不指定任何标记就调用Parse时，它会使用如图6-2所示的默认值。

如果不希望使用如图6-2所示的默认值，那么必须明确指定NumberStyles：

```
int thousand = int.Parse ("3E8", NumberStyles.HexNumber);
int minusTwo = int.Parse ("(2)", NumberStyles.Integer|NumberStyles.AllowParentheses);
double aMillion = double.Parse ("1,000,000", NumberStyles.Any);
decimal threeMillion = decimal.Parse ("3e6", NumberStyles.Any);
decimal fivePointTwo = decimal.Parse ("$5.20", NumberStyles.Currency);
```

因为我们没有指定格式提供者，所以这个例子只支持本地货币符号、分组符号、小数点等。下一个例子是以硬编码方式使用欧元符号和空格分组符号来表示货币：

```
NumberFormatInfo ni = new NumberFormatInfo();
ni.CurrencySymbol = "€";
ni.CurrencyGroupSeparator = " ";
double million = double.Parse ("€1 000 000", NumberStyles.Currency, ni);
```

	Default parsing flags	AllowLeadingSign	AllowTrailingSign	AllowParenthesis	AllowDecimalPoint	AllowThousands	AllowExponent	AllowCurrencySymbol	AllowHexSpecifier
Integral types	`Integer`	✓							
`double` and `float`	`Float | AllowThousands`	✓			✓	✓	✓		
`decimal`	`Number`	✓	✓		✓	✓			

图6-2：数字类型的默认解析标记

框架基础

6.4.3 Date/Time格式字符串

DateTime/DateTimeOffset的格式字符串可以按照它们使用的文化和格式提供者设置分成两组。采用这种方式的格式字符串列在表6-4中；其他格式字符串列在表6-5中。示例输出结果来自下面DateTime（表6-4所列的是不变文化情况）的格式化：

```
new DateTime(2000,1,2,17,18,19)
```

表6-4：与文化相关的日期/时间格式字符串

格式字符串	意义	示例输出
d	短日期	01/02/2000
D	长日期	Sunday, 02 January 2000
t	短时间	17:18
T	长时间	17:18:19
f	长日期 + 短时间	Sunday, 02 January 2000 17:18
F	长日期 + 长时间	Sunday, 02 January 2000 17:18:19
g	短日期 + 短时间	01/02/2000 17:18
G（默认）	短日期 + 长时间	01/02/2000 17:18:19
m、M	月与日	January 02
y、Y	年与月	2000 January

表6-5：与文化无关的日期/时间格式字符串

格式字符串	意义	示例输出	说明
o	圆整	2000-01-02T17:18:19.0000000	除非DateTimeKind为Unspecified，否则附加时区信息
r、R	RFC 1123标准	Sun, 02 Jan 2000 17:18:19 GMT	必须使用DateTime.ToUniversalTime将它显式转换为UTC
s	可排序；ISO 8601	2000-01-02T17:18:19	兼容基于文本的排序
u	"全局"排序	2000-01-02 17:18:19Z	与上相同，必须显式转换为UTC
U	UTC	Sunday, 02 January 2000 17:18:19	长日期+短时间，转换为UTC

格式字符串"r"、"R"和"u"会添加一个隐含表示UTC的后缀，但是它们不能将一个本地时间自动转换为UTC DateTime（所以必须自行转换）。奇怪的是，"U"会自动转移为UTC，但是它不会添加时区后缀！事实上，"o"是这组符号中唯一一个不需要干预就能够产生一个明确的DateTime的格式说明符。

DateTimeFormatInfo也支持自定义格式字符串，与数字的自定义格式字符串相似。这个清单是非常详细的，也可以在MSDN上找到。下面是一个自定义格式字符串的示例：

```
yyyy-MM-dd HH:mm:ss
```

解析与解析错误的DateTime

字符串可以任意选择将月份或日期放在前面，但是很可能出现解析错误，特别是在位于美国和加拿大以外地区时。在用户界面控制上不会出现问题，因为当格式化时会强制使用相同的设置进行解析。但是，当写入文件时，日期/月份就会出现解析错误的问题。下面是两种解决方法：

- 在格式化和解析时总是指定相同的显式文化（例如，不变文化）。
- 以一种与文化无关的方式格式化DateTime和DateTimeOffsets。

第二种方法更加可靠，特别是当选择4位年份数字在前的格式时，这种字符串不可能被解析错误。而且，使用符合标准的年份在前格式的字符串（如"o"）能够正确解析本地格式化的字符串，这与“万能格式”不同。使用"s"或"o"格式化的日期还具有可排序的优点。

为了说明这一点，假设生成下面一个与文化无关的DateTime字符串：

```
string s = DateTime.Now.ToString ("o");
```

提示："o"格式字符串的输出中包括毫秒。下面的自定义字符串结果与"o"相同，但是不包括毫秒：

```
yyyy-MM-ddTHH:mm:ss K
```

我们可以用两种方法来重新解析这个字符串。ParseExact要求严格匹配指定的格式字符串：

```
DateTime dt1 = DateTime.ParseExact (s, "o", null);
```

（可以使用XmlConvert的ToString和ToDateTime方法得到类似的结果。）

然而，Parse则隐式接受"o"格式和CurrentCulture格式：

```
DateTime dt2 = DateTime.Parse (s);
```

这种方法同时支持DateTime和DateTimeOffset。

提示：如果已知正在解析的字符串的格式，那么使用ParseExact会更好。它表示如果字符串格式不正确，那么就会抛出异常，这远好于得到一个解析错误的日期。

6.4.4 DateTimeStyles

DateTimeStyles是一种标记枚举类型，它在调用Parse处理DateTime(Offset)时提供了一些额外的指令。下面是它的成员：

```
None,
AllowLeadingWhite, AllowTrailingWhite, AllowInnerWhite,
AssumeLocal, AssumeUniversal, AdjustToUniversal,
NoCurrentDateDefault, RoundTripKind
```

还有一个复合成员AllowWhiteSpaces：

```
AllowWhiteSpaces = AllowLeadingWhite | AllowTrailingWhite | AllowInnerWhite
```

默认值是None。这意味着多余的空格通常都是禁止的（属于标准DateTime模式的空格除外）。

如果字符串不包含时区后缀（如Z或+9:00），那么就会应用AssumeLocal和AssumeUniversal。AdjustToUniversal仍然包含时区后缀，但是会使用当前区域设置转换为UTC。

如果解析一个包含时间但不包含日期的字符串，那么默认使用今天的日期。然而，如果应用了NoCurrentDateDefault标记，则使用1st January 0001。

6.4.5 Enum格式字符串

在本章的“Enums”中，我们将探讨枚举值的格式化和解析。表6-6列出了每个格式字符串以及将它应用到以下表达式的结果：

```
Console.WriteLine (System.ConsoleColor.Red.ToString (formatString));
```

表6-6：Enum格式字符串

格式字符串	意义	示例输出	说明
G或g	“一般”	Red	默认
F或f	视为设置了Flags属性	Red	即使枚举值没有Flags属性也处理组合成员
D或d	十进制值	12	获取底层整数值
X或x	十六进制值	0000000C	获取底层整数值

6.5 其他转换机制

在前面的两节中，我们介绍格式提供者——.NET中主要的格式化和解析机制。另外一个重要的转换机制是分散到各种类型和命名空间中的。有些可以与字符串相互转换，有些则采用其他的的转换方式。在本节中，我们将讨论以下内容：

- Convert类及其方法：
  - 实数到整数转换采用圆整而非截取方式
  - 解析二、八和十六进制数字
  - 动态转换
  - Base 64转换
- XmlConvert及其在XML格式化和解析中的作用
- 类型转换器及其在设计器和XAML格式化和解析中的作用
- 支持二进制转换的BitConverter

6.5.1 Convert

.NET Framework将以下类型称为基本类型：

- bool、char、string、System.DateTime和System.DateTimeOffset
- 所有C#数值类型

静态Convert类定义了将每一个基本类型转换成其他基本类型的方法。可是，这些方法大多数都是无

用的：它们或者抛出异常，或者是隐式转换的冗余方法。然而，其中还是有一些方法是很有用的，下面的章节将对它们进行介绍。

提示： 所有基本类型都（显式地）实现了IConvertible，它定义了转换到其他基本类型的方法。在大多数情况中，每一种方法的实现都直接调用Convert类中的方法。在少数情况中，编写一个接受IConvertible类型的参数是很有用的。

---

1. 实数到整数的圆整转换

在第2章中，我们介绍了允许在数字类型之间执行的隐式和显式转换。概括为：

- 隐式转换只支持无值丢失的转换（例如，int到double的转换）
- 只有会出现值丢失的转换才需要使用显式转换（例如，double到int的转换）

转换是经过效率优化的，因此它们将截断不符合要求的数据。当从一个实数转换成一个整数时，可能会出现问题，因为通常希望进行*圆整*，而不是截断。Convert的数值转换方法采用*圆整*的方式，因此能够解决这个问题：

```
double d = 3.9;
int i = Convert.ToInt32 (d); // i == 4
```

Convert采用银行的圆整方式，将中间值转换为偶整数（这样可以避免正负偏差）。如果这种方法存在问题，那么首先要调用Math.Round处理这个实数；它可以接受另一个参数，可以用它来控制中间值的圆整方式。

2. 解析二、八和十六进制数字

To（*整数类型*）方法隐含了一些重载方法，它们可以将数字转换为其他进制：

```
int thirty = Convert.ToInt32 ("1E", 16); // 以十六进制方式解析
uint five = Convert.ToUInt32 ("101", 2);  // 以二进制方式解析
```

第二个参数指定了进制数。它可以是任何一种进制（二、八、十或十六进制）！

3. 动态转换

有时候，需要在各种类型之间进行转换，但是只有在运行时才知道具体的类型。为此，Convert类提供了一个ChangeType方法：

```
public static object ChangeType (object value, Type conversionType);
```

源类型和目标类型都必须是一种基本类型。ChangeType也接受一个可选的IFormatProvider参数。例如：

```
Type targetType = typeof (int);
object source = "42";

object result = Convert.ChangeType (source, targetType);

Console.WriteLine (result);                 // 42
Console.WriteLine (result.GetType());       // System.Int32
```

这种方法适用于编写一个反序列化器来处理多种类型。它还能够将任意枚举类型转换为整型（见第3章的“Enums”）。

ChangeType的缺点是无法指定一个格式字符串或解析标记。

4. Base 64转换

有时候需要转换一些二进制数据，如XML文件或电子邮件内容中的文本文档的位图。Base 64是普遍使用的方法，它使用ASCII字符集的64个字符将二进制数据编码为可读字符。

Convert的ToBase64String方法能够将一个字节数组转换为Base 64；FromBase64String则执行相反操作。

6.5.2 XmlConvert

如果需要处理XML文件读写的数据，那么XmlConvert（位于System.Xml命名空间）提供了最适合用的格式化和解析方法。XmlConvert的方法不需要使用特殊的格式字符串就能够处理XML格式化的细微差别。例如，true在XML中是“true”而不是“True”。.NET Framework内部也经常使用XmlConvert。XmlConvert也很适合用来进行通用的与文化无关的序列化。

XmlConvert中的格式化方法都重载了ToString方法；解析方法为ToBoolean、ToDateTime等。例如：

```
string s = XmlConvert.ToString (true);        // s = "true"
bool isTrue = XmlConvert.ToBoolean (s);
```

DateTime的转换方法接受一个XmlDateTimeSerializationMode参数，它是一个包含下列值的枚举值：

```
Unspecified, Local, Utc, RoundtripKind
```

Local和Utc在格式化时（如果DateTime不在该时区）会发生转换。然后，时区信息会被附加到字符串上，例如：

```
2010-02-22T14:08:30.9375              // Unspecified
2010-02-22T14:07:30.9375+09:00        // Local
2010-02-22T05:08:30.9375Z             // Utc
```

Unspecified会在格式化之前删除DateTime（例如DateTimeKind）中包含的所有时区信息。RoundtripKind支持DateTime的DateTimeKind，所以当重新解析时，产生的DateTime结构结果与之前是完全相同的。

6.5.3 类型转换器

类型转换器用来在设计时间环境中执行格式化和解析。它们也能够解析XAML（可扩展应用程序标记语言）文档中的一些值——Windows Presentation Foundation和Workflow Foundation中会使用。

在.NET Framework中有超过100种类型转换器，用于处理颜色、图像和URL等数据。相反，格式提供者只针对一些简单的值类型实现。

类型转换器通常会采用许多方式来解析数据，而不需要提示。例如，在Visual Studio的一个ASP.NET应用程序中，如果在属性窗口中通过输入"Beige"来设置BackColor，那么Color的类型转换器会判断引用的是一个颜色名称，而不是RGB字符或系统颜色值。这种灵活性使类型转换器在设计器和XAML

文档以外的环境中很有用。

所有类型转换器的子类TypeConverter位于System.ComponentModel中。如果要获得一个TypeConverter，我们需要调用TypeDescriptor.GetConverter。下面的例子获得了一个包含Color类型（位于System.Drawing. dll的System.Drawing命名空间）的TypeConverter：

```
TypeConverter cc = TypeDescriptor.GetConverter (typeof (Color));
```

TypeConverter的其他方法还包括ConvertToString和ConvertFromString。我们可以按以下方式调用这两个方法：

```
Color beige = (Color) cc.ConvertFromString ("Beige");
Color purple = (Color) cc.ConvertFromString ("#800080");
Color window = (Color) cc.ConvertFromString ("Window");
```

按照惯例，类型转换器的名称以Converter结尾，并且通常与它们所转换类型位于同一个命名空间中。类型是通过一个TypeConverterAttribute与转换器发生联系的，允许设计器自动获得对应的转换器。

类型转换器也提供了一些设计时间功能，如为设计器的下拉列表或代码序列化提示生成标准值清单。

6.5.4 BitConverter

大多数基本类型都可以通过调用BitConverter.GetBytes转换为字节数组：

```
foreach (byte b in BitConverter.GetBytes (3.5))
  Console.Write (b + " ");                    // 0 0 0 0 0 0 12 64
```

BitConverter也提供了其他类型转换的方法，如ToDouble。

BitConverter不支持decimal和DateTime(Offset)类型。然而，可以通过调用decimal.GetBits将一个decimal转换为int数组。另外，decimal也提供了一个可以接受int数组的构造方法。

对于DateTime，可以调用一个实例的ToBinary方法，它会返回一个long，然后可以使用BitConverter进行转换。静态的DateTime.FromBinary方法可以执行相反的操作。

6.6 全球化

应用程序的国际化包括两个方面：*全球化*和*本地化*。

*全球化*注重于三个任务（重要性由大到小）：

1. 保证程序在其他文化环境中运行时不会出错。
2. 采用一种本地文化的格式化规则，例如日期显示。
3. 设计程序，使之能够从将来可能编写和部署的附属程序集读取与文化相关的数据和字符串。

*本地化*表示为特定文化编写附属程序集以结束最终任务。这可以在程序编写完成之后进行——我们将在第17章的“资源与附属程序集”中进行详细介绍。

.NET Framework默认能够通过应用与特定文化规则来完成第二个任务。我们已经知道如何调用ToString来处理一个采用本地格式化规则的DateTime或数字。可是，在执行第一个任务时很容易出

现错误，并导致程序中断，因为希望日期或数字按照一个假定文化设置进行格式化。正如我们之前介绍的，解决的方法是在格式化和解析时指定一个文化（如不变文化），或者使用与文化无关的方法，如XmlConvert方法。

6.6.1 全球化清单

我们已经介绍了本章的一些重要内容。下面是需要完成的基本工作的总结：

- 认识Unicode和文本编码（见本章的“文本编码与Unicode”）
- 要注意字符和字符串的一些方法是与文化相关的，如ToUpper和ToLower，除非希望区分不同文化，否则要使用ToUpperInvariant/ToLowerInvariant。
- DateTime和DateTimeOffsets的一些与文化无关的格式化和解析机制，如ToString("o")和XmlConvert。
- 除非默认使用本地文化，否则，在格式化/解析数字时指定一个文化。

6.6.2 测试

可以通过重新指定Thread的CurrentCulture属性（在System.Threading中）再次测试不同的文化。下面的代码将当前文化修改为Turkey（土耳其）：

```
Thread.CurrentThread.CurrentCulture = CultureInfo.GetCultureInfo ("tr-TR");
```

Turkey是非常好的测试用例，因为：

- "i".ToUpper()!= "I"和"I".ToLower()!= "i"。
- 日期格式为日/月/年，带有句号分隔符。
- 小数点符号是逗号，而不是句号。

还可以通过在Windows控制面板中修改数字和日期格式设置来进行测试：控制面板会修改默认文化设置（CultureInfo.CurrentCulture）。

CultureInfo.GetCultures()会返回一个包含所有可用文化的数组。

提示：Thread和CultureInfo也支持CurrentUICulture属性，它是与本地化关系更密切的属性，我们将在第18章进行介绍。

6.7 操作数字

6.7.1 转换

我们在前面的章节中已经介绍了数值转换，表6-7总结了所有可能的转换。

表6-7：数值转换总结

任务	函数	示例
解析十进制数	Parse	double d = double.Parse ("3.5");
	TryParse	int i; bool ok = int.TryParse ("3", out i);
解析二、八、十六进制数	Convert.ToIntegral	int i = Convert.ToInt32 ("1E", 16);
格式化为十六进制	ToString ("X")	string hex = 45.ToString ("X");
无值丢失的数值转换	隐式转换	int i = 23; double d = d;
截断值的数值转换	显式转换	double d = 23.5; int i = (int) d;
圆整值的数值转换	Convert.ToIntegral	double d = 23.5; int i = Convert.ToInt32 (d);

6.7.2 Math

表6-8列出了静态Math类的成员。三角函数接受double类型的参数；其他方法则经过重载后支持所有数值类型，如Max。Math类也定义了数学常量E(*e*)和PI。

表6-8：静态Math类的方法

类别	方法
圆整	Round、Truncate、Floor、Ceiling
最大值/最小值	Max、Min
绝对值和符号	Abs、Sign
平方根	Sqrt
幂运算	Pow、Exp
对数运算	Log, Log10
三角函数	Sin, Cos, Tan Sinh, Cosh, Tanh Asin, Acos, Atan

Round方法能够指定圆整的小数位数以及如何处理中间值（远离0，或者使用银行的圆整方式）。Floor和Ceiling会圆整到最接近的整数：Floor总是向下圆整，而Ceiling则总是向上圆整——即使是负数。

Max和Min只接受两个参数。如果是一个数组或数字序列，那么要使用System.Linq.Enumerable中的Max和Min扩展函数。

6.7.3 BigInteger

BigInteger结构体是.NET Framework 4.0新增的特殊数值类型。它位于*System.Numerics.dll*中新的System.Numerics命名空间，可以用于表示一个任意大的整数而不会丢失精度。

C#并不提供BigInteger的原生支持，所以无法表示BigInteger值。然而，可以从任意整数类型隐式地转换到BigInteger。例如：

```
BigInteger twentyFive = 25;  // 从整型隐式转换而来
```

为了表示更大的数字，如天文数字（10^{100}），可以使用BigInteger的一个静态方法，如Pow（提高幂次）。

```
BigInteger googol = BigInteger.Pow (10, 100);
```

此外，还可以Parse一个字符串：

```
BigInteger googol = BigInteger.Parse ("1".PadRight (100, '0'));
```

调用ToString()可以打印所有数字：

```
Console.WriteLine (googol.ToString()); // 10000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
```

可以将一个BigInteger隐式地转换为标准数值类型，也可以显式地进行反向转换。例如：

```
double g2 = (double)googol;           // 显式转换
BigInteger g3 = (BigInteger) g2;      // 显式转换
Console.WriteLine (g3);
```

这个例子的输出结果演示了精度丢失情况：

```
9999999999999999673361688041166912...
```

BigInteger重载了所有的算术运算符，以及比较、等式、求模（%）和负值运算符。

也可以从一个字节数组创建一个BigInteger。下面的代码生成一个适合加密的32字节的随机数，然后将它赋值给一个BigInteger：

```
// 使用了System.Security.Cryptography命名空间：
RandomNumberGenerator rand = RandomNumberGenerator.Create();
byte[] bytes = new byte [32];
rand.GetBytes (bytes);
var bigRandomNumber = new BigInteger (bytes);        // 转换为BigInteger
```

将一个数字存储到一个BigInteger中而不是字节数组的优点是可以获得值类型的语义。调用ToByteArray可以将一个BigInteger转换回字节数组。

6.7.4 Complex

Complex结构体是Framework 4.0新增的另一个特殊数值类型，用来表示用double类型的实数和虚数构成的复数。Complex位于*程序集*（与BigInteger相同）。

要使用Complex，我们需要实例化这个结构体，指定实数和虚数值：

```
var c1 = new Complex (2, 3.5);
var c2 = new Complex (3, 0);
```

标准的数值类型可以有很多种Complex的隐式转换方式。

Complex结构体具有实数和虚数值的属性，以及阶和量级：

```
Console.WriteLine (c1.Real);          // 2
Console.WriteLine (c1.Imaginary);     // 3.5
Console.WriteLine (c1.Phase);         // 1.05165021254837
Console.WriteLine (c1.Magnitude);     // 4.03112887414927
```

还可以通过指定量级和阶来创建复数：

```
Complex c3 = Complex.FromPolarCoordinates (1.3, 5);
```

复数也重载了标准的算术操作符：

```
Console.WriteLine (c1 + c2);          // (5, 3.5)
Console.WriteLine (c1 * c2);          // (6, 10.5)
```

Complex结构体具有一些支持更高级功能的静态方法，其中包括：

- 三角函数（Sin、Asin、Sinh、Tan等）
- 取对数与求幂
- 共轭

6.7.5 Random

Random类能够生成一个随机byte、integer或double类型的伪随机数序列。

要使用Random，首先要实例化，可选择提供一个种子来实例化随机数序列。使用相同的种子一定会产生相同序列的数字，当希望有可再现性时，是非常有用的：

```
Random r1 = new Random (1);
Random r2 = new Random (1);
Console.WriteLine (r1.Next (100) + ", " + r1.Next (100));     // 24, 11
Console.WriteLine (r2.Next (100) + ", " + r2.Next (100));     // 24, 11
```

如果不希望可再现性，那么可以不使用种子来创建Random而是使用当前系统时间来创建。

警告： 因为系统时钟只有有限的粒度，创建时间间隔很小（一般是10ms内）的两个Random将会产生相同序列的值。常用的方法是每次需要一个随机数时才实例化一个新的Random对象，而不是重用同一个对象。

一个很好的模式是声明一个静态Random实例。然而，在多线程情况下，可能出现问题，因为Random对象并不是线程安全的。我们将在第22章中介绍“线程本地存储”工作区。

调用Next(n)可以生成一个0至n-1之间的随机整数。NextDouble可以生成一个0至1之间的随机double数值。NextBytes会用随机数填充一个字节数组。

Random对于高安全性要求的应用程序而言随机性还不够高，如加密。为此，.NET Framework提供了

一种密码加强的随机数生成器，它位于System.Security.Cryptography命名空间。使用方式如下：

```
var rand = System.Security.Cryptography.RandomNumberGenerator.Create();
byte[] bytes = new byte [32];
rand.GetBytes (bytes);        // 给字节数组填充随机数
```

这种方法的缺点是不够灵活：填充字节数组是唯一获得随机数的方法。要获得一个整数，必须使用BitConverter：

```
byte[] bytes = new byte [4];
rand.GetBytes (bytes);
int i = BitConverter.ToInt32 (bytes, 0);
```

6.8 枚举类型

在第3章，我们介绍了C#的枚举类型，并说明了如何组合成员、判断相等、使用逻辑运算符和执行转换。Framework通过System.Enum类型扩展了C#的枚举类型支持。这种类型有两种作用：支持所有枚举类型的类型统一和定义静态的实用方法。

- 为所有enum类型提供类型统一化
- 定义静态实用方法

*类型统一*表示可以将任何枚举成员隐式转换为一个System.Enum实例：

```
enum Nut  { Walnut, Hazelnut, Macadamia }
enum Size { Small, Medium, Large }

static void Main()
{
  Display (Nut.Macadamia);   // Nut.Macadamia
  Display (Size.Large);      // Size.Large
}

static void Display (Enum value)
{
  Console.WriteLine (value.GetType().Name + "." + value.ToString());
}
```

System.Enum的静态实用方法主要是与转换和获取成员清单相关。

6.8.1 枚举值转换

有三种形式可以表示枚举值：

- 作为一个枚举成员
- 作为它的基本整型值
- 作为一个字符串

在这一节，我们将介绍如何采用各种方法进行转换。

1. 枚举型转换为整型

回顾一下枚举成员及其整型值的显式转换方法。如果在编译时已知枚举类型，那么显式转换是正确的

方法。

```
[Flags] public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }
...
int i = (int) BorderSides.Top;          // i == 4
BorderSides side = (BorderSides) i;   // side == BorderSides.Top
```

可以用相同的方法将一个System.Enum实例转换为它的整型。这个方法的关键在于先将它转换为一个对象，然后再转换为整型：

```
static int GetIntegralValue (Enum anyEnum)
{
  return (int) (object) anyEnum;
}
```

这基于已知所处理的是整型：如果传入一个long整型枚举值，那么上述方法就会出错。要编写一个方法来支持所有整型的枚举值，可以采用以下三种方法。第一种方法是调用Convert.ToDecimal：

```
static decimal GetAnyIntegralValue (Enum anyEnum)
{
  return Convert.ToDecimal (anyEnum);
}
```

使用原因是每一种整型（包括ulong）都可以转换为十进制数而不会丢失值。第二种方法是调用Enum.GetUnderlyingType以获得枚举值的整数类型，然后再调用Convert.ChangeType：

```
static object GetBoxedIntegralValue (Enum anyEnum)
{
  Type integralType = Enum.GetUnderlyingType (anyEnum.GetType());
  return Convert.ChangeType (anyEnum, integralType);
}
```

这个方法会保护原始的整数类型，如下所示：

```
object result = GetBoxedIntegralValue (BorderSides.Top);
Console.WriteLine (result);                              // 4
Console.WriteLine (result.GetType());                    // System.Int32
```

提示： GetBoxedIntegralType方法实际上没有执行值转换；相反，它将同一个值重新装箱在另一种类型中。它将一个*枚举类型*的整型值转换成一个*整数类型*的整型值。我们将在下一节“枚举类型工作方式”中进行介绍。

第三种方法是通过指定"d"或"D"格式字符串来调用Format或ToString。这会将枚举变量的整型值转换成一个字符串，这在编写自定义序列化格式时是很有用的：

```
static string GetIntegralValueAsString (Enum anyEnum)
{
  return anyEnum.ToString ("D");      // 返回类似于"4"的数字
}
```

2. 整型转换为枚举型

Enum.ToObject能够将一个整型值转换为一个指定类型的enum实例：

```
object bs = Enum.ToObject (typeof (BorderSides), 3);
Console.WriteLine (bs);                               // Left, Right
```

这个方法等同于：

```
BorderSides bs = (BorderSides) 3;
```

ToObject已经重载，可以接受所有的整数类型和对象（后者支持任何装箱的整数类型）。

3. 字符串转换

要将一个enum转换为一个字符串，可以调用静态的Enum.Format方法或调用实例的ToString。每一种方法都接受一个格式字符串，其中"G"表示默认的格式化行为，"D"表示将实际的整型值作为字符串使用，"X"表示对十六进制数执行相同的操作，而"F"表示格式化一个不带Flags属性的enum的组合成员。

Enum.Parse可以将一个字符串转换为一个enum。它接受enum类型和一个包含多个成员的字符串：

```
BorderSides leftRight = (BorderSides) Enum.Parse (typeof (BorderSides),
                                                  "Left, Right");
```

可以选择传递第三个参数来执行区分大小写的解析。如果成员不存在，那么抛出一个ArgumentException。

6.8.2 列举枚举值

Enum.GetValues返回一个包含某特定enum类型的所有成员：

```
foreach (Enum value in Enum.GetValues (typeof (BorderSides)))
  Console.WriteLine (value);
```

它也包含一些复合成员，如LeftRight = Left|Right。

Enum.GetNames执行相同的操作，但是返回的是一个字符串数组。

提示： 在内部，CLR通过反射enum类型的字段实现GetValues和GetNames，其结果会被缓存以提高效率。

6.8.3 枚举类型工作方式

枚举类型的语义很大程度上是由编译器决定的。在CLR中，enum实例（未拆箱）与它实际的整型值在运行时是没有任何区别的。而且，在CLR中定义的enum仅仅是System.Enum的子类型，它的每个成员都是静态的整型域。

这个方法的缺点是枚举值可以支持静态方式，但是不具备强类型安全性。我们在第3章介绍了这样一个例子：

```
public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }
...
BorderSides b = BorderSides.Left;
b += 1234;                          // 没有错误！
```

当编译器无法执行验证时（如上例），就会抛出一个运行时异常。

我们所谓的enum实例与其整型值在运行时没有任何区别，但是在下面这个例子中可能有些不一样：

```
[Flags] public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }
...
Console.WriteLine (BorderSides.Right.ToString());          // Right
Console.WriteLine (BorderSides.Right.GetType().Name);      // BorderSides
```

由enum实例的运行时特征，可能认为它会打印出2和Int32。产生这种行为的原因更多是由一些编译时问题决定的。C#会在调用enum实例的虚方法之前对它进行显式装箱，如ToString或GetType。而且，当enum实例被装箱后，它会获得一个引用其enum类型的封装。

6.9 元组

Framework 4.0提供了一组新的泛型类来保存不同类型的元素集，称为*元组*：

```
public class Tuple <T1>
public class Tuple <T1, T2>
public class Tuple <T1, T2, T3>
public class Tuple <T1, T2, T3, T4>
public class Tuple <T1, T2, T3, T4, T5>
public class Tuple <T1, T2, T3, T4, T5, T6>
public class Tuple <T1, T2, T3, T4, T5, T6, T7>
public class Tuple <T1, T2, T3, T4, T5, T6, T7, TRest>
```

每种元组都有名为Item1、Item2等的只读属性，分别对应一种类型参数。

可以通过它的构造方法实例化一个元组：

```
var t = new Tuple<int,string> (123, "Hello");
```

或者通过静态帮助方法Tuple.Create：

```
Tuple<int,string> t = Tuple.Create (123, "Hello");
```

后者使用的是泛型推断方法。可以将这种方法与隐式类型转换结合使用：

```
var t = Tuple.Create (123, "Hello");
```

然后，可以按照以下方法访问这些属性（注意每一个方法都是静态输入的）：

```
Console.WriteLine (t.Item1 * 2);          // 246
Console.WriteLine (t.Item2.ToUpper());    // HELLO
```

元组可以很方便地用来实现从一个方法返回多个值或者创建值对集合（我们将在下面的章节介绍集合）。

元组的替代方法是使用对象数组。然而，这种方法会影响静态类型安全性，增加了值类型的装箱/开箱开销，并且需要作一些编译器无法验证的复杂转换。

```
object[] items = { 123, "Hello" };
Console.WriteLine ( ((int) items[0]) * 2 );          // 246
Console.WriteLine ( ((string) items[1]).ToUpper() ); // HELLO
```

元组比较

元组是一些类（也就是引用类型）。所以，使用等式运算符比较两个不同的实例会返回false。然而，重载的Equals方法可以用来比较每个元素：

```
var t1 = Tuple.Create (123, "Hello");
var t2 = Tuple.Create (123, "Hello");
Console.WriteLine (t1 == t2);          // False
Console.WriteLine (t1.Equals (t2));    // True
```

也可以利用实现了IStructuralEquatable的元组传入一个自定义等式运算符。我们将在本章后面介绍等值比较和顺序比较。

6.10 Guid结构体

Guid结构体表示一个全局唯一标识符：一个随机生成的16位值，几乎可以肯定具有唯一性。Guid在应用程序和数据库中通常用作各种排序的键。Guid唯一的数量为2^{128}或3.4×10^{38}。

我们可以调用静态的Guid.NewGuid方法创建一个新的随机Guid：

```
Guid g = Guid.NewGuid ();
Console.WriteLine (g.ToString());     // 0d57629c-7d6e-4847-97cb-9e2fc25083fe
```

我们可以使用构造方法实例化一个已有值。其中最常用的两个构造方法是：

```
public Guid (byte[] b);     // 接收一个16字节数组
public Guid (string g);     // 接收一个格式化字符串
```

当表示为一个字符串时，Guid会格式化为一个32位十六进制数，并且可以选择在第8、12、16和20位数字后添加连字符。整个字符串可以选择封装在方括号或花括号中：

```
Guid g1 = new Guid ("{0d57629c-7d6e-4847-97cb-9e2fc25083fe}");
Guid g2 = new Guid ("0d57629c7d6e484797cb9e2fc25083fe");
Console.WriteLine (g1 == g2);          // True
```

作为一个结构体，Guid支持值类型语法，因此，前面的例子也可以使用等式运算符。

ToByteArray方法可以将一个Guid转换为一个字节数组。

静态的Guid.Empty属性会返回一个空的Guid（全为零），通常用来替换null。

6.11 等值比较

到现在为止，我们假设==和!=运算符都是等式运算符。然而，等式问题是复杂且模糊的，有时候我们需要借助其他的方法和接口。这一节将介绍C#和.NET标准的等值比较方法，主要关注两个问题：

- ==和!=在什么时候能够满足或不满足等值比较的需要，以及有哪些替代方法？
- 可以如何以及应该在何时自定义一个类型的等值比较逻辑？

但是在详细介绍等值比较方法及如何自定义之前，我们首先要了解值相等与引用相等的基本概念。

6.11.1 值相等与引用相等

相等有两种类型：

值相等

两个值在某种意义上是相等的。

引用相等

两个引用指向完全相同的对象。

默认情况下：

- 值类型采用的是*值相等*。
- 引用类型采用的是*引用相等*。

事实上，值类型只能使用值相等形式进行比较（除非已装箱）。最简单的值相等例子就是比较两个数字：

```
int x = 5, y = 5;
Console.WriteLine (x == y);  // True（通过值比较得出）
```

更复杂的例子就是比较两个DateTimeOffset结构体。下面的输出结果是True，因为两个DateTimeOffset指向同一个时间点，所以它们被认为是相等的：

```
var dt1 = new DateTimeOffset (2010, 1, 1, 1, 1, 1, TimeSpan.FromHours(8));
var dt2 = new DateTimeOffset (2010, 1, 1, 2, 1, 1, TimeSpan.FromHours(9));
Console.WriteLine (dt1 == dt2);       // True
```

提示：DateTimeOffset是一个结构体，它的等值语义是比较复杂的。默认情况下，结构体有一种特殊形式的值相等称为*结构相等*，即当两个值的所有成员都相等时，它们就被认为是相等的。（可以创建一个结构体，然后调用它的Equals方法来验证这一点，后面将对此进行更详细的介绍。）

框架基础

引用类型默认是采用引用相等的比较形式。在下面的例子中，尽管它们的对象具有相同的内容，但f1和f2是不相等的：

```
class Foo { public int X; }
...
Foo f1 = new Foo { X = 5 };
Foo f2 = new Foo { X = 5 };
Console.WriteLine (f1 == f2);                  // False
```

相反，f3和f1是相等的，因为它们引用了同一个对象：

```
Foo f3 = f1;
Console.WriteLine (f1 == f3);                  // True
```

我们将在后面的内容中介绍如何自定义引用类型以实现值相等。System命名空间的Uri类就是一种自定义值相等的例子：

```
Uri uri1 = new Uri ("http://www.linqpad.net");
Uri uri2 = new Uri ("http://www.linqpad.net");
Console.WriteLine (uri1 == uri2);              // True
```

6.11.2 标准等值比较方法

有三种标准方法可以实现等值比较：

- ==和!=运算符
- 对象的虚方法Equals
- IEquatable<T>接口

此外，我们还将在第7章讨论可插拔的方法和IStructuralEquatable接口。

1. ==和!=

我们已经看到了一些标准的==和!=运算符是如何执行相等/不等比较的例子。==和!=都是操作符，所以它们是可以静态解析的（实际上，它们是作为静态函数实现的）。所以，当使用==或!=时，C#会进行编译时判断，决定使用哪种类型来执行比较，并且不使用任何虚行为。这通常是可以接受的。在下面的例子中，编译器会将==强制绑定到int类型，因为x和y都是int类型：

```
int x = 5;
int y = 5;
Console.WriteLine (x == y);  // True
```

但是在下一个例子中，编译器将==运算符绑定为object类型：

```
object x = 5;
object y = 5;
Console.WriteLine (x == y);  // False
```

因为object是一个类（也就是引用类型），object的==运算符使用*引用相等*的方式来比较x和y。其结果是false，因为x和y分别指向堆中的不同装箱对象。

2. 虚方法Object.Equals

为了正确比较上一个例子的x和y，我们可以使用虚方法Equals。Equals在System.Object中定义，所以所有类型都支持这个方法：

```
object x = 5;
object y = 5;
Console.WriteLine (x.Equals (y));    // True
```

Equals是在运行时根据对象的实际类型解析的。在这个例子中，它会调用Int32的Equals方法，在操作对象上进行值相等比较，返回true；对于结构体，Equals会调用每个字段的Equals执行结构比较。

> **为什么要这样复杂？**
>
> 你可能会疑惑为什么C#的设计人员不将==设计成虚函数以避免这个问题呢，这样它在功能上就等同于Equals。有以下三个原因：
>
> - 如果第一个操作对象是null，那么Equals会抛出NullReferenceException；而静态运算符则不会。

- 因为==运算符是静态解析的，所以它执行速度非常快。这意味着可以轻松编写出计算密集型的代码而不需要学习其他语言，如C++。
- 有时候分别使用==和Equals实现不同的等值定义是非常有用的。我们将在本节后面的内容介绍这一点。

因此，Equals很适合用来比较两个未知类型的对象。下面的方法用于比较两个任意类型的对象：

```
public static bool AreEqual (object obj1, object obj2)
{
  return obj1.Equals (obj2);
}
```

然而这个例子在一种情况下会出错，即如果第一个参数为null，那么它会抛出一个NullReference Exception。下面是修改方法：

```
public static bool AreEqual (object obj1, object obj2)
{
  if (obj1 == null) return obj2 == null;
  return obj1.Equals (obj2);
}
```

3. 静态方法object.Equals

object类提供了一个静态的帮助方法，它能够实现前一个例子中的AreEqual操作。它的名称是Equals，与虚方法相同，但是不会有冲突，因为它接受两个参数：

```
public static bool Equals (object objA, object objB)
```

如果在处理编译时未知类型对象，这是一种能够避免null值异常的等值比较算法。例如：

```
object x = 3, y = 3;
Console.WriteLine (object.Equals (x, y));    // True
x = null;
Console.WriteLine (object.Equals (x, y));    // False
y = null;
Console.WriteLine (object.Equals (x, y));    // True
```

这在编写泛型类型时是很有用的。如果将object.Equals替换成==或!=运算符，下面的代码就无法正常编译：

```
class Test <T>
{
  T _value;
  public void SetValue (T newValue)
  {
    if (!object.Equals (newValue, _value))
    {
      _value = newValue;
      OnValueChanged();
    }
  }
  protected virtual void OnValueChanged() { ... }
}
```

这里是禁止使用运算符的，因为编译器无法绑定一个未知类型的静态方法。

提示： 实现这个比较的一个更复杂方法是使用EqualityComparer<T>类。它的优点是不需要装箱：

```
if (!EqualityComparer<T>.Default.Equals (newValue, _value))
```

我们将在第7章的“插入等式和顺序”中详细地讨论EqualityComparer<T>。

---

4. 静态方法object.ReferenceEquals

有时候，需要进行引用等值比较。静态方法object.ReferenceEquals可以实现这种比较：

```
class Widget { ... }

class Test
{
  static void Main()
  {
    Widget w1 = new Widget();
    Widget w2 = new Widget();
    Console.WriteLine (object.ReferenceEquals (w1, w2));          // False
  }
}
```

可能希望进行这种比较，因为这样Widget就可以重载虚方法Equals，这时w1.Equals(w2)就会返回true。而且，Widget也可以重载==运算符，这样w1==w2也会返回true。在这两种情况中，调用object.ReferenceEquals都能够保证正常的引用相等的语义。

提示： 另一种采用引用等值比较的方法是将值转换为object，然后再使用==运算符。

---

5. IEquatable<T>接口

调用object.Equals的结果是强制对值类型执行装箱。这在对性能高度敏感的情况下是不太适合的，因为装箱操作相对于实际比较操作的开销还要高。C# 2.0引入了一个解决方法，那就是使用IEquatable<T>接口：

```
public interface IEquatable<T>
{
  bool Equals (T other);
}
```

关键在于实现IEquatable<T>所返回的结果与调用object的虚方法Equals是一样的，但是执行速度会更快。大多数.NET基本类型都实现了IEquatable<T>。可以在泛型中使用IEquatable<T>作为一个约束：

```
class Test<T> where T : IEquatable<T>
{
  public bool IsEqual (T a, T b)
  {
    return a.Equals (b);      // 不需要装箱为泛型T
  }
}
```

如果我们删除泛型约束，那么这个类仍然可以编译，但是a.Equals(b)会被绑定到速度较慢的object.Equals（如果T是一个值类型，那么速度会较慢一些）。

6. Equals和== 不等价

我们之前提到有时候使用==和Equals来实现不同的等值定义是很有用的。例如：

```
double x = double.NaN;
Console.WriteLine (x == x);            // False
Console.WriteLine (x.Equals (x));      // True
```

double类型的==运算符规定了NaN不等于任何东西——即使是另一个NaN。这从数学角度而言是很自然的，而且它反映了实际的CPU行为。然而，Equals必须支持自反等值；换句话说：

x.Equals (x) *必须总是*返回true。

依赖于Equals的集合和字典也采用这种方法；否则，它们可能无法找到之前存储的数据项。

使Equals和==分别支持不同的等值定义实际上很少用于值类型。这种方式一般用于引用类型，并且在开发人员自定义Equals时使用，从而使Equals执行值相等比较，而==执行（默认的）引用相等比较。StringBuilder类就是采用这种方式：

```
var sb1 = new StringBuilder ("foo");
var sb2 = new StringBuilder ("foo");
Console.WriteLine (sb1 == sb2);              // False（引用相等）
Console.WriteLine (sb1.Equals (sb2));        // True（值相等）
```

接下来，让我们学习如何自定义等值。

6.11.3 等值与自定义类型

我们知道默认的等值比较操作有：

- 值类型采用的是值相等
- 引用类型采用的是引用相等

此外：

- 结构体的Equals方法默认采用的是结构值相等（例如，它会比较结构体中的每个字段）。

有时创建一个类型时重载这个行为是很有用的。有以下两种情况我们需要这样做：

- 修改相等的语义
- 提高结构体的等值比较的执行速度

1. 修改相等的语义

当==和Equals默认行为不符合要求的类型，并且这种行为一般人难以想象时，修改相等的语义是很有用的。一个例子是DateTimeOffset，这是一个具有两个私有字段的结构体：UTC DateTime和整数偏移值。编写这个类型时可能希望保证等值比较只考虑UTC DateTime字段，而不考虑偏移字段。另一个例子是支持NaN值的数值类型，如float和double。实现这些类型时希望保证等值比较支持NaN比较逻辑。

对于类而言，有时将值相等作为默认的比较方法会比使用引用相等更自然。这通常指的是具有一个简单数据类型的小型类，如System.Uri或System.String。

2. 提高结构体的等值比较

结构体的默认*结构等值*比较算法相对较慢。通过重载Equals来实现这个过程可以将性能提高20%。重载==运算符和实现IEquatable<T>接口可以实现等值比较的拆箱，并且同样能够将比较速度提高20%。

提示： 重载引用类型的等值语义并不能提高性能。引用等值比较的默认算法已经非常快速，因为它只比较两个32位或64位引用。

实际上还有另一个特殊原因需要自定义等值，那就是为了改进结构体的散列算法，以提高散列表的性能。这是因为等值比较和散列在内部是相关联的。我们将在后面章节介绍散列。

3. 如何重载等值语义

下面是操作步骤总结：

（1）重载GetHashCode()和Equals()。

（2）（可选）重载!=和==。

（3）（可选）实现IEquatable<T>。

4. 重载GetHashCode

在System.Object（只有很少成员）中定义一个具有特殊用途且使用范围不广的方法是不常用的。GetHashCode是Object的一个虚方法，它就属于这一类方法，存在的主要目的只是为了服务下面两种类型：

```
System.Collections.Hashtable
System.Collections.Generic.Dictionary<TKey,TValue>
```

这些是*散列表*即一些使用键来存储和查询元素的集合。散列表支持一个基于键的高效分配元素的特殊方法。它要求每一个键都是Int32数字，或者称为散列码。散列码不需要保持唯一，但是为了实现最佳散列性能，它要尽可能地保持差异性。在System.Object中定义的GetHashCode对于散列表而言非常重要，所以每一种类型都具有一个散列码。

提示： 我们将在第7章的“字典”中详细介绍散列表。

引用类型和值类型都只有默认的GetHashCode实现，这意味着不需要重载这个方法——除非重载了Equals。（反之亦然，如果重载了GetHashCode，那么也必须重载Equals）。

下面是重载object.GetHashCode的其他规则：

- 它必须为Equals方法都返回true的两个对象返回相同的值，因此，GetHashCode和Equals必须同时重载。
- 它不能抛出异常。

- 如果重复调用相同对象，必须返回相同的值（除非对象*改变*）。

为了实现最佳性能的散列表，所编写的GetHashCode应该尽量避免为两个不同的值返回相同的散列码。这是在结构体中重载Equals和GetHashCode的第三个原因，其目的是为了实现更高效的散列算法。结构体的默认散列方法只是在每个字段上执行按位异或操作，通常会比编写的算法产生更多的重复码。

相反，类的默认GetHashCode实现基于一个内部对象标识，它在CLR当前实现中的每一个实例上都是唯一的。

警告：如果一个对象的散列码在它作为键添加到字典后发生变化，那么这个对象在字典中将不可访问。可以在不可变字段中计算散列码以避免出现这个问题。

稍后我们将举例说明如何重载GetHashCode。

5. 重载Equals

object.Equals的执行逻辑如下：

- 对象不能是null（除非它是可空类型）。
- 相等是*自反性的*（对象与其本身相等）。
- 相等是*可交换的*（如果a.Equals(b)，那么b.Equals(a)）。
- 相等是*可传递的*（如果a.Equals(b)且b.Equals(c)，那么a.Equals(c)）。
- 等值操作是可重复且可靠的（它们不会抛出异常）。

6. 重载==和!=

除了重载Equals，还可以选择重载相等和不等运算符。这种重载几乎都发生在结构体上，否则==和!=运算符无法正确判断类型。

对于类，有两种方法可以处理：

- 保留==和!=，这样它们会应用引用相等。
- 重载Equals同时重载==和!=。

第一种方法通常用于自定义类型，特别是可变类型。它能够保证类型是符合要求的，即==和!=应该对引用类型应用引用相等方法，可以避免产生歧义。我们在前面看到过一个例子：

```
var sb1 = new StringBuilder ("foo");
var sb2 = new StringBuilder ("foo");
Console.WriteLine (sb1 == sb2);              // False（引用比较）
Console.WriteLine (sb1.Equals (sb2));        // True（值比较）
```

第二个方法对于不需要使用引用相等的类型是很有意义的。一般是不可变类型，如string和System.Uri类，并且它们有时是很好的候选结构体。

提示：虽然我们可以重载!=，使之变成相反的操作（==），但是在实际中这几乎是不可能发生的，除非在比较float.NaN时。

7. 实现IEquatable<T>

为了保持完整性，在重载Equals时，最好也要实现IEquatable<T>，其结果应该总是与被重载对象的Equals方法保持一致。如果自己编写Equals方法实现，那么实现IEquatable<T>并没有任何的程序开销，如下面的例子所示。

8. 示例：Area结构体

假设我们需要用一个结构体来表示面积，宽度和高度应该是可交换的。换句话说，5×10等于10×5。这种类型很适合用在排列矩形图形的算法中。

下面是完整的代码：

```
public struct Area : IEquatable <Area>
{
  public readonly int Measure1;
  public readonly int Measure2;

  public Area (int m1, int m2)
  {
    Measure1 = Math.Min (m1, m2);
    Measure2 = Math.Max (m1, m2);
  }

    public override bool Equals (object other)
  {
    if (!(other is Area)) return false;
    return Equals ((Area) other);              // 调用下面的方法
  }

  public bool Equals (Area other)              // 实现IEquatable<Area>
  {
    return Measure1 == other.Measure1 && Measure2 == other.Measure2;
  }

  public override int GetHashCode()
  {
    return Measure2 * 31 + Measure1;           // 31 = 某个素数
  }

  public static bool operator == (Area a1, Area a2)
  {
    return a1.Equals (a2);
  }

  public static bool operator != (Area a1, Area a2)
  {
    return !a1.Equals (a2);
  }
}
```

提示：下面是另一种实现Equals的方法，即使用可空类型值：

```
Area? otherArea = other as Area?;
return otherArea.HasValue && Equals (otherArea.Value);
```

在实现GetHashCode时，在添加两个尺寸值之前，我们用一个较大的尺寸值乘以某个素数（忽略溢出），以提高唯一性的可能程序。当有两个以上的字段时，下面这个Josh Bloch推荐的模式能够在执行时得到很好的结果：

```
int hash = 17;                               // 17 = 某个素数
hash = hash * 31 + field1.GetHashCode();     // 31 = 另一个素数
hash = hash * 31 + field2.GetHashCode();
hash = hash * 31 + field3.GetHashCode();
...
return hash;
```

（参见*http://albahari.com/hashprimes*，了解关于素数与散列码的信息。）

下面是Area结构体的应用示例：

```
Area a1 = new Area (5, 10);
Area a2 = new Area (10, 5);
Console.WriteLine (a1.Equals (a2));  // True
Console.WriteLine (a1 == a2);        // True
```

9. 可插入等值比较符

如果希望在一个类型上针对特殊情况实现不同的等值语义，那么可以使用可插入的IEqualityComparer。这在与标准的集合类一起使用时特别有用，我们将在第7章的“等值和顺序插入”中介绍这个方法。

6.12 顺序比较

除了标准等值协议，C#和.NET还定义了用于确定对象之间相对顺序的协议。基本的协议包括：

- IComparable接口（IComparable和IComparable<T>）
- >和<运算符

IComparable接口可用于普通的排序算法。在下面的例子中，静态的Array.Sort方法之所以有效是因为System.String实现了IComparable接口：

```
string[] colors = { "Green", "Red", "Blue" };
Array.Sort (colors);
foreach (string c in colors) Console.Write (c + " ");        // Blue Green Red
```

<和>操作符比较特殊，它们大多数情况用于比较数字类型。因为它们是静态解析的，所以可以转换为高效的字节码，适用于一些密集型算法。

.NET Framework也通过IComparer接口实现了可插入的排序协议，我们将在第7章的最后一节介绍这个接口。

6.12.1 IComparable

IComparable接口是按如下方式定义的：

```
public interface IComparable       { int CompareTo (object other); }
public interface IComparable<in T> { int CompareTo (T other); }
```

这两个接口实现了相同的功能。对于值类型，泛型安全的接口执行速度比非泛型接口更快。在这两个接口中，CompareTo方法按如下方式执行：

- 如果a在b之后，那么a.CompareTo(b)返回一个正数。
- 如果a与b位置相同，那么a.CompareTo(b)返回0。
- 如果a在b之前，那么a.CompareTo(b)返回一个负数。

例如：

```
Console.WriteLine ("Beck".CompareTo ("Anne"));       // 1
Console.WriteLine ("Beck".CompareTo ("Beck"));       // 0
Console.WriteLine ("Beck".CompareTo ("Chris"));      // -1
```

大多数基本类型都实现了这两个IComparable接口。在编写自定义类型时，有时也需要实现这些接口。稍后我们将介绍一个这样的例子。

IComparable与Equals

假设一种类型重载了Equals，并且实现了IComparable接口。当Equals返回true时，CompareTo返回0。但是，也有例外的情况：

当Equals返回false，CompareTo可以返回任意结果。

换句话说，等值比比较严格，但是反之不然（违反这一点，排序算法就会出错）。所以，CompareTo结果可能是所有对象都相同，而Equals则得到"有一些对象与其他对象更接近！"

最好的例子就是System.String。String的Equals方法和==运算符采用顺序比较，它会比较每一个字符的Unicode值。然而，它的CompareTo方法使用一种较为宽泛的与文化无关的比较。例如，在大多数计算机中，字符串"ǖ"和"ǖ"应用Equals得到的结果是不同的，但是CompareTo会认为是相同的。

在第7章中，我们将讨论可插入排序协议IComparer，它允许在排序或实例化一个按顺序排列集合时指定其他的排序算法。自定义的IComparer可以进一步明确CompareTo和Equals的区别，例如，区分大小写的字符串比较在比较"A"和"a"时会返回0。然而，相反的规则也有效：CompareTo一定比Equals简单。

提示： 当在一个自定义类型中实现IComparable接口时，可以将下面一行代码添加到CompareTo开头来避免违反这个规则：

```
if (Equals (other)) return 0;
```

然后，它可以返回符合这个规则的任意值。

6.12.2 < 和 >

有些类型定义了<和>运算符。例如：

```
bool after2010 = DateTime.Now > new DateTime (2010, 1, 1);
```

实现过程中，<和>运算符在功能上与IComparable接口是一致的。这是在整个.NET Framework中都适用的标准方法。

在重载<和>后，同时实现IComparable接口，这也是一种标准方法，但是反之不成立。事实上，大多数实现了IComparable的.NET类型都没有重载<和>。与等值的处理方法不同的是，在等值中如果重载了Equals，一般也会重载==。

通常，<和>只有在以下情况中进行重载：

- 类型具有固定的"大于"和"小于"规定（对应于IComparable宽泛的"之前"和"之后"）。
- 只有唯一方法或环境能够执行这个比较。
- 比较结果在各种文化中保持不变。

System.String不符合最后一种情况：字符串比较的结果可能会由于语言的不同而不同。因此，字符串不支持<和>运算符，例如。

```
bool error = "Beck" > "Anne";              // 编译时错误
```

6.12.3 实现IComparable接口

在下面的结构体中，我们实现了用IComparable接口来表示一个音符，并且重载了<和>运算符。为了保持完整性，我们还重载了Equals/GetHashCode以及==与!=：

```
public struct Note : IComparable<Note>, IEquatable<Note>, IComparable
{
  int _semitonesFromA;
  public int SemitonesFromA { get { return _semitonesFromA; } }

  public Note (int semitonesFromA)
  {
    _semitonesFromA = semitonesFromA;
  }

  public int CompareTo (Note other)           // 泛型IComparable<T>
  {
    if (Equals (other)) return 0;             // 异常安全检测
    return _semitonesFromA.CompareTo (other._semitonesFromA);
  }

  int IComparable.CompareTo (object other)    // 非泛型IComparable
  {
    if (!(other is Note))
      throw new InvalidOperationException ("CompareTo: Not a Note");
    return CompareTo ((Note) other);
  }

  public static bool operator < (Note n1, Note n2)
  {
```

```
        return n1.CompareTo (n2) < 0;
      }
      public static bool operator > (Note n1, Note n2)
      {
        return n1.CompareTo (n2) > 0;
      }
      public bool Equals (Note other)          // 实现IEquatable<Note>
      {
        return _semitonesFromA == other._semitonesFromA;
      }
      public override bool Equals (object other)
      {
        if (!(other is Note)) return false;
        return Equals ((Note) other);
      }
      public override int GetHashCode()
      {
        return _semitonesFromA.GetHashCode();
      }
      public static bool operator == (Note n1, Note n2)
      {
        return n1.Equals (n2);
      }
      public static bool operator != (Note n1, Note n2)
      {
        return !(n1 == n2);
      }
    }
```

6.13 实用类

6.13.1 Console

静态的Console类能够处理基于控制台应用程序的标准输入/输出。在一个命令行（Console）应用程序中，输入是利用键盘通过Read、ReadKey和ReadLine得到的，而输出结果则通过Write和WriteLine显示到文本窗口上。可以通过属性WindowLeft、WindowTop、WindowHeight和WindowWidth控制窗口的位置和尺寸。也可以修改BackgroundColor和ForegroundColor属性，并且通过CursorLeft、CursorTop和CursorSize属性控制鼠标指针。

```
Console.WindowWidth = Console.LargestWindowWidth;
Console.ForegroundColor = ConsoleColor.Green;
Console.Write ("test... 50%");
Console.CursorLeft -= 3;
Console.Write ("90%");        // test... 90%
```

Write和WriteLine方法经过重载，也可以接受一个复合格式字符串（见本章“字符串和文本处理”的String.Format）。然而，这两个方法都不接受格式提供者，所以无法使用CultureInfo.CurrentCulture。（当然，解决方法是要显式地调用string.Format。）

Console.Out属性会返回一个TextWriter。将Console.Out传递给一个接受TextWriter的方法是实现诊断目的Console输出的最佳方法。

还可以通过SetIn和SetOut方法重定向Console的输入和输出流。

```
// 首先保存现有的输出写入流：
System.IO.TextWriter oldOut = Console.Out;

// 将控制台输出重定向到一个文件：
using (System.IO.TextWriter w = System.IO.File.CreateText("e:\\output.txt"))
{
  Console.SetOut (w);
  Console.WriteLine ("Hello world");
}

// 恢复标准的控制台输出
Console.SetOut (oldOut);

// 在记事本中打开output.txt：
System.Diagnostics.Process.Start ("e:\\output.txt");
```

在第15章中，我们将介绍流和文本输入器是如何工作的。

提示： 在Visual Studio Windows应用程序中，Console的输出会（在调试模式中）自动重定向到Visual Studio的输出窗口，这使得Console.Write很适合用于诊断问题。但是在大多数情况下，System.Diagnostics命名空间中的Debug和Trace类更加适合（见第13章）。

6.13.2 Environment

静态的System.Environment类具有很多重要的属性：

文件与文件夹

CurrentDirectory、SystemDirectory、CommandLine

计算机与操作系统

MachineName、ProcessorCount、OSVersion

用户登录

UserName、 UserInteractive、UserDomainName

诊断

TickCount、StackTrace、WorkingSet、Version

可以通过调用GetFolderPath获得更多的文件夹；我们将在第15章的“文件与目录操作”中介绍这个方法。

可以使用下面三个方法来访问操作系统环境变量（在命令提示符中输入“set”时得到的结果）：GetEnvironmentVariable、GetEnvironmentVariables和SetEnvironmentVariable。

可以使用ExitCode属性设置返回码，当程序被命令行或批处理文件调用时，FailFirst方法会在不执行清理操作的情况下立即停止程序。

6.13.3 Process

System.Diagnostics中的Process类可以用于启动一个新的进程。

警告：出于安全性的考虑，Metro模板不包含Process类，因此无法随意启动进程。相反，必须使用Windows.System.Launcher类，“启动”一个具有访问权限的URI或文件，例如：

```
Launcher.LaunchUriAsync (new Uri ("http://albahari.com"));

var file = await KnownFolders.DocumentsLibrary
                             .GetFileAsync ("foo.txt");
    Launcher.LaunchFileAsync (file);
```

这样可以打开这个URI或文件，然后就可以使用该URI模式中文件扩展名所关联的程序。当前程序必须位于前台才能执行这个操作。

静态Process.Start方法有很多重载方法，最简单的方法是在可选参数中接受一个文件名，例如：

```
Process.Start ("Notepad.exe");
Process.Start ("Notepad.exe", "e:\\file.txt");
```

还可以只指定一个文件名，而注册其扩展名的程序将会执行：

```
Process.Start ("e:\\file.txt");
```

最灵活的重载方法是能够接受一个ProcessStartInfo实例。通过这个方法，能够获取并重新启动进程的输入、输出和错误输出（如果将UseShellExecute设置为false），并将它重定向到其他位置。下面的代码是获取执行ipconfig命令的输出：

```
ProcessStartInfo psi = new ProcessStartInfo
{
  FileName = "cmd.exe",
  Arguments = "/c ipconfig /all",
  RedirectStandardOutput = true,
  UseShellExecute = false
};
Process p = Process.Start (psi);
string result = p.StandardOutput.ReadToEnd();
Console.WriteLine (result);
```

如果将Filename设置为如下所示的值，那么可以用同样的方式调用csc编译器：

```
psi.FileName = System.IO.Path.Combine (
  System.Runtime.InteropServices.RuntimeEnvironment.GetRuntimeDirectory(), "csc.exe");
```

如果没有重定向输出，那么Process.Start会与调用者并行地执行这个程序。如果希望等待新程序完成，可以通过传入一个可选的超时时间来调用Process对象的WaitForExit方法。

Process类也允许查询计算机上运行的其他进程，并与之交互（见第13章）。

第7章 集合

.NET Framework提供了标准的存储和管理对象集合的类型集。其中包括可变大小列表、链表和排序或不排序字典以及数组。在这些类型中，只有数组属于C#语言；其余的集合只是一些类，可以像使用其他类一样进行实例化。

Framework中的集合类型可以分成以下三类：

- 定义标准集合协议的接口
- 随时可用的集合类（列表、字典等）
- 编写应用程序特有集合的基类

本章将介绍其中的每一类集合，并介绍如何确定元素数量和顺序的类型。

集合命名空间有以下几种：

命名空间	包括
System.Collections	非泛型集合类和接口
System.Collections.Specialized	强类型非泛型集合类
System.Collections.Generic	泛型集合类和接口
System.Collections.ObjectModel	自定义集合的委托和基类
System.Collections.Concurrent	线程安全的集合（见第23章）

7.1 枚举

在计算中，集合类型有很多，包括简单的数据结构（如数组或链表）和复杂的结构（如红/黑树和散列表）。虽然这些数据结构的内部实现和外部特征差别很大，但是它们几乎都需要实现遍历集合内容的功能。Framework通过一系列的接口（`IEnumerable`、`IEnumerator`及其泛型对应结构）来支持这个需求，它们允许不同的数据结构使用一组通用的遍历API。部分集合接口的说明如图7-1所示。

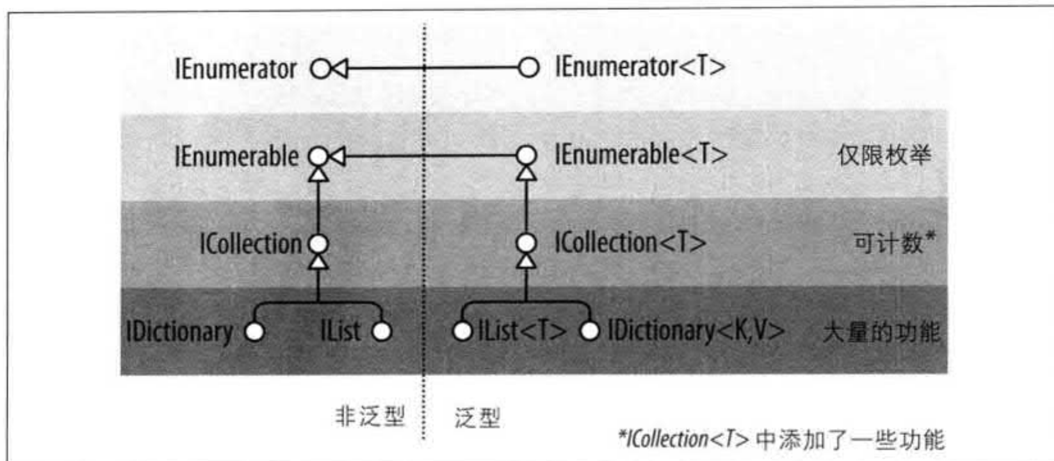


图7-1：集合接口

7.1.1 IEnumerable和IEnumerator

IEnumerator接口定义了以向前方式遍历或枚举集合元素的基本底层协议。声明方式如下：

```
public interface IEnumerator
{
  bool MoveNext();
  object Current { get; }
  void Reset();
}
```

MoveNext将当前元素或“游标”向前移动到下一个位置，如果集合没有更多的元素，那么它会返回false。Current返回当前位置的元素（通常需要从object转换为更具体的类型）。在取出第一个元素之前，我们必须先调用MoveNext——即使是空集合也支持这个操作。如果Reset方法实现了，那么它的作用就是将位置移回到起点，允许再一次遍历集合。（通常是不需要调用Reset的，因为并非所有枚举器都支持这个方法。）

Collections并未实现枚举器，而是通过接口IEnumerable提供枚举器：

```
public interface IEnumerable
{
  IEnumerator GetEnumerator();
}
```

通过定义一个返回枚举器的方法，IEnumerable实现了将迭代逻辑转交由另一类完成的灵活性。而且，这意味着多个使用者可以同时遍历集合，而不会互相干扰。IEnumerable可以看作是“IEnumerator的提供者”，它是集合类需要实现的最基础接口。

下面的例子说明了IEnumerable和IEnumerator的用法：

```
string s = "Hello";

// 因为string实现了IEnumerable，我们可以调用GetEnumerator()：
IEnumerator rator = s.GetEnumerator();
```

```
while (rator.MoveNext())
{
  char c = (char) rator.Current;
  Console.Write (c + ".");
}
// 输出：H.e.l.l.o.
```

然而，我们很少采用这种直接调用枚举器的方法，因为C#提供了一个快捷语法：foreach语句。下面是使用foreach语句重写的具有相同效果的示例：

```
string s = "Hello"; // String类实现了IEnumerable

foreach (char c in s)
Console.Write (c + ".");
```

7.1.2 IEnumerable<T>和IEnumerator<T>

IEnumerator和IEnumerable几乎总是和它们扩展的泛型版本同时实现：

```
public interface IEnumerator<T> : IEnumerator, IDisposable
{
  T Current { get; }
}

public interface IEnumerable<T> : IEnumerable
{
  IEnumerator<T> GetEnumerator();
}
```

通过定义一个类型化的Current和GetEnumerator，这些接口强化了静态类型安全性，避免了装箱值类型元素的额外负载，这对于用户而言更为方便。数组已经自动实现了IEnumerable<T>（其中T指的是数组成员类型）。

由于静态类型安全性得到改进，调用下面的方法来处理字符数组将会出现编译时错误：

```
void Test (IEnumerable<int> numbers) { ... }
```

集合类公开提供IEnumerable<T>接口，而通过显式接口实现“隐藏”非泛型的IEnumerable，这是一种标准的方法。这样，如果直接调用GetEnumerator()，那么返回到类型安全的泛型IEnumerator<T>。但是，有时候这个规则会由于向后兼容性而被破坏（C# 2.0以前不支持泛型）。一个很好的反映这个问题的例子是数组——它们必须返回非泛型（更贴切的说法是“经典的”） IEnumerator，以避免破坏之前的代码。为了获得一个泛型IEnumerator<T>，必须强制转换为显式接口：

```
int[] data = { 1, 2, 3 };
var rator = ((IEnumerable <int>)data).GetEnumerator();
```

幸好，我们可以使用foreach语句，所以很少需要编写这种代码。

IEnumerable<T>和IDisposable

IEnumerable<T>实现了IDisposable。它允许枚举器保存资源引用（如数据库连接），并保证这些资源在枚举结束或者中途停止时能够被释放。Foreach语句能够识别这个细节，并将下面的语句：

```
foreach (var element in somethingEnumerable) { ... }
```

转换为：

```
using (var rator = somethingEnumerable.GetEnumerator())
  while (rator.MoveNext())
  {
    var element = rator.Current;
    ...
  }
```

using语句保证清理操作的执行——第12章将更详细介绍IDisposable。

何时使用非泛型接口

由于诸如IEnumerable<T>的泛型接口具有更高的类型安全性，所以这里会有一个问题：还需要使用非泛型IEnumerable（或ICollection或IList）吗?

对于IEnumerable接口，实现这个接口时需要同时实现IEnumerable<T>，因为后者是从前者继承而来的。然而，实际上很少需要从零开始实现这些接口：几乎在所有情况中，都可以使用迭代器方法、Collection<T>和LINQ等高级实现途径。

所以，作为用户应该如何选择？几乎在所有情况中，都可以完全管理泛型接口。但是，有时非泛型接口仍然是很有用的，因为它们能够在集合中实现所有元素类型的类型统一性。例如，下面的方法能够递归地计算所有集合的元素个数：

```
public static int Count (IEnumerable e)
{
  int count = 0;
  foreach (object element in e)
  {
var subCollection = element as IEnumerable;
if (subCollection != null)
  count += Count (subCollection);
else
  count++;
  }
  return count;
}
```

因为C#与泛型接口协同变化，因此使用这个方法来代替接受IEnumerable<object>也是有效的。然而，在遇到值类型元素时可能会出错。而在使用未实现IEnumerable<T>的传统集合时也可能会出错，例如Windows Forms的ControlCollection。

注意，例子中存在一个潜在的bug：循环引用将导致无限递归，从而使这个方法崩溃。最简单的修复方法是使用HashSet（参见第7章的“HashSet<T>和Sorted-Set<T>”）。

7.1.3 实现Enumeration接口

有时由于下面一个或多个原因而希望实现IEnumerable或IEnumerable<T>：

- 为了支持foreach语句

- 为了与任何使用标准集合的组件交互
- 作为一个更复杂集合接口实现的一部分
- 为了支持集合初始化器

为了实现IEnumerable/IEnumerable<T>，必须提供一个枚举器。可以采用以下三个方法来实现：

- 如果这个类“包装”了任何一个集合，那么就返回所包装集合的枚举器
- 使用yield return的迭代器
- 实例化IEnumerator/IEnumerator<T>

提示： 还可以创建一个现有集合类的子类，Collection<T>正是基于此目而设计的（见本章的“可定制集合和委托”）。另一种方法是使用我们将在下一章介绍的LINQ查询操作数。

返回另一个集合的枚举器就是调用内部集合的GetEnumerator。然而，这种方法仅仅适合一些最简单的情况，那就是内部集合的元素正好是所需要的类型。更好的方法是使用C#的yield reture语句编写迭代器。迭代器是C#语言的一个特性，它能够协助完成集合编写，与foreach语句协助完成集合遍历的方式是一样的。迭代器会自动处理IEnumerable和IEnumerator或者它们的泛型类的实现。

下面是一个简单的例子：

```
public class MyCollection : IEnumerable
{
  int[] data = { 1, 2, 3 };

  public IEnumerator GetEnumerator()
  {
foreach (int i in data)
yield return i;
  }
}
```

注意，GetEnumerator实际上不返回一个枚举器。通过解析yield reture语句，编译器编写一个隐藏的枚举器类，然后重构GetEnumerator来实例化和返回这个类。迭代器很强大，也很简单，并且是LINQ实现的基础。

使用这种方法，我们也能够实现泛型接口IEnumerable<T>：

```
public class MyGenCollection : IEnumerable<int>
{
  int[] data = { 1, 2, 3 };

  public IEnumerator<int> GetEnumerator()
  {
    foreach (int i in data)
      yield return i;
  }

  IEnumerator IEnumerable.GetEnumerator()     // 显式实现
  {                                           // 使它隐藏
    return GetEnumerator();
  }
}
```

集合

因为IEnumerable<T>实现了IEnumerable，所以必须同时实现泛型和非泛型的GetEnumerator。为了与标准方法保持一致，我们还显式地实现了非泛型版本。它能够直接调用泛型的GetEnumerator，因为IEnumerable<T>可以实现IEnumerator。

我们刚刚编写的类很适合作为编写更复杂集合的基础。然而，如果我们只需要一个简单的IEnumerable<T>实现，那么yield return语句可以实现更简单的变化。不需要编写类，但是可以将迭代逻辑转移到一个返回泛型IEnumerable<T>的方法，并由编译器完成剩余工作。如下面这个例子：

```
public class Test
{
  public static IEnumerable <int> GetSomeIntegers()
  {
    yield return 1;
    yield return 2;
    yield return 3;
  }
}
```

下面是我们使用的方法：

```
foreach (int i in Test.GetSomeIntegers())
  Console.WriteLine (i);

// 输出结果
1
2
3
```

最后一种编写GetEnumerator的方法是编写一个直接实现IEnumerator的类。这与编译器在解析迭代器时的工作是完全相同的（大多数情况下不需要这样做）。下面的例子定义了一个集合，它以硬编码方式包含三个整数即1、2和3：

```
public class MyIntList : IEnumerable
{
  int[] data = { 1, 2, 3 };

  public IEnumerator GetEnumerator()
  {
    return new Enumerator (this);
  }

  class Enumerator : IEnumerator      // 定义一个内部类
  {                                   // 对于枚举器。
    MyIntList collection;
    int currentIndex = -1;

    internal Enumerator (MyIntList collection)
    {
      this.collection = collection;
    }

  public object Current
  {
  get
  {
    if (currentIndex == -1)
      throw new InvalidOperationException ("Enumeration not started!");
```

```
      if (currentIndex == collection.data.Length)
        throw new InvalidOperationException ("Past end of list!");
      return collection.data [currentIndex];
    }
  }
  public bool MoveNext()
  {
    if (currentIndex > collection.data.Length) return false;
    return ++currentIndex < collection.data.Length;
  }
  public void Reset() { currentIndex = -1; }
    }
  }
```

提示： 实现Reset方法不是必需的，相反，可以抛出一个NotSupportedException。

注意，第一次调用MoveNext会将位置移到列表的第一个（而非第二个）元素。

为了与迭代器在功能上保持一致，还必须实现IEnumerator<T>。下面是一个为了保持简洁而省略了边界检查代码的示例：

```
  class MyIntList : IEnumerable<int>
  {
    int[] data = { 1, 2, 3 };
    // 泛型枚举器同时兼容IEnumerable和IEnumerable<T>。
    // 我们可以显式地实现非泛型的GetEnumerator方法来避免命名冲突。
    public IEnumerator<int> GetEnumerator() { return new Enumerator(this); }
    IEnumerator IEnumerable.GetEnumerator() { return new Enumerator(this); }
    class Enumerator : IEnumerator<int>
    {
  int currentIndex = -1;
  MyIntList collection;
  internal Enumerator (MyIntList collection)
  {
    this.collection = collection;
  }
  public int Current { get { return collection.data [currentIndex]; } }
  object IEnumerator.Current { get { return Current; } }
  public bool MoveNext()
  {
    return ++currentIndex < collection.data.Length;
  }
  public void Reset() { currentIndex = -1; }
  // 既然我们不需要Dispose方法，那么最好显式地实现这个方法，所以它不是公开的接口。
  void IDisposable.Dispose() {}
    }
  }
```

这个使用泛型的例子执行速度很快，因为IEnumerator<int>.Current不需要将int转换为object，所以避免了一些装箱负载。

集合

7.2 ICollection和IList接口

虽然枚举接口提供一种向前遍历集合的协议，但是它们并没有实现用于确定集合大小、根据索引访问元素、搜索或修改集合的方法。为了实现这些功能，.NET Framework定义了ICollection、IList和IDictionary接口。这些接口都支持泛型和非泛型版本；然而，非泛型版本的存在只是为了兼容遗留代码。

这些接口的继承层次如图7-1所示。最简单的方法是将它们总结为：

IEnumerable<T>（和IEnumerable）

　　支持最少的功能（只支持枚举）。

ICollection<T>（和ICollection）

　　支持一般的功能（例如，Count属性）。

IList<T>/IDictionary <K,V>*及其非泛型版本*

　　支持最多的功能（包括根据索引/键实现“随机”访问）。

提示： 大多数情况下不需要实现这些接口。几乎在需要编写一个集合类的任何时候，都可以使用子类Collection<T>替代（见本章的“自定义集合与代码”）。LINQ还提供了另一个适合许多情况的方法。

泛型和非泛型版本的差别很大，特别是对于ICollection。这其中很多是由于历史原因造成的：因为泛型出现在后，而泛型接口是为了利用后台出现的泛型而开发的。为此，ICollection<T>并没有继承ICollection，IList<T>也没有继承IList，而且IDictionary<TKey, TValue>也同样不继承IDictionary。当然，在有利的情况下，集合类本身通常是可以实现某个接口的两个版本的。

提示： IList<T>不扩展IList的另一个原因是因为转换到IList<T>会返回一个同时支持Add(T)和Add(object)成员的接口。这实际上会破坏静态类型安全性，因为可能会使用任意类型的object调用Add。

这一节介绍了ICollection<T>、IList<T>及其非泛型版本；本章的“字典”将介绍字典接口。

提示： .NET Framework中并没有一种统一使用*集合*（collection）和*列表*（list）这两个词的方法。例如，由于IList<T>在功能表述上优于ICollection<T>，所以可能认为List<T>类会比Collection<T>类具有更多的功能。但是实际并非如此。我们通常将*集合*（collection）和*列表*（list）这两个术语看作在很多方面是同义的，只有在使用具体类型时例外。

7.2.1 ICollection<T>和ICollection

ICollection<T>是对象的可计数集合的标准接口。它提供了很多功能，包括确定集合大小（Count）、确定集合中是否存在某个元素（Contains）、将集合复制到一个数组（ToArray）以及确定集合是否为只读（IsReadOnly）。对于可写集合，可能还需要对集合元素执行Add、Remove和Clear操作。而且，由于它继承了IEnumerable<T>，所以也支持通过foreach语句进行遍历：

```
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
  int Count { get; }

  bool Contains (T item);
  void CopyTo (T[] array, int arrayIndex);
  bool IsReadOnly { get; }

  void Add(T item);
  bool Remove (T item);
  void Clear();
}
```

非泛型的ICollection具有与可计数集合类似的功能，但是它不支持修改列表或检查元素成员的功能：

```
public interface ICollection : IEnumerable
{
  int Count { get; }
  bool IsSynchronized { get; }
  object SyncRoot { get; }
  void CopyTo (Array array, int index);
}
```

非泛型接口也定义了一些协助同步性（见第14章）的属性，而泛型版本没有这些属性，因为集合本身不再考虑线程安全性。

这两种接口的实现都非常简单。如果要实现一个只读的ICollection<T>，Add、Remove和Clear方法应该抛出一个NotSupportedException。

这些接口通常与IList或IDictionary一起实现。

7.2.2 IList<T>和IList

IList<T>是标准的可按位置索引的接口。除了从ICollection<T>和IEnumerable<T>继承的功能，它还提供了按位置（通过一个索引器）读写元素和按位置插入/删除元素的功能：

```
public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
  T this [int index] { get; set; }
  int IndexOf (T item);
  void Insert (int index, T item);
  void RemoveAt (int index);
}
```

IndexOf方法可以对列表执行线性搜索，如果未找到指定项，那么返回-1。

IList的非泛型版本具有更多的成员方法，因为它继承了少量的ICollection成员方法：

```
public interface IList : ICollection, IEnumerable
{
  object this [int index] { get; set }
  bool IsFixedSize { get; }
  bool IsReadOnly { get; }
  int Add (object value);
  void Clear();
  bool Contains (object value);
```

```
    int IndexOf (object value);
    void Insert (int index, object value);
    void Remove (object value);
    void RemoveAt (int index);
  }
```

非泛型IList接口的Add方法返回一个整数，这是最新添加元素的索引。相反，ICollection<T>的Add方法的返回类型为void。

通用的List<T>类是IList<T>和IList的典型实现。C#数组也同时实现了泛型和非泛型的IList（虽然添加或删除元素的方法是通过显式接口实现隐藏的，并且在调用时会抛出一个NotSupportedException）。

警告：如果试图通过IList的索引访问一个多维数组，程序就会抛出一个ArgumentException异常。这是故意的，可以将方法修改为：

```
public object FirstOrNull (IList list)
{
if (list == null || list.Count == 0) return null;          281
      return list[0];
}
```

这段代码似乎毫无破绽，但是如果传入一个多维数据，就会抛出一个异常。我们可以在运行时使用下面的表达式测试一个多维数组（第19章有更详细的介绍）：

```
list.GetType().IsArray && list.GetType().GetArrayRank()>1
```

IReadOnlyList<T>

为了与只读的Windows Runtime集合实现互操作，Framework 4.5引入了一个新的集合接口IReadOnlyList<T>。这个接口本身很有用，并且可以看作为IList<T>的缩简版本，它只包含列表只读操作所需要的成员：

```
public interface IReadOnlyList<out T> : IEnumerable<T>, IEnumerable
{
int Count { get; }
T this[int index] { get; }
}
```

因为它的类型参数只用在输出位置，所以它被标记为协变式（covariant）。例如，这样就可以将一个小猫链表示为一个只读的动物链表。相反，IList<T>的T并没有标记为协变式，因为T既可用在输入位置，也可用在输出位置。

提示：IReadOnlyList<T>表示一个链表的只读版本。它并不意味着底层实现也是只读的。

如果IList<T>从IReadOnlyList<T>派生，也合乎情理。然而，微软无法做这样的修改，因为这样就要求将IList<T>的成员转移到IReadOnlyList<T>上，从而会给CLR 4.5造成破坏性变化（使用者可能需要重新编译程序，才能消除编译错误）。相反，IList<T>的实现者需要手动将IReadOnlyList<T>添加到在他们实现的接口中。

IReadOnlyList<T>与Windows运行时类型IVectorView<T>相对应。

7.3 Array类

Array类是所有一维和多维数组的隐式基类，它是实现标准集合接口的最基本类型之一。Array类提供了类型统一性，所以常见的方法都适用于所有的数组，而与它们声明或实际的元素类型无关。

由于数组是基本类型，所以C#提供了明确的声明和初始化语法，我们已经在第2章和第3章中介绍过了。当使用C#语法声明一个数组时，CLR会在内部将它转换为Array的子类——合成一个对应该数组维数和元素类型的伪类型。这个伪类型实现了类型化的泛型集合接口，如IList<string>。

CLR也会特别处理数组类型的创建，将它们分配到一块连续的内存空间。因此数组的索引非常高效，但是不允许在创建后修改数组大小。

Array实现了IList<T>的泛型与非泛型的集合接口。但是，IList<T>本身是显式实现的，以保证Array的公开接口中不包含其中一些方法，如Add或Remove，这些方法会在固定长度的集合（如数组）上抛出一个异常。Array类实例也提供了一个静态的Resize方法，但是它实际上是创建一个新数组，然后将每一个元素复制到新数组中。Resize方法是很低效的，而且程序的数组引用无法修改为新位置。实现可变大小集合的最好方法是使用List<T>类（将在下一节介绍）。

数组可以包含值类型或引用类型元素。值类型元素存储在数组中，所以一个有三个long整数（每个占用8字节）的数组将会占用24个字节的连续内存空间。然而，引用类型在数组中只占用一个引用所需要的空间（32位环境中是4个字节，64位环境中是8个字节）。图7-2说明了下面这个程序在内存中的作用：

```
StringBuilder[] builders = new StringBuilder [5];
builders [0] = new StringBuilder ("builder1");
builders [1] = new StringBuilder ("builder2");
builders [2] = new StringBuilder ("builder3");

long[] numbers = new long [3];
numbers [0] = 12345;
numbers [1] = 54321;
```

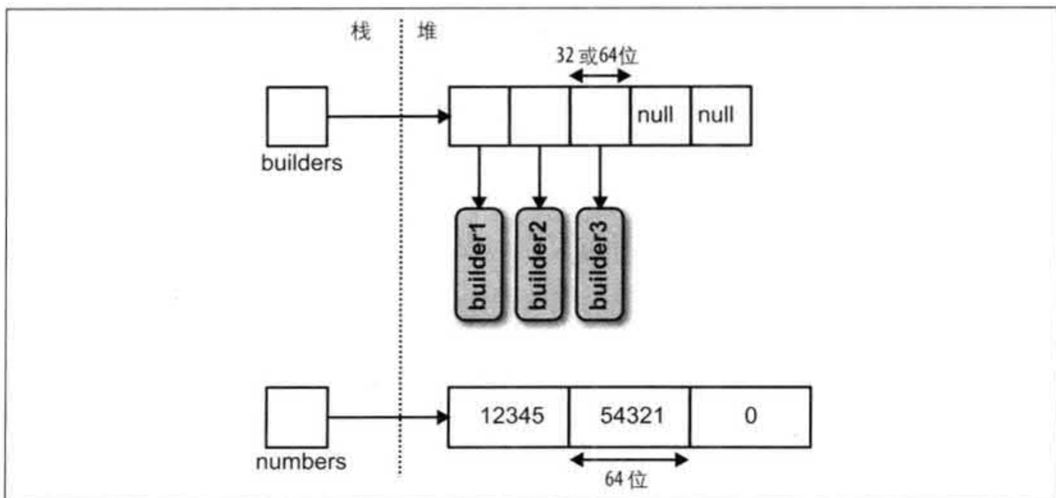


图7-2：内存中的数组

因为Array是一个类，所以无论数组的元素是什么类型，数组（本身）总是引用类型。这意味着语句arrayB=arrayA的结果是将两个变量指向同一个数组。类似地，两个不同的数组在等值比较中总是不相等的——除非使用自定义的等值比较。Framework 4.0提供了一种用于比较数组或元组元素的比较方式，可以通过StructuralComparisons类型进行访问：

```
object[] a1 = { "string", 123, true };
object[] a2 = { "string", 123, true };

Console.WriteLine (a1 == a2);                  // False
Console.WriteLine (a1.Equals (a2));            // False

IstructuralEquatable se1 = a1;
Console.WriteLine (a1.Equals (a2,
 StructuralComparisons.StructuralEqualityComparer)); // True
```

数组可以通过Clone方法进行复制，例如arrayB = arrayA.Clone()。然而，这是一个浅克隆，表示只有数组本身表示的内存会被复制。如果数组包含的是值类型的对象，那么这些值会被复制；如果数组包含的是引用类型的对象，那么只有引用被复制（结果就是两个数组的元素都引用相同的对象）。图7-3演示了将下面的代码添加到例子中的效果：

```
StringBuilder[] builders2 = builders;
StringBuilder[] shallowClone = (StringBuilder[]) builders.Clone();
```

如果要进行深度复制即复制引用类型子对象，必须遍历整个数组，然后手动克隆每个元素。相同的规则也适用于其他.NET集合类型。

虽然Array主要是针对32位索引器设计的，但是它也通过一些能够接受Int32和Int64参数的方法实现对64位索引的部分支持（在理论上允许一个数组最多支持2^{64}个元素）。这些重载方法在实例中的用处是很小的，因为CLR不允许任何对象（包括数组）在大小上超过2GB（无论是运行在32位或是64位环境上）。

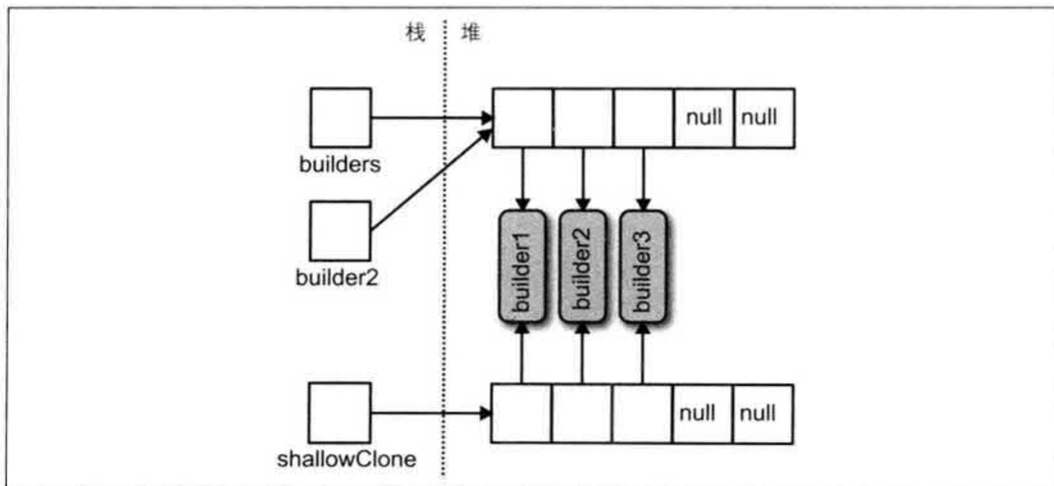


图7-3：数组的浅克隆

警告： 你可能会以为Array类的许多方法是实例方法，但是实际上它们是静态方法。这是一个奇怪的设计方法，意味着在寻找Array方法时，应该同时查看静态方法和实例方法。

7.3.1 创建和索引

最简单的创建和索引数组的方法是使用C#的语言构造：

```
int[] myArray = { 1, 2, 3 };
int first = myArray [0];
int last = myArray [myArray.Length - 1];
```

此外，可以通过调用Array.CreateInstance动态实例化一个数组，可以在运行时指定元素类型和维数以及为非零开始索引的数组指定下界。非零开始索引的数组不符合CLS（Common Language Specification，公共语言规范）。

静态的GetValue和SetValue方法访问动态创建的数组的元素（它们也支持普通数组的元素访问）：

```
// 创建一个长度为2的字符串数组：
Array a = Array.CreateInstance (typeof(string), 2);
a.SetValue ("hi", 0);                          // → a[0] = "hi";
a.SetValue ("there", 1);                       // → a[1] = "there";
string s = (string) a.GetValue (0);            // → s = a[0];

// 也可以按以下方式转换为一个C#数组：
string[] cSharpArray = (string[]) a;
string s2 = cSharpArray [0];
```

动态创建的从零开始索引的数组可以转换为一种类型匹配或兼容（兼容标准数组变化规则）的C#数组。例如，如果Apple是Fruit的子类，Apple[]就可以转换为Fruit[]。这就产生一个问题，为什么不使用object[]作为统一的数组类型，而要使用Array类呢？原因就是object[]既不兼容多维数组，也不兼容值类型数组以及非零开始索引的数组。int[]数组不能够转换为obejct[]。因此，我们需要使用Array类实现完整的类型统一性。

GetValue和SetValue也支持编译器创建的数组，并且它们对于编写能够处理任意类型和任意维数数组的方法是很有用的。对于多维数组，它们接受一个索引器数组：

```
public object GetValue (params int[] indices)
public void SetValue (object value, params int[] indices)
```

下面的方法会打印出任意数组的第一个元素，数组的维数无关：

```
void WriteFirstValue (Array a)
{
  Console.Write (a.Rank + "-dimensional; ");

  // 索引器数组将全部自动初始化为0，所以把它传递给GetValue或SetValue
  // 将会get/set从零开始的数组元素（例如，第一个）。
  int[] indexers = new int[a.Rank];
  Console.WriteLine ("First value is " + a.GetValue (indexers));
}

void Demo()
{
  int[] oneD = { 1, 2, 3 };
```

```
  int[,] twoD = { {5,6}, {8,9} };

  WriteFirstValue (oneD); // 第1维：第1个值为1
  WriteFirstValue (twoD); // 第2维：第1个值为5
}
```

提示：对于已知维数但未知类型的数组，泛型提供了一种更加简单且高效的方法：

```
void WriteFirstValue<T> (T[] array)
{
  Console.WriteLine (array[0]);
}
```

如果元素与数组类型不一致，SetValue方法会抛出一个异常。

当实例化数组时，无论是通过语言语法还是Array.CreateInstance，数组元素都会自动初始化。对于引用类型元素的数组，这意味着写入null值；对于值类型元素的数组，这意味着调用值类型的默认构造函数（实际上是成员的"归零"操作）。Array类也通过Clear方法实现归零功能：

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

这种方法不会改变数组的大小。不同于通常意义的Clear用法（如ICollection<T>.Clear）的是，这些方法一般会将集合元素删除。

7.3.2 枚举

数组可以通过foreach语句进行枚举：

```
int[] myArray = { 1, 2, 3};
foreach (int val in myArray)
Console.WriteLine (val);
```

也可以使用静态的Array.ForEach方法进行枚举，如下所示：

```
public static void ForEach<T> (T[] array, Action<T> action);
```

这种方法使用一个Action delegate及其以下签名：

```
public delegate void Action<T> (T obj);
```

下面使用Array.ForEach重写第一个例子：

```
Array.ForEach (new[] { 1, 2, 3 }, Console.WriteLine);
```

7.3.3 长度和维数

Array提供了下面的方法和属性来查询长度和维数：

```
public int GetLength          (int dimension);
public long GetLongLength     (int dimension);

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

```
public int GetLowerBound (int dimension);
public int GetUpperBound (int dimension);

public int Rank { get; }     // 返回数组的维数
```

GetLength和GetLongLength会返回一个指定维度的长度（0表示一维数组），而Length和LongLength返回数组的元素总数（包括所有维数）。

GetLowerBound和GetUpperBound在处理非零开始索引的数组时是很有用的。GetUpperBound返回的结果与任意维度的GetLowerBound和GetLength相加的结果是相同的。

7.3.4 查找

Array类提供了许多用于查找一维数组元素的方法：

BinarySearch*方法*

用于快速搜索排序数组中的特定元素

IndexOf / LastIndex*方法*

用于搜索未排序数组中的特定元素

Find / FindLast / FindIndex / FindLastIndex / FindAll / Exists / TrueForAll

用于搜索未排序数组中满足指定Predicate<T>的一个或多个元素

如果指定值未找到，这里的每一个数组搜索方法都不会抛出异常。相反，如果一个元素未找到，那么这些方法会返回一个整数-1（假设是索引从0开始的数组），而返回泛型类型的方法则返回该类型的默认值（例如，0对应int，或者null对应string）。

二分查找方法速度快，但是它们只适用于排序数组，而且要求元素能够比较顺序，而不只是比较是否相等。要实现这个效果，二分查找方法可以接受一个IComparer或IComparer<T>对象，用于判断元素的顺序（参见本章后面“插入等值和顺序”小节）。这个判断必须与原先数组排序时所使用的比较器保持一致。如果没有比较器，则根据它的IComparable / IComparable<T>实现使用该类型的默认排序算法。

IndexOf和LastIndexOf方法会对数组执行简单的枚举过程，返回与给定值匹配的第一个（或最后一个）元素的位置。

基于断言的查找方法允许使用一个方法委托或lambda表达式来判断给定的元素是否匹配。断言实际上是一个可以接受对象和返回true或false的委托：

```
public delegate bool Predicate<T> (T object);
```

在下例中，我们在一个字符串数组中查找包含字母“a”的姓名：

```
static void Main()
{
  string[] names = { "Rodney", "Jack", "Jill" };
  string match = Array.Find (names, ContainsA);
  Console.WriteLine (match);       // Jack
}
static bool ContainsA (string name) { return name.Contains ("a"); }
```

下面是使用匿名方法缩写的同一段代码：

```
string[] names = { "Rodney", "Jack", "Jill" };
string match = Array.Find (names, delegate (string name)
  { return name.Contains ("a"); } );
```

lambba表达式可以将代码进一步缩写为：

```
string[] names = { "Rodney", "Jack", "Jill" };
string match = Array.Find (names, n => n.Contains ("a")); // Jack
```

FindAll会返回包含所有满足断言的元素的数组。事实上，它等同于System.Linq命名空间的Ennmerable.Where，只是FindAll返回一个包含匹配项的数组，而不是一个IEnumerable<T>。

如果所有数组成员都满足给定的断言，那么Exists会返回true，它等同于System.Linq.Enumerable的Any。

如果所有项都满足这个断言，那么TrueForAll会返回true，它等同于System.Linq.Enumerable的All。

7.3.5 排序

Array有以下内置排序方法：

```
// 用于排序的一个数组：

public static void Sort<T> (T[] array);
public static void Sort    (Array array);

// 用于排序的一对数组：

public static void Sort<TKey,TValue> (TKey[] keys, TValue[] items);
public static void Sort              (Array keys, Array items);
```

每一个方法重载后都可以接受：

```
int index                    // 排序的开始位置
int length                   // 排序的元素个数
IComparer<T> comparer        // 判断顺序的对象
Comparison<T> comparison     // 执行顺序比较的代码
```

下面的代码演示了Sort最简单的用法：

```
int[] numbers = { 3, 2, 1 };
Array.Sort (numbers);                  // 数组现在变成 { 1, 2, 3 }
```

这种接受一对数组的方法实际上是通过调整每个数组的元素及基于第一个数组的顺序判断而实现的。在下面的例子中，数字及其对应的单词都是按照数字顺序排序的：

```
int[] numbers = { 3, 2, 1 };
string[] words = { "three", "two", "one" };
Array.Sort (numbers, words);

// 数字数组现在变成 { 1, 2, 3 }
// 单词数组现在变成 { "one", "two", "three" }
```

Array.Sort要求数组中的元素实现IComparable（见第6章的“顺序比较”），这意味着C#的最基本类型（如前一个例子使用的整数）都可以进行排序。如果元素是不可比较的，或者希望重写默认的

顺序比较，那么必须给Sort提供一个自定义的比较提供者，用来判断两个元素的相对位置。可以采用以下方法：

- 通过一个实现IComparer/IComparer<T>的帮助对象（见本章的“等值和顺序插入”）
- 通过一个Comparison 委托：

```
public delegate int Comparison<T> (T x, T y);
```

Comparison 委托采用与IComparer<T>.CompareTo相同的语义：如果x在y之前，那么返回一个负数；如果x在y后面，那么返回一个正数；如果x和y具有相同位置顺序，那么返回0。

在这个例子中，我们对一个整数数组进行排序使奇数在偶数前面：

```
int[] numbers = { 1, 2, 3, 4, 5 };
Array.Sort (numbers, (x, y) => x % 2 == y % 2 ? 0 : x % 2 == 1 ? -1 : 1);

// 数字数组现在变成 { 3, 5, 1, 2, 4 }
```

提示：作为Sort的替代方法，可以使用LINQ的OrderBy和ThenBy运算符。与Array.Sort不同的是，LINQ运算符不会修改原始数组，而是将排序结果保存在一个新的IEnumerable<T>序列中。

7.3.6 倒转元素

这些Array方法会倒转数组中所有或者部分元素的顺序：

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

7.4 复制

Array有4个方法可以执行浅拷贝操作：Clone、CopyTo、Copy和ConstrainedCopy。前两个方法都是实例方法；后两个方法是静态方法。

Clone方法返回一个全新（浅拷贝）的数组。CopyTo和Copy方法复制数组的若干连续元素。复制多维矩阵则需要将多维索引映射为线性索引。例如，一个3 × 3数组的中间矩阵（position[1,1]）可以用索引4表示，计算方法是：1*3 + 1。源与目标范围可以重叠，而不会产生问题。

ConstrainedCopy执行一个*原子*操作：如果所有请求的元素都无法成功复制（如由于类型错误），那么操作会回滚。

Array还有一个AsReadOnly方法，它会返回一个包装器，可以防止元素被重新赋值。

转换和调整大小

Array.ConvertAll会创建和返回一个包含元素类型TOutput的新数组，调用所提供的Converter代理就可以复制元素。Converter的定义如下：

```
public delegate TOutput Converter<TInput,TOutput> (TInput input)
```

下面的代码将一个浮点数数组转换为一个整数数组：

集合

```
float[] reals = { 1.3f, 1.5f, 1.8f };
int[] wholes = Array.ConvertAll (reals, r => Convert.ToInt32 (r));

// wholes array is { 1, 2, 2 }
```

Resize方法的实现方式是创建一个新数组，然后将全部元素复制到新数组中，再通过引用参数返回这个新数组。然而，在其他对象中指向原始数组的任意引用都保持不变。

提示：System.Linq命名空间包含另外一些适合用于执行数组转换的扩展方法。这些方法会返回一个IEnumerable<T>，它可以通过Enumerable的ToArray方法转换回一个数组。

7.5 List、Queue、Stack和Set

Framework提供了全面的具体集合类，它们可以实现本章所介绍的接口。这一节主要介绍列表型的集合（而字典型的集合将在本章的“字典”中介绍）。和我们前面介绍的接口一样，对于每一种类型，通常都可以选择使用泛型或非泛型实现。在灵活性和性能方面，泛型类更具优势，而它们的非泛型冗余实现则是为了实现向后兼容。这与集合接口是不一样的，非泛型的集合接口有时还是很有用的。

在本节所介绍的类中，泛型List类是最常使用的。

7.5.1 List<T>和ArrayList

泛型List和非泛型ArrayList类提供了一种动态调整大小的对象数组实现，它们是集合类中使用最广泛的类。ArrayList实现了IList，而List<T>同时实现了IList和IList<T>。与数组不同，所有接口都是公开实现的，而且诸如Add和Remove等方法也是公开可用的。

在内部，List<T>和ArrayList都维护了一个对象数组，并在超出容量时替换为一个更大的数组。添加元素是很高效的（因为数组末尾通常还有空闲存储位置），但是插入元素的速度会慢一些（因为插入位置之后的所有元素都必须向后移动才能留出插入空间）。与数组一样，如果对已排序列表执行BinarySearch方法，那么查找是很高效的，但是其他情况效率就不高，因为查找时必须检查每一个元素。

提示：如果T是一种值类型，那么List<T>的速度会比ArrayList快好几倍，因为List<T>不需要元素执行装箱和开箱操作。

List<T>和ArrayList具有可以接受已有元素集合的构造函数，它们会将已有集合的每一个元素复制到新的List<T>或ArrayList中。

```
public class List <T> : IList <T>
{
  public List ();
  public List (IEnumerable<T> collection);
  public List (int capacity);

  // 添加及插入
  public void Add (T item);
  public void AddRange(IEnumerable<T> collection);
  public void Insert  (int index, T item);
  public void InsertRange (int index, IEnumerable<T> collection);
```

```
  // 删除
  public bool Remove     (T item);
  public void RemoveAt   (int index);
  public void RemoveRange    (int index, int count);
  public int RemoveAll   (Predicate<T> match);

  // 索引
  public T this [int index] { get; set; }
  public List<T> GetRange (int index, int count);
  public Enumerator<T> GetEnumerator();

  // 导出、复制和转换
  public T[] ToArray();
  public void CopyTo (T[] array);
  public void CopyTo (T[] array, int arrayIndex);
  public void CopyTo (int index, T[] array, int arrayIndex, int count);
  public ReadOnlyCollection<T> AsReadOnly();
  public List<TOutput> ConvertAll<TOutput> (Converter <T,TOutput> converter);
  // 其他
  public void Reverse();                          // 反向排序列表元素
  public int Capacity { get;set; }                // 强制扩大内部数组
  public void TrimExcess();                       // 将内部数组缩小为原始大小
  public void Clear();                            // 删除所有元素，所以Count=0
}

public delegate TOutput Converter <TInput, TOutput> (TInput input);
```

除了这些成员方法，List<T>还提供了所有Array查找和排序方法的实例版本。

下面的代码演示了List的属性和方法。请参考本章“Array类”的查找与排序示例：

```
List<string> words = new List<string>();            // 新的字符串类型列表

words.Add ("melon");
words.Add ("avocado");
words.AddRange (new[] { "banana", "plum" } );
words.Insert (0, "lemon");                          // 插入到开头
words.InsertRange (0, new[] { "peach", "nashi" });  // 插入到开头

words.Remove ("melon");
words.RemoveAt (3);                                 // 删除第4个元素
words.RemoveRange (0, 2);                           // 删除前2个元素

// 删除所有以'n'开头的字符串：
words.RemoveAll (s => s.StartsWith ("n"));

Console.WriteLine (words [0]);                      // 第一个单词
Console.WriteLine (words [words.Count - 1]);        // 最后一个单词
foreach (string s in words) Console.WriteLine (s);  // 所有单词
List<string> subset = words.GetRange (1, 2);        // 第2～3个单词

string[] wordsArray = words.ToArray();              // 创建一个新的类型化数组

// 将前两个元素复制到已有数组的末尾：
string[] existing = new string [1000];
words.CopyTo (0, existing, 998, 2);

List<string> upperCastWords = words.ConvertAll (s => s.ToUpper());
List<int> lengths = words.ConvertAll (s => s.Length);
```

非泛型ArrayList类主要用于向后兼容Framework 1.x代码，并且需要进行一些复杂的转换，如下面例子所示：

```
ArrayList al = new ArrayList();
al.Add ("hello");
string first = (string) al [0];
string[] strArr = (string[]) al.ToArray (typeof (string));
```

编译器无法验证这些转换；虽然下面的例子能够编译成功，但是在运行时会出错：

```
int first = (int) al [0]; // 运行时异常
```

提示：ArrayList的功能与List<object>类型相似。当需要一个包含不共享任何相同基类的混合类型元素时，这两种类型是很有用的。在这种情况下，如果需要使用反射机制（见第19章）处理列表，那么选择使用ArrayList更具优势。相比于List<object>，反射机制更容易处理非泛型的ArrayList。

如果定义System.Linq命名空间，那么可以通过先调用Cast再调用ToList的方式将一个ArrayList转换为一个泛型List：

```
ArrayList al = new ArrayList();
al.AddRange (new[] { 1, 5, 9 } );
List<int> list = al.Cast<int>().ToList();
```

Cast和ToList是System.Linq.Enumerable的扩展方法，是从.NET Framework 3.5开始支持的。

7.5.2 LinkedList<T>

LinkedList<T>是一个泛型的双向链表（见图7-4）。双向链表是一系列互相引用的节点，其中每个节点都引用前一个节点、后一个节点及实际存储数据的元素。它的主要优点是元素总是能够高效地插入到链表的任意位置，因为插入节点只需要创建一个新节点，然后修改引用值。然而，查找插入节点的位置可能减慢执行速度，因为链表本身没有直接索引的内在机制；我们必须遍历每一个节点，并且无法执行二叉查找。

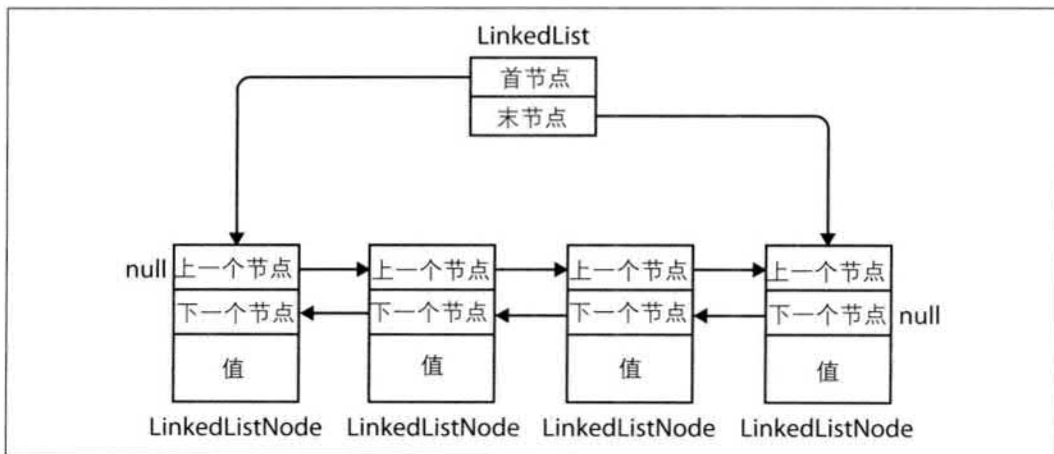


图7-4：LinkedList<T>

LinkedList<T>实现了IEnumerable<T>和ICollection<T>及其非泛型版本，但是没有实现IList<T>，因为它不支持根据索引进行访问。列表节点是通过下面这个类实现的：

```
public sealed class LinkedListNode<T>
{
  public LinkedList<T> List { get; }
  public LinkedListNode<T> Next { get; }
  public LinkedListNode<T> Previous { get; }
  public T Value { get; set; }
}
```

当添加一个节点时，可以指定它相对于其他节点的位置，或者指定列表的开始/结束位置。LinkedList<T>提供下面一些添加节点的方法：

```
public void AddFirst(LinkedListNode<T> node);
public LinkedListNode<T> AddFirst (T value);

public void AddLast (LinkedListNode<T> node);
public LinkedListNode<T> AddLast (T value);

public void AddAfter (LinkedListNode<T> node, LinkedListNode<T> newNode);
public LinkedListNode<T> AddAfter (LinkedListNode<T> node, T value);

public void AddBefore (LinkedListNode<T> node, LinkedListNode<T> newNode);
public LinkedListNode<T> AddBefore (LinkedListNode<T> node, T value);
```

删除元素也有类似的方法：

```
public void Clear();

public void RemoveFirst();
public void RemoveLast();

public bool Remove (T value);
public void Remove (LinkedListNode<T> node);
```

LinkedList<T>内部有些字段用来跟踪列表元素的个数及列表的头和尾。它们是通过下面的公开属性实现的：

```
public int Count { get; }                   // 快速
public LinkedListNode<T> First { get; }     // 快速
public LinkedListNode<T> Last { get; }      // 快速
```

LinkedList<T>也支持以下查找方法（每一个方法的内部都要求枚举该列表）：

```
public bool Contains (T value);
public LinkedListNode<T> Find (T value);
public LinkedListNode<T> FindLast (T value);
```

另外，LinkedList<T>支持将元素复制到一个数组中，以支持索引处理和获得一个支持foreach语句的枚举器：

```
public void CopyTo (T[] array, int index);
public Enumerator<T> GetEnumerator();
```

下面的代码演示了LinkedList<string>的用法：

```
var tune = new LinkedList<string>();
tune.AddFirst ("do");                           // do
tune.AddLast ("so");                            // do - so

tune.AddAfter (tune.First, "re");               // do - re - so
tune.AddAfter (tune.First.Next, "mi");          // do - re - mi - so
tune.AddBefore (tune.Last, "fa");               // do - re - mi - fa - so

tune.RemoveFirst();                             // re - mi - fa - so
tune.RemoveLast();                              // re - mi - fa

LinkedListNode<string> miNode = tune.Find ("mi");
tune.Remove (miNode);                           // re - fa
tune.AddFirst (miNode);                         // mi - re - fa

foreach (string s in tune) Console.WriteLine (s);
```

7.5.3 Queue<T>和Queue

Queue<T>和Queue是一种先进先出（FIFO）的数据结构，它们提供了Enqueue（将一个元素添加到队列末尾）和Dequeue（取出并删除队列的第一个元素）方法。它们还包括一个只返回而不删除队列第一个元素的Peek方法，以及一个Count属性（可用来检查出列前的元素个数）。

虽然队列是可枚举的，但是它们都没有实现IList<T>/IList，因为不能够直接通过索引访问它的成员。然而，它提供了一个ToArray方法，可用于将元素复制到一个数组中，然后进行随机访问：

```
public class Queue<T> : IEnumerable<T>, ICollection, IEnumerable
{
  public Queue();
  public Queue (IEnumerable<T> collection);       // 复制已有元素
  public Queue (int capacity);                    // 减少自动修改大小次数
  public void Clear();
  public bool Contains (T item);
  public void CopyTo (T[] array, int arrayIndex);
  public int Count { get; }
  public T Dequeue();
  public void Enqueue (T item);
  public Enumerator<T> GetEnumerator();           // 为了支持foreach语句
  public T Peek();
  public T[] ToArray();
  public void TrimExcess();
}
```

以下是Queue<int>的应用示例：

```
var q = new Queue<int>();
q.Enqueue (10);
q.Enqueue (20);
int[] data = q.ToArray();              // 导出一个数组
Console.WriteLine (q.Count);           // "2"
Console.WriteLine (q.Peek());          // "10"
Console.WriteLine (q.Dequeue());       // "10"
Console.WriteLine (q.Dequeue());       // "20"
Console.WriteLine (q.Dequeue());       // 抛出一个异常（队列为空）
```

队列内部是使用一个可根据需要调整大小的数组来操作的，这与一般的List类很类似。队列具有一

个直接指向头和尾元素的索引，因此，入列和出列操作是极其快速的（除非内部的大小需要调整）。

7.5.4 Stack<T>和Stack

Stack<T>和Stack是后进先出（LIFO）的数据结构，它们提供了Push（添加一个元素到堆栈的顶部）和Pop（从堆栈顶部取出并删除一个元素）方法。它们还提供了一个只读取而不删除元素的Peek方法，以及Count属性和用于导出数据以实现随机访问的ToArray方法：

```
public class Stack<T> : IEnumerable<T>, ICollection, IEnumerable
{
  public Stack();
  public Stack (IEnumerable<T> collection);        // 复制已有元素
  public Stack (int capacity);                     // 减少自动修改大小的次数
  public void Clear();
  public bool Contains (T item);
  public void CopyTo (T[] array, int arrayIndex);
  public int Count { get; }
  public Enumerator<T> GetEnumerator();            // 为了支持foreach语句
  public T Peek();
  public T Pop();
  public void Push (T item);
  public T[] ToArray();
  public void TrimExcess();
}
```

下面是Stack<int>的使用示例：

```
var s = new Stack<int>();
s.Push (1);                      //           Stack = 1
s.Push (2);                      //           Stack = 1,2
s.Push (3);                      //           Stack = 1,2,3
Console.WriteLine (s.Count);     // Prints 3
Console.WriteLine (s.Peek());    // Prints 3, Stack = 1,2,3
Console.WriteLine (s.Pop());     // Prints 3, Stack = 1,2
Console.WriteLine (s.Pop());     // Prints 2, Stack = 1
Console.WriteLine (s.Pop());     // Prints 1, Stack = <empty>
Console.WriteLine (s.Pop());     // 抛出异常
```

堆栈内部也是使用一个可根据需要调整大小的数组来操作，这一点和Queue<T>与List<T>类似。

7.5.5 BitArray

BitArray是一个保存压缩bool值的可动态调整大小的集合。它具有比简单的bool数组和bool泛型List更高的内存使用效率，因为它的每个值只占用一位，而bool类型的每个值占用一个字节：

BitArray的索引器可以读写各个位：

```
var bits = new BitArray(2);
bits[1] = true;
```

位操作方法有4种（And、Or、Xor和Not）。只有最后一个方法接受一个BitArray作为参数：

```
bits.Xor (bits);               // 按位异或
Console.WriteLine (bits[1]); // False
```

7.5.6 HashSet<T>和SortedSet<T>

HashSet<T>和SortedSet<T>分别是Framework 3.5和4.0新增加的泛型集合。这两个类都具有以下特点：

- 它们的Contains方法都使用基于散列的查找而实现快速执行。
- 它们都不保存重复元素，并且都忽略添加重复值的请求。
- 无法根据位置访问元素。

SortedSet<T>按一定顺序保存元素，而HashSet<T>则不是。

提示： 这些类型的共同点是由接口ISet<T>提供的。

由于历史原因，HashSet<T>位于*System.Core.dll*，而SortedSet<T>和ISet<T>位于*System.dll*。

HashSet<T>是通过使用只存储键的散列表实现的；而SortedSet<T>则是通过一个红/黑树实现的。

下面是HashSet<T>的定义：

两个集合都实现了ICollection<T>接口，并且有一些常用方法，如Contains、Add和Remove。此外，还有一个基于断言的删除方法RemoveWhere。

下面的语句从现有集合构造一个HashSet<char>，测试其成员，然后列举集合的元素（注意没有重复元素）：

```
var letters = new HashSet<char> ("the quick brown fox");

Console.WriteLine (letters.Contains ('t'));          // true
Console.WriteLine (letters.Contains ('j'));          // false

foreach (char c in letters) Console.Write (c);       // the quickbrownfx
```

（之所以可以给HashSet<char>的构造方法传递一个字符串，是因为字符串实现了IEnumerable<char>接口。）

真正有意思的方法是设置操作。以下设置操作是破坏性的，因为它们修改设置：

```
public void UnionWith           (IEnumerable<T> other); // Adds
public void IntersectWith       (IEnumerable<T> other); // Removes
public void ExceptWith          (IEnumerable<T> other); // Removes
public void SymmetricExceptWith (IEnumerable<T> other); // Removes
```

而下面的方法只是查询集合，所以没有破坏性：

```
public bool IsSubsetOf         (IEnumerable<T> other);
public bool IsProperSubsetOf   (IEnumerable<T> other);
public bool IsSupersetOf       (IEnumerable<T> other);
public bool IsProperSupersetOf (IEnumerable<T> other);
public bool Overlaps           (IEnumerable<T> other);
public bool SetEquals          (IEnumerable<T> other);
```

UnionWith会将第二个集合的所有元素添加到原始集合上（不包含重复元素）。IntersectWith会删除不是两个集合共有的元素。我们用下面的代码提取字符集中所有的元音字母：

```
var letters = new HashSet<char> ("the quick brown fox");
letters.IntersectWith ("aeiou");
foreach (char c in letters) Console.Write (c);        // euio
```

ExceptWith会删除源集合的指定元素。在这里会删除集合中所有的元音字母：

```
var letters = new HashSet<char> ("the quick brown fox");
letters.ExceptWith ("aeiou");
foreach (char c in letters) Console.Write (c);        // th qckbrwnfx
```

SymmetricExceptWith会删除所有元素中不是某个集合独有的元素：

```
var letters = new HashSet<char> ("the quick brown fox");
letters.SymmetricExceptWith ("the lazy brown fox");
foreach (char c in letters) Console.Write (c);        // quicklazy
```

因为HashSet<T>和SortedSet<T>实现了IEnumerable<T>接口，所以可以将另一种集合作为任意集合操作方法的参数：

SortedSet<T>有HashSet<T>的所有成员，以及下面的成员：

```
public virtual SortedSet<T> GetViewBetween (T lowerValue, T upperValue)
public IEnumerable<T> Reverse()
public T Min { get; }
public T Max { get; }
```

SortedSet<T>的构造函数还接受一个可选的IComparer<T>参数（而非一个*等值比较器*）。

下面这个例子将相同的字母加载到一个SortedSet<char>中：

```
var letters = new SortedSet<char> ("the quick brown fox");
foreach (char c in letters) Console.Write (c);        // bcefhiknoqrtuwx
```

可以用下面的方法获得*f*与*j*之间的字母：

```
foreach (char c in letters.GetViewBetween ('f', 'j'))
Console.Write (c);                                    // fhi
```

7.6 字典

字典是一种所包含元素均为键/值对的集合。字典通常都用来执行列表查找和排序。

Framework通过接口IDictionary和IDictionary<TKey, TValue>及一组通用的字典类定义了一个标准字典协议。这些类在以下方面有区别：

- 元素是否按有序序列存储
- 元素是否按位置（索引）或按键访问
- 类是泛型还是非泛型的
- 集合变大时的性能

表7-1总结了所有字典类以及它们在这些方面的区别。性能时间是用毫秒表示的，指的是在1.5 GHz PC上的一个保存整型键和值的字典中执行50,000次操作的时间。使用相同底层集合结构的泛型和非泛

型类的性能区别是由装箱操作引起的，并且只有在值类型元素的集合中才会出现。

表7-1：Dictionary类

类型	内部结构	是否按索引取值	内存负载（每个元素的平均字节数）	速度（随机插入）	速度（顺序插入）	速度（按键检索）
不排序的						
Dictionary<K,V>	散列表	否	22	30	30	20
Hashtable	散列表	否	38	50	50	30
ListDictionary	链表	否	36	50,000	50,000	50,000
OrderedDictionary	散列表 + 数组	是	59	70	70	40
排序的						
SortedDictionary<K,V>	红/黑树	否	20	130	100	120
SortedList<K,V>	二维数组	是	2	3,300	30	40
SortedList	二维数组	是	27	4,500	100	180

在Big-O符号中，按键检索的时间是：

- `Hashtable`、`Dictionary`和`OrderedDictionary`为O(1)。
- `SortedDictionary`和`SortedList`为O(log *n*)。
- `ListDictionary`和非字典类型（如`List<T>`）为O(*n*)。

其中*n*是集合中的元素个数。

7.6.1 IDictionary<TKey,TValue>

`IDictionary<TKey,TValue>`定义了所有基于键/值的集合的标准协议。它扩展了`ICollection<T>`，增加了一些基于任意类型的键访问元素的方法和属性：

```
public interface IDictionary <TKey, TValue> :
  ICollection <KeyValuePair <TKey, TValue>>, IEnumerable
{
  bool ContainsKey (TKey key);
  bool TryGetValue (TKey key, out TValue value);
  void Add         (TKey key, TValue value);
  bool Remove      (TKey key);

  TValue this [TKey key]      { get; set; }        // 主索引器－根据键操作
  ICollection <TKey> Keys     { get; }             // 只返回键
  ICollection <TValue> Values { get; }             // 只返回值
}
```

提示：从Framework 4.5开始，还出现了一个接口IReadOnlyDictionary<TKey,TValue>，它定义了字典成员的只读子集。它与Windows Runtime类型IMapView<K,V>相对应，当时也是因为相同原因而引入的。

要给字典增加一个元素，可以调用Add或者使用索引的set操作方法，后者会在该键不存在时将元素添加到字典中（或者当该键存在时修改该元素）。重复的键在所有字典实现中都是禁止的，所以用相同的键调用两次Add会抛出一个异常。

我们可以使用索引器或者TryGetValue方法，从字典中取出一个元素。如果这个键不存在，那么索引器会抛出一个异常，而TryGetValue会返回false。可以先调用ContainsKey明确地检测成员是否存在；然而，如果检测后再取出该元素，那么一共需要两次查询过程。

直接通过一个IDictionary<TKey,TValue>进行枚举会返回一个KeyValuePair结构体序列：

```
public struct KeyValuePair <TKey, TValue>
{
  public TKey Key     { get; }
  public TValue Value { get; }
}
```

可以通过字典的Keys/Values枚举键或值。

我们将在下一节说明这个接口在处理泛型Dictionary时的用法。

7.6.2 IDictionary

非泛型的IDictionary接口在原理上与IDictionary<TKey,TValue>相同，但是存在以下两个重要的功能区别。我们一定要意识到这些区别，因为一些遗留代码会使用IDictionary（包括.NET Framework本身）：

- 通过索引器查找一个不存在的键会返回null（而不是抛出一个异常）。
- 使用Contains而非ContainsKey来检测成员是否存在。

枚举一个非泛型IDictionary会返回一个DictionaryEntry结构体序列：

```
public struct DictionaryEntry
{
  public object Key   { get; set; }
  public object Value { get; set; }
}
```

7.6.3 Dictionary<TKey,TValue>和Hashtable

泛型Dictionary类（和List<T>集合一样）是使用最广泛的集合之一。它使用一个散列表结构来存储键和值，而且快速、高效。

提示：Dictionary<TKey,TValue>的非泛型版本是Hashtable；Framework中不存在名为Dictionary的非泛型类。当我们提到Dictionary时，指的是泛型的Dictionary<TKey,TValue>类。

Dictionary同时实现了泛型和非泛型的IDictionary接口，而泛型IDictionary是公开的接口。事

实上，Dictionary是泛型IDictionary的一个标准实现。

下面演示了它的用法：

```
var d = new Dictionary<string, int>();

d.Add("One", 1);
d["Two"] = 2;     // 由于"two"不存在，所以会添加到字典中
d["Two"] = 22;    // 由于"two"已经存在，所以会修改字典
d["Three"] = 3;

Console.WriteLine (d["Two"]);                    // 打印 "22"
Console.WriteLine (d.ContainsKey ("One"));       // true（快速的操作）
Console.WriteLine (d.ContainsValue (3));         // true（较慢的操作）
int val = 0;
if (!d.TryGetValue ("onE", out val))
  Console.WriteLine ("No val");                  // "No val"（区分大小写）

// 三个不同的枚举字典的方法：

foreach (KeyValuePair<string, int> kv in d)          // One ; 1
  Console.WriteLine (kv.Key + "; " + kv.Value);      // Two ; 22
                                                     // Three ; 3

foreach (string s in d.Keys) Console.Write (s);      // OneTwoThree
Console.WriteLine();
foreach (int i in d.Values) Console.Write (i);       // 1223
```

它的底层散列表会将每个元素的键转换为一个整型散列码——伪随机值，然后使用算法将散列码转换为一个散列键。这个散列键在内部用来决定元素属于哪一个“桶”。如果这个桶不止包含一个值，那么散列表会在其中执行线性查找。散列表一般会将桶与值的比较控制在1:1（即1:1的负载因数），表示每个桶只包含一个值。然而，随着越来越多的值进入散列表，负载因数会动态地增加，其增长方式是针对插入和查询性能及内存需求进行优化的。

字典支持任何类型的键，能够确定键和所获得的散列码是否相等。默认情况下，相等是通过键的object.Equals方法判断的，而伪随机散列码是通过键的GetHashCode方法获得的。这个行为可以通过重写这些方法或者在创建字典时提供一个IEqualityComparer对象而进行修改。它的一个常见应用就是在使用字符串键时指定一个不区分大小写的等值比较器：

```
var d = new Dictionary<string, int> (StringComparer.OrdinalIgnoreCase);
```

我们将在本章的“等值和顺序插入”中进一步讨论这个问题。

和其他类型的集合一样，字典的性能可以通过在构造函数中指定集合的预期大小而得到改进，从而避免或减少内部调整大小的操作。

非泛型版本是Hashtable，它在功能上类似，区别在于之前讨论的它提供的非泛型IDictionary接口。

Dictionary和Hashtable的缺点是元素是无序的。而且，添加元素时不保存原始顺序。此外，所有字典类型都不允许出现重复键。

提示：当Framework 2.0引入泛型集合时，CLR团队选择根据它们所表示的内容进行命名（Dictionary, List），而不是根据它们的内部实现进行命名（Hashtable, ArrayList）。虽然这种做法很好，因为它们可以在将来自由修改其实现方式，但是这也意味着它们的名称无法反映其性能表现（通常这是选择一种集合的最重要依据）。

7.6.4 OrderedDictionary

OrderedDictionary是一种非泛型字典，它能够保存添加元素时的原始顺序。通过使用OrderedDictionary，既可以根据索引访问元素，也可以根据键进行访问。

提示：OrderedDictionary并不是一个有序的字典。

OrderedDictionary是Hashtable和ArrayList的组合。因此它具有Hashtable的所有功能，还有诸如RemoveAt以及整数索引器等功能。它还具有Keys和Values属性，可以按原始顺序返回元素。

这个类是在.NET 2.0中引入的，特殊的是，它没有泛型版本。

7.6.5 ListDictionary和HybridDictionary

ListDictionary使用一个独立链表来存储实际的数据。虽然它能够保存添加元素时的原始顺序，但是它不支持排序。ListDictionary在处理较大列表时速度非常慢。它存在的真正意义是它在处理非常小的列表（小于10个元素）时效率很高。

HybridDictionary是一个在达到一定大小后会自动转换为Hashtable的ListDictionary，目的是为了解决ListDictionary的性能问题。这个概念的产生主要是为了提高字典很小时的内存使用效率以及字典变大时维持良好性能。然而，考虑到中间存在转换的负载，而且Dictionary在这两种情况下都不会太重或太慢，所以即使直接使用Dictionary也是合理的。

ListDictionary和HybridDictionary这两个类都只有非泛型形式。

7.6.6 排序的字典

Framework只支持两种在内部结构中将内容根据键进行排序的字典：

- SortedDictionary<TKey,TValue>
- SortedList<TKey,TValue>（注1）

（在这一节中，我们把<TKey,TValue>缩写为<,>。）

SortedDictionary<,>使用一个红/黑树数据结构：这种结构在任何插入或检索情况中都采用一致的执行方式。

SortedList<,>在内部是通过一个有序数组实现的，具有很高的读取效率（通过一个二叉查找），但是插入性能很差，因为必须移动现有的值才能够留出空间存储新的元素。

SortedDictionary<,>在随机序列（特别是大型列表）中插入元素的速度比SortedList<,>快很多。然而，SortedList<,>也有突出的功能，即按索引和按键访问元素。可以使用一个排序列表直接访问排序序列的第n个元素（通过键/值属性的索引器）。如果要使用SortedDictionary<,>实现相同的操作，必须手动遍历n个元素。（此外，可以编写一个类来组合使用一个排序字典和一个列表类。）

这三种集合都不允许出现重复键（这与所有字典类一样）。

注1：SortedList是具有相同功能的非泛型版本。

下面的例子使用反射机制将System.Object中的所有方法加载到一个以名称为键的排序列表中，然后枚举出它们的键和值：

```
// MethodInfo位于System.Reflection命名空间

var sorted = new SortedList <string, MethodInfo>();

foreach (MethodInfo m in typeof (object).GetMethods())
  sorted [m.Name] = m;

foreach (string name in sorted.Keys)
  Console.WriteLine (name);

foreach (MethodInfo m in sorted.Values)
  Console.WriteLine (m.Name + " returns a " + m.ReturnType);
```

第一个枚举的结果是：

```
Equals
GetHashCode
GetType
ReferenceEquals
ToString
```

第二个枚举的结果是：

```
Equals returns a System.Boolean
GetHashCode returns a System.Int32
GetType returns a System.Type
ReferenceEquals returns a System.Boolean
ToString returns a System.String
```

注意我们是通过它的索引器来填充字典的。如果使用Add方法，那么它会抛出一个异常，因为我们所反射的object类重载Equals方法，所以无法两次将相同的键添加到一个字典中。通过使用索引器，后面进入的元素会重写前面的元素，从而不会出现错误。

提示： 通过将每个值元素变成一个列表，可以用相同的键存储多个成员：

```
SortedList <string, List<MethodInfo>>
```

对我们的例子进行扩展，下面的代码会查询键为"GetHash Code"的MethodInfo，这和普通的字典是一样的：

```
Console.WriteLine (sorted ["GetHashCode"]);     // Int32 GetHashCode()
```

目前，我们所做的每一个操作都支持SortedDictionary<,>。然而，下面两行查询最后一个键和值的代码只适用于排序列表：

```
Console.WriteLine (sorted.Keys [sorted.Count - 1]);            // ToString
Console.WriteLine (sorted.Values[sorted.Count - 1].IsVirtual);  // True
```

7.7 可定制的集合和委托

前一节所讨论的集合类都可以直接实例化，所以使用很方便。但是无法控制当集合添加或删除一个元

素时所发生的变化。在应用程序中使用强类型化集合时，有时会需要进行以下控制：

- 当添加或删除一个元素时触发一个事件
- 当添加或删除一个元素时更新一些属性
- 检测不合法的添加/删除操作，并抛出一个异常（例如，如果操作违反商业规则）

.NET Framework在System.Collections.ObjectModel命名空间中提供了一些专门针对以上问题的集合类。实际上是一些通过将方法转发到一个底层集合而实现IList<T>或IDictionary<,>的委托或包装。每一个Add、Remove或Clear操作都会通过一个虚方法执行，作为重写时的“入口”。

可定制的集合类通常用在一些公开的集合，例如，在System.Windows.Form类上公开的一系列控制。

7.7.1 Collection<T>和CollectionBase

Collection<T>类是一个可定制的List<T>包装类。

除了实现IList<T>和IList，Collection<T>还定义了四个虚方法和一个受保护的属性，如下所示：

```
public class Collection<T> :
  IList<T>, ICollection<T>, IEnumerable<T>, IList, ICollection, IEnumerable
{
  // ...

  protected virtual void ClearItems();
  protected virtual void InsertItem (int index, T item);
  protected virtual void RemoveItem (int index);
  protected virtual void SetItem (int index, T item);

  protected IList<T> Items { get; }
}
```

这些虚方法提供了一个入口，可以通过这个入口“强行”修改或改进列表的普通行为。通过这个受保护的Items属性可以直接访问内部列表，而不需要使用虚方法，可用于执行一些内部修改。

这些虚方法不需要重写，它们可以保持不变，直到需要修改列表的默认行为。下面的例子说明了Collection<T>的典型使用方法：

```
public class Animal
{
  public string Name;
  public int Popularity;

  public Animal (string name, int popularity)
  {
    Name = name; Popularity = popularity;
  }
}

public class AnimalCollection : Collection <Animal>
{
  // AnimalCollection已经是一个功能完整的动物列表。这里不需要额外的代码。
}

public class Zoo       // 这个类将显示AnimalCollection
{                      // 这里一般添加其他一些成员
  public readonly AnimalCollection Animals = new AnimalCollection();
```

```
}
class Program
{
  static void Main()
  {
    Zoo zoo = new Zoo();
    zoo.Animals.Add (new Animal ("Kangaroo", 10));
    zoo.Animals.Add (new Animal ("Mr Sea Lion", 20));
    foreach (Animal a in zoo.Animals) Console.WriteLine (a.Name);
  }
}
```

按照这种方式，AnimalCollection在功能上并不比一个简单的List<Animal>多，它的作用是提供一个基类以便扩展。为了说明这一点，我们给Animal添加一个Zoo属性，因此它能够引用它所在的Zoo，以及重载Collection<Animal>中的每一个虚方法，以便自动维护这个属性：

```
public class Animal
{
  public string Name;
  public int Popularity;
  public Zoo Zoo { get; internal set; }
  public Animal(string name, int popularity)
  {
    Name = name; Popularity = popularity;
  }
}

public class AnimalCollection : Collection <Animal>
{
  Zoo zoo;
  public AnimalCollection (Zoo zoo) { this.zoo = zoo; }
  protected override void InsertItem (int index, Animal item)
  {
    base.InsertItem (index, item);
    item.Zoo = zoo;
  }
  protected override void SetItem (int index, Animal item)
  {
    base.SetItem (index, item);
    item.Zoo = zoo;
  }
  protected override void RemoveItem (int index)
  {
    this [index].Zoo = null;
    base.RemoveItem (index);
  }
  protected override void ClearItems()
  {
    foreach (Animal a in this) a.Zoo = null;
    base.ClearItems();
  }
}

public class Zoo
{
  public readonly AnimalCollection Animals;
```

```
    public Zoo() { Animals = new AnimalCollection (this); }
  }
```

Collection<T>也具有一个接受已有IList<T>的构造函数。与其他集合类不同的是，它所提供的列表是委托的，而非复制的，所以所做的修改将会反映在封装的Collection<T>中（虽然没有触发Collection<T>的虚函数）。相反，通过Collection<T>所做的修改将会修改底层的列表。

CollectionBase

CollectionBase是Framework 1.0引入的Collection<T>的非泛型版本。它提供了大多数与Collection<T>相似的特性，但是使用方式不太灵活。CollectionBase没有模板方法InsertItem、RemoveItem、SetItem和ClearItem，但是有一些“钩子”方法可以满足各种需要，包括OnInsert、OnInsertComplete、OnSet、OnSetComplete、OnRemove、OnRemoveComplete、OnClear和OnClearComplete。因为CollectionBase是非泛型的，所以在继承它时还必须实现类型化方法，至少需要一个类型化索引器和Add方法。

7.7.2 KeyedCollection<TKey,TItem>和DictionaryBase

KeyedCollection<TKey,TItem>是Collection<TItem>的子类。它增加也删去了一些功能。它增加的功能是按键访问元素，这与字典很相似；删去的功能是委托自己的内部列表。

使用键的集合与OrderedDictionary相似之处是，它们都组合使用线性列表和散列表。然而，与OrderedDictionary不同的是，它并没有实现IDictionary，也不支持键/值对的概念。相反，键是通过提供的GetKeyForItem方法从元素本身获得的。这意味着枚举一个使用键的集合就像是在枚举一个普通的列表。

KeyedCollection<TKey,TItem>通常看作是实现了按键进行快速查找的Collection<TItem>。

因为它继承了Collection<>，所以使用键的集合继承了Collection<>的所有功能，除了在创建时指定一个现有列表。它定义的其他成员有：

```
public abstract class KeyedCollection <TKey, TItem> : Collection <TItem>

  // ...

  protected abstract TKey GetKeyForItem(TItem item);
  protected void ChangeItemKey(TItem item, TKey newKey);

  // 按键进行快速查找，这是对根据索引进行查找的补充。
  public TItem this[TKey key] { get; }

  protected IDictionary<TKey, TItem> Dictionary { get; }
}
```

GetKeyForItem是实现者重载以便从底层对象获得元素的键。如果元素的键属性发生变化，必须调用ChangeItemKey方法，才能更新内部字典。Dictionary属性会返回用于实现查找的内部字典，这是在添加第一个元素时创建的。这个行为可以通过在构造函数中指定一个生成临界值进行修改，这使内部字典的修改延迟到达到这个临界值时才执行（在这期间，如果一个元素是按照键请求的，那么它会执行一次线性查找）。不指定生成临界值的理由是使用有效字典通过Dictionary的Keys属性获得ICollection<>的键是很有用的，然后这个集合可以传递到一个公开的属性。

KeyedCollection<,>通常用来实现按照索引或名称访问元素集合。为了说明这一点，我们将再次使

集合

用动物园例子，将AnimalCollection实现为一个KeyedCollection<string,Animal>：

```
public class Animal
{
  string name;
  public string Name
  {
    get { return name; }
    set {
  if (Zoo != null) Zoo.Animals.NotifyNameChange (this, value);
  name = value;
    }
}

  public int Popularity;
  public Zoo Zoo { get; internal set; }

  public Animal (string name, int popularity)
  {
    Name = name; Popularity = popularity;
  }
}

public class AnimalCollection : KeyedCollection <string, Animal>
{
  Zoo zoo;
  public AnimalCollection (Zoo zoo) { this.zoo = zoo; }

  internal void NotifyNameChange (Animal a, string newName)
  {
    this.ChangeItemKey (a, newName);
  }

  protected override string GetKeyForItem (Animal item)
  {
    return item.Name;
  }

  // 以下方法的实现同前面的例子
  protected override void InsertItem (int index, Animal item)...
  protected override void SetItem (int index, Animal item)...
  protected override void RemoveItem (int index)...
  protected override void ClearItems()...
}

public class Zoo
{
  public readonly AnimalCollection Animals;
  public Zoo() { Animals = new AnimalCollection (this); }
}

class Program
{
  static void Main()
  {
    Zoo zoo = new Zoo();
    zoo.Animals.Add (new Animal ("Kangaroo", 10));
    zoo.Animals.Add (new Animal ("Mr Sea Lion", 20));
    Console.WriteLine (zoo.Animals [0].Popularity);               // 10
    Console.WriteLine (zoo.Animals ["Mr Sea Lion"].Popularity);   // 20
```

```
    zoo.Animals ["Kangaroo"].Name = "Mr Roo";
    Console.WriteLine (zoo.Animals ["Mr Roo"].Popularity);    // 10
  }
}
```

DictionaryBase

KeyedCollection的非泛型版本称为DictionaryBase。这个类采用不同的实现方法：它实现了IDictionary，并使用一些类似于CollectionBase的钩子方法：OnInsert、OnInsertComplete、OnSet、OnSetComplete、OnRemove、OnRemoveComplete、OnClear和OnClearComplete（及额外的OnGet）。实现IDictionary相对于KeyedCollection方法的主要优点是不需要创建它的子类来获得键。但是，由于DictionaryBase的真正用途是用来创建子类，所以这个优点没有任何意义。KeyedCollection中改进的模型因为出现的时间晚，从而具有后来的优势。DictionaryBase存在的目的就是为了向后兼容。

7.7.3 ReadOnlyCollection<T>

ReadOnlyCollection<T>是一个包装器，或者称为委托，它提供了集合的一种只读视图。它的用处是允许一个类公开地显示集合的只读访问，但是同时这个类仍然可以在内部进行修改。

一个只读的集合可以在它的构造函数中接受输入集合，并持久引用。它不会静态地复制输入集合，所以输入集合的后续修改都会通过只读包装器显示出来。

为了说明这一点，假设类提供一个名为Names的字符串列表的公开只读访问：

```
public class Test
{
  public List<string> Names { get; private set; }
}
```

这仅仅完成了一半的工作。虽然其他类型无法给Names属性重新赋值，但是仍然可以调用列表的Add、Remove或Clear。ReadOnlyCollection<T>能够解析这个类：

```
public class Test
{
  List<string> names;
  public ReadOnlyCollection<string> Names { get; private set; }

  public Test()
  {
    names = new List<string>();
    Names = new ReadOnlyCollection<string> (names);
  }

  public void AddInternally() { names.Add ("test"); }
}
```

现在，只有Test类的成员能够修改名称列表：

```
Test t = new Test();

Console.WriteLine (t.Names.Count);         // 0
t.AddInternally();
Console.WriteLine (t.Names.Count);         // 1

t.Names.Add ("test");                      // 编译错误
((IList<string>) t.Names).Add ("test");    // NotSupportedException
```

7.8 等值和顺序插入

在第6章的“等值比较”和“顺序比较”中，我们介绍使一个类型可相等、可散列和可比较的.NET标准协议。实现这些协议的类型可以在一个“开箱即用”的字典或排序列表中正确发挥功能。更具体的是：

- 一个Equals和GetHashCode返回的有意义的结果的类型可用作字典或Hashtable的键。
- 一个实现了IComparable/IComparable<T>的类型可用作任意排序字典或列表的键。

一个类型的默认等值或比较实现一般都反映了该类型最基本特性，但是它的默认行为并非是你想要的。你可能需要这样一个字典，它的字符串类型键是不区分大小写的。或者你可能希望有一个排序的顾客列表，它按照每个顾客的邮政编码进行排序。为此，.NET Framework还定义了一组相匹配的“插件”协议。这些插件协议实现了以下两个功能：

- 它们允许使用替代的等值或比较行为。
- 它们允许使用一个本质上键类型不可相等或不可比较的字典或排序集合。

这些插件协议由以下接口构成：

IEqualityComparer*和*IEqualityComparer<T>

- 执行插入等值比较和散列化
- 由Hashtable和Dictionary识别

IComparer*和*IComparer<T>

- 执行插入*顺序比较*
- 由排序字典和集合识别；以及Array.Sort

每一个接口都有泛型和非泛型形式。IEqualityComparer接口还具有一个名为EqualityComparer的默认实现类。

此外，Framework 4.0还新增加了两个名为IStructuralEquatable和IStructuralComparable的接口，用于在类和数组中进行结构比较。

7.8.1 IEqualityComparer和EqualityComparer

等值比较器会使用非默认等值和散列行为，主要是针对Dictionary和Hashtable类。

回顾一下基于散列表的字典的要求。它需要解决任意指定键的以下两个问题：

- 它与另一个键是否相同？
- 它的整数散列码是什么？

等值比较器通过实现IEqualityComparer接口解决这两个问题：

```
public interface IEqualityComparer<T>
{
  bool Equals (T x, T y);
  int GetHashCode (T obj);
}
```

```
public interface IEqualityComparer       // 非泛型版本
{
  bool Equals (object x, object y);
  int GetHashCode (object obj);
}
```

如果要编写自定义比较器，需要实现一个或多个此类接口（同时实现两个接口可以得到最大的互操作性）。这种方法有一些复杂，替代方法是创建抽象类EqualityComparer的子类，它的定义如下所示：

```
public abstract class EqualityComparer<T> : IEqualityComparer,
                                            IEqualityComparer<T>
{
  public abstract bool Equals (T x, T y);
  public abstract int GetHashCode (T obj);

  bool IEqualityComparer.Equals (object x, object y);
  int IEqualityComparer.GetHashCode (object obj);

  public static EqualityComparer<T> Default { get; }
}
```

EqualityComparer实现了两个接口；需要重写两个抽象方法。

Equals和GetHashCode的语义与第6章介绍的object.Equals和object.GetHashCode采用了相同的规则。在下面的例子中，我们定义了一个带有两个域的Customer类，然后编写一个等值比较器来同时匹配姓与名：

```
public class Customer
{
  public string LastName;
  public string FirstName;

  public Customer (string last, string first)
  {
    LastName = last;
    FirstName = first;
  }
}

public class LastFirstEqComparer : EqualityComparer <Customer>
{
  public override bool Equals (Customer x, Customer y)
  {
    return x.LastName == y.LastName && x.FirstName == y.FirstName;
  }

  public override int GetHashCode (Customer obj)
  {
    return (obj.LastName + ";" + obj.FirstName).GetHashCode();
  }
}
```

为了演示它的工作原理，我们创建以下两个顾客：

```
Customer c1 = new Customer ("Bloggs", "Joe");
Customer c2 = new Customer ("Bloggs", "Joe");
```

因为我们还没有重写object.Equals，所以这里使用普通的引用类型等值语义：

```
Console.WriteLine (c1 == c2);                // False
Console.WriteLine (c1.Equals (c2));          // False
```

当在Dictionary中使用这些顾客而不指定等值比较器时，应用相同的默认等值语义：

```
var d = new Dictionary<Customer, string>();
d [c1] = "Joe";
Console.WriteLine (d.ContainsKey (c2));      // False
```

现在我们使用自定义的等值比较器：

```
var eqComparer = new LastFirstEqComparer();
var d = new Dictionary<Customer, string> (eqComparer);
d [c1] = "Joe";
Console.WriteLine (d.ContainsKey (c2));      // True
```

在这个例子中，需要注意的是，不能在字典使用顾客的FirstName或LastName时对它们进行修改。否则，它的散列码会发生变化，Dictionary会被破坏。

EqualityComparer<T>.Default

调用EqualityComparer<T>.Default会返回一个通用的等值比较器，它可以替代静态的object.Equals方法。它的优点是首先检查T是否实现了IEquatable<T>，如果是，那么就调用实现类，从而避免出现装箱负载。这在泛型方法中特别有用：

```
static bool Foo<T> (T x, T y)
{
  bool same = EqualityComparer<T>.Default.Equals (x, y);
...
```

7.8.2 IComparer和Comparer

比较器用于接入排序字典和集合的自定义顺序逻辑。

注意，比较器对于不排序的字典是无用的，如Dictionary和Hashtable，这些字典需要用一个IEqualityComparer来获取散列码。类似地，等值比较器对于排序字典和集合也是没有用处的。

下面是IComparer接口的定义：

```
public interface IComparer
{
  int Compare(object x, object y);
}
public interface IComparer <in T>
{
  int Compare(T x, T y);
}
```

与等值比较符一样，可以选择创建一个抽象类的子类，而不需要实现这些接口：

```
public abstract class Comparer<T> : IComparer, IComparer<T>
{
  public static Comparer<T> Default { get; }
```

```
  public abstract int Compare (T x, T y);      // 自行实现
  int IComparer.Compare (object x, object y); // 已经实现
}
```

下面的例子演示了一个描述愿望的类，以及一个按优先级对愿望进行排序的比较器：

```
class Wish
{
  public string Name;
  public int Priority;

  public Wish (string name, int priority)
  {
    Name = name;
    Priority = priority;
  }
}

class PriorityComparer : Comparer <Wish>
{
  public override int Compare (Wish x, Wish y)
  {
    if (object.Equals (x, y)) return 0;      // 异常安全检测
    return x.Priority.CompareTo (y.Priority);
  }
}
```

object.Equals检查会保证绝不与Equals方法产生冲突。在这里调用静态的object.Equals比调用x.Equals更好，这是因为当x为null时，它仍然有效！

下面例子介绍PriorityComparer是如何用来对一个List执行排序的：

```
var wishList = new List<Wish>();
wishList.Add (new Wish ("Peace", 2));
wishList.Add (new Wish ("Wealth", 3));
wishList.Add (new Wish ("Love", 2));
wishList.Add (new Wish ("3 more wishes", 1));

wishList.Sort (new PriorityComparer());
foreach (Wish w in wishList) Console.Write (w.Name + " | ");

// 输出: 3 more wishes | Love | Peace | Wealth |
```

在下一个例子中，SurnameComparer允许对姓氏字符串进行排序，以便供电话簿清单使用：

```
class SurnameComparer : Comparer <string>
{
  string Normalize (string s)
  {
    s = s.Trim().ToUpper();
    if (s.StartsWith ("MC")) s = "MAC" + s.Substring (2);
    return s;
  }

  public override int Compare (string x, string y)
  {
    return Normalize (x).CompareTo (Normalize (y));
  }
}
```

下面使用的SurnameComparer位于一个排序字典中：

```
var dic = new SortedDictionary<string,string> (new SurnameComparer());
dic.Add ("MacPhail", "second!");
dic.Add ("MacWilliam", "third!");
dic.Add ("McDonald", "first!");

foreach (string s in dic.Values)
  Console.Write (s + " ");          // first! second! third!
```

7.8.3 StringComparer

StringComparer是一个支持等值和比较字符串的预定义插件类，它允许指定语言和是否区分大小写。StringComparer同时实现了IEqualityComparer和IComparer（及其泛型版本），所以它可以与任何类型的字典或排序集合一起使用：

```
// CultureInfo 定义在System.Globalization命名空间中

public abstract class StringComparer : IComparer, IComparer <string>,
                                       IEqualityComparer,
                                       IEqualityComparer <string>
{
  public abstract int Compare (string x, string y);
  public abstract bool Equals (string x, string y);
  public abstract int GetHashCode (string obj);

  public static StringComparer Create (CultureInfo culture,
                                       bool ignoreCase);
  public static StringComparer CurrentCulture { get; }
  public static StringComparer CurrentCultureIgnoreCase { get; }
  public static StringComparer InvariantCulture { get; }
  public static StringComparer InvariantCultureIgnoreCase { get; }
  public static StringComparer Ordinal { get; }
  public static StringComparer OrdinalIgnoreCase { get; }
}
```

因为StringComparer是抽象的，所以必须通过它的静态方法和属性来获得一个实例。StringComparer.Ordinal会复制字符串的默认行为和StringComparer.CurrentCulture的比较方式。

在下一个例子中，我们创建了一个顺序的不区分大小写的字典，例如dict["Joe"]和dict["JOE"]表示相同的信息：

```
var dict = new Dictionary<string, int> (StringComparer.OrdinalIgnoreCase);
```

在下一个例子中，数组的名称是按照澳大利亚英语进行排序的：

```
string[] names = { "Tom", "HARRY", "sheila" };
CultureInfo ci = new CultureInfo ("en-AU");
Array.Sort<string> (names, StringComparer.Create (ci, false));
```

最后一个例子是之前介绍的与文化相关的SurnameComparer类（以便适合电话簿清单的名称）。

```
class SurnameComparer : Comparer <string>
{
  StringComparer strCmp;
```

```
  public SurnameComparer (CultureInfo ci)
  {
    // 创建一个区分大小写、与文化相关的字符比较器
    strCmp = StringComparer.Create (ci, false);
  }

  string Normalize (string s)
  {
    s = s.Trim();
    if (s.ToUpper().StartsWith ("MC")) s = "MAC" + s.Substring (2);
    return s;
  }

  public override int Compare (string x, string y)
  {
    // 直接在与文化相关的环境中调用
    return strCmp.Compare (Normalize (x), Normalize (y));
  }
}
```

7.8.4 IStructuralEquatable和IStructuralComparable

正如第6章所介绍的，结构体默认实现*结构比较*：如果所有字段都相等，那么两个结构必然是相等的。但是，有时结构等值和顺序比较在其他类型的插件方法中也是很有用的，例如，数组和元组。Framework 4.0引入了两个新的接口来实现这一点：

```
public interface IStructuralEquatable
{
  bool Equals (object other, IEqualityComparer comparer);
  int GetHashCode (IEqualityComparer comparer);
}

public interface IStructuralComparable
{
  int CompareTo (object other, IComparer comparer);
}
```

传递的IEqualityComparer/IComparer都会应用到组合对象的每一个元素中。我们可以使用数组和元组来演示它们实现这些接口。在下面的例子中，我们比较了两个数组的等值性：首先使用默认的Equals方法，然后使用IStructuralEquatable的Equals方法。

```
int[] a1 = { 1, 2, 3 };
int[] a2 = { 1, 2, 3 };
Console.Write (a1.Equals (a2));                                    // False
Console.Write (a1.Equals (a2, EqualityComparer<int>.Default));     // True
```

下面是另一个例子：

```
string[] a1 = "the quick brown fox".Split();
string[] a2 = "THE QUICK BROWN FOX".Split();
bool isTrue = a1.Equals (a2, StringComparer.InvariantCultureIgnoreCase);
```

元组也采用相同的方式：

```
var t1 = Tuple.Create (1, "foo");
var t2 = Tuple.Create (1, "FOO");
```

集合

```
IStructuralEquatable se1 = t1;
bool isTrue = t1.Equals (t2, StringComparer.InvariantCultureIgnoreCase);
IStructuralComparable sc1 = t1;
int zero = t1.CompareTo (t2, StringComparer.InvariantCultureIgnoreCase);
```

与元组的区别是它们的默认等值和顺序比较实现也应用了结构比较：

```
var t1 = Tuple.Create (1, "FOO");
var t2 = Tuple.Create (1, "FOO");
Console.WriteLine (t1.Equals (t2)); // True
```

第8章 LINQ查询

LINQ是Language Integrated Query的简写，它可以被视为一组语言和框架特性的集合，我们可以使用LINQ对本地对象和远程数据源进行结构化的类型安全的查询操作。在C#3.0和Framework 3.5中引入了LINQ。

LINQ可用于查询任何实现了IEnumerable<T>接口的集合类型，无论是数组、列表还是XML DOM以及远程数据源，例如SQL Server数据库中的表。LINQ具有编译时的类型检查及动态查询组合这两大优点。

在本章中我们主要讨论LINQ的架构及查询语句的基本写法。LINQ中所有核心类型都包含在System.Linq和System.Linq.Expressions这两个命名空间中。

提示： 本章及其后两章中关于LINQ的所有示例都在LINQPad中进行调试。可以在*www.linqpad.net*下载。

8.1 入门

LINQ数据源的基本组成部分是序列和元素。在这里，序列是指任何实现了IEnumerable<T>接口的对象，其中的每一项则称为一个元素。在下面这个示例中，names就是一个序列，而其中的Tom、Dick和Harry就是这个序列中的元素：

```
string[] names = { "Tom", "Dick", "Harry" };
```

names表示内存中的一个对象集合，在这里我们可以称之为*本地序列*。

*查询运算符*是LINQ中用于转换序列的方法。通常，查询运算符可接收一个*输入序列*，并将其转换为一个*输出序列*。在System.Linq命名空间的Enumerable类中定义了约40种查询运算符，这些运算符都是以静态扩展方法的形式来实现的，称为*标准查询运算符*。

提示： 我们把对本地序列进行的查询操作称为本地查询或者是LINQ到对象查询。

LINQ还支持对那些从远程数据源（如SQL Server）中动态获取的序列进行查询。这些序列需要实现IQueryable<T>接口，而在Queryable类中则有一组相应的标准查询运算符对其进行支持。关于这方面的内容，在本章的“解释型的查询”一节中进行详述。

一个查询可以理解为一个使用查询运算符对所操作的序列进行转换的表达式。最简单的查询包含一个输入序列和一个运算符。例如，我们可以使用Where运算符找出一个简单数组中字符个数大于4的那些元素，如下所示：

```
string[] names = { "Tom", "Dick", "Harry" };
IEnumerable<string> filteredNames = System.Linq.Enumerable.Where
                                    (names, n => n.Length >= 4);
foreach (string n in filteredNames)
  Console.WriteLine(n);
```

输出的结果是：

Dick
Harry

由于标准查询运算符都是以静态扩展方法的方式来实现的，因此我们可以像使用对象的实例方法那样直接在names之上调用Where：

```
IEnumerable<string> filteredNames = names.Where (n => n.Length >= 4);
```

这段代码要顺利通过编译，需要导入System.Linq命名空间，下面是这个示例完整的代码：

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

class LinqDemo
{
  static void Main()
  {
    string[] names = { "Tom", "Dick", "Harry" };

    IEnumerable<string> filteredNames = names.Where (n => n.Length >= 4);
    foreach (string name in filteredNames)
      Console.WriteLine(name);
  }
}
```

Dick
Harry

提示：实际上我们可以通过声明隐式类型的filteredNames来进一步精简代码：

```
var filteredNames = names.Where ( n => n.Length >= 4);
```

但这样做会降低可读性，并且在IDE以外的环境下将无法获得自动提示信息。

因此在本章中，我们会尽量避免使用隐式类型来定义查询结果，除非确有必要（我们将在“映射策略”小节中进行讨论）或者当查询结果类型不是现有的类型时，也可以使用隐式类型来定义结果集。

大多数查询运算符都接受一个Lambda表达式作为参数。Lambda表达式用于对查询进行格式化。本例中的Lambda表达式如下：

```
n => n.Length >= 4
```

这里输入参数对应于一个输入元素。本例中，输入参数n表示数组中的每一个名字，其类型是string。Where运算符要求这个Lambda表达式返回一个bool值，如果返回值是true，说明当前元素应该包含在输出序列里。下面是Where运算符的方法签名：

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)
```

下面这个查询会返回那些包含字母“a”的名字：

```
IEnumerable<string> filteredNames = names.Where (n => n.Contains ("a"));

foreach (string name in filteredNames)
  Console.WriteLine (name);            // Harry
```

到目前为止，我们使用扩展方法（Where）以及Lambda表达式编写LINQ查询语句。这种查询方式是可以被拼接起来连续使用的，在本书中，我们称这种用法为*运算符流语法*（注1）。C#还提供了另外一种语法来编写LINQ语句，这种写法称为*查询表达式*语法。使用查询表达式，我们可以将之前的LINQ语句编写成：

```
IEnumerable<string> filteredNames = from n in names
                                    where n.Contains ("a")
                                    select n;
```

运算符流语法和查询表达式语法是两种互补的LINQ表达方式。我们会在接下来的两节中深入探讨这两种书写方式。

8.2 运算符流语法

运算符流是最基本同时也是最灵活的书写LINQ表达式的方式。在这一节中，我们将介绍如何连续使用多个查询运算符来构造更复杂的查询语句，从中我们可以看到扩展方法的强大功能。此外，还会介绍Lambda表达式的书写规则及几个新的查询运算符。

8.2.1 连续使用查询运算符

在前一节中，我们创建了两个简单的LINQ查询表达式，每个表达式只包含一个查询运算符。如果想创建更复杂的查询表达式，只需在前面的表达式后面添加新的查询运算符。为了说明这种用法，以下查询从这个集合中查出所有包含字母“a”的元素，并且将这些元素按字符串长度从小到大排序，再把这些元素转化成大写字母后输出：

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

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

注1：运算符流语法（Fluent Syntax）来自于Eric Evans和Martin Fowler有关运算符流接口（Fluent Interface）的工作结果。

```
    string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

    IEnumerable<string> query = names
      .Where(n => n.Contains("a"))
      .OrderBy(n => n.Length)
      .Select(n => n.ToUpper());

    foreach (string name in query)
    Console.WriteLine(name);
  }
}

JAY
MARY
HARRY
```

提示： 上面示例中的变量n在每个Lambda表达式中都是私有的。可以用下面示例中的c说明这三个n之间的关系：

```
void Test()
{
  foreach (char c in "string1") Console.Write(c);
  foreach (char c in "string2") Console.Write(c);
  foreach (char c in "string3") Console.Write(c);
}
```

Where、OrderBy和Select都是标准的LINQ查询运算符，它们对应着Enumerable类中的三个扩展方法（当然需要导入System.Linq命名空间）。

我们前面已经介绍过Where运算符的作用，它会筛选传进来的输入序列；OrderBy运算符会对传过来的集合进行排序；而Select运算符的作用是将集合转换或者映射成Lambda表达式中指定的形式（在这个示例中是n.ToUpper()即转换为大写形式）输出。在整个查询过程中，数据是从左到右依次被各个查询运算符处理的，所以names中的数据的操作顺序是：首先筛选，再排序，最后转换成大写形式。

警告： 查询运算符绝不会修改输入序列，相反，它会返回一个新序列。这种设计是符合函数式编程规范的，LINQ的思想实际上就起源于*函数式编程*。

下面是这三种查询运算符所对应的扩展方法的签名（这里对OrderBy的签名稍微进行了简化）：

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)

public static IEnumerable<TSource> OrderBy<TSource,TKey>
  (this IEnumerable<TSource> source, Func<TSource,TKey> keySelector)

public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
```

当我们以上面示例中的方式连续使用三个查询运算符时，可以将各个运算符的处理过程理解成一个传送带，从中可以更直观地看出每个运算符的输入及输出，如图8-1所示。

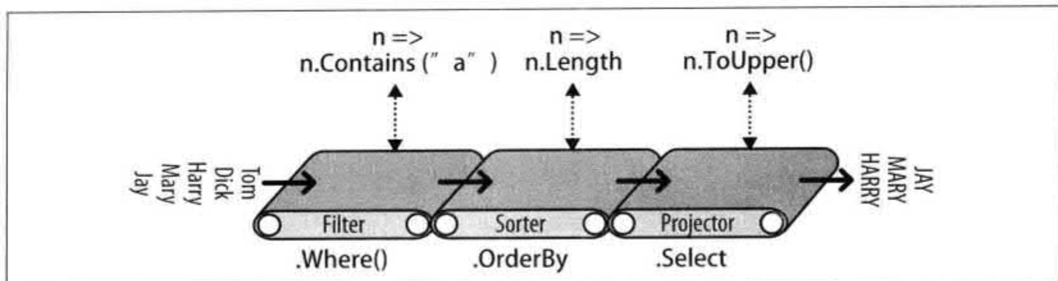


图8-1：连续使用查询运算符

当然我们也可以分三步完成以上三项操作，而不用把它们放在同一个表达式中，分步写的方式如下：

```
//首先必须导入System.Linq命名空间

IEnumerable<string> filtered = names .Where (n => n.Contains ("a"));
IEnumerable<string> sorted = filtered.OrderBy (n => n.Length);
IEnumerable<string> finalQuery = sorted .Select (n => n.ToUpper());
```

这里的finalQuery中会得到与之前的query相同的查询结果，这个结果是在前两项操作（filtered、sorted）的基础上得到的。实际上，这三步中每步都可以得到一个可用的集合，例如：

```
foreach (string name in filtered)
  Console.Write (name + "|");  // Harry|Mary|Jay|

Console.WriteLine();
foreach (string name in sorted)
  Console.Write (name + "|"); // Jay|Mary|Harry|

Console.WriteLine();
foreach (string name in finalQuery)
  Console.Write (name + "|"); // JAY|MARY|HARRY|
```

扩展方法的重要性

前面我们提到，每个查询运算符对应着一个扩展方法。前面的示例中我们都是使用查询运算符对集合进行操作，实际上，也可以直接使用这些扩展方法来完成查询，例如：

```
IEnumerable<string> filtered = Enumerable.Where (names,
                                       n => n.Contains ("a"));
IEnumerable<string> sorted = Enumerable.OrderBy (filtered, n => n.Length);
IEnumerable<string> finalQuery = Enumerable.Select (sorted,
                                         n => n.ToUpper());
```

实际上这也是编译器在遇到查询运算符时的处理方式，把它们编译成对应的函数进行调用。但是，这种调用方式是不可取的。举个示例来说，如果我们想在一个LINQ表达式中使用多个运算符，可以这样写：

```
IEnumerable<string> query = names.Where   (n => n.Contains ("a"))
                                 .OrderBy (n => n.Length)
                                 .Select  (n => n.ToUpper());
```

这种写法可以很清楚地表示出数据是从左到右依次处理的，但是如果直接调用Enumerable类中的方法来完成上面这个查询，表达式就不再那么容易理解了：

```
IEnumerable<string> query =
  Enumerable.Select (
    Enumerable.OrderBy (
      Enumerable.Where (
        names, n => n.Contains ("a")
      ), n => n.Length
   ), n => n.ToUpper()
);
```

8.2.2 使用Lambda表达式

在前面的示例中，我们把如下Lambda表达式作为参数传给了Where运算符：

```
n => n.Contains ("a")        // 输入类型为string，输出类型为bool.
```

提示： 返回一个bool值的表达式我们称之为“*断言*”。

对于不同的查询运算符来说，Lambda表达式的作用也不同。在Where运算符中，Lambda表达式用于判断一个元素是否应该被包含在输出序列中，也就是判断元素是否符合筛选条件；在OrderBy运算符中，Lambda表达式用于将集合中的每个元素映射到它的排序键上；在Select运算符中，Lambda表达式用于定义输入序列以何种格式输出。

提示： 查询运算符的Lambda表达式针对的是集合中的每个元素，而不是集合整体。

实际上，运算符会自动识别传递给它的Lambda表达式的意义，典型的情况是，Lambda表达式会作用于序列中的每个元素，并且在操作每个元素的时候都会对Lambda表达式进行解析。这使得查询运算符看起来非常神奇，但它的底层实现是很容易理解的，以Enumerable类中的Where方法为例，它的实现如下：

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)
{
  foreach (TSource element in source)
  {
    if (predicate (element))
      yield return element;
  }
}
```

1. Lambda表达式及Func的方法签名

标准的查询运算符使用了一个泛型Func委托。Func是System.Linq命名空间中一组通用的泛型委托，它的作用是保证Func中的参数顺序和Lambda表达式中的参数顺序一致。

因此，Func<TSource,bool>所对应的Lambda表达式为TSource=>bool，也就是接受一个TSource型的输入，返回一个bool型的结果。

类似的，Func<TSource,TResult>所对应的Lambda表达式为TSource=>TResult。在第4章的“Lambda表达式”一节中我们列出了所有的Func委托。

2. Lambda表达式和元素类型

标准的查询运算符使用下面这些泛型：

泛型类型	名称意义
TSource	输入集合的元素类型
TResult	输出集合的元素类型（不同于TSource）
TKey	在排序、分组或者连接操作中所用的键

这里的TSource由输入集合的元素类型决定。而TResult和TKey则*由*我们给出的Lambda表达式指定。

以Select查询运算符的方法签名为例：

```
public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
```

在Func<TSource,TResult>所对应的Lambda表达式是TSource=>Result，这个表达式定义了*输入元素*和*输出元素*之间的映射关系，实际上TSource和TResult可以是不同的数据类型。更进一步说，Lambda表达式可以指定输出序列的类型，也就是说Select运算符可以根据Lambda表达式中的定义将输入类型转化成输出类型。下面这个示例中使用Select运算符将string类型的集合元素转换成int类型的数据来输出：

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<int> query = names.Select (n => n.Length);

foreach (int length in query)
  Console.Write (length + "|");     // 3|4|5|4|3|
```

编译器通过判断Lambda表达式中返回值的类型，推断出TResult的类型，在这个示例中，推断TResult为int型。

Where查询运算符的内部操作比Select查询运算符要简单一些，因为它只筛选集合，不对集合中的元素进行类型转换，因此不需要进行类型推断。它的方法签名如下：

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)
```

最后，我们看一下OrderBy运算符的方法签名：

```
// 更加精简一些：
public static IEnumerable<TSource> OrderBy<TSource,TKey>
  (this IEnumerable<TSource> source, Func<TSource,TKey> keySelector)
```

Func<TSource,TKey>将每个输入元素关联到一个*排序键TKey*，TKey的类型也是由Lambda表达式中推测出来的，但它的类与同输入类型、输出类型是没有关系的，三者是独立的，类型可以相同也可以不同。例如，我们可以选择对names集合按照名字的长度进行排序（TKey是int型的），也可以对names集合按字母排序（TKey是string类型的）：

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<string> sortedByLength, sortedAlphabetically;
sortedByLength       = names.OrderBy (n => n.Length);      // int型的键
sortedAlphabetically = names.OrderBy (n => n);              // string型的键
```

提示： 实际上我们可以使用传统的方式直接调用Enumerable中的各种方法来实现查询运算符的功能，此时在查询过程可以不使用Lambda表达式。这种直接调用的方式在对本地集合进行查询时非常好用，尤其是在LINQ to XML这种操作中应用最为方便，会在第10章中介绍。但传统调用方式并不适合对IQueryable<T>类型集合的查询，最典型的就是对数据库的查询，因为在对IQueryable<T>类型数据进行查询时，Queryable类中的运算符需要Lambda表达式来生成完整的查询表达式树，没有Lambda表达式，这个表达式树将不能生成。这部分内容我们在"解释型的查询"一节中详细介绍。

8.2.3 自然排序

LINQ中集成了对集合的排序功能，这种内置的排序对整个LINQ体系来说有重要意义。因为一些查询操作直接依赖于这种排序，例如Take、Skip和Reverse。

Take运算符会输出集合中前x个元素，这个x以参数的形式指定。例如：

```
int[] numbers = { 10, 9, 8, 7, 6 };
IEnumerable<int> firstThree = numbers.Take(3);       // { 10, 9, 8 }
```

Skip运算符会跳过集合中的前x个元素，输出其余元素。例如：

```
IEnumerable<int> lastTwo = numbers.Skip(3);          // { 7, 6 }
```

Reverse运算符则会将集合中的所有元素反转，也就是按照元素当前顺序的逆序排列。如：

```
IEnumerable<int> reversed = numbers.Reverse();       // { 6, 7, 8, 9, 10 }
```

Where和Select这两个查询运算符在执行时，会将集合中元素按照原有的顺序进行输出。实际上，在LINQ中，除非有必要，否则各个查询运算符都不会改变集合中元素的排序方式。

8.2.4 其他查询运算符

在LINQ中，并不是所有的查询运算符都会返回一个集合，一些针对元素的运算符可以从输入集合中返回单个元素，如First、Last和ElementAt等查询运算符，下面是几个简单示例：

```
int[] numbers = { 10, 9, 8, 7, 6 };
int firstNumber = numbers.First();                        // 10
int lastNumber = numbers.Last();                          // 6
int secondNumber = numbers.ElementAt(1);                  // 9
int lowestNumber = numbers.OrderBy(n => n).First();       // 7
```

而*聚合*运算符则返回一个表示数量的值，如：

```
int count = numbers.Count();        // 5
int min = numbers.Min();            // 6
```

下面这些*量词*运算符则返回bool型的结果：

```
bool hasTheNumberNine = numbers.Contains (9);            // true
bool hasMoreThanZeroElements = numbers.Any();            // true
bool hasAnOddElement = numbers.Any (n => n % 2 == 1);    // true
```

由于这些运算符返回的不是一个集合，所以无法在它们的后面再使用其他的查询运算符，这一点很容易理解。所以这些运算符一般都出现在一个查询的末尾。

有些查询运算符同时接受两个输入集合，例如Concat运算符会把一个集合中的元素添加到另一个集合中，另外还有Union运算符，它和Concat运算符的作用是相同的，唯一的区别在于，Union运算符会将结果集合中相同的元素去掉。如：

```
int[] seq1 = { 1, 2, 3 };
int[] seq2 = { 3, 4, 5 };
IEnumerable<int> concat = seq1.Concat (seq2);   // { 1, 2, 3, 3, 4, 5 }
IEnumerable<int> union = seq1.Union (seq2);     // { 1, 2, 3, 4, 5 }
```

实际上，连接运算符也属于这一类，在第9章中再详细介绍。

8.3 查询表达式

C#中新增了一组专门用于LINQ查询的语法结构，这种语法结构和C#原有的语法差别很显著。这种语法结构用到的查询运算符如From、Select、Where等和SQL中的关键字类似，但实际上这种语法结构并不是基于SQL设计出来的，而是来源于诸如LISP和Haskell这样的函数式编程语言，C#借鉴了这些语言中的列表解析方式。

提示：在本书中，当提到查询表达式时，可以简单理解为“查询语法”。

在之前的章节中，我们以查询表达式流的方式写过这样一段代码，它可以筛选出names集合中所有含字母“a”的元素，并把这些元素排序后以大写形式输出。下面使用查询表达式语法来完成相同的操作：

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

class LinqDemo
{
  static void Main()
  {
    string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

    IEnumerable<string> query =
      from n in names
      where n.Contains("a")      // 筛选元素
      orderby n.Length           // 对元素排序
      select n.ToUpper();        // 将元素转换为大写

      foreach (string name in query)
      {
        Console.WriteLine(name);
      }
  }
}
```

```
JAY
MARY
HARRY
```

查询表达式一般以from子句开始，最后以select或者group子句结束。from子句的作用是定义一个*范围变量*（本例中是n），这个变量会分别被输入序列中的每个元素赋值，通过这个变量，可以操作序列中的所有元素。实际上和foreach循环中的临时变量的作用是相同的。图8-2展示了完整的语法。

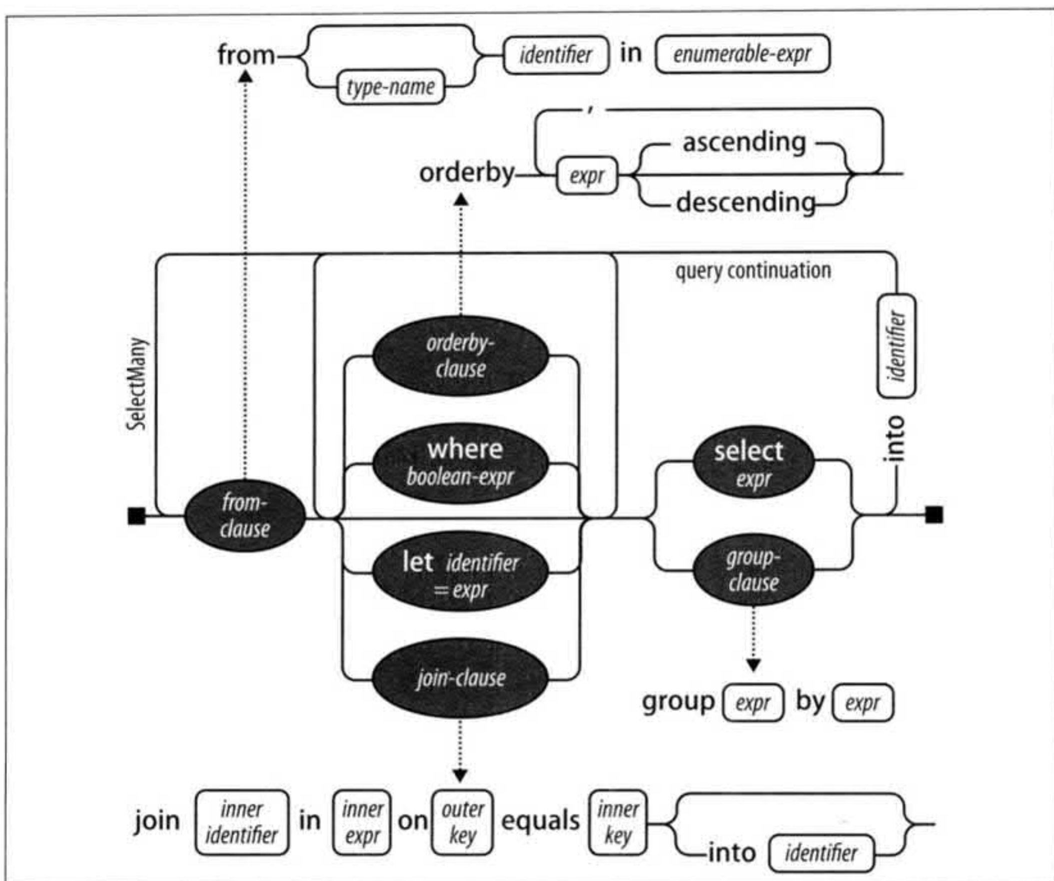


图8-2：查询语法

提示：要理解上面的查询表达式中的逻辑关系，可以从表达式的最左边开始把整个表达式的各个子句看作一个队列。例如，在一个表达式中，在from子句之后，可以选择性地使用orderby、where、let或者join子句。在所有这些子句之后，我们可以接着使用select或者group来结束整个查询。也可以不直接结束查询，而是把上次的查询结果视作一个输入，重新使用from、orderby、where、let或者join子句进行第二轮的查询。

编译器在执行查询表达式之前会把它编译成运算符流的形式，这样更接近它的原始状态。这个过程非常机械化，并没有使用什么特别操作，都是最常用的基本操作。foreach语句也是一样，就是通过多

次调用GetEnumberator和MoveNext方法来完成内部逻辑。因此查询表达式中的所有逻辑都可以用运算符流语法来书写。下面这个语句是上面的查询表达式经编译器编译之后的结果：

```
IEnumerable<string> query = names.Where   (n => n.Contains("a"))
                                  .OrderBy (n => n.Length)
                                  .Select  (n => n.ToUpper());
```

Where、OrderBy以及Select这些运算符在执行前会被解析成运算符流语法中相对的运算符。能达到这个效果，是因为这些关键字绑定了Enumerable类中对应的查询方法。如何知道绑定到哪个方法呢？在一开始导入了System.Linq命名空间，并且输入集合names也实现了Enumerable<string>接口，这时使用Where、OrderBy和Select这些运算符时，编译器就会依次找到Enumerable类中绑定的方法，最终完成查询的是这些方法而不是运算符，只是以一种更抽象的方式定义查询表达式。这种做法不仅让代码更易懂，而且还可以在执行的时候判断到底需要调用哪个类中的Where、Select方法。还有其他类也实现了这些方法，例如下一节中的Queryable类。

提示： 如果我们从代码中删除对System.Linq命名空间的引用，那么查询表达式就不能顺利编译，因为编译器在编译Where、OrderBy和Select这些运算符时，找不到与之对应的方法去绑定。编译器需要的是可以进行绑定的方法，这时需要导入命名空间，或者为每个查询运算符实现一个相应的方法。

8.3.1 范围变量

在之前用到的LINQ表达式中，紧跟在from关键字之后的标识符n实际上是一个范围变量。范围变量指向当前序列中将要进行操作的元素。

在之前的示例中，在每一个查询子句中都有范围变量，每个范围变量都会在各个子句中被重新定义，后面的子句并没有重用前面子句中的范围变量，下面是一个简单示例：

```
from n in names            // 这里n是范围变量
where n.Contains ("a")     // n直接使用from子句中的元素
orderby n.Length           // n使用经where子句筛选过的集合
select n.ToUpper()         // n使用经orderby子句排序过的集合
```

经过上面的解释，就很容易理解编译器将LINQ表达式转换成运算符流形式后的代码是：

```
names.Where (n => n.Contains ("a"))          // 私有的局域变量n
     .OrderBy (n => n.Length)                // 一个新的私有局域变量n
     .Select (n => n.ToUpper())              // 又一个新的私有局域变量n
```

正如我们看到的，在每个子查询的Lambda表达式中，n都会被重新定义。

如果有必要，在查询中可以将结果集暂存于某个变量中作为中间结果集，然后再对这个中间结果集进行新的查询。要定义这种存储中间结果的变量，需要使用下面几个子句：

- let
- into
- 一个新的from子句
- jion

我们会在本章稍后的内容中介绍这几个子句的使用方式。在第9章的“映射”和“连接”两节中还会有进一步介绍。

8.3.2 LINQ查询语法与SQL语法的对比

LINQ查询语法看起来和SQL语法非常相似，但它们完全不同。LINQ查询表达式本质上是一种C#表达式，遵循C#的语法规则，例如，在LINQ中，变量必须在声明之后才能使用。与之不同的是在SQL中，SELECT后面可以直接跟列名，而这些列所在的表在稍后的FROM子句中才能确定。

在LINQ中，一个子查询实际上就是一个新的C#表达式，因此不需要使用专门的子查询运算符。而在SQL中，使用子查询要遵循特殊的语法规则。

在LINQ查询过程中，数据的处理顺序一直是从左向右进行的；而在SQL中，数据的处理顺序则是不确定的。

另外，在LINQ中，在处理数据过程中，集合中的元素始终保持一种有序的状态；而在SQL的查询过程中，数据始终是以一种网状的结构存在，没有顺序的概念。

8.3.3 查询表达式语法与运算符流语法的区别

查询表达式语法和运算符流语法各有优势。

在包含以下运算符的查询操作中，使用查询表达式语法更加方便：

- 在查询中使用let子句导入新的查询变量。
- 在查询中用到SelectMany、Join或者GroupJoin这些运算符。

（我们会在本章的“LINQ构造方式”一节中介绍let子句的用法，在第9章中介绍SelectMany、Join以及GroupJoin子句的用法。）

对于只包含Where、OrderBy或者Select的查询语句，这两种查询方式都可以。

一般来说，查询表达式语法由单个的运算符组成，结构比较清晰；而运算符流语法写出的代码相对简洁。

最后是适合使用运算符流语法书写的情况，很多查询运算符在表达式语法中没有相应的关键字，这时只能使用运算符流的方式进行查询，至少应该是部分使用。在不含以下运算符的查询中，选用运算符流语法进行查询会更加方便：

```
Where, Select, SelectMany
OrderBy, ThenBy, OrderByDescending, ThenByDescending
GroupBy, Join, GroupJoin
```

8.3.4 混合查询语法

如果一个查询运算符没有适合的查询语法，那么我们可以混合使用上面介绍的两种查询方式来得到最终结果，这样做的唯一限制是，在整个查询中，每个查询表达式的表述必须是完整的（例如，必须由from子句开始，由select或者group子句结束）。

假如有这样一个数组：

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
```

下面这个表达式用于查询集合names中包含字母“a”的单词的个数：

```
int matches = (from n in names where n.Contains ("a") select n).Count();  // 3
```

下面这个表达式首先对集合进行排序，然后返回排序数组中的第一个元素：

```
string first = (from n in names orderby n select n).First();        // Dick
```

在比较复杂的查询中，这种混合使用两种查询语法进行查询的方式非常高效。上面的两个示例，实际上也可以直接使用运算符流语法书写：

```
int matches = names.Where (n => n.Contains ("a")).Count();          // 3
string first = names.OrderBy (n => n).First();                      // Dick
```

提示：有时候，即使混合使用了两种查询语法，也没有写出真正简练的LINQ查询，但注意不要因此养成只使用一种查询语法的习惯。如果习惯只使用一种语法形式的，在遇到复杂查询情况时，很难找到一种真正高效的方式去解决问题。

在本章余下的内容中，我们会尽量在示例中混合使用两种查询语法。

8.4 延迟执行

在LINQ中，另一个很重要的特性是延迟执行，也可以说是延迟加载，它是指查询操作并不是在查询运算符定义的时候执行，而是在真正使用集合中的数据时才执行（例如在遍历集合时调用MoveNext方法时会触发查询操作）。下面是一个简单的示例：

```
var numbers = new List<int>();
numbers.Add(1);

IEnumerable<int> query = numbers.Select(n => n * 10);    // 创建查询语句

numbers.Add(2);                        // 向numbers中添加一个新的元素

foreach (int n in query)
  Console.Write(n + "|");              // 10|20|
```

在上面示例中，在创建查询语句后又向集合中加入新元素，这个新元素最终也出现在查询结果中。这就是因为查询语句是在遇到foreach之后才真正执行的，而foreach在*numbers.Add(2)*之后，所以输出集合中包含了这个元素。这是所谓的推迟或延迟执行，它与代理实现的效果相同：

```
Action a = () => Console.WriteLine ("Foo");
//没有在控制台输出任何内容。现执行此方法：
a();// 推迟执行!
```

这种特性就是所谓的延迟加载，绝大部分标准的LINQ查询运算符都具有延迟加载这种特性，当然也有例外，以下是几个例外的运算符：

- 那些返回单个元素或者返回一个数值的运算符，如First或者Count。

- 转换运算符：ToArray、ToList、ToDictionary、ToLookup。

以上这些运算符都会触发LINQ语句立即执行，因为它们的返回值类型不支持延迟加载。例如Count运算符，它返回一个简单的整型数据，而一个整型数据不会用到遍历操作，因此Count运算符会被立即执行，如下面这个示例所示：

```
int matches = numbers.Where (n => n < 2).Count();    // 1
```

在LINQ中，延迟加载特性有很重要的意义，这种设计将查询的创建和查询的执行进行了解耦，这使得我们可以将查询分成多个步骤来创建，有利于查询表达式的书写，而且在执行的时候按照一个完整的结构去查询，减少了对集合的查询次数，这种特性在对数据库的查询中尤为重要。

提示：子查询中的表达式有额外的延迟加载限制。无论是聚合运算符还是转换运算符，如果出现在子查询中，它们都会被强制地进行延迟加载。我们会在本章“子查询”一节中详细介绍这部分内容。

8.4.1 重复执行

延迟执行导致的后果是：当两次遍历同一个集合时，查询被重复执行。也就是当多次查询同一个数据集时，它后一次遍历不会使用前一次遍历的结果，而是再到数据源中进行一次新的查询。如下面这个示例：

```
var numbers = new List<int>() { 1, 2 };

IEnumerable<int> query = numbers.Select (n => n * 10);
foreach (int n in query) Console.Write (n + "|");    // 10|20|

numbers.Clear();

foreach (int n in query) Console.Write (n + "|");    // <无>
```

这种重复执行是LINQ的缺点，它带来以下两个方面的影响：

- 无法缓存集合在某一个时刻的状态，供后面的代码使用。
- 对于一些远程数据源，例如远端的数据库，这种重复执行会大大降低执行效率。

这种问题是可以解决的，一般使用转换运算符来绕过这种重复执行，如ToArray或者ToList。ToArray把结果集中的元素拷贝到一个新的数组中；ToList会将结果集中的元素拷贝到一个List<>集合中。这两种方式都可以保存查询结果，下次需要使用结果集时，直接使用这两种集合中保存的数据即可，下面是一段示例：

```
var numbers = new List<int>() { 1, 2 };

List<int> timesTen = numbers
  .Select (n => n * 10)

  .ToList();                          // 立即执行，将结果存放于List<int>

numbers.Clear();
Console.WriteLine (timesTen.Count);    // 仍然输出2
```

8.4.2 捕获的变量

延迟加载还会带来另一个问题，如果在Lambda表达式中使用了本地变量，那么Lambda表达式使用的是这个本地变量的引用，而不是它的拷贝。这就意味着，如果在其他地方改变了本地变量的值，Lambda表达式中的结果也会发生改变，如下面示例所示：

```
int[] numbers = { 1, 2 };

int factor = 10;
IEnumerable<int> query = numbers.Select(n => n * factor);
factor = 20;
foreach (int n in query)
{
  Console.Write(n + "|");    // 20|40|
}
```

尽管很多人知道存在这个问题，但是在实际应用中还是很容易犯错，特别是在foreach循环中更容易出错。如下面这个示例，去掉一个字符串中的元音字符。首先是下面这种写法，看起来效率不高，但可以得到正确的结果：

```
IEnumerable<char> query = "Not what you might expect";

query = query.Where(c => c != 'a');
query = query.Where(c => c != 'e');
query = query.Where(c => c != 'i');
query = query.Where(c => c != 'o');
query = query.Where(c => c != 'u');

foreach (char c in query) Console.Write(c); // Nt wht y mght xpct
```

如果使用foreach语句来重构这段代码，会使整个查询更简洁高效，代码如下：

```
IEnumerable<char> query = "Not what you might expect";

string vowels = "aeiou";

for (int i = 0; i < vowels.Length; i++)
query = query.Where (c => c != vowels[i]);

foreach (char c in query) Console.Write (c);
```

列举查询时会抛出一个`IndexOutOfRangeException`异常，正如在第4章所介绍的（参见第136页的“捕捉外部变量”），这是因为编译器将`for`循环中的迭代变量作用域定义为与循环外部声明一样。因此，每一个闭包都会捕捉到相同的变量（i），查询在执行实际列举步骤时，它的值是5。为了解决这个问题，必须将循环变量赋值给语句块内部声明的另一个变量：

```
for (int i = 0; i < vowels.Length; i++)
{
char vowel = vowels[i];
query = query.Where (c => c != vowel);
}
```

这样就强制在每一次循环迭代中获得一个新变量值。

提示： 在C# 5.0中，解决这个问题的另一个方法是将for循环替换成foreach循环：

```
foreach (char vowel in vowels)
query = query.Where (c => c != vowel);
```

这个方法在C# 5.0可以生效，但是在旧版本C#上会失败，其原因在第4章中介绍过。

8.4.3 延迟加载的工作原理

LINQ查询运算符之所以有延迟加载功能，是因为每个运算符的返回值不是一个一般的数组或者集合，而是一个经过封装的序列，这种序列通常情况下并不直接存储数据元素，它封装并使用运行时传递给它的集合，元素也由其他集合来存储它实际上只是维护自己与数据集合的一种依赖关系，当有查询请求时，再到它依赖的序列中进行真正的查询。

提示： 查询运算符实际上是封装一系列的转换函数，这种转换函数可以将与之关联的数据集转换为各种形式的序列。如果输出集合不需要转换的话，那么就不用执行查询运算符封装的转换操作，这个时候查询运算符实际上就是一个*委托*，进行数据转发而已。

例如调用Where运算符的时候，在Where内部所做的操作非常简单，根据Lambda表达式中指定的查询条件，对输入集合重新进行了筛选，保留那些符合条件的元素的指针引用，当外部遍历Where的返回值时，Where会到它所关联的集合中进行真正的查询，然后返回查询结果。

图8-3演示了下面这个查询语句的执行过程：

```
IEnumerable<int> lessThanTen = new int[] { 5, 12, 3 }.Where(n => n < 10);
```

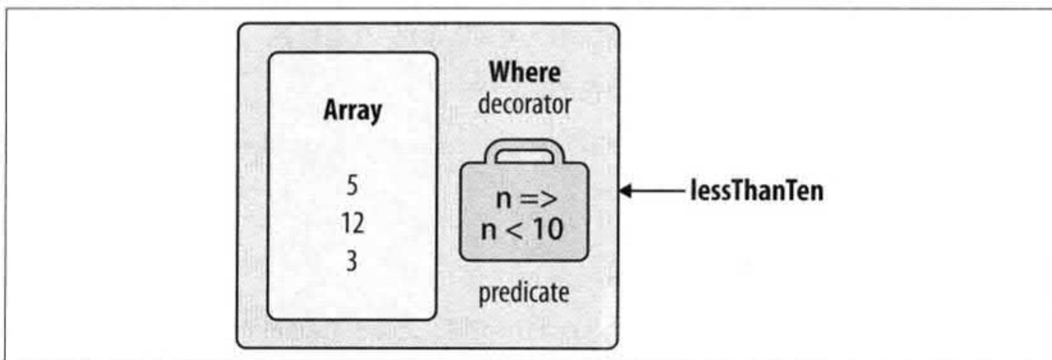


图8-3：封装序列

当执行lessThanTen操作时，实际上，就是使用Where运算符对封装序列进行筛选。

当需要使用一些有特殊功能的运算符时，我们可以自己动手使用C#中的迭代器轻松地实现。例如下面这个示例所示，实现一个自定义的Select运算符：

```
public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
{
```

```
    foreach (TSource element in source)
        yield return selector (element);
  }
```

上面这个方法实际上就是在循环中返回yield类型的元素。它和下面这个方法所实现的功能是相同的：

```
public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
{
  return new SelectSequence (source, selector);
}
```

上面这段代码中用到了*SelectSequence*方法，实际上是完成了之前代码中循环返回yield类型元素的功能，这个方式是系统内置的，它的功能就是实现对集合中元素的遍历。

所以当调用Select和Where查询运算符时，内部操作就是创建一个序列，然后将查询得到的元素存入这个新的序列中。

8.4.4 连续使用封装集合

如果使用运算符流语法对集合进行查询，会创建多个层次的封装集合。以下面这个查询语句为例：

```
IEnumerable<int> query = new int[] { 5, 12, 3 }.Where (n => n < 10)
                                               .OrderBy (n => n)
                                               .Select (n => n * 10);
```

图8-4演示了执行这个语句时系统对集合的处理过程，这是一个完整的对象处理模型，需要注意的是，这个对象模型一定是在查询语句被真正执行前就创建好的。

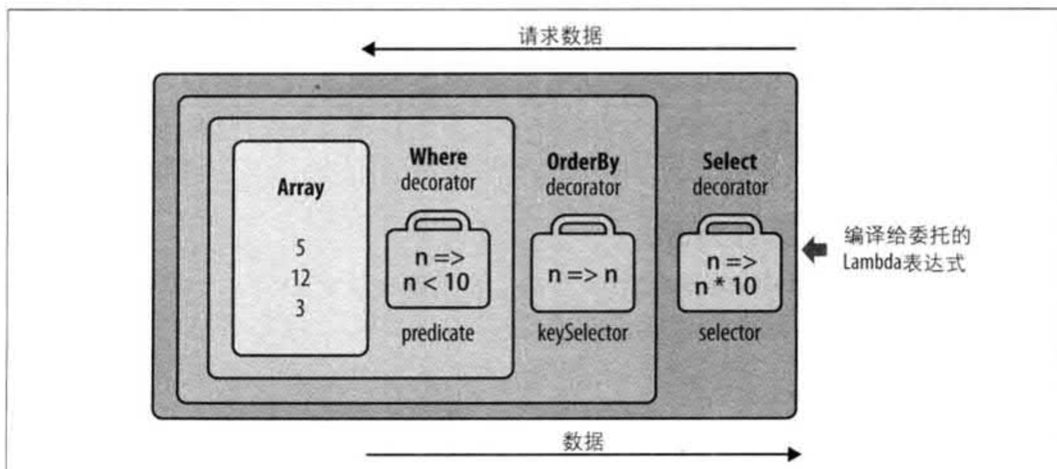


图8-4：查询中多个封装类的层次结构

在使用LINQ语句的返回集合时，实际是在原始的输入集合中进行查询，只不过在进入原始集合之前，会经过上面这些封装类的处理，在不同层次的封装类中，系统都会对查询做相应的修改，这使得LINQ语句使用的各种查询条件会被反映到最终的查询结果中。

提示： 如果在LINQ查询语句的最后加上ToList方法，会强制LINQ语句立刻执行，查询结果会被保存到一个List类型的集合中。

图8-5使用UML的方式重新演示了封装类之间的层次结构。从中可以看出，Select子句的封装类指向OrderBy子句的封装类，OrderBy子句的封装类指向Where子句的封装类，而Where子句最终指向一个实际的数组。LINQ的延迟加载特性有这样一种功能：不论查询语句是连续书写的还是分多个步骤完成的，在执行之前，都会被组合成一个完整的对象模型，而且两种书写方式所产生的对象模型是一样的，例如下面这种书写方式并不会导致查询被多次执行：

```
IEnumerable<int>
  source    = new int[] { 5, 12, 3 },
  filtered  = source   .Where   (n => n < 10),
  sorted    = filtered .OrderBy (n => n),
  query     = sorted   .Select  (n => n * 10);
```

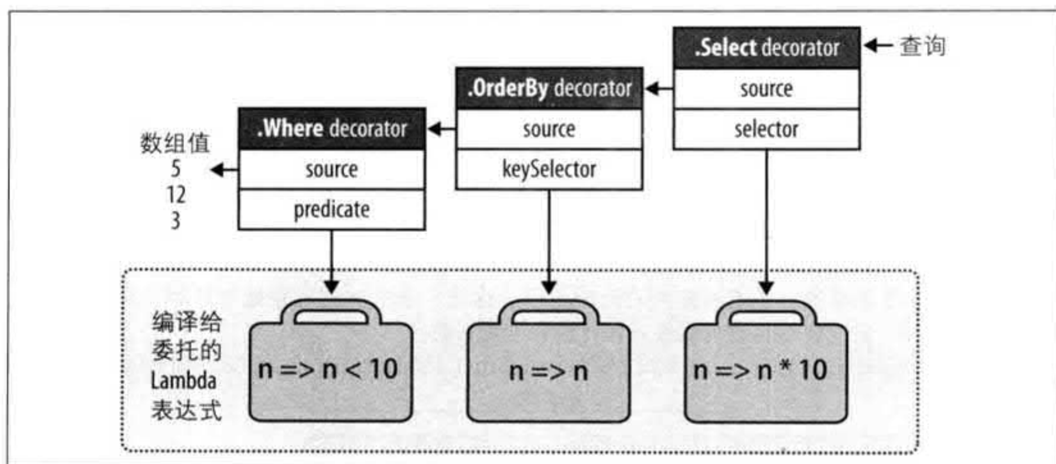


图8-5：封装对象模型的UML表示

8.4.5 查询语句的执行方式

下面这几行代码遍历了前面得到的query集合：

```
foreach (int n in query) Console.WriteLine (n);

30
50
```

这个foreach遍历了query集合，并输出了集合中的元素，分析代码可知，foreach会调用Select运算符（整个LINQ语句最外部的运算符）中的GetEnumerator方法，这个方法会返回原始集合中所有的元素。返回的元素再经过Where、OrderBy等运算符的层层处理，最终返回一个结果集。图8-6演示了使用foreach遍历LINQ集合的内部执行过程。

在本章的第一节，我们将LINQ查询比作一个流水线，在流水线的不同阶段对集合进行不同的操作，最后得到结果，所以实际上LINQ查询是一个低效率的流水线。因为流水线并不是一直在运行，它只有在需要启动的时候才开始工作。也就是说，先创建一条完整的流水线，但是它并没有开始运转。当

客户需要使用集合中的元素时（从代码角度看，就是foreach中输入集合元素时），流水线才被启动。流水线的最后一个环节需要元素，但是它上面没有元素，于是它请求前一个步骤给它传递数据，前一个流水线又将这个请求传递给更靠前的流水线，直到找到数据源中的元素，然后再一级级传递下去。LINQ使用的是需求驱动的模型，先请求再有数据。这种模型很重要，在接下来的内容中我们会介绍它的作用。

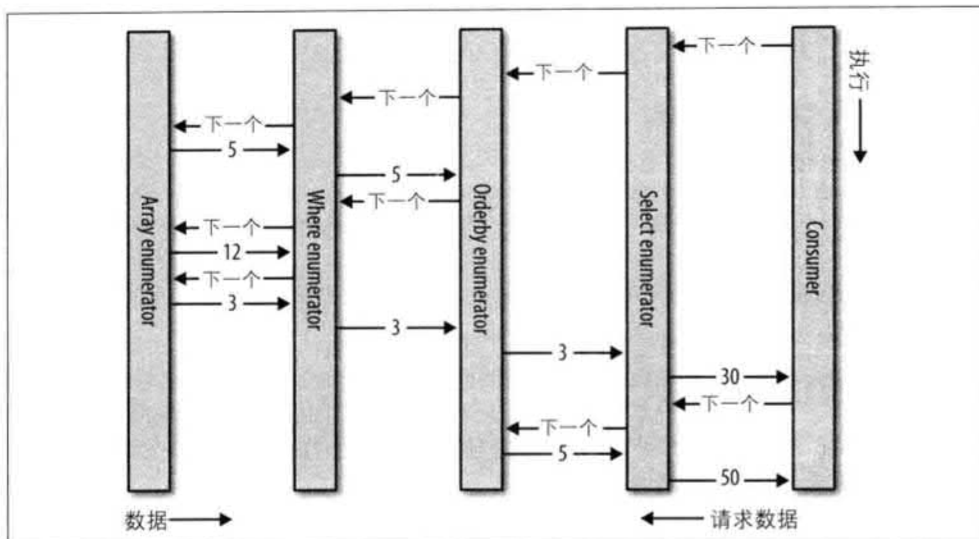


图8-6：执行本地查询

8.5 子查询

在LINQ中，所谓子查询就是包含在另一个查询的Lambda表达式中的查询语句。下面这个示例中就是使用子查询对一组音乐家的名字按姓氏排序：

```
string[] musos = { "David Gilmour", "Roger Waters", "Rick Wright" };
IEnumerable<string> query = musos.OrderBy(m => m.Split().Last());
```

`m.Split`将musos中的每个名字转换成一个字符型的数组，然后使用Last运算符取出数组的最后一个元素，这里`m.Split().Last`就是一个子查询，这个子查询的结果可以供外部查询使用。

一个子查询实际上就是一个独立的C#表达式，可以是LINQ表达式，也可以是普通的逻辑判断，所以只要是符合C#语法规则的内容，都可以放在Lambda表达式的右侧作为子查询来使用。也就是说，子查询的使用规则是由Lambda表达式的规则所决定的。

提示： “子查询”这个词，在通常意义下，概念非常宽泛，我们只关注LINQ下的子查询。在运算符流语法中，子查询是指包含在Lambda表达式中的查询语句。在查询表达式语法中，只要包含在其他查询语句中的查询，都是子查询，但是`from`子句除外。

子查询一般有两个作用：一个是为父查询确定查询范围，一般是一个较小的查询范围，另一个作用是为外层查询的Lambda表达式提供参数。

m.Split().Last是一个非常简单的子查询，下面这个示例稍微复杂一点，它使用子查询取出一个字符串数组中长度最短的元素：

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IEnumerable<string> outerQuery = names
  .Where(n => n.Length == names.OrderBy(n2 => n2.Length)
                               .Select(n2 => n2.Length).First());

Tom, Jay
```

使用查询表达式语法可以实现同样的功能：

```
IEnumerable<string> outerQuery =
  from n in names
  where n.Length ==(from n2 in names orderby n2.Length select n2.Length).First()
  select n;
```

在上面的表达式中，范围变量n已经作为父查询的变量被使用，所以在子查询中就不能再使用n作为范围变量了，这里使用了n2。

子查询在什么时候执行完全是由外部查询决定的，当外部查询开始执行时，子查询也同时执行，它们是同步的，在整个查询中，子查询的执行结果被作为父查询的某个组成部分。我们可以认为查询的开始命令是*从外向内*传递的，对本地集合的查询严格按照这种由外向内的顺序进行；但对数据库的查询，则没有那么严格，只是原则上按照这种方式进行。

另一种理解方式是，子查询会在需要返回查询结果时执行，那什么时候需要子查询返回查询结果决定于外部查询什么时候被执行，在上面的示例中，子查询（图8-7中上面的传送带）会在每次外部循环中都执行，也就是这里的子查询是由外部循环触发的。结合图8-7、图8-8以及下面的代码，我们可以把之前用到的子查询定义为：

```
IEnumerable<string> query =
  from n in names
  where n.Length == names.OrderBy (n2 => n2.Length).First().Length
  select n;
```

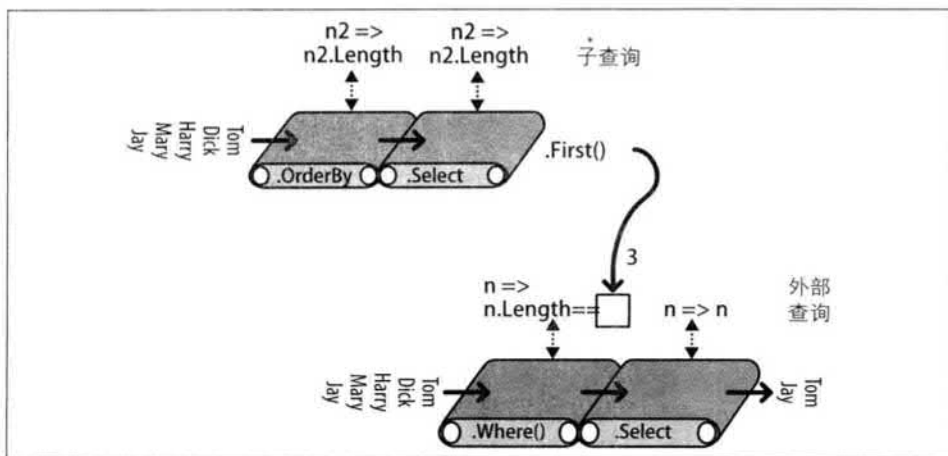


图8-7：子查询的执行过程

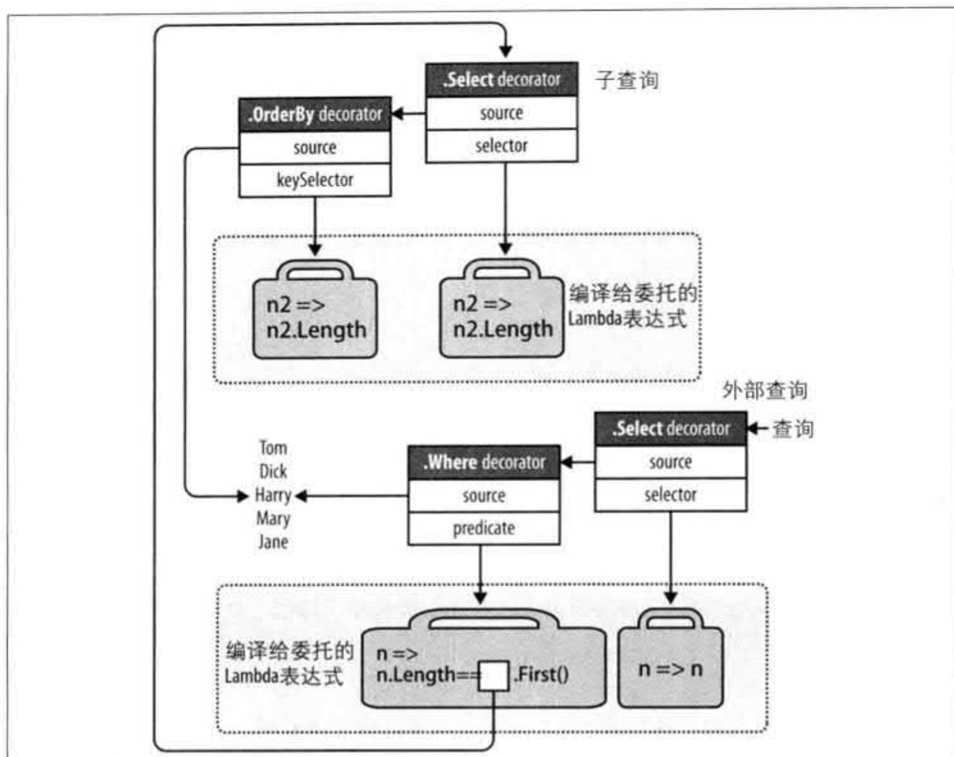


图8-8：子查询过程的UML表述

使用Min聚合函数之后，整个查询将更加简单：

```
IEnumerable<string> query =
  from n in names
  where n.Length == names.Min (n2 => n2.Length)
  select n;
```

在本章的"解释型的查询"一节中，会介绍如何使用LINQ查询远端的数据源，如对SQL数据库表的查询。上面的代码段是一个非常标准的查询语句，使用一个LINQ语句就完成了所有的操作，当对数据库进行查询时，只需访问一次数据库就可以全部完成。但对于本地数据源来说，上面这个查询语句在运行时效率比较低，因为字符串的长度会被重复计算多次，我们可以单独运行子查询来避免这种重复操作（单独运行的子查询实际上已经不是子查询，而是一个新的查询语句）。代码如下：

```
int shortest = names.Min(n => n.Length);

IEnumerable<string> query = from n in names
                            where n.Length == shortest
                            select n;
```

提示：在执行本地查询时，单独书写子查询是一种常用的查询方式。但是当子查询中的数据和外部查询有紧密关联的时候，即内部数据需要用到外部数据的值时，这种方式不适合，最好写成一个表达式。这种查询会在第9章的"映射"一节中进行详述。

子查询和延迟加载

在子查询中使用单个元素或者聚合函数（如Count或者First）的时候，整个LINQ查询语句并不会被强制执行，外部查询还是以延迟加载的方式执行。这是因为子查询是被间接执行的，在本地集合查询中，它通过委托的驱动来执行；而在远程数据源的查询中，它通过表达式树的方式执行。

如果Select语句中已经包含了子查询，在这种情况下如果是本地查询，那么相当于将源序列重新封装到一个新的序列中，集合中的每个元素都是以延迟加载的方式执行的。关于在Select语句中使用子查询更多的内容，在第9章中会有详细介绍。

8.6 LINQ构造方式

在本节中将通过以下三个方面介绍如何书写复杂的LINQ查询表达式：

- 递增式的书写方式
- 使用into关键字
- 包装查询语句

实际上无论用何种书写方式，在运行时，LINO查询表达式都会被编译成相同的查询语句来运行。

8.6.1 递增式的查询

在本章的一开始，曾经演示过如何将运算符流语法的查询转换成递增式的查询：

```
var filtered   = names    .Where   (n => n.Contains("a"));
var sorted     = filtered .OrderBy (n => n);
var query      = sorted   .Select  (n => n.ToUpper());
```

在上面这三行示例代码中，每个查询运算符都会返回一个封装序列，这些查询语句在执行时会被组合成一个语句，实际上对于编译器来说，这种分步写和使用一个查询语句的方式没有任何区别。但对于开发者来说，这种递增式的书写方式有两个好处：

- 这种方式使得查询语句更加清晰、更容易书写和阅读。
- 我们可以根据需要添加新的运算符。例如：

```
if (includeFilter) query = query.Where (...)
```

比下面这种方式更高效：

```
query = query.Where (n => !includeFilter || <expression>)
```

这是因为当includeFilter是false的时候，第一种方式不用在LINQ查询中添加新的查询运算符。在使用多个查询条件进行查询的时候，这种递增式的书写方式比较实用。为了说明这个问题，我们还是用前面的一个示例，移除一组名字中的元音字母，然后将长度大于2的名字按字母升序排列。使用运算符流语法，下面这个语句可以完成这些操作：

```
IEnumerable<string> query = names
  .Select (n => n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                 .Replace ("o", "").Replace ("u", ""))
  .Where (n => n.Length > 2)
```

```
    .OrderBy (n => n);
```

查询结果：{ "Dck", "Hrry", "Mry" }

提示： 在上面的查询中，为了移除元音字母，调用了五次Replace方法，实际上可以用下面这个表达式更高效地移除元音字母：

```
n => Regex.Replace (n, "[aeiou]", "")
```

当然字符串的Replace方法也是有优点的，它可以在对数据库的查询中使用，但上面这种表达式只能用于本地序列的查询中。

如果想要直接将上面这段查询代码查询结果：转换成查询表达式语法有些困难，因为在查询表达式的书写方式中，select语句必须出现在where和orderby子句之后，如果我们在最后一步才将名字映射到一个新集合，那么得到的结果和期望的结果就不同了：

```
IEnumerable<string> query =
  from n in names
  where n.Length > 2
  orderby n
  select n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
          .Replace ("o", "").Replace ("u", "");
```

查询结果：{ "Dck", "Hrry", "Jy", "Mry", "Tm" }

两次结果查询的不同在于，在使用n.Length > 2对名字进行筛选的时候，第一次查询是在移除元音字母之后进行判断，第二次查询是在移除元音字母之前进行判断，因此第二次的结果集中多了两个名字。但是，递增式的查询方式可以绕过上面这个限制，得到正确的结果集：

```
IEnumerable<string> query =
  from n in names
  select n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
          .Replace ("o", "").Replace ("u", "");

query = from n in query where n.Length > 2 orderby n select n;
```

查询结果：{ "Dck", "Hrry", "Mry" }

8.6.2 into关键字

提示： 根据上下文的不同，into关键字在查询表式达中有两种完全不同的功能。这里首先介绍如何使用into关键字延长查询（另一种是和GroupJoin配合使用）。

在LINQ查询中，一般会用到集合的映射，也就是在Select方法中将查询结果直接组装成新的集合，这种映射一般在查询的最后执行。但是如果在映射之后还想对新集合执行查询的话，就可以使用into关键字来完成。这里使用into关键字重写前面移除元音字符的示例：

```
IEnumerable<string> query =
  from   n in names
  select n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
          .Replace ("o", "").Replace ("u", "")
```

```
  into noVowel
    where noVowel.Length > 2orderby noVowel select noVowel;
```

注意，into关键字只能出现在select和group关键字之后，into会重新创建一个新的查询，在新的查询中，我们可以再次使用where、orderby、select关键字。

提示：into关键字的作用就是在原来的查询中重新创建一次新的查询，在执行前，这种带into的查询表达式会被编译成运算符流的查询语句，因此使用into运算符并不会带来性能上的损失。

实际上的查询方式并没有特别之处，在效率上也没有改进，只是连续使用多个查询运算符的一种书写方式。

范围规则

在同一个语句中，into关键字后面的查询语句不能再使用在其之前定义的变量。下面这个查询语句就是因为这个原因不能编译通过：

```
var query =
  from n1 in names
  select n1.ToUpper()
  into n2                                 // 在下面查询中只有n2是可用的
    where n1.Contains("x")                // 错误：n1不能在这里被使用
    select n2;
```

为什么n1不能在into关键字之后继续使用呢，这段代码被编译成运算符流语法之后的格式如下：

```
var query = names
  .Select(n1 => n1.ToUpper())
  .Where(n2 => n1.Contains("x"));      // 错误：n1已经不能再使用
```

当代码运行到Where子句的时候，定义了一个新的局域变量n2，这个时候n1已经失去作用，因为Select子句将一个大写的名字集合传递给了Where子句，此时Select子句的作用已经完成，所以它定义的局部变量在Where子句中就不能再使用。

8.6.3 封装查询表达式

以递增方式书写的LINQ查询表达式都可以用一个查询语句来完成，只需将各个子句放在不同的层次中，一层一层地进行查询。一个常见的情况：

```
var tempQuery = tempQueryExpr
var finalQuery = from ... in tempQuery ...
```

上面这个查询可以表示成：

```
var finalQuery = from ... in (tempQueryExpr)
```

这种包含了多个层次的查询表达式，在语义和执行上都和递增式的LINQ查询语句相同，它们本质上没有区别，唯一的区别就是查询关键字的使用顺序，如下面这个示例：

```
IEnumerable<string> query =
  from n in names
  select n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
```

```
            .Replace ("o", "").Replace ("u", "");

query = from n in query where n.Length > 2 orderby n select n;
```

如果使用多层次查询来表示，代码如下：

```
IEnumerable<string> query =
  from n1 in
  (
    from n2 in names
    select n2.Replace ("a", "").Replace ("e", "").Replace ("i", "")
             .Replace ("o", "").Replace ("u", "")
  )
  where n1.Length > 2 orderby n1 select n1;
```

如果用运算符流语法完成上面的操作，各个运算符出现的顺序和多层次查询是相同的：

```
IEnumerable<string> query = names
  .Select  (n => n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                  .Replace ("o", "").Replace ("u", ""))
  .Where   (n => n.Length > 2)
  .OrderBy (n => n);
```

在上面这个查询语句中，编译器并不会返回.Select(n=>n)中查询的内容，因为在这里Select并不是平时意义上的Select。

多层次的查询结构和子查询的结构非常类似，很容易混淆，因为这两种查询结构中都有内部查询和外部查询。但是这两种结构转换成编译器能识别的运算符流的方式之后，它们的不同之处就比较明显了。多层次的查询结构经过翻译后会形成一个循环结构，对一组运算符重复执行多次来得到查询结果，这个查询结果不依赖于内部查询的结果。

还是用前面提到的传送带作示例说明：在多层次查询中，内部查询是在传送带之前执行的。而子查询则是传送带上的一部分，它会随着整个传送带的运行而执行（如图8-7所示）。

8.7 映射策略

8.7.1 对象初始化器

到目前为止，我们在Select语句中定义的结果集形式都非常简单。实际上，我们可以使用C#的对象初始化器将LINQ的返回结果定义成更复杂的结构。还是以移除一组名字中的元音字母为例，如果我们现在要求结果集中同时保留没有元音字母的名字和原来的名字，显然这时的返回值不再是一个简单类型，这个需求应该怎么完成呢，可以首先定义一个类用于存放返回值的每个元素：

```
class TempProjectionItem
{
  public string Original;    // 原来的名字
  public string Vowelless;   // 移除元音字母的名字
}
```

然后用对象初始化器将LINQ查询结果映射成期望的结果即可：

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
```

```
IEnumerable<TempProjectionItem> temp =
  from n in names
  select new TempProjectionItem
  {
    Original  = n,
    Vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                 .Replace ("o", "").Replace ("u", "")
};
```

上面这个查询的返回值类型是IEnumerable<TempProjectionItem>，如果想要输出结果，可以再使用下面这个查询：

```
IEnumerable<string> query = from   item in temp
                            where  item.Vowelless.Length > 2
                            select item.Original;
输出结果是：
Dick
Harry
Mary
```

8.7.2 匿名类型

所谓匿名类型指的是没有显式定义过的类型，在查询过程中，可以使用这种类型来封装查询结果。实际上这个类并不是没有定义，只是不用我们自己定义，编译器会自动定义这个类型。下面用匿名类型重写前面的查询，注意TempProjectionItem这个类是如何被替代的：

```
var intermediate = from n in names

  select new
  {
    Original = n,
    Vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                 .Replace ("o", "").Replace ("u", "")
};

IEnumerable<string> query = from   item in intermediate
                            where  item.Vowelless.Length > 2
                            select item.Original;
```

使用了匿名类的查询语句得到的结果和之前的查询结果是相同的，只是在匿名类这种查询方法中定义了一次性类，它只能在这个查询中使用。别的查询如果想使用这个类的话，必须重新定义。前面提到，这个类在运行时实际上是存在的，编译器会根据我们写的映射结构，在运行时构建一个相同结构的类供查询使用，那么在查询的第一步，LINQ语句的返回结果集的类型是：

```
IEnumerable <编译器随机生成的类名>
```

要在C#代码中定义一种编译时才能确定的类型，唯一的选择是使用var关键字，此时var关键字就不仅仅是为了便于书写，而是不得不这么写，因为我们不知道匿名类型的名字到。

使用into关键字可以使上面的查询更加简洁：

```
var query = from n in names
  select new
  {
```

```
      Original = n,
      Vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                  .Replace ("o", "").Replace ("u", "")
    }
    into temp
    where temp.Vowelless.Length > 2
    select temp.Original;
```

实际上LINQ表达式还提供了一种更简单的方式来书写匿名类型，即let关键字。

8.7.3 Let关键字

使用let关键字，可以在查询中定义一个新的临时变量来存放某些步骤的查询结果。

使用let关键字来查询一组名字中，去除元音字母之后长度大于2的名字，查询方法如下：

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IEnumerable<string> query =
  from n in names
  let vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                   .Replace ("o", "").Replace ("u", "")
  where vowelless.Length > 2
  orderby vowelless
  select n;          // 由于Let关键字，变量n仍然可用
```

编译器在编译let关键字的时候，会把它翻译成一个匿名类型，这个匿名类型中包含了之前的范围变量n和一个新的表达式变量。也就是说，编译器将n翻译成了前面的匿名类型查询。

let还有以下两个优点：

- 保留了前面查询中的范围变量。
- 在一个查询中可以重复使用它定义的变量。

Let关键字的这两个优点在上面的示例中也有所体现，在select语句中，我们可以使用前面的范围变量(n)，也可以使用let定义的变量(vowelless)。

在LINQ查询中，在where关键字之前或之后可以使用任意多个let关键字（见图8-2）。后面的let关键字会使用前面let关键字的返回类型，显然，let关键字会在每一次使用时重新组成结果集。

Let关键字一般不用来返回数值类型的结果，更多使用在子查询中。

8.8 解释型的查询

前面我们已经提到过，LINQ包含两种查询：对本地集合的本地查询以及对远程数据源的解释型查询。到目前为止，我们使用的LINQ查询都是对本地集合的查询，这种查询调用IEnumerable<>接口中定义的Enumerable方法实现了接口中所有的方法来完成具体的查询。这个过程中，无论LINQ是用查询表达式语法还是运算符流语法定义的，整个查询都会被编译器编译成中间语言，这时的LINQ查询语句完全按照C#表达式来处理。

与此不同的是，在解释型的查询中，所有的查询操作都是通过IQueryable<T>接口中的方法完成的，具体的方法实现是在Queryable类中。在这种查询中，LINQ语句不会被编译成.NET Framework

中间语言（IL），而会在运行时被解释成查询表达式树来执行。

提示：实际上，可以使用Enumerable中的方法来查询IQueryable<T>类型的数据源，但会遇到一个问题，那就是查询的时候，远端的数据源必须被加载到本地内存中，然后以本地数据源的方式进行处理。可以想象，这种查询的效率非常低，每次都需要读取大量的数据，在本地进行筛选。这正是创建解释型查询的原因。

在.NET Framework中有两个类都实现了IQueryable<T>接口，这两个类用于实现两种不同的查询：

- LINQ to SQL
- Entity Framework (EF)

这两种*LINQ-to-db*的查询技术实际上非常相似，在本书中，除非特别说明，我们的LINQ查询语句在两种查询中都可以使用。

在对本地数据源的查询中，也可以使用IQueryable<T>接口中的方法进行查询，只要在本地集合的最后使用一个AsQueryable方法即可。在本章的“查询表达式的创建”一节中我们会介绍AsQueryable方法。

在本节中，重点使用LINQ to SQL来介绍解释型查询的结构，因为LINQ to SQL这种方式不用事先定义查询的对象模型，表述起来会方便很多，但是本节中用到的查询语句同样适用于EF查询。

提示：IQueryable<T>实际上是对IEnumerable<T>方法的扩展，它添加了一些用于生成查询表达式树的方法。我们并不需要关心这些方法是什么时候被调用的或者如何被调用的，因为调用工作是LINQ查询框架自动完成的，开发者只需像查询本地集合那样操作集合。在“查询表达式的创建”一节中我们会对IQueryable<T>方法有更深入的解释。

使用下面的SQL脚本在数据库中创建一个的结构简单的数据表，只包含一个主键和一个用于存放名字的列，再向这个表中添加五条数据：

```
create table Customer
(
  ID int not null primary key,
  Name varchar(30)
)
insert Customer values (1, 'Tom')
insert Customer values (2, 'Dick')
insert Customer values (3, 'Harry')
insert Customer values (4, 'Mary')
insert Customer values (5, 'Jay')
```

在数据库中创建了这个表，并添加了数据之后，我们可以使用下面这段代码来取出名字中含有“a”的元素：

```
using System;
using System.Linq;
using System.Data.Linq;            // 在System.Data.Linq.dll
using System.Data.Linq.Mapping;
```

```
[Table] public class Customer
{
  [Column(IsPrimaryKey = true)] public int ID;
  [Column]                      public string Name;
}

class Test
{
  static void Main()
  {
    DataContext dataContext = new DataContext("connection string");
    Table<Customer> customers = dataContext.GetTable<Customer>();

    IQueryable<string> query = from c in customers
      where c.Name.Contains("a")
      orderby c.Name.Length
      select c.Name.ToUpper();

    foreach (string name in query) Console.WriteLine(name);
  }
}
```

LINQ to SQL的内部逻辑会将上面代码中的查询结构转化成以下标准的SQL查询语句：

```
SELECT UPPER([t0].[Name]) AS [value]
FROM [Customer] AS [t0]
WHERE [t0].[Name] LIKE @p0
ORDER BY LEN([t0].[Name])
```

上面这段代码的执行结果为：

```
JAY
MARY
HARRY
```

8.8.1 解释型查询的工作机制

下面我们看一下前面这个查询是如何执行的。

首先，编译器会将最初的查询表达式转换成如下运算符流格式：

```
IQueryable<string> query = customers.Where   (n => n.Name.Contains("a"))
                                    .OrderBy (n => n.Name.Length)
                                    .Select  (n => n.Name.ToUpper());
```

然后，编译器会进一步解析上面这个查询中的各个运算符，这就体现出了本地查询和远程数据源查询的区别：在解析这些运算符的时候，编译器使用的是Queryable类中的方法，而不是Enumerable类。

编译器通过判断变量customers的类型来决定使用哪个类来解析查询运算符。在上面这个示例中，变量customers是一个Table<>类型的变量，这个类型继承自IQueryable<T>，而IQueryable又继承自IEnumerable<T>。那么在编译器要去解释Where关键字的时候，就会面对两个Where方法的实现，一个来自Enumerable类，另一个来自Queryable类，并且这两个类中实现的Where具有完全相同的方法签名：

```
public static IQueryable<TSource> Where<TSource>
```

```
(this IQueryable<TSource> source, Expression <Func<TSource,bool>> predicate)
```

在上面示例中，编译器会选择`Queryable.Where`方法，因为这个方法和Table<>类型的匹配度更高一些。

`Queryable.Where`方法可以接收`Expression<TDelegate>`类型的变量作为参数，在本例中这个参数是一个Lambda表达式：`n=>n.Name.Contains("a")`，这个Lambda表达式在运行时会被编译成查询表达式树的形式，而不会作为C#中的方法来处理。查询表达式树是`System.Linq.Expressions`命名空间下的一种对象模型，这种对象是在运行时被解释运行的（这也是为什么LINQ to SQL和EF支持延迟加载）。

由于`Queryable.Where`方法的返回类型是`IQueryable<T>`，因此接下来的运算符`OrderBy`以及`Select`使用相同的逻辑进行解释，它们都与Queryable类中相应的方法对应。图8-9演示了这段查询的详细执行过程，在灰色的区域中，都是以一个查询表达式树的方式来描述的，这个表达式树可以在运行时被解释执行。

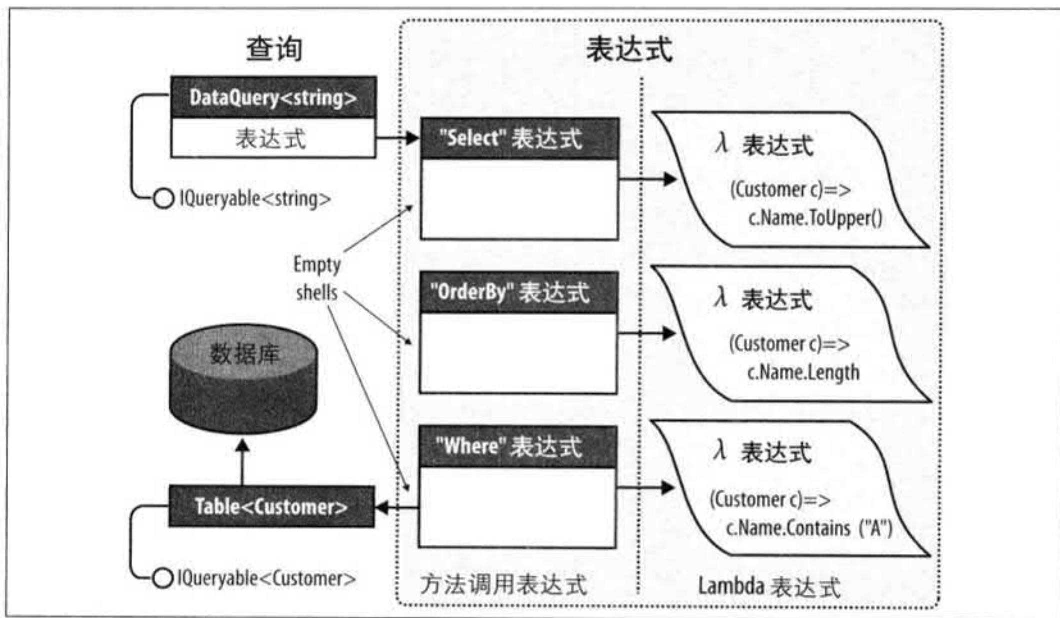


图8-9：解释型查询组成

执行

和对本地数据源的查询类似，解释型的查询也遵循延迟加载的处理方式。这意味着在LINQ中定义的查询逻辑只有在数据被真正使用时才会执行，这里的真正使用一般是遍历序列。需要注意的是，如果对一个集合在不同的地方遍历两次，那么定义的LINQ查询会被执行两次，也就是说会执行两次数据库查询。

解释型的查询和本地数据查询的本质不同在于它们的执行方式。在遍历解释型的集合时，整个LINQ查询语句会被编译成一个完整的查询表达式树来加以执行。在上面这个示例中，LINQ to SQL机制将LINQ表达式编译成SQL命令后加以执行，然后将返回在数据库中得到的结果集。

提示：为了使LINQ to SQL在远程数据的查询中能够正常运行，实际上需要导入一些特殊的标记来帮助编译器正常工作，在前面示例中的Customer类中定义了Table和Column标签，这些标签都是供编译器使用的。在“LINQ to SQL和Entity Framework”一节中我们将详细地介绍这些标签。Entity Framework也需要类似的标签，但是除了这些之外，它还需要一个额外的XML文件Entity Data Model (EDM)，在这个文件中定义了数据表和实体类之间的对应关系。

在本章的一开始，我们将LINQ表达式描述为一个流水线。当遍历IQueryable类型的集合时，查询过程和前面的IEnumerable类型的数据集合有所区别，IQueryable类型的变量并不会启动整条流水线，而是启动流水线上IQueryable的那部分，这部分是由产品经理专门负责的，产品经理首先会检查整个流水线上以查询表达式树方式定义的代码，那些被编译的代码这时不予考虑。检查查询表达式树上的代码之后，会将所有的这种表达式组合在一起，翻译成一个完整的SQL查询语句，然后执行这个SQL语句，将得到的结果返回给调用者。这个过程中，在流水线上只有取数据的那部分内容运行结束了，流水线上其他的部分会在此之后被使用，剩下的部分定义了一组处理逻辑，用于处理数据库返回的数据。

这里的规则实际上是可以自定义的。例如在对本地集合的查询中，可以实现自定义的查询方法，实现的方式很简单，就是遍历集合，在遍历中对集合元素进行需要的处理即可。但是在对远程数据的查询中，很难让编译器执行我们自定义的方法，即使可以执行，实际上也是没有必要的。举例说明，假如我们通过继承IQueryable<T>接口，并实现了一个新的MyWhere方法，方法的定义没有问题，但在执行查询时，流水线的“产品经理”在检查和组合查询表达式树的时候，不知道该如何处理MyWhere这个方法，因为在他的产品列表中，并没有这个部件，不知道该将它放在产品的哪个部位。即使我们想尽办法克服了这个困难，还存在一个问题，那就是这个方法只能针对特定的数据库进行查询，例如只能在LINQ to SQL中使用，其他实现了IQueryable接口的数据源还是不能处理这个方法。因此在Queryable类中定义一套标准的查询运算符非常重要，因为使用这组标准查询运算符定义的LINQ查询可以查询任意类型的数据源，这有很大的好处：只需定义一次LINQ查询，就可以在不同数据源查询中使用它。我们自定义的查询方法，显然就不会有这样的功能。

还需要注意的是，即使使用了标准的查询运算符，也要注意这个运算符能否被翻译成数据库能识别的查询运算符，也就是说，LINQ to SQL 和EF中可能定义了30种查询方式，但是在SQL Server的SQL查询中只有10种查询方式，而最终LINQ查询表达式要被翻译成SQL来执行，那么只能在10种查询方法中选一种来使用。如果在LINQ使用了一个功能很强大的运算符，但是在SQL中却没有相同功能的运算符，那么LINQ中的这个运算符就会被翻译成其他的SQL语句来完成这项功能。如果你比较熟悉SQL查询，就会知道SQL查询中哪些可以做、哪些做不到。那么在定义LINQ查询的时候就会联想到，某个关键字在执行的时候会被翻译成什么形式，是否会影响效率。有时候我们会遇到莫名其妙的查询错误，当深入查看LINQ翻译出来的SQL语句时，可能看到和想象完全不同的查询语句。

8.8.2 综合使用解释型的查询和本地查询

一个LINQ查询中可以同时使用解释型查询运算符和本地查询运算符。应用的典型方式就是把本地查询操作放在外层，将解释型的查询操作放在内层，在执行查询的时候，解释型的操作先执行，返回一个结果集合给外层的本地查询使用。这种查询模式经常用于LINQ对数据库的查询操作。

查询运算符绝不会修改输入序列，相反，它会返回一个新序列。这种设计是符合函数式编程规范的，LINQ的思想实际上就起源于函数式编程。

例如，假设编写一个自定义扩展方法，将集合中的字符串组合成对：

```
public static IEnumerable<string> Pair (this IEnumerable<string> source)
{
    string firstHalf = null;
    foreach (string element in source)
        if (firstHalf == null)
            firstHalf = element;
        else
        {
        yield return firstHalf + ", " + element;
        firstHalf = null;
    }
}
```

可以在一个混合使用LINQ to SQL和本地操作符的查询中使用这个扩展方法：

```
DataContext dataContext = new DataContext ("connection string");
Table<Customer> customers = dataContext.GetTable <Customer>();

IEnumerable<string> q = customers
    .Select (c => c.Name.ToUpper())
    .OrderBy (n => n)
    .Pair()           // Local from this point on.
    .Select ((n, i) => "Pair " + i.ToString() + " = " + n);
foreach (string element in q) Console.WriteLine (element);

Pair 0 = HARRY, MARY
Pair 1 = TOM, DICK
```

因为customers是一个实现IQueryable<T>的类型，所以Select操作符会解析为Queryable.Select。这同样会返回一个类型为IQueryable<T>的输出序列。但是一个查询操作符Pair并没有重载为接受IQueryable<T>——只有更一般化的IEnumerable<T>。所以，它会解析为本地方法Pair——将解释的查询封装在一个本地查询中。Pair还会返回IEnumerable，所以OrderBy封装了另一个本地操作符。

在LINQ to SQL方面，最终产生的SQL语句相当于：

```
SELECT UPPER (Name) FROM Customer ORDER BY UPPER (Name)
```

剩余的工作将在本地完成。实际上，最终会得到一个本地查询（在外部），它的来源是一个解析得到的查询（在内部）。

8.8.3 AsEnumerable方法

Enumerable.AsEnumerable可以说是一个最简单的查询运算符，下面是它的完整定义：

```
public static IEnumerable<TSource> AsEnumerable<TSource>
              (this IEnumerable<TSource> source)
{
    return source;
}
```

它的作用就是将一个IQueryable<T>类型的变量转化成IEnumerable<T>类型，在解析LINQ查询语句中的运算符时，强制使用Enumerable类中的方法。这样会使接下来的查询按照本地查询的方式进行。

举例来说，假如在SQL Server数据库中有一个名为MedicalArticles的表，我们想要从这张表中取出

所有关于感冒的文章，并且文章的概要部分要少于100个字。可以先定义一个正则表达式给后面的判断语句使用，查询代码如下：

```
Regex wordCounter = new Regex(@"\b(\w|[-'])+\b");

var query = dataContext.MedicalArticles
.Where(article => article.Topic == "influenza" &&
                    wordCounter.Matches(article.Abstract).Count < 100);
```

但上面的查询在运行时会遇到一个问题，那就是SQL Server不支持正则表达式查询，所以LINQ-to-db机制在运行时会抛出异常，异常提示不能将LINQ查询语句转换成SQL。为了避免这个异常，可以将查询分两步来进行。第一步：使用LINQ to SQL查出所有与感冒相关的文章，第二步：将得到的数据集转化为本地数据，然后再使用正则表达式进行筛选。实现方式如下：

```
Regex wordCounter = new Regex(@"\b(\w|[-'])+\b");

IEnumerable<MedicalArticle> sqlQuery = dataContext.MedicalArticles
  .Where(article => article.Topic == "influenza");

IEnumerable<MedicalArticle> localQuery = sqlQuery
  .Where(article => wordCounter.Matches(article.Abstract).Count < 100);
```

由于变量sqlQuery是IEnumerable<MedicalArticle>类型的，第二个查询表达式在对它进行查询的时候，就会按照本地集合来处理，并且在编译器解析LINQ语句中的运算符时，会优先执行Enumerable类中的对应方法。

使用AsEnumerable方法可以将两个查询简化成一个：

```
Regex wordCounter = new Regex(@"\b(\w|[-'])+\b");

var query = dataContext.MedicalArticles
  .Where(article => article.Topic == "influenza")

  .AsEnumerable()
  .Where(article => wordCounter.Matches(article.Abstract).Count < 100);
```

还有两种方式可以间接地调用AsEnumerable方法，那就是ToArray方法和ToList方法。使用AsEnumerable方法有下面两点好处，一是这个方法不会强制查询立即执行，但是如果希望查询立即执行的话，就要使用另外两个方法了；二是它不会创建本地的存储结构，因此它会比较节省资源。

提示：将查询逻辑从数据库移到本地会降低查询的性能，特别是当查询的数据量比较大的时候，效率损失更加严重。同样针对上面这个示例，有一个更高效（同时也更复杂）的方式来完成上面的查询，那就是使用SQL CLR在数据库端实现正则表达式的查询。

我们将在第10章中对解释型的查询和本地查询的综合使用再做进一步介绍。

8.9 LINQ to SQL 和 Entity Framework

在接下来的章节中，我们都使用LINQ to SQL（L2S）和Entity Framework（EF）来演示解释型查询的使用方式，在这一节中介绍这些技术的重要特性。

提示：如果你已经对L2S很熟悉了，可以看一下本章最后的表8-1中列出的L2S的API。

LINQ to SQL 和Entity Framework的区别

LINQ to SQL和EF都是用LINQ来实现的对象的映射工具，它们之间的不同在于映射的方式，我们知道，在数据库查询中，映射的一端是数据库表，LINQ to SQL可以将数据库表结构映射成对象，然后供调用者使用，这种映射严格按照数据库表结构，映射成的对象不需要我们定义。与之不同的是，EF对这种映射做了一些改进，那就是允许我们定义实体类，也就是允许开发者定义数据库表被映射成什么类型。这种映射提供了一种更灵活的解决方案，但是它会降低查询性能，也增加了使用的复杂度，因为需要占用额外的时间去维护数据库和自定义的实体类间的映射关系。

L2S是由微软的C#团队完成的，在Framework3.5中发布，而EF是由ADO.NET团队在ADO.NET SP1中发布的。后来L2S的开发和维护由ADO.NET团队来接管，由于开发重心的不同，在.NET Framework 4.0中对L2S的改变很少，而主要的改进集中在EF方面。

尽管在性能上和易用性上，EF在.NET Framework 4.0中已经有了极大的改进，但是两种技术还是各有优势。L2S的优点是简单易用、执行性能好、此外它生成的SQL语句的解释质量更好一些。EF的优点是允许我们创建自定义的持久化的实体类，用于数据库的映射。另外EF允许使用同一个查询机制查询SQL Server之外的数据源，实际上L2S也支持这个功能，但是为了鼓励第三方的查询机制的出现，L2S中没有对外公布这些机制。

EF4.0突出的改进是它支持几乎所有的L2S中的查询方法。这意味着，在本书前面演示的所有LINQ to DB的查询语句都适用于EF的查询。这个改进还有一个好处，就是让学习者可以更轻松地了解LINQ的查询机制，因为在LINQ to DB的查询中，我们不用考虑实体类以及它和数据表之间的映射关系而把主要精力集中在LINQ本身。

8.9.1 LINQ to SQL实体类

L2S允许任何类来承载数据，只要类中加入了合适的标签即可。下面是一个简单的示例：

```
[Table]
public class Customer
{
  [Column(IsPrimaryKey = true)]
  public int ID;

  [Column]
  public string Name;
}
```

这里的[Table]标签定义在System.Data.Linq.Mapping命名空间中，它定义的类型用来承载数据表中的一行数据。默认情况下，L2S会认为这个类名和它对应的表名是相同的，如果想让两者不同的话，由于表名已经固定，只能更改对应的类名，更改方式是在[Table]标签中显式地指定类名：

```
[Table (Name="Customers")]
```

在L2S中，如果一个类具有[Table]标签，就称这个类为实体，为了能够顺利使用，这个实体的结构必

须与数据表的结构相匹配，多字段或少字段都不行。这种限制使得这种映射是一种低级别的映射。

[Column]标签用来指示数据表中的某列，如果实体中定义的列名和数据表中的别名不同，那么需要在[Column]标签中特别指出所对应的列名，如下面的两行代码所示：

```
[Column(Name = "FullName")]
public string Name;
```

[Column]标签中的IsPrimaryKey属性用于指示当前列是主键，在数据库中这列用于唯一标识一条数据，在程序中也用这列区分不同的实体，将实体中的变化更新到数据库的时候，也需要使用这一列来确定写入的目标。

在实体类中，除了可以定义公共的属性外，我们还可以定义私有的字段来对应数据库中的列。因此，我们可以在属性的set结构中定义判断逻辑，用于检查能否给这个字段赋值，而从数据库读取数据的时候，则直接将数据赋值给私有字段即可。下面是一个简单示例：

```
string _name;

[Column(Storage = "_name")]
public string Name { get { return _name; } set { _name = value; } }
```

这里的Column(Storage="_name")表示L2S在读取数据库的的时候，直接向_name中赋值即可，不用通过Name属性。总的来讲，在定义实体类的时候，L2S允许将数据库的字段映射对象（实体中的属性）定义成私有的，它可以访问到实体类中的私有变量。

提示： 实际上与数据库表对应的实体类是可以自动生成的，不用逐行书写，常用的生成工具有Visual Studio（需要在“工程”菜单添加一个“LINQ to SQL Classes”选项）和命令行工具*SqlMetal*。

8.9.2 Entity Framework的实体类

和L2S中的实体类相似，EF中允许开发者定义自己的实体类用于承载数据，不同的是，EF中的实体类的定义要灵活得多，在理论上允许任何类型的类来作为实体类使用（在某些特殊情况下需要实现一些接口，例如定义导航属性的时候），也就是说实体类中的结构不用和数据表中的字段完全对应。下面是一个和Customer表对应的实体类的代码段，这段代码就不是严格和数据库表结构对应的：

```
// 在使用时需要导入System.Data.Entity.dll命名空间

[EdmEntityType(NamespaceName = "NutshellModel", Name = "Customer")]
public partial class Customer
{
  [EdmScalarPropertyAttribute(EntityKeyProperty = true, IsNullable = false)]
  public int ID { get; set; }

  [EdmScalarProperty(EntityKeyProperty = false, IsNullable = false)]
  public string Name { get; set; }
}
```

和L2S不同的是，在EF中，要完成数据的映射和查询，只定义上面这个实体类是不够的。因为在EF中，查询并不是直接针对数据库进行的，它使用了一种更高级别的抽象模型，称为*实体数据模型*（EDM，Entity Data Model），我们的查询语句是针对这个模型来定义的。EDM实际上是使用XML定义的一个*.edmx*类型的文件，这个文件包含三部分内容：

- 概念模型：定义了数据库的信息，不同的数据库有不同的概念模型内容。
- 存储模型：定义了数据库的表结构。
- 映射：定义了数据库表和实体类之间的映射关系。

创建*.edmx*文件最简单的方式是使用Visual Studio，在“项目”菜单中点击“添加新项”，在弹出的窗口中选择“ADO.NET Entity Data Model”。之后使用向导就可以完成实体类到数据表的映射配置。这一系列操作不仅添加一个*.edmx*文件，还会创建涉及到的实体类。

提示： 在EF中实体类都是映射到概念模型上，所有对概念模型的查询和更新操作，都是由*Object Services*发起的。

EF的设计者在设计的时候将映射关系想得比较简单，他们假设数据表和实体类之间的映射关系是1:1的，所以并没有提供专门的机制去完成一对多或者多对一的映射。尽管这样，如果确实需要这种特殊的映射关系，还是可以通过修改*.edmx*文件中的相关内容来实现。下面是几个常用的修改操作：

- 多个表映射到一个实体类。
- 一个表映射到多个实体类。
- 按照ORM世界中的三种继承方式将继承的类映射到表。

三种继承策略是：

每个分层结构一张表
: 一张表映射到整个类分层结构。该表中包含分隔符列，用于指出每个行应该映射到哪个类。

每个类一张表
: 一张表映射到一个类，意味着继承的类映射到多张表。查询某个实体时，EF生成SQL JOIN，以合并其所有基类。

每个具体类一张表
: 一张单独的表映射到每个具体的类。这意味着基类映射到多张表，并且在查询基类的实体时，EF生成SQL UNION。

（比较一下，L2S仅支持每个分层结构一张表。）

提示： EDM非常复杂，包含很多的细节，如果要涉及到所有的细节部分，可能要用几百页的篇幅来描述。对这方面内容感兴趣的读者可以参考Julia Lerman的《*Programming Entity Framework*》。

此外，EF还支持LINQ之外的查询方式，有一种语言叫Entity SQL（ESQL），使用这种语言，我们可以通过EDM查询数据库。这种查询方式非常便于动态地构建查询语句。

8.9.3 DataContext和ObjectContext

在创建了实体类之后（如果是EF的话还需要有EDM文件），就可以对数据库进行查询了。在查询之前，首先要创建`DataContext`（L2S）或者`ObjectContext`（EF）对象，这个对象用于指定数据库连接字符串。

```
var l2sContext = new DataContext("database connection string");
var efContext = new ObjectContext("entity connection string");
```

提示： 直接创建DataContext/ObjectContext实例是一种很底层的使用方式，它可以展示出这两种类型是如何工作的。但在实际应用中，更常用的方式是创建类型化的*context*（继承自*DataContext* /*ObjectContext*）来使用，它的使用方式会在接下来的内容中简要介绍。

对于L2S来说，我们只需为DataContext传递一个数据库连接字符串即可；而对于EF，传递的是数据库连接实体，这个实体中除了数据库连接字符串外，还包括EDM文件的路径信息。（如果通过Visual Studio创建EDM文件，那么系统会自动在项目的*app.config*文件中添加完整的数据库连接实体，可以从这个文件得到需要的信息）。

然后我们就可以使用GetTable（L2S）或者CreateObjectSet（EF）对象了，这两个对象都是用于从数据库中读取数据，下面是两个简单的示例，演示这两个对象如何使用：

```
var context = new DataContext("database connection string");
Table<Customer> customers = context.GetTable<Customer>();

Console.WriteLine(customers.Count());                // 表中的行数

Customer cust = customers.Single(c => c.ID == 2);    // 检索用户ID为2的记录
```

下面是EF的示例：

```
var context = new ObjectContext("entity connection string");
context.DefaultContainerName = "NutshellEntities";
ObjectSet<Customer> customers = context.CreateObjectSet<Customer>();

Console.WriteLine(customers.Count());                // 表中的行数

Customer cust = customers.Single(c => c.ID == 2);    // 检索用户ID为2的记录
```

提示： Single运算符会根据主键从结果集中取出一行记录。和First关键字不同的是，Single运算符要求结果集中只有一条记录，当结果集中的结果多于一行时，它会抛出异常；而First关键字在这种情况下则不会抛出异常。

DataContext/ObjectContext这两个对象实际上只做两件事情。第一，它作为一个工厂，将我们查询的数据组合成对象。第二，它会维护实体类的状态，如果查询出的实体类中的值在类外改变了，它会记录下这个字段，然后便于更新回数据库。下面我们接着上面的代码完成数据的更新操作：

```
Customer cust = customers.OrderBy(c => c.Name).First();
cust.Name = "Updated Name";
context.SubmitChanges();
```

在EF中，唯一的不同点是使用SaveChanges方法代替SubmitChanges方法：

```
Customer cust = customers.OrderBy(c => c.Name).First();
cust.Name = "Updated Name";
context.SaveChanges();
```

1. 类型化的contexts

在对数据库的查询中，每次都调用GetTable<Customer>()或者CreateObjectSet<Customer>()显然不是一个很好的选择，一个更好的方式是为每个数据库定义一个继承自DataContext/ObjectContext的子类，一般会为每个实体类都添加一个这样的属性，这种属性我们称之为类型化的contexts：

```
class NutshellContext : DataContext   // 对于LINQ to SQL
{
  public Table<Customer> Customers
  {
    get { return GetTable<Customer>(); }
  }
  // 对于数据库中的每个表
}
```

在EF中这样定义：

```
class NutshellContext : ObjectContext   // 对于实体框架
{
  public ObjectSet<Customer> Customers
  {
    get { return CreateObjectSet<Customer>(); }
  }
  // 对于概念模型中的每个实体
}
```

接下来就可以这样使用这个对象了：

```
var context = new NutshellContext("connection string");
Console.WriteLine(context.Customers.Count());
```

如果通过Visual Studio在项目中创建"LINQ to SQL Classes"或者添加"ADO.NET Entity Data Model"项，VS会自动创建这种类型化的contexts。除此之外设计器还会做一些额外的工作，例如将标识符复数化。所谓复数化，具体到这个示例来讲，就是在使用Customer实体的时候，应该以复数的方式调用context.Customers，而不是context.Customer，即使数据库表和实体类都是单数Customer，系统也会自动要求使用复数。

销毁DataContext/ObjectContext对象

尽管DataContext/ObjectContext都实现了IDisposable接口，而且Dispose方法会强制断开数据库连接，但是我们一般不通过调用Dispose方法来销毁这两个对象，因为L2S和EF在返回查询结果后会自动断开连接。实际上销毁这两个对象会带来一些问题，例如下面这个示例：

```
IQueryable<Customer> GetCustomers (string prefix)
{
    using (var dc = new NutshellContext ("connection string"))
    return dc.GetTable<Customer>()
             .Where (c => c.Name.StartsWith (prefix));
}
...
foreach (Customer c in GetCustomers ("a"))
    Console.WriteLine (c.Name);
```

上面这段代码在执行时会出现错误，因为在遍历GetCustomers的返回结果时，DataContext对象已经销毁了。

下面是一些关于释放context对象需要注意的地方：

- 在系统调用Close方法来释放连接的时候，有个很重要的问题是系统中非托管的资源会不会同时被释放掉。在SqlConnection中，可以保证不再使用的非托管资源也会被释放。但在理论上，第三方的连接在调用了Close方法之后，并不能保证释放掉非托管资源，尽管这违反了接口方法IDbConnection.Close的初衷，但是第三方的连接确实存在这样的问题。
- 如果在一个查询中使用GetEnumerator而不是foreach来遍历集合，并且在循环结束后没有释放枚举对象也没有释放结果集的使用者的话，那么数据库连接不会被断开。在这个时候可以通过销毁DataContext/ObjectContext对象来释放连接。
- 有的开发者会觉得在一个对象结束使用之后，只要它实现了IDisposable接口，就应该显式地调用Dispose方法来销毁对象，当然查询结束之后也应该销毁context对象，这样会使得代码更加简洁、准确。

如果想显式地销毁contexts对象，那么必须传递一个DataContext/ObjectContext对象到诸如GetCustomers这样的方法中，以避免出现上面描述的问题。

2. 对象状态跟踪

DataContext/ObjectContext对象有跟踪实体类状态的功能，当取出一个表中的数据保存到本地内存之后，如果下次再到数据库中查询某条已经存在的数据，DataContext/ObjectContext并不会去数据库中读取数据，而是直接从内存中取出需要的数据。也就是说，在一个context的生命周期中，它不会将数据库中的某行记录返回两次（数据记录之间使用主键进行区分）。这个很容易理解，因为如果context为某条记录返回两个实体对象的话，需要同时维护两个对象的状态。如果两个数据在不同的地方被更改，那么就会造成非常多的潜在问题，例如更新的时候选用哪个实体中的数据；如果有人再请求这条数据，应该返回哪个实体等。

提示：L2S和EF都允许关闭对象状态跟踪功能，为避免这些限制，在L2S中将DataContext对象的ObjectTrackingEnabled属性设置成false即可。在EF中禁用对象跟踪的功能要麻烦一点，它需要在每个实体中都添加下面的代码：

```
context.Customers.MergeOption = MergeOption.NoTracking;
```

关闭对象状态跟踪功能之后，为了数据安全，通过context向数据库中提交更新的功能也同时被禁用。

为了说明对象状态跟踪的工作方式，假设一个customer的名字是字母表中的第一个字母a，并且它的ID也是数据库表中最小的。那么在下面这个示例中，a和b两个对象指向的是同一个内存对象：

```
var context = new NutshellContext("connection string");

Customer a = context.Customers.OrderBy (c => c.Name).First();
Customer b = context.Customers.OrderBy (c => c.ID).First();
```

这种查询机制有两个副作用，第一，想象一下在L2S和EF中，第二个查询语句的执行过程，它首先

会到数据库中查询出需要的数据，然后得到这条数据的主键，再到context缓存的实体类中查看有没有相同主键的记录，如果有，则返回缓存的那条记录而删掉数据库中的那条记录。所以如果数据库中的那条记录的Name字段被别的用户更新过，这个时候取出的Name还是未更新过的，它成了过时的数据。由于Customer对象有可能在任何地方被跟新，这个时候，我们就需要想办法控制这种预想之外的更新，以及在并发情况下如何保持数据一致性。实体类在创建的时候一般都有SubmitChanges/SaveChanges这样的属性，这种属性用于指示当前实体数据的状态，通过它们可以避免上述问题的出现。

提示： 如果要从数据库中得到最新的数据，必须定义一个新的context对象，将旧的实体类传给这个对象，然后调用Refresh方法，这样，最新的数据就会被更新到实体类中。

第二个副作用是我们不能将查询结果映射到自己定义的实体类上，例如想要得到整个表的一部分列的时候，就会出现一些问题。举例来说，如果只想通过一次查询得到customer的名字，下面这三种方式都是可以的：

```
customers.Select (c => c.Name);
customers.Select (c => new { Name = c.Name } );
customers.Select (c => new MyCustomType { Name = c.Name } );
```

但下面这种方式不可以：

```
customers.Select (c => new Customer { Name = c.Name } );
```

最后的这个查询会导致查询的Customer实体都只包含一个名字属性，如果想要通过查询customer表得到更多的数据，会发现查询结果中不会包含更多的数据，Customer实体中只有名字一个属性。

提示： 在一个多层次的系统中，不能在系统的中间层定义一个静态的DataContext或者ObjectContext实例完成所有的数据库查询操作，因为context对象不能保证线程安全。正确的做法是在中间层的方法中，为每个请求的客户创建一个context，这样做的好处是可以减轻数据库的负担，因为维护和更新实体类的任务被多个context对象分担。对于数据库来说，更新操作会通过多个事务执行完成，这显然比一个很大的事务要高效很多。

8.9.4 联合查询

使用实体类生成工具还有一个特点，当表之间有关联关系的时候，我们可以直接使用关联表中的属性，实体类自动完成了关联的字段和关联表的映射。举例来说，假如我们定义了Customer和Purchase两个表，这两个表是一对多的关系，定义这两个表的SQL脚本如下：

```
create table Customer
(
  ID int not null primary key,
  Name varchar(30) not null
)

create table Purchase
(
  ID int not null primary key,
  CustomerID int references Customer (ID),
```

```
  Description varchar(30) not null,
  Price decimal not null
)
```

使用自动生成的实体类，查询应该这样定义：

```
var context = new NutshellContext ("connection string");

// 返回第一个客户所有的购买记录

Customer cust1 = context.Customers.OrderBy (c => c.Name).First();
foreach (Purchase p in cust1.Purchases)
  Console.WriteLine (p.Price);
// 返回购买东西最少的客户

Purchase cheapest = context.Purchases.OrderBy (p => p.Price).First();
Customer cust2 = cheapest.Customer;
```

此外，如果cust1和cust2指向同一个customer的时候，c1和c2也将指向同一个对象，也就是说cust1==cust2的结果是true。

我们来看一下自动生成的customer实体中Purchases的结构类型。首先在L2S中：

```
[Association (Storage="_Purchases", OtherKey="CustomerID")]
public EntitySet <Purchase> Purchases { get {...} set {...} }
```

在EF中：

```
[EdmRelationshipNavigationProperty ("NutshellModel", "FK...", "Purchase")]
public EntityCollection<Purchase> Purchases { get {...} set {...} }
```

两个Purchases的类型分别是EntitySet和EntityCollection，在这两种类型中有一个内置的Where查询，只返回符合条件的实体。L2S查询中[Association]标签的作用是提供生成SQL语句所需的信息；而EF中的[EdmRelationshipNavigationProperty]标签的作用是告诉EF要到EDM中去查找两个表的关联关系。

上面这种查询，和前面其他查询类似，L2S和EF的查询方式仍然是延迟加载。在L2S查询中，真正的查询会在遍历结果集时进行，而EF的查询则是在显式地调用了Load方法之后才会执行。

下面定义L2S中的Purchases.Customer属性，也就是关联关系的另一侧：

```
[Association (Storage="_Customer",ThisKey="CustomerID",IsForeignKey=true)]
public Customer Customer { get {...} set {...} }
```

虽然Purchases.Customer属性的类型是Customer，但是它真正用于存储数据的字段（_Customer）是EntityRef类型的。EntityRef类型实现了延迟加载这一特性，所以直到真正被使用的时候，Customer属性中的数据才从数据库中取出。

在这个特性上EF的工作方式和L2S是相同的，唯一的不同点是，L2S是在访问Customer数据的时候直接从数据库中读取数据，而EF则需要显式地调用EntityReference对象中的Load方法。这意味着EF必须同时向Customer的父类型和EntityReference都提供Customer这个属性：

```
[EdmRelationshipNavigationProperty ("NutshellModel", "FK..., "Customer")]
public Customer Customer { get {...} set {...} }

public EntityReference<Customer> CustomerReference { get; set; }
```

提示： 可以通过设置下面这个属性使EF和L2S以相同的方式返回EntityCollection和EntityRefere nces：

```
context.ContextOptions.DeferredLoadingEnabled = true;
```

8.9.5 L2S和EF的延迟加载

和对本地集合的查询相同，L2S和EF在原则上也是遵循延迟加载的查询方式，这种查询方式允许开发者以渐进的方式构建查询。除了常规的延迟加载查询之外，L2S和EF还支持其他类型的延迟加载，例如子查询出现在Select表达式中时：

- 在本地集合查询中，使用的是双重的延迟加载，因为从查询的执行过程来看，我们是在对前一个查询的结果执行查询，所以如果外层的查询结束，但内层的查询没有执行的时候，实际上外层查询没有结果。
- 在L2S/EF查询中，外层查询和内层查询是作为一个整体运行的，不存在内层查询先执行，外层查询后执行的问题，这样可以避免一部分数据的重复传递。

例如，下面这个示例中，整个查询会在第一个foreach语句第一次循环的时候完全执行：

```
var context = new NutshellContext("connection string");

var query = from c in context.Customers
            select
               from p in c.Purchases
               select new { c.Name, p.Price };

foreach (var customerPurchaseResults in query)
  foreach (var namePrice in customerPurchaseResults)
     Console.WriteLine(namePrice.Name + " spent " + namePrice.Price);
```

并且显式定义的EntitySets/EntityCollections映射对象也会在查询中全部返回：

```
var query = from c in context.Customers
        select new { c.Name, c.Purchases };

foreach (var row in query)
  foreach (Purchase p in row.Purchases)    // 无额外的往返开销
     Console.WriteLine(row.Name + " spent " + p.Price);
```

但是如果循环中处理的EntitySet/EntityCollection是未经映射的结构，那么整个查询就会按照普通的延迟加载来执行，下面这个示例演示了使用L2S和EF在循环中处理Purchases查询：

```
context.ContextOptions.DeferredLoadingEnabled = true; // 只需对EF查询设置这个属性

foreach (Customer c in context.Customers)
  foreach (Purchase p in c.Purchases) // 另一个SQL查询
    Console.WriteLine(c.Name + " spent " + p.Price);
```

这种查询模式的好处是可以在代码中加入判断逻辑，根据判断结果，有选择地输出数据，而不是全部输出，下面这个示例演示了这种操作方式，结果只输出那些有不良信用记录的消费者的消费记录：

```
foreach (Customer c in context.Customers)
  if (myWebService.HasBadCreditHistory (c.ID))
    foreach (Purchase p in c.Purchases) // 另一个SQL查询
      Console.WriteLine (...);
```

（在第9章的“映射”一节中我们将对Select运算符中的子查询做更深入的探讨。）

我们已经看到，通过显式地映射结果集，可以避免查询操作的重复执行。实际上，在L2S和EF中，还提供了其他的机制完成同样的功能，在接下来的两章中将会介绍这些方法。

8.9.6 DataLoadOptions

DataLoadOptions类是L2S中一个特有的类，它有两个作用：

- 它允许我们为EntitySet所关联的类指定一个筛选条件。
- 它可以强制加载特定的EntitySets，这样可以减少整个数据查询的次数。

1. 设置更多的过滤条件

这里重构前面的部分查询代码：

```
foreach (Customer c in context.Customers)
  if (myWebService.HasBadCreditHistory (c.ID))
    ProcessCustomer (c);
```

ProcessCustomer方法的定义如下：

```
void ProcessCustomer (Customer c)
{
  Console.WriteLine (c.ID + " " + c.Name);
  foreach (Purchase p in c.Purchases)
    Console.WriteLine (" - purchased a " + p.Description);
}
```

如果想要通过查询输出那些信用较好的消费者的消费记录，可以使用下面这种定义方式：

```
foreach (Customer c in context.Customers)
{
  if (myWebService.HasBadCreditHistory (c.ID))
     ProcessCustomer (c.ID,
                      c.Name,
                      c.Purchases.Where (p => p.Price > 1000));
...
void ProcessCustomer (int custID, string custName, IEnumerable<Purchase> purchases)
{
  Console.WriteLine (custID + " " + custName);
  foreach (Purchase p in purchases)
     Console.WriteLine (" - purchased a " + p.Description);
}
```

这种实现方式不是很简洁清晰，如果ProcessCustomer方法需要更多参数的话，那么上面的代码将更加杂乱。一个比较好的替代方案是使用DataLoadOptions的AssociateWith方法：

```
DataLoadOptions options = new DataLoadOptions();
options.AssociateWith <Customer>
 (c => c.Purchases.Where (p => p.Price > 1000));
context.LoadOptions = options;
```

这样，DataContext会根据指定的判别式来筛选Customer对象中的Purchases属性。这个时候就可以使用简洁清晰的ProcessCustomer方法了。

LINQ查询

AssociateWith方法并不会影响延迟加载这一特性。如果需要增加新的筛选条件，只需要更改AssociateWith中的判断条件。

2. 立即加载

DataLoadOptions的第二个用处是强制加载EntitySets及其父实体对象的数据。例如，我们希望在一个查询中得到所有的Customers和其中的Purchases信息，就可以使用下面的代码：

```
DataLoadOptions options = new DataLoadOptions();
options.LoadWith <Customer> (c => c.Purchases);
context.LoadOptions = options;

foreach (Customer c in context.Customers)     // 循环一次
  foreach (Purchase p in c.Purchases)
     Console.WriteLine (c.Name + " bought a " + p.Description);
```

上面这段代码的结构可以保证，每次在Customer对象被返回的时候，它关联的Purchases对象同时被返回。实际上我们可以同时使用LoadWith和AssociateWith这两个方法。下面这个示例演示的是在得到Customer结果的同时，同时返回符合条件的purchases，返回的purchases只是整体的一部分：

```
options.LoadWith <Customer> (c => c.Purchases);
options.AssociateWith <Customer>
  (c => c.Purchases.Where (p => p.Price > 1000));
```

8.9.7 Entity Framework中的立即加载

在EF中可以使用Include方法实现数据的立即加载。下面这段代码用于输出Customer中的Purchases属性的信息，我们可以使用Include方法让整个查询在一个SQL查询中完成：

```
foreach (var c in context.Customers.Include ("Purchases"))
  foreach (var p in c.Purchases)
    Console.WriteLine (p.Description);
```

Include方法没有广度和深度的限制，也就是可以在查询中连续使用多个Include方法。例如，假如Purchases属性和PurchaseDetails以及SalesPersons相关联，我们可以使用Include方法一次返回所有的关联数据：

```
context.Customers.Include ("Purchases.PurchaseDetails")
                 .Include ("Purchases.SalesPersons")
```

8.9.8 更新

前面已经提到过，L2S和EF都会跟踪实体类的状态，如果实体中的数据有所改变，我们可以将这些改变更新回数据库，更新的方式是调用DataContext类中的SubmitChanges方法，在EF中则是使用ObjectContext对象的SaveChanges方法。

除此之外，L2S的Table<T>类还提供了InsertOnSubmit和DeleteOnSubmit方法用于插入和删除数据表中的记录；而EF的ObjectSet<T>类提供了AddObject和DeleteObject方法来完成相同的功能。下面演示如何使用这些方法向数据库中添加记录：

```
var context = new NutshellContext ("connection string");
```

```
Customer cust = new Customer { ID=1000, Name="Bloggs" };
context.Customers.InsertOnSubmit (cust);      // EF中使用AddObject方法
context.SubmitChanges();                      // EF中使用SaveChanges方法
```

添加完成之后，再从数据库中取出这条数据，更新后再删除这条记录：

```
var context = new NutshellContext ("connection string");

Customer cust = context.Customers.Single (c => c.ID == 1000);
cust.Name = "Bloggs2";

context.SubmitChanges();                   // 更新记录

context.Customers.DeleteOnSubmit (cust); // DeleteObject方法
context.SubmitChanges();                   // 删除记录
```

SubmitChanges/SaveChanges会记录context创建以来实体类中的所有数据变化，然后将这些变化更新回数据库中，在更新的过程中，需要创建一个TransactionScope对象来帮助完成，以免更新过程中造成的错误数据。

也可以使用EntitySet/EntityCollection类中的Add方法向数据库中添加新的记录。在调用了SubmitChanges或者SaveChanges方法之后，实体中新添加的记录的外键信息会被自动取出来。下面演示Add方法的使用：

```
Purchase p1 = new Purchase { ID=100, Description="Bike", Price=500 };
Purchase p2 = new Purchase { ID=101, Description="Tools", Price=100 };

Customer cust = context.Customers.Single (c => c.ID == 1);

cust.Purchases.Add (p1);
cust.Purchases.Add (p2);

context.SubmitChanges();  // (在EF中使用SaveChanges方法)
```

提示： 为新添加的实体对象添加主键值比较繁琐，因为我们需要保证这个主键是唯一的，解决方法是可以在数据库中定义自增类型的主键，或者使用Guid作为主键。

在上面这个示例中，L2S/EF会自动将purchases对象中的CustomerID属性设置成1。L2S能够识别它们的关联关系并赋值是因为实体类中有这样的关联定义，而EF之所以可以自动识别关联关系并赋值是因为EDM中存储了这两种实体间的关联关系以及关联字段。

```
[Association (Storage="_Purchases", OtherKey="CustomerID")]
public EntitySet <Purchase> Purchases { get {...} set {...} }
```

如果这里用到的Customer和Purchase实体是由Visual Studio设计器或者*SqlMetal*自动生成的，那么生成的实体还会包含一些额外的代码，这些代码用于同步关联实体间数据。具体来说，当向Purchase.Customer属性赋值之后，在关联的另一侧，Customer.Purchases属性也会同时被自动赋值，反之亦然。还是使用前面用过的示例来解释这一点：

```
var context = new NutshellContext("connection string");

Customer cust = context.Customers.Single(c => c.ID == 1);
```

```
new Purchase { ID = 100, Description = "Bike ", Price = 500, Customer = cust };
new Purchase { ID = 101, Description = "Tools", Price = 100, Customer = cust };

context.SubmitChanges();    // (在EF中是SaveChanges方法)
```

当从EntitySet/EntityCollection对象中移除一行后，它的外键的值会自动被设置成null。下面这段代码将移除前面添加到Customer中的Purchases对象：

```
var context = new NutshellContext("connection string");

Customer cust = context.Customers.Single(c => c.ID == 1);

cust.Purchases.Remove(cust.Purchases.Single(p => p.ID == 100));
cust.Purchases.Remove(cust.Purchases.Single(p => p.ID == 101));

context.SubmitChanges(); // 将SQL语句提交给数据库执行(EF中是SaveChanges方法)
```

由于在删除关联信息的时候，每个purchase对象的CustomerID属性会被设置成null，因此就要求数据表Purchase的CustomerID字段是可空的，否则在进行设置的时候，会抛出异常。如果数据库中的字段是非空的，那么将实体中的CustomerID属性的类型设置成可空的避免抛出异常。

或者从Table<T>或者ObjectSet<T>中将涉及到的实体对象全部移除，就不会牵涉到设置外键的操作了。在L2S中，代码如下：

```
var c = context;
c.Purchases.DeleteOnSubmit (c.Purchases.Single (p => p.ID == 100));
c.Purchases.DeleteOnSubmit (c.Purchases.Single (p => p.ID == 101));
c.SubmitChanges();          // Submit SQL to database
```

在EF中，代码如下：

```
var c = context;
c.Purchases.DeleteObject(c.Purchases.Single(p => p.ID == 100));
c.Purchases.DeleteObject(c.Purchases.Single(p => p.ID == 101));
c.SaveChanges();          // 将SQL语句提交给数据库执行
```

8.9.9 L2S和EF的API对比

从前面的大量示例我们可以看出，L2S和EF具有极其相似的用法，但是二者在具体的API上有一定的区别，表8-1列出了这两种技术API上的对比。

表8-1：L2S和EF的API对比

功能	LINQ to SQL	Entity Framework
各种操作的基础类	DataContext	ObjectContext
从数据库中取出指定类型的所有记录	GetTable	CreateObjectSet
方法的返回类型	Table<T>	ObjectSet<T>
将实体中的属性值的变化（添加、删除等）更新回数据库	SubmitChanges	SaveChanges
使用context更新的方式向数据库中添加新的记录	InsertOnSubmit	AddObject

表8-1：L2S和EF的API对比（续）

功能	LINQ to SQL	Entity Framework
使用context更新的方式删除记录	DeleteOnSubmit	DeleteObject
关联表中用于存放多条关联记录的属性	EntitySet<T>	EntityCollection<T>
关联表中用于存放单条关联记录的属性	EntityRef<T>	EntityReference<T>
加载关联属性的默认方式	Lazy	Explicit
构建立即加载的查询方式	DataLoadOptions	Include()

8.10 查询表达式的创建

到目前为止，本章已经用到了很多动态创建的查询表达式，这种查询表达式连续使用多个查询运算符。这种查询方式在大多数情况下是非常实用的，但有些情况下，例如需要根据上下文的逻辑选择不同的Lambda表达式的时候，这种方式就有局限了。

在本节中，将介绍一些其他用于构建动态查询的方法，本节中我们将使用下面这个Product类作为示例：

```
[Table] public partial class Product
{
  [Column(IsPrimaryKey=true)]        public int ID;
  [Column]                           public string Description;
  [Column]                           public bool Discontinued;
  [Column]                           public DateTime LastSale;
}
```

8.10.1 委托与查询表达式树

回忆以下内容：

- 在对本地集合的查询中，使用Enumerable类中的操作作为各种查询操作的委托。
- 在解释型的查询中，使用Queryable类中的运算符创建查询，最终这些查询语句会转换成查询表达式树的形式被执行。

下面这几行代码是Where运算符在Enumerable和Queryable中的方法签名，比较它们之间的区别：

```
public static IEnumerable<TSource> Where<TSource> (this
  IEnumerable<TSource> source, Func<TSource,bool> predicate)

public static IQueryable<TSource> Where<TSource> (this
  IQueryable<TSource> source, Expression<Func<TSource,bool>> predicate)
```

我们可以在查询表达式中嵌入Lambda表达式，而无论使用Enumerable中的方法查询，还是IQueryable中的方法查询，嵌入的Lambda表达式在两种查询中的形式是完全相同的：

```
IEnumerable<Product> q1 = localProducts.Where (p => !p.Discontinued);
IQueryable<Product>  q2 = sqlProducts.Where   (p => !p.Discontinued);
```

对中间变量使用Lambda表达式的时候，必须清楚这个Lambda表达式会被委托方法使用（例如

Func<>）还是被查询表达式树使用（例如 Expression<Func<>>）。在下面这个示例中，predicate1和predicate2两个变量是不可以互换的：

```
Func <Product, bool> predicate1 = p => !p.Discontinued;
IEnumerable<Product> q1 = localProducts.Where (predicate1);

Expression <Func <Product, bool>> predicate2 = p => !p.Discontinued;
IQueryable<Product> q2 = sqlProducts.Where (predicate2);
```

1. 查询表达式树的编译

可以通过调用Compile方法将查询表达式树的查询转换成委托方法的查询。在一些特定的情况下，例如需要定义一个方法来返回一些可重用的查询表达式时，Compile方法可以发挥很好的作用。举例说明这一点，我们在Product类中添加一个静态的方法，这个方法用于判断某种产品是否一直在生产，如果一直在生产并且在过去30天内有销售，那么就返回true，代码如下：

```
public partial class Product
{
  public static Expression<Func<Product, bool>> IsSelling()
  {
    return p => !p.Discontinued && p.LastSale > DateTime.Now.AddDays (-30);
  }
}
```

这里将自己定义的方法写在一个用partial关键字定义的类中，这样可以防止在自动生成DataContext对应的实体类时将这个方法覆盖。

上面定义的这个方法在本地查询和解释型的查询中都可以使用：

```
void Test()
{
  var dataContext = new NutshellContext ("connection string");
  Product[] localProducts = dataContext.Products.ToArray();

  IQueryable<Product> sqlQuery = dataContext.Products.Where (Product.IsSelling());

  IEnumerable<Product> localQuery = localProducts.Where (Product.IsSelling.Compile());
}
```

提示：Compile方法无法实现反向转换，也就是说不能将一个委托方法的查询转换成一个查询表达式树的查询。

---

2. AsQueryable运算符

AsQueryable运算符让查询语句既可以在本地集合上执行，也可以在数据库查询中执行。如下面代码所示：

```
IQueryable<Product> FilterSortProducts (IQueryable<Product> input)
{
  return from p in input
         where ...
         order by ...
```

```
        select p;
}

void Test()
{
  var dataContext = new NutshellContext ("connection string");
  Product[] localProducts = dataContext.Products.ToArray();

  var sqlQuery = FilterSortProducts (dataContext.Products);
  var localQuery = FilterSortProducts (localProducts.AsQueryable());
  ...
}
```

AsQueryable运算符将本地查询的结果集封装了IQueryable<T>接口，在随后的查询中，查询语句会被编译成查询表达式树的形式，在最后遍历结果集的时候，这些查询表达式树会被后台编译（这种编译的消耗很小），这样，对结果集的遍历还是按照对本地的集合来进行。

8.10.2 查询表达式树

前面我们已经提到过，如果将一个Lambda表达式赋值给一个Expression<TDelegate>类型的变量，那么编译器会将整个语句编译成一个查询表达式树的形式。由于编程技术的发展，现在开发者可以在运行时从零开始手动地组装查询表达式树，创建的查询表达式树可以被转换成Expression<TDelegate>类型，然后在LINQ-to-db的查询中直接使用，或者通过Compile方法转换成委托方法的形式。

表达式DOM

一个查询表达式树是由一个微型的DOM（Document Object Model，文档对象模型）来描述的。这个DOM中每个节点都代表了System.Linq. Expressions命名空间中的一个类型。图8-10中描述了这些类型。

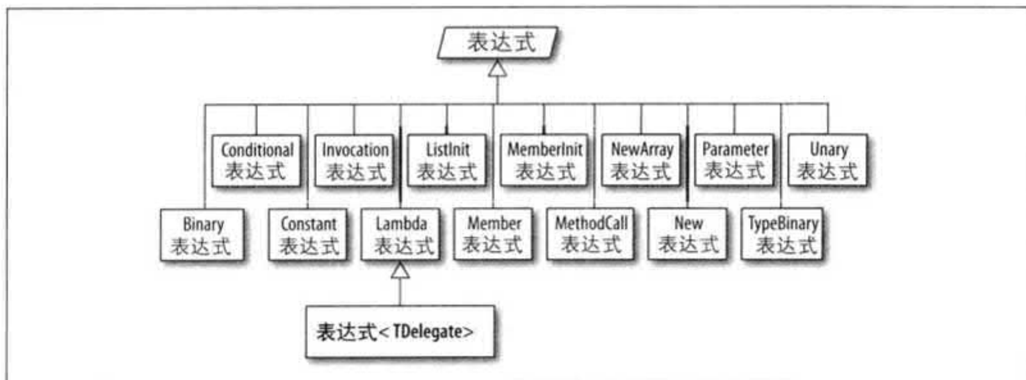


图8-10：表达式类型

提示： 从.NET Framework 4.0以来，命名空间中增加了新的类型和方法，以支持在代码中直接使用这些内置的类型。这些方法和类型对DLR是有好处的，但对Lambda表达式来说并无意义。也就是说，下面这种Lambda表达式仍然不能被转换成查询表达式树的形式：

```
Expression<Func<Customer,bool>> invalid =
  { return true; }  //不允许使用这种方式
```

查询表达式树中的节点的基类是Expression，一般来说Expression<TDelegate>表示Lambda表达式类型，实际上有专用的Lambda表达式类型LambdaExpression<TDelegate>，但是由于这种类型在使用时需要这样定义：

```
LambdaExpression<Func<Customer,bool>> f = ...
```

这种定义方式看起来不是很美观，所以就用Expression<TDelegate>来表示Lambda类型。Expression<T>的基类是LambdaExpression，LambdaExpression是Lambda表达式树中所有节点的基类型，所有的节点类型都可以转换成这种基类型，因此保证了表达式树中节点的类型一致性。

如何区分LambdaExpressions和普通的Expressions呢，方法是：Lambda表达式需要接收参数，而普通的表达式则没有参数。

在手工创建查询表达式树的时候，不要直接实例化一个节点来使用，而应该调用Expression类中提供的静态方法来获取一个新节点。下面列出了所有可以使用的方法：

Add	ElementInit	MakeMemberAccess	Or
AddChecked	Equal	MakeUnary	OrElse
And	ExclusiveOr	MemberBind	Parameter
AndAlso	Field	MemberInit	Power
ArrayIndex	GreaterThan	Modulo	Property
ArrayLength	GreaterThanOrEqual	Multiply	PropertyOrField
Bind	Invoke	MultiplyChecked	Quote
Call	Lambda	Negate	RightShift
Coalesce	LeftShift	NegateChecked	Subtract
Condition	LessThan	New	SubtractChecked
Constant	LessThanOrEqual	NewArrayBounds	TypeAs
Convert	ListBind	NewArrayInit	TypeIs
ConvertChecked	ListInit	Not	UnaryPlus
Divide	MakeBinary	NotEqual	

图8-11演示了下面这行代码生成的表达式树：

```
Expression<Func<string, bool>> f = s => s.Length < 5;
```

可以使用下面两行代码验证上面描述的表达式树：

```
Console.WriteLine (f.Body.NodeType);                    // LessThan
Console.WriteLine (((BinaryExpression) f.Body).Right);// 5
```

下面从零开始构建这个查询表达式树。构建的原则是从表达式树的下面依次向上构建。在这里，表达式树的最下面是一个ParameterExpression，Lambda表达式使用的变量是一个字符串类型的s：

```
ParameterExpression p = Expression.Parameter (typeof (string), "s");
```

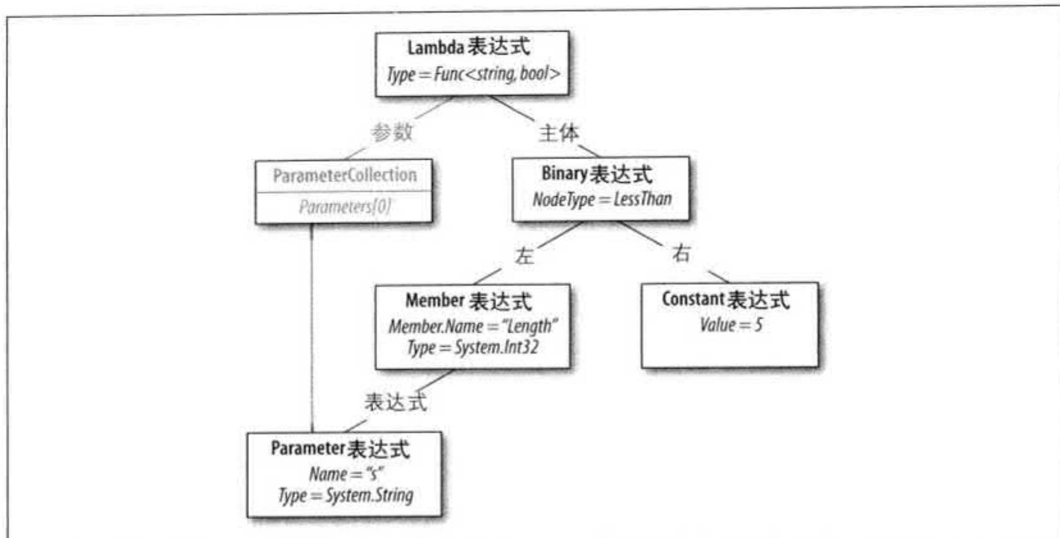


图8-11：查询表达式树

下一步是创建MemberExpression和ConstantExpression对象。在示例中，我们需要得到变量“s”的Length属性，方式如下：

```
MemberExpression stringLength = Expression.Property(p, "Length");
ConstantExpression five = Expression.Constant(5);
```

接下来是使用LessThan方法创建一个比较：

```
BinaryExpression comparison = Expression.LessThan(stringLength, five);
```

最后一步就是创建Lambda表达式，我们需要将表达式的主体和参数组合到一起：

```
Expression<Func<string, bool>> lambda
  = Expression.Lambda<Func<string, bool>> (comparison, p);
```

如果想要测试创建Lambda是否正确，简便方式是使用Compile方法将其转换成委托方法的形式，代码如下所示：

```
Func<string, bool> runnable = lambda.Compile();

Console.WriteLine (runnable ("kangaroo"));          // False
Console.WriteLine (runnable ("dog"));               // True
```

提示： 在创建表达式树的时候，要想知道节点的类型，最简单的方式是使用Visual Studio中的调试器去调试现有的Lambda表达式，看它里面都用到了哪些类型。

网上提供了这些方面更详细的内容：*http://www.albahari.com/expressions/*。

第9章

LINQ运算符

本章介绍LINQ中查询运算符的使用。后面的“映射”和“连接”是两节补充内容，介绍其他一些概念性的内容：

- 映射对象的层次结构。
- 使用Select、SelectMany、Join和GroupJoin完成查询中的连接。
- 查询表达式中的范围变量。

本章所有本地查询的示例都基于names数组，这个数组的定义如下：

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
```

在对数据库查询的示例中，使用DataContext类型的变量dataContext，这个对象以及相关实体类的定义如下：

```
var dataContext = new NutshellContext ("connection string...");
...
public class NutshellContext : DataContext
{
    public NutshellContext (string cxString) : base (cxString) {}

    public Table<Customer> Customers { get { return GetTable<Customer>(); } }
    public Table<Purchase> Purchases { get { return GetTable<Purchase>(); } }
}

[Table] public class Customer
{
  [Column(IsPrimaryKey=true)] public int ID;
  [Column]                    public string Name;

  [Association (OtherKey="CustomerID")]
  public EntitySet<Purchase> Purchases = new EntitySet<Purchase>();
}

[Table] public class Purchase
{
    [Column(IsPrimaryKey=true)] public int ID;
    [Column]                    public int? CustomerID;
```

```
        [Column]                public string Description;
        [Column]                public decimal Price;
        [Column]                public DateTime Date;

        EntityRef<Customer> custRef;

        [Association (Storage="custRef",ThisKey="CustomerID",IsForeignKey=true)]
        public Customer Customer
        {
          get { return custRef.Entity; } set { custRef.Entity = value; }
        }
    }
```

提示： 本章所有的示例都在LINQPad中进行调试。可以到*http://www.linqpad.net*下载LINQPad。

上面定义的两个实体类是LINQ to SQL工具通常生成的简化版本，不包括实体中的字段发生变化时更新数据库的代码。

下面是相应SQL表的定义：

```
create table Customer
(
  ID int not null primary key,
  Name varchar(30) not null
)
create table Purchase
(
  ID int not null primary key,
  CustomerID int references Customer (ID),
  Description varchar(30) not null,
  Price decimal not null
)
```

提示： 除了特别说明的地方，所有的示例都适用于Entity Framework。可以通过在Visual Studio中创建一个新的实体数据模型（EDM），把表结构拖拽到VS的设计器界面中，根据这些表创建`ObjectContext`对象。

9.1 概述

在这一节中，会对所有的标准查询运算符进行概述。

标准查询运算符可以分为三类：

- 输入是集合，输出是集合
- 输入是集合，输出是单个元素或者标量值
- 没有输入，输出是集合（生成方法）

先按分类说明每类查询运算符的特点，然后再详细介绍每类中的查询运算符。

9.1.1 集合→集合

大多数的查询运算符都属于这一类，接受一个或多个序列作为输入，经过处理后，将输入序列并且转换成一个输出序列。图9-1演示了这种运算符如何重新形成序列。

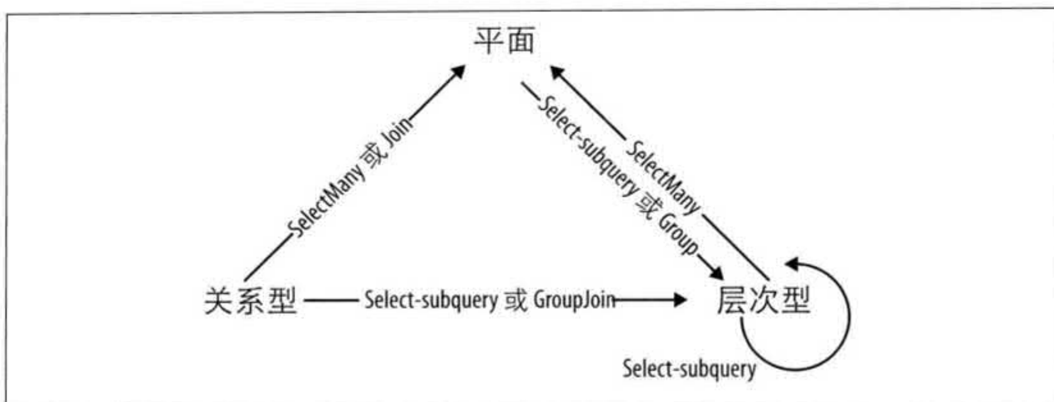


图9-1：内部查询过程

1. 筛选运算符

`IEnumerable<TSource>→IEnumerable<TSource>`

返回原始序列的一个子集。使用的运算符有：

`Where`、`Take`、`TakeWhile`、`Skip`、`SkipWhile`、`Distinct`

2. 映射运算符

`IEnumerable<TSource>→IEnumerable<TResult>`

这种运算符可以按照Lambda表达式指定的形式，将每个输入元素转换成输出元素。`SelectMany`用于查询嵌套的集合；在LINQ to SQL和EF中`Select`和`SelectMany`运算符可以执行内连接、左外连接、交叉连接以及非等连接等各种连接查询。使用的运算符有：

`Select`、`SelectMany`

3. 连接运算符

`IEnumerable<TOuter>`、`IEnumerable<TInner>→IEnumerable<TResult>`

用于将两个集合连接之后，取得符合条件的元素。连接运算符支持内连接和左外连接，非常适合对本地集合的查询。使用运算符有：

`Join`、`GroupJoin`、`zip`

4. 排序运算符

`IEnumerable<TSource>→IOrderedEnumerable<TSource>`

返回一个经过重新排序的集合。使用的运算符有：

```
OrderBy、ThenBy、Reverse
```

5. 分组运算符

IEnumerable<TSource>→IEnumerable<IGrouping<TSource,TElement>>

将一个集合按照某种条件分成几个不同的子集。使用的运算符有：

```
GroupBy
```

6. 集合运算符

IEnumerable<TSource>, IEnumerable<TSource>→IEnumerable<TSource>

主要用于对两个相同类型集合的操作，可以返回两个集合中共有的元素、不同的元素或者两个集合的所有元素。使用的运算符有

```
Concat、Union、Intersect、Except
```

7. 转换方法：Import

IEnumerable→IEnumerable<TResult>，这种方法包括：

```
OfType、Cast
```

8. 转换方法：Export

将IEnumerable<TSource>类型的集合转换成一个数组、清单、字典、检索或者序列，这种方法包括：

```
ToArray、ToList、ToDictionary、ToLookup、AsEnumerable、AsQueryable
```

9.1.2 集合→单个元素或标量值

下面这些查询运算符接收一个集合作参数，返回单个元素或者标量值。

1. 元素运算符

IEnumerable<TSource>→TSource

从集合中取出单个特定的元素，使用的运算符有：

```
First、FirstOrDefault、Last、LastOrDefault、Single、SingleOrDefault、
ElementAt、ElementAtOrDefault、DefaultIfEmpty
```

2. 聚合方法

IEnumerable<TSource>→标量值

对集合中的元素进行某种计算，然后返回一个标量值（通常是一个数字）。使用的运算符有：

Aggregate、Average、Count、LongCount、Sum、Max、Min

3. 数量词

IEnumerable<TSource>→布尔值

一种返回true或者false的聚合方法，使用的运算符有：

All、Any、Contains、SequenceEqual

9.1.3 空→集合

第三种查询运算符不需要输入但可以输出一个集合。

生成方法

void→IEnumerable<TResult>

生成一个简单的集合，使用的方法有：

Empty、Range、Repeat

9.2 筛选

IEnumerable<TSource>→IEnumerable<TSource>

方法	描述	等价SQL
Where	返回所有符合给定条件的元素的集合	WHERE
Take	取出前x个元素，丢弃其他元素	WHERE ROW_NUMBER() 或者TOP n subquery
Skip	跳过前x个元素，返回剩余的元素	WHERE ROW_NUMBER()... 或者NOT IN (SELECT TOP n)
TakeWhile	返回集合中的元素直到判断为false	抛出异常
SkipWhile	如果判断为true的时候，忽略集合中的元素；为false的时候，开始输出剩余的元素	抛出异常
Distinct	返回一个没有重复元素的集合	SELECT DISTINCT

提示： 上面这个表中的“等价的SQL语句”这一列并不是IQueryable类型的LINQ查询翻译出来的SQL语句，而是显示了如何使用SQL完成第一列关键字的功能。

在本章接下来的内容中，在展示Enumberable类的内部代码细节的时候，我们只给出核心部分的代码，那些用于非空判断以及索引判断的代码固然重要，但不是本章要关心的内容。为节省篇幅，所以省略这部分代码。

经过各种方法的筛选，最终得到的序列中的元素只能比原始序列少或者相等，绝不可能比原始序列还多。在筛选过程中，集合中的元素类型及元素值是不会改变的，和输入时始终保持一致。

9.2.1 Where

参数	类型
输入序列	IEnumerable<TSource>
给定的判断	TSource => bool 或者(TSource,int) => bool[注]

注：上表第二行的判断在LINQ to SQL和Entity Framework中禁止使用。

1. Where的查询语法

```
where 布尔型表达式
```

2. Enumerable.Where方法的实现

看一下Enumerable.Where方法的内部实现方式，省略非空判断逻辑之后，Where方法的内部实现逻辑如下：

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func <TSource, bool> predicate)
{
  foreach (TSource element in source)
  {
    if (predicate (element))
       yield return element;
   }
}
```

3. 概述

Where返回输入序列中满足筛选条件的那些元素。

例如：

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<string> query = names.Where (name => name.EndsWith ("y"));

// 查询结果: { "Harry", "Mary", "Jay" }
```

使用查询语法查询：

```
IEnumerable<string> query = from n in names
                            where n.EndsWith ("y")
                            select n;
```

如果和let语句配合使用的话，where语句可以在一个查询中出现多次，例如：

```
from n in names
where n.Length > 3
let u = n.ToUpper()
where u.EndsWith ("Y")
```

```
select u;             // 查询结果：{ "HARRY", "MARY" }
```

标准的C#变量作用域规则同样适用于LINQ查询。也就是说，在使用一个查询变量前，必须先声明，否则不能使用。

4. 索引筛选

Where判断选择性地接受一个int型的第二参数。这个参数用于指定输入序列中特定位置上的元素，在查询中可以使用这个数值进行元素的筛选。例如，在下例中，查询会跳过偶数位置上的元素：

```
IEnumerable<string> query = names.Where ((n, i) => i % 2 ==  0);
// 查询结果：{ "Tom", "Harry", "Jay" }
```

5. 比较LINQ to SQL和EF中SQL的LIKE关键字

下面几个关键字如果用在string类型的查询中将会被转换成SQL中的LIKE关键字：

```
Contains、StartsWith、EndsWith
```

例如，c.Name.Contains ("abc")转换成SQL语句就是：customer.Name LIKE '%abc%'（或者更准确地说是转化成一个参数化的版本）。Contains关键字仅用于本地集合的比较。如果想要比较两个不同列的数据，则需要使用SqlMethods.Like方法：

```
... where SqlMethods.Like (c.Description, "%" + c.Name + "%")
```

SqlMethods.Like也可以进行更复杂的比较操作。（例如LIKE 'abc%def%'）。

6. LINQ to SQL和EF中如何使用SQL中的< 和 >

在对字符串进行比较的时候，如果需要关注字母顺序，那么可以使用字符串类型的CompareTo方法，这个方法会被翻译成SQL中的<和>关键字，请看下面这个示例：

```
dataContext.Purchases.Where (p => p.Description.CompareTo ("C") < 0)
```

7. WHERE x IN (..., ..., ...) 在LINQ to SQL和EF的实现

在LINQ to SQL和EF中，可以使用Contains方法来查询一个本地集合，例如：

```
string[] chosenOnes = { "Tom", "Jay" };

from c in dataContext.Customers
where chosenOnes.Contains (c.Name)
...
```

上面这段代码会被翻译成SQL中的IN关键字，也就是：

```
WHERE customer.Name IN ("Tom", "Jay")
```

如果本地集合是一个对象集合或者其他非数值类型的集合，LINQ to SQL或者EF，也可能把Contains关键字翻译成一个EXISTS子查询。

9.2.2 Take 和 Skip运算符

参数	类型
输入集合	IEnumerable<TSource>
Take或Skip的元素个数	int

Take返回集合的前*n*个元素，并且放弃其余元素；Skip则是跳过前*n*个元素，并且返回其余元素。如果一个网页中有大量的数据要显示，并且需要频繁地使用这组数据的子集，那么同时使用Take和Skip关键字很容易实现。例如，用户在一个书籍数据库中搜索包含"mercury"这个词的图书，有100条记录被搜索出来。下面的操作可以取出其中的前20条：

```
IQueryable<Book> query = dataContext.Books
  .Where   (b => b.Title.Contains ("mercury"))
  .OrderBy (b => b.Title)
  .Take (20);
```

接下来的查询返回结果集中的第21～40条记录：

```
IQueryable<Book> query = dataContext.Books
  .Where (b => b.Title.Contains ("mercury"))
  .OrderBy (b => b.Title)
  .Skip (20).Take (20);
```

在SQL Server 2005中，LINQ to SQL和EF中的Take和Skip运算符会被翻译成ROW_NUMBER方法，而在更早的SQL Server 数据库版本中则会被翻译成Top *n*查询。

9.2.3 TakeWhile和SkipWhile

参数	类型
输入集合	IEnumerable<TSource>
给定的判断	TSource => bool 或者 (TSource,int) => bool

Takewhile运算符会遍历输入集合，然后输出每个元素，直到给定的判断为false时停止输出，并忽略剩余的元素：

```
int[] numbers= { 3, 5, 2, 234, 4, 1 };
var takeWhileSmall = numbers.TakeWhile (n => n < 100); // { 3, 5, 2 }
```

SkipWhile运算符会遍历输入集合，忽略判断条件为真之前的每个元素，直到给定的判断为false时输出剩余的元素：

```
int[] numbers = { 3, 5, 2, 234, 4, 1 };
var skipWhileSmall = numbers.SkipWhile (n => n < 100); // { 234, 4, 1 }
```

在SQL中没有与TakeWhile和SkipWhile对应的查询方式。如果在LINQ-to-db查询中使用，将会导致一个运行时错误。

9.2.4 Distinct

Distinct的作用是返回一个没有重复元素的序列，它会删除输入序列中的重复元素。在这里，判断两个元素是否重复的规则是可以自定义的，如果没有定义，那么就使用默认的判断规则。下面的示例返回字符串中没有重复的字母：

```
char[] distinctLetters = "HelloWorld".Distinct().ToArray();
string s = new string(distinctLetters);                    // HeloWrd
```

因为string实现了IEnumerable<char>接口，所以我们可以在一个字符串上直接使用LINQ方法。

9.3 映射

IEnumerable<TSource>→IEnumerable<TResult>

方法	功能	等价SQL语句
Select	将输入的每个元素按照Lambda表达式给定的格式进行转换	SELECT
SelectMany	按照Lambda表达式指定的要求转化输入元素，然后和其他集合进行连接操作。	INNER JOIN, LEFT OUTER JOIN, CROSS JOIN

提示：在查询一个数据库时，Select和SelectMany是最常用的连接操作方法；对于本地查询来说，使用Join和Group Join的效率最好。

9.3.1 Select

参数	类型
输入集合	I Enumerable<TSource>
结果选择器	TSource => TResult 或者 (TSource,int) => TResult[注]

注：第二行的查询方式不能在LINQ to SQL和Entity Framework中使用。

1. 查询语法

Select *映射表达式*

2. Select运算符在Enumerable类中的实现方式

```
public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
{
  foreach (TSource element in source)
    yield return selector (element);
}
```

3. 概述

在使用Select时，通常不会减少序列中的元素数量。每个元素可以被转换成需要的形式，并且这个

形式需要通过Lambda表达式来定义。

下面的示例会查询出当前电脑上安装的所有字体的名字（从System.Drawing）：

```
IEnumerable<string> query = from f in FontFamily.Families
                            select f.Name;

foreach (string name in query)
{
  Console.WriteLine (name);
}
```

在这个示例中，select语句将FontFamily对象转换成它的名字。下面是一个相同功能的lambda表达式：

```
IEnumerable<string> query = FontFamily.Families.Select (f => f.Name);
```

select语句经常用来映射到匿名类型：

```
var query =
  from f in FontFamily.Families
  select new { f.Name, LineSpacing = f.GetLineSpacing (FontStyle.Bold) };
```

在条件查询中，一般不需要对查询结果进行映射，之所以要使用select运算符，是为了满足LINQ查询必须以select或者group语句结尾的语法要求。下面这个示例只查询那些支持“Strikeout”的字体：

```
IEnumerable<FontFamily> query =
  from f in FontFamily.Families
  where f.IsStyleAvailable (FontStyle.Strikeout)
  select f;

foreach (FontFamily ff in query) Console.WriteLine (ff.Name);
```

在这种情况下，当编译器将它翻译成运算符流形式的查询语句时会忽略映射操作。

4. 索引映射

selector表达式还接受一个整型的可选参数，这个参数实际上是一个索引，使用它可以得到输入序列中元素的位置。需要注意的是，这种参数只能在本地查询中使用：

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
  IEnumerable<string> query = names
    .Select ((s,i) => i + "=" + s);        //  { "0=Tom", "1=Dick", ... }
```

5. 多层次的Select查询

可以在select语句中再嵌套Select子句来构成嵌套查询，这种嵌套查询的结果是一个多层次的对象集合。在下面这个示例中，外层的Select查询会返回*D:\source*目录下的所有文件夹名，而内层的Select子查询则返回每个文件夹下的所有文件信息：

```
DirectoryInfo[] dirs = new DirectoryInfo (@"d:\source").GetDirectories();

var query =
  from d in dirs
  where (d.Attributes & FileAttributes.System) == 0
  select new
  {
```

```
      DirectoryName = d.FullName,
      Created = d.CreationTime,

      Files = from f in d.GetFiles()
              where (f.Attributes & FileAttributes.Hidden) == 0
              select new { FileName = f.Name, f.Length, }
    };

  foreach (var dirFiles in query)
  {
    Console.WriteLine ("Directory: " + dirFiles.DirectoryName);
    foreach (var file in dirFiles.Files)
      Console.WriteLine ("  " + file.FileName + "Len: " + file.Length);
  }
```

上面这段查询中的内部子查询可以叫做“关联子查询”。因为内部的子查询总是针对外部查询的某个元素进行的，在上面这个示例中，这个外部元素是d，代表一个文件夹信息。

提示： Select内部的子查询可以将一个多层次的对象映射成另一个多层次的对象，也可以将一组关联的单层次对象映射成一个多层次的对象模型。

在对本地集合的查询中，如果Select语句中包含Select子查询，那么整个查询是双重的延迟加载。例如上面这个示例，如果只是遍历路径信息而不是路径下的文件信息，那么内层的Select子查询是不会被执行的，只有在遍历文件信息时才会被执行。

6. 子查询、连接在LINQ to SQL和EF中的实现方式

子查询的映射在LINQ to SQL和EF中都可以实现，并且可以用来实现SQL的连接功能。下面这个示例检索每个消费者的名字和他们大于1000的消费记录：

```
  var query =
    from c in dataContext.Customers
    select new {
                 c.Name,
                 Purchases = from p in dataContext.Purchases
                             where p.CustomerID == c.ID && p.Price > 1000
                             select new { p.Description, p.Price }
               };

  foreach (var namePurchases in query)
  {
    Console.WriteLine ("Customer: " + namePurchases.Name);
    foreach (var purchaseDetail in namePurchases.Purchases)
    {
      Console.WriteLine (" - $$$: " + purchaseDetail.Price);
    }
  }
```

警告： 这个嵌套查询特别适合解释型的查询。外部查询和子查询都被当作一个整体来执行，这样可以避免查询过程中不必要的循环。在本地查询中，因为需要遍历外部元素和内部元素的所有组合情况，而符合条件的元素可能非常少，但无论结果集的大小，都需要完整地遍历所有元素，所以这种查询方式的效率不是很高。对于本地查询来说，较好的选择是使用Join或者GroupJoin，它们的用法会在下一部分介绍。

这个查询可以从两个不同的序列中取出符合条件的元素，因此它可以看作是一种连接操作。 这个查询和常见的数据库连接（或者子查询）的不同之处在于，在输出结果的时候，结果集没有被连换成一个二维的结果集，而是被映射成一个多层次的对象结构。

使用Customer对象关联Purchase对象，可以将上面的代码简化为如下形式：

```
from c in dataContext.Customers
select new
{
  c.Name,
  Purchases = from p in c.Purchases    // Purchases是EntitySet<Purchase>型的
              where p.Price > 1000
              select new { p.Description, p.Price }
};
```

上面这种查询方式相当于SQL中的左外连接，也就是不论Purchases的多少，所有的消费者都会被查询出来。下面演示一下如何模拟SQL中的内连接，只有那些有高消费记录的客户被返回，忽略剩余消费者。可以在外部查询中添加一个用于限制Purchases的语句来实现这个要求：

```
from c in dataContext.Customers
where c.Purchases.Any (p => p.Price > 1000)
select new {
             c.Name,
             Purchases = from p in c.Purchases
                         where p.Price > 1000
                         select new { p.Description, p.Price }
};
```

在上面的查询中使用了两次（Price > 1000）判定，代码看起来稍显凌乱。我们可以使用let语句来避免这种重复：

```
from c in dataContext.Customers
let highValueP = from p in c.Purchases
                 where p.Price > 1000
                 select new { p.Description, p.Price }
where highValueP.Any()
select new { c.Name, Purchases = highValueP };
```

使用let子句之后，会使整个查询变得比较灵活。例如，通过将Any改成Count，就可以查询出那些有两个以上高消费记录的消费者：

```
...
where highValueP.Count() >= 2
select new { c.Name, Purchases = highValueP };
```

7. 将数据映射到具体的类型

前面的一些示例中，我们将查询结果映射到匿名类中，这种映射方式适用于查询过程中暂存中间结果集的情况，但是当需要将结果返回给客户端使用的时候，这种映射方式就不能满足需求了，因为匿名类型只能在一个方法内作为本地变量存在。解决方法就是将结果集映射到具体的类型，例如DataSets或者自定义业务实体类。方法自定义业务实体就是一些附带属性标签的简单类，就像LINQ to SQL和EF中的[Table]属性，同时，这种类隐藏了一些跟数据库密切相关的细节。举例来说，在业务实体类中我们可以不使用一个ID来标注键字段。假设我们自定义了Customer的实体类，这些类包括两个实体

类：CustomerEntity和PurchaseEntity，下面我们演示一下如何将查询结果映射到这两个实体类中：

```
IQueryable<CustomerEntity> query =  from c in dataContext.Customers
  select new CustomerEntity
  {
    Name = c.Name,
    Purchases =
      (from p in c.Purchases
      where p.Price > 1000
      select new PurchaseEntity {
                                             Description = p.Description,
                                             Value = p.Price
                                           }
      ).ToList()
};

// 强制执行查询，将输出转换为一个更方便的List：
List<CustomerEntity> result = query.ToList();
```

到目前为止，还没有用到Join或者SelectMany关键字进行查询。这是因为我们一直在处理多层次结构的对象模型，如图9-2所示。在使用LINQ时，由于是在处理对象，因此很容易表达出对象之间的逻辑关系，而不像标准的SQL查询需要将查询结果转换成二维的结果集。

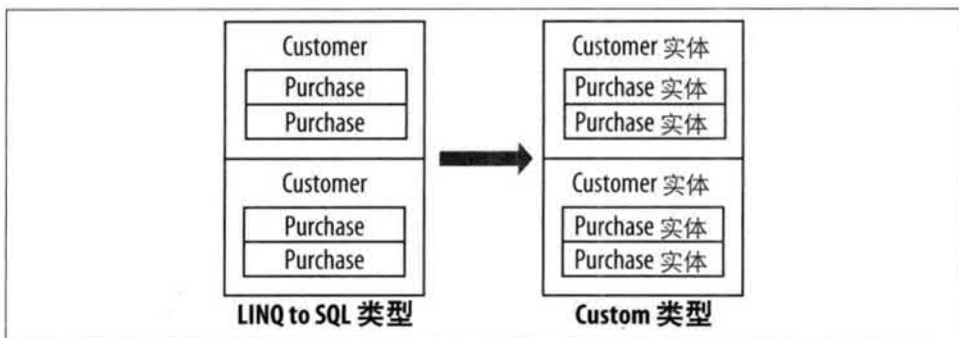


图9-2：将结果集映射成对象层次结构

9.3.2 SelectMany运算符

参数	类型
输入集合	IEnumerable<TSource>
结果选择器	TSource => IEnumerable<TResult>或者 (TSource,int) => IEnumerable<TResult>[注]

注：不能在LINQ to SQL和Entity Framework中使用第二行的查询方式。

1. SelectMany的查询语法

```
from 标识符1  in 可遍历序列表达式
from 标识符2  in 可遍历序列表达式
...
```

2. From运算符在Enumerable中的实现方式

```
public static IEnumerable<TResult> SelectMany<TSource,TResult>
  (IEnumerable<TSource> source,
   Func <TSource,IEnumerable<TResult>> selector)
{
  foreach (TSource element in source)
    foreach (TResult subElement in selector (element))
      yield return subElement;
}
```

3. 概述

SelectMany可以将两个集合组合成一个更大的集合。

回忆一下之前的Select运算符的实现方式，对于每个输入元素，经过处理之后，只有一个元素作为结果被返回。与此不同的是，SelectMany则一次返回*0...n*个元素，这*0...n*个元素则是由后续集合中的Lambda表达式提供的。

SelectMany主要有三项用途：将子序列中的元素进行扩展，就是将嵌套结构的对象转换成非嵌套结构，然后合并两个序列。针对传送带示例，SelectMany的作用相当于一个漏斗，这个漏斗会将原料送上流水线，在这个过程中，每个输入元素都会触发SelectMany。这些原料是由selector的Lambda表达式生成的，并且必须是序列形式的。也就是说，这个Lambda表达式会为每个输入元素创建一个小的序例，SelectMany最终的返回结果就是这些小序列连接在一起形成的大序列。

举个简单的示例，假如有一个用于存储名字的数组，包含下面这些元素：

```
string[] fullNames = { "Anne Williams", "John Fred Smith", "Sue Green" };
```

如果要求将这个数组中的元素重新分配，以每个单词为单位存储在一个新的数组中,也就是说将上面这个数组转换成下面这种形式：

```
"Anne", "Williams", "John", "Fred", "Smith", "Sue", Green"
```

SelectMany可以实现这样的要求，因为在SelectMany中，每个输入元素会被当成一个新的集合来处理，然后再输出。我们要做的就是使用一个循环，在循环中处理输入元素，使之符合输出要求，在这个示例中，要做的就是将名字拆分成单词，string.Split方法可以将一个字符串分割成一个个的单词，并将这些单词存放在一个数组中，下面这几行代码演示了string.Split的用法：

```
string testInputElement = "Anne Williams";
string[] childSequence = testInputElement.Split();

// childSequence的内容是 { "Anne", "Williams" };
```

所以，使用SelectMany完成对数组中元素拆分的代码如下：

```
IEnumerable<string> query = fullNames.SelectMany (name => name.Split());

foreach (string name in query)
  Console.Write (name + "|");
输出结果： Anne|Williams|John|Fred|Smith|Sue|Green|
```

提示： 在分层次的数据查询中，使用SelectMany和Select得到的结果是相同的，但是在查询单层次的数据源（如数组）的时候，Select要完成同样的任务，就需要使用嵌套循环了，如下所示：

```
IEnumerable<string[]> query =
  fullNames.Select (name => name.Split());

foreach (string[] stringArray in query)
  foreach (string name in stringArray)
    Console.Write (name + "|");
```

SelectMany的好处就是在于，无论输入集合是什么类型的，它输出的集合肯定是一个数组类型的二维集合，结果集的数据不会有层次关系。

在LINQ查询中，除了使用SelectMany运算符进行查询之外，还有一些情形，即使没有使用SelectMany运算符，编译器也会自动生成一个SelectMany查询，例如说，连续使用两个from子句就会被翻译成SelectMany。在查询表达式语法中，from运算符有两个作用，在查询一开始的from的作用都是引入查询集合和范围变量；其他任何位置再出现from子句，编译器都会将其翻译成SelectMany。下面是使用两个from子句完成名字分割的代码：

```
IEnumerable<string> query =
  from fullName in fullNames
  from name in fullName.Split()      // 被翻译成SelectMany
  select name;
```

注意，在使用第二个from子句的时候，引入了一个新的查询变量name，这个变量会在接下来的查询中使用，而fullName则成为一个外部变量。

4. 外部范围变量

前面的示例中，在SelectMany之后的fullName变成了外部变量。外部变量的作用范围比较广，在查询语句的结尾或者into语句之前一直有效。像这种外部的范围变量在查询表达式语法的LINQ查询中很好用，但是如果想在运算符流语法中使用这种变量，则非常困难。

为了说明这一点，还使用前面的示例，只是在最后将加上fullName进行映射：

```
IEnumerable<string> query =
  from fullName in fullNames            // fullName是外部变量
  from name in fullName.Split()         // 是范围变量
  select name + " came from " + fullName;

输出结果：
Anne came from Anne Williams
Williams came from Anne Williams
John came from John Fred Smith
...
```

为了解决外部变量的范围问题，编译器编译上面这段代码的时候使用了一些障眼法。如果想验证的话，可以使用运算符流语法完成上面那段代码。由于编译器在编译过程中使用的是障眼法，本来很普通的查询，想要添加where或者orderby运算符就变得非常困难，请看下面这个示例：

```
from fullName in fullNames
from name in fullName.Split()
orderby fullName, name
```

```
select name + " came from " + fullName;
```

问题出现在哪里呢？因为SelectMany运算符为每个元素返回一个一维的集合。这样在返回集合中，那种层次关系必然就丢失了，因此这时已经无法使用SelectMany返回集合中的外部的范围变量，当前示例外部的范围变量就是fullName。编译器的解决方案是将SelectMany运算符的查询结果再封装一次，让结果序列中的每个元素都带着外部范围变量funName，如下面代码所示：

```
from fullName in fullNames
from x in fullName.Split().Select (name => new { name, fullName } )
orderby x.fullName, x.name
select x.name + " came from " + x.fullName;
```

这里的唯一变化是，每个元素（name）连同它的fullName被封装到了一个匿名类型中，这个过程和let关键字的解析过程非常相似。下面是最终转换成运算符流形式的查询语句：

```
IEnumerable<string> query = fullNames
  .SelectMany (fName => fName.Split()
                           .Select (name => new { name, fName } ))
  .OrderBy (x => x.fName)
  .ThenBy (x => x.name)
  .Select (x => x.name + " came from " + x.fName);
```

5. 查询表达式语法

像前面演示的，在需要用到外部变量的情况下，选择使用查询表达式语法是最佳选择。因为在这种情况中，这种语法不仅便于书写，而且表达方式也更接近查询逻辑。

一般在两种情形下，我们在查询中需要用到额外的生成器，第一种情形是需要将多层次的数据集合转换成单层次数据集合的时候，要达到这个目的，可以在外部的查询变量上调用变量的属性和方法。就像前面示例中所做的：

```
from fullName in fullNames
from name in fullName.Split()
```

通过上面的代码，我们可以改变查询的返回结果，本来是返回人的全名，而现在返回单个的单词。在LINQ-to-db中有类似的查询，这种查询同时使用多个对象的属性。下面这个查询就演示了如何同时查询客户姓名和他们购买的商品：

```
IEnumerable<string> query = from c in dataContext.Customers
                            from p in c.Purchases
                            select c.Name + " bought a " + p.Description;

输出结果：
Tom bought a Bike
Tom bought a Holiday
Dick bought a Phone
Harry bought a Car
...
```

在这里，我们扩展了Customer对象，同时使用了Customer和它的Purchases属性来查询所需信息。

第二种情形是为了实现交叉连接的查询效果，取两个序列的笛卡尔积，即让两个序列中的所有元素两两结合。要在LINQ中实现这个功能，可以使用一个额外的from子句返回和外部变量无关的新变量，

如下所示：

```
int[] numbers = { 1, 2, 3 }; string[] letters = { "a", "b" };

IEnumerable<string> query = from n in numbers
                            from l in letters
                            select n.ToString() + l;

查询结果: { "1a", "1b", "2a", "2b", "3a", "3b" }
```

实际上这种连接方式就是SelectMany运算符的内部实现方式。

6. 使用SelectMany进行连接查询

我们可以使用SelectMany对两个集合进行连接查询，然后在连接结果中查找符合条件的记录。例如，当需要将参加游戏的人两两配对时，可以使用下面的查询：

```
string[] players = { "Tom", "Jay", "Mary" };

IEnumerable<string> query = from name1 in players
                            from name2 in players
                            select name1 + " vs " + name2;

查询结果: { "Tom vs Tom", "Tom vs Jay", "Tom vs Mary",
           "Jay vs Tom", "Jay vs Jay", "Jay vs Mary",
           "Mary vs Tom", "Mary vs "Jay", "Mary vs Mary" }
```

这个查询执行的操作是：从第一个集合中取出一个参赛者，再从第二个集合中取出一个参赛者，将取出的参赛者两两配对。虽然得到了元素两两配对之后的交叉集合，但这个集合对我们来说是没有意义的，因为它包含了所有可能的配对情况，加上下面这个筛选条件之后这个查询才真正有效：

```
IEnumerable<string> query = from name1 in players
                            from name2 in players
                            where name1.CompareTo (name2) < 0
                            orderby name1, name2
                            select name1 + " vs " + name2;

查询结果: { "Jay vs Mary", "Jay vs Tom", "Mary vs Tom" }
```

上面示例中使用的筛选条件也可以认为是连接条件。这里的连接条件叫做非等值连接，因为在where子句中使用的是一个不等表达式（<）。

在接下来的内容中，我们会演示LINQ to SQL中其他的连接方式（除了显式地使用外键作为连接条件的查询，展示的所有的连接同样适用于EF）。

7. 在LINQ to SQL和EF中使用SelectMany

在LINQ to SQL和EF中，SelectMany可以实现交叉连接、不等连接、内连接以及左外连接。实际上SelectMany运算符还支持其他一些特殊的连接，例如完成和Select关键字相同的查询效果。不同的是，SelectMany返回的是一个没有层次关系的数据集，而Select则可以返回具有层次关系的数据集。

在前一节中已经使用过LINQ-to-db中的交叉连接。下面这个查询使用交叉连接将每个消费者和每种产品进行配对：

```
var query = from c in dataContext.Customers
            from p in dataContext.Purchases
            select c.Name + " might have bought a " + p.Description;
```

在实际应用中，我们只希望得到某个客户和他购买的商品的配对信息。这个时候就需要在连接查询中加上where子句来筛选结果集。下面就是它的实现方式，这是一个标准的SQL式的连接查询：

```
var query = from c in dataContext.Customers
            from p in dataContext.Purchases
            where c.ID == p.CustomerID
            select c.Name + " bought a " + p.Description;
```

提示：上面这个LINQ查询语句会被翻译成标准的SQL语句。在下一节中，我们将演示如何使用外连接。如果将上面的查询用join关键字改写的话，功能也可以实现，但是扩展性会差很多，这是一个LINQ和标准SQL不匹配的地方，在标准SQL中，所有的连接都要通过join关键字实现。

如果在实体类中有关联属性，那么在查询的时候，就可以直接使用这个属性中的数据，而不必使用外键去连接其他的表来得到需要的数据，如：

```
from c in dataContext.Customers
from p in c.Purchases
select new { c.Name, p.Description };
```

提示：前一章中我们已经提过，在Entity Framework的实体类中，并不会直接存储一个外键值，而是存储外键所关联对象的集合，所以当需要使用外键所关联的数据时，直接使用实体类属性中附带的数据集合即可，不用像LINQ to SQL查询中那样手动地进行连接来得到外键集合中的数据。

上面这种查询的好处是，它不再需要使用where判断，也就是说不再需要从交叉连接的结果集中筛选所需的数据。但实际上，当编译器将这两种LINQ语句翻译成标准SQL查询的时候，翻译出来的SQL语句还是需要进行连接查询。

还可以为上面的查询添加其他的where子句以添加新的筛选条件。例如，如果我们只想得到名字以字母“T”开头的客户，可以像下面这样添加一个新的where子句：

```
from c in dataContext.Customers
where c.Name.StartsWith ("T")
from p in c.Purchases
select new { c.Name, p.Description };
```

可能读者已经注意到，这个where子句是添加在两个from子句之间的，在LINQ-to-db中，这样写对查询结果没有影响。但是在对本地集合的查询中，如果将这个where子句向下移一行的话，会降低查询效率。在对于本地集合的查询中，为了提高执行效率，应该尽量先筛选，再连接。

如果有需要的话，可以引入新的表来进行连接，查询时的连接并不限于两个表之间，多个表也可以进行，在LINQ中，可以通过添加一个from子句来实现。以前面消费者和商品记录为例，如果客户购买的商品不止一件，而是多个商品记录，我们想要得到每个客户和他购买的所有商品，可以使用下面这个语句进行查询：

```
from c in dataContext.Customers
from p in c.Purchases
from pi in p.PurchaseItems
select new { c.Name, p.Description, pi.DetailLine };
```

上面这个查询中，每个from子句都会引入一个数据表。如果实体类中使用了关联属性，那么就可以直接使用关联属性中的数据，而不必使用多个from子句进行连接查询。例如，在Customer表中使用外键关联了一个SalesPerson表，那么当需要通过Customer对象得到它所关联的SalesPerson的名字时，可以使用下面这种方式：

```
from c in dataContext.Customers
select new { Name = c.Name, SalesPerson = c.SalesPerson.Name };
```

在这里没有使用SelectMany运算符，这是因为这个查询中，并没有子集需要展开。Customer对象的关联属性返回的不是一个集合，而是单个元素。

8. 使用SelectMany实现外连接

从前面的示例中可以知道，Select运算符下面的子查询会被翻译成一个左外连接，如下面这个查询所示：

```
from c in dataContext.Customers
select new {
             c.Name,
             Purchases = from p in c.Purchases
                         where p.Price > 1000
                         select new { p.Description, p.Price }
};
```

在这个示例中，无论customer中是否包含Purchases，每个外部元素（customer对象）都会被遍历，这显然有点浪费性能。假设我们使用SelectMany关键字来重写上面这个查询，得到的结果集就是一个单层次的数据集，而不再是多层次的数据集：

```
from c in dataContext.Customers
from p in c.Purchases
where p.Price > 1000
select new { c.Name, p.Description, p.Price };
```

在上面这个查询执行过程中，内部的实现机制实际上是内连接，只有那些有高消费记录的客户（p.Price > 1000）会被真正查询到。如果既想使用左连接，又想让得到的结果集是单层次结构的，可以在内连接中添加一个DefaultIfEmpty运算符来实现。当输入集合中没有元素时，这个运算符会返回null。下面是一个使用了DefaultIfEmpty关键字的示例：

```
from c in dataContext.Customers
from p in c.Purchases.DefaultIfEmpty()
select new { c.Name, p.Description, Price = (decimal?) p.Price };
```

这种查询方式在LINQ to SQL和EF中都能正确地执行左外连接查询，取出所有的客户，不论他有没有消费记录。但是在查询本地集合的时候，上面的查询语句可能会出错，因为当变量p为空的时候，再使用p.Description和p.Price时会抛出一个NullReferenceException类型的异常。为了让查询语句在两种状况下都能正常运行，可以采用下面这种写法：

```
from c in dataContext.Customers
from p in c.Purchases.DefaultIfEmpty()
select new {
              c.Name,
              Descript = p == null ? null : p.Description,
              Price = p == null ? (decimal?) null : p.Price
};
```

如果要对查询结果进行筛选的话，不可以只简单地使用一个where语句。因为这个时候where语句会在DefaultIfEmpty之后执行，如下面的查询所示：

```
from c in dataContext.Customers
from p in c.Purchases.DefaultIfEmpty()
where p.Price > 1000...
```

正确的做法是在DefaultIfEmpty运算符之前使用Where语句，如：

```
from c in dataContext.Customers
from p in c.Purchases.Where (p => p.Price > 1000).DefaultIfEmpty()
select new {
              c.Name,
              Descript = p == null ? null : p.Description,
              Price = p == null ? (decimal?) null : p.Price
};
```

在LINQ to SQL和EF中，上面的查询会被翻译成左外连接，在类似的查询中，这种查询方式的效率最好。

提示： 如果标准SQL查询的经验比较丰富的话，可能在LINQ中也会以SQL查询的思路进行数据查询，也就是说喜欢使用SelectMany而不是Select运算符，因为SelectMany运算符返回的结果集是没有层次关系的数据，和SQL中的查询结果类似。然而，当查询中需要用到外连接的时候，Select运算符所返回的带层次结构的结果集更容易被后续步骤使用，因为不需要处理那些空元素。

9.4 连接

IEnumerable<TOuter>, IEnumerable<TInner>→IEnumerable<TResult>

方法	描述	等价SQL
Join	对两个集合进行连接查询，返回非嵌套的数据集合	内连接
GroupJoin	对两个集合进行连接查询，返回嵌套的数据集合	内连接 左外连接
Zip	同时列举了2个序列（像拉链一样），并为每一对元素执行一个函数	

9.4.1 Join和GroupJoin

IEnumerable<TOuter>, IEnumerable<TInner>→IEnumerable<TResult>

1. Join的参数

参数	类型
外部集合	IEnumerable<TOuter>
内部集合	IEnumerable<TInner>
外键选择器	TOuter => TKey
内键选择器	TInner => TKey
结果选择器	(TOuter,TInner) => TResult

2. GroupJoin 的参数

参数	类型
外部集合	IEnumerable<TOuter>
内部集合	IEnumerable<TInner>
外键选择器	TOuter => TKey
内键选择器	TInner => TKey
结果选择器	(TOuter,IEnumerable<TInner>) => TResult

3. 查询语法

```
from 外部变量 in 外部集合
join 内部变量 in 内部集合 on 外部表的键 equals 内部表的键
[into 标识符]
```

4. 概述

Join和GroupJoin的作用是连接两个集合进行查询，然后返回一个查询结果集。它们的不同点在于，join返回的是非嵌套结构的数据集合，而GroupJoin返回的则是嵌套结构的数据集合。

前面一节介绍了使用Select和SelectMany连接两个集合，在本节中介绍的是使用Join和GroupJoin进行连接，这两种连接方式在查询中都可以使用，各有利弊。Join和GroupJoin的长处在于对本地集合的查询，也就是对内存中数据的查询效率比较高。它们的缺点是目前只支持内连接和左连接，并且连接条件必须是相等连接。需要用到交叉连接或者非等值连接时，就只能选择Select或者SelectMany运算符。在LINQ to SQL或者EF查询中，Join和GroupJoin运算符在功能上与Select和SelectMany是没有什么区别的。

表9-1总结了上面这几种连接方式的异同。

表9-1：连接策略

查询方式	结果集形式	本地集合查询效率	是否支持内连接连	是否支持左外接	是否支持交叉连接	非等值连接
Select+SelectMany	非嵌套的	低	是	是	是	是
Select + Select	嵌套的	低	是	是	是	是
Join	非嵌套的	高	是	-	-	-
GroupJoin	嵌套的	高	是	是	-	-
GroupJoin + SelectMany	非嵌套的	高	是	是	-	-

5. join

下面介绍join运算符，这个运算符会执行一个内连接，并返回一个非嵌套结构的结果集。

提示： 由于Entity Framework中隐藏了数据表的外键，所以我们没有办法手动地在EF中执行一个内连接来取数据。实际上可以直接使用EF实体类中的关联属性得到需要的数据。

使用Join运算符，最简单的情形是LINQ to SQL的查询。下面这个查询使用join运算符，从Customers表和Purchases表中取出了用户的购买记录：

```
IQueryable<string> query =
  from c in dataContext.Customers
  join p in dataContext.Purchases on c.ID equals p.CustomerID
  select c.Name + " bought a " + p.Description;
```

下面是查询执行的结果，这个结果和前面使用SelectMany运算符取得的结果是一致的：

```
Tom bought a Bike
Tom bought a Holiday
Dick bought a Phone
Harry bought a Car
```

为了比较Join和SelectMany之间的优缺点，下面对本地集合进行查询，从中可以看出Join在性能上的优势。首先将需要用到的数据存放到本地数组中，然后再执行上面的查询，代码如下：

```
Customer[] customers = dataContext.Customers.ToArray();
Purchase[] purchases = dataContext.Purchases.ToArray();
var slowQuery = from c in customers
                from p in purchases where c.ID == p.CustomerID
                select c.Name + " bought a " + p.Description;

var fastQuery = from c in customers
                join p in purchases on c.ID equals p.CustomerID
                select c.Name + " bought a " + p.Description;
```

上面这两种查询返回结果相同，区别在于使用Join运算符的查询略快一些。这是因为在对本地集合查询时，Join运算符会被翻译成Enumerable中对应的方法，这个方法会将purchases表中的数据加载到一个集合中，这个集合是一种键值对的结构，非常便于数据的检索，因此整个查询效率会高一些。

Join运算符的使用方法可以总结为下面这种格式：

```
join 内部变量 in 内部集合 on 外部表的键 equals 内部表的键
```

在LINQ中，使用Join运算符的时候是有外部集合和内部集合之分的，它们的区别在于：

- *外部集合*是输入集合，在本例中是customers。
- *内部集合*是新引入的集合，在本例中是purchases。

Join运算符执行的是内连接，也就是说只有那些含有消费记录的消费者才会被查询出来。在对两个集合进行内连接查询的时候，这两个集合出现的先后顺序是没有关系的，例如查询中可以互换customers和pruchases的位置，查询结果不会改变：

```
from p in purchases                                   //p是外部变量
join c in customers on p.CustomerID equals c.ID       // c是内部变量
...
```

在一个查询中可以使用多个join关键字。例如，当每个消费记录还有子记录的时候，在查询中，可以使用下面这种方式查出这些子记录和消费者的对应关系：

```
from c in customers
join p in purchases on c.ID equals p.CustomerID           // 第一次连接
join pi in purchaseItems on p.ID equals pi.PurchaseID    // 第二次连接
...
```

注意，在上面的查询中，purchases和customers连接时是内部集合，而在和purchaseItems连接时是外部集合。它在customers和purchaseItems中间起了一个过渡的作用。实际上还可以使用多个嵌套的foreach来实现，这种查询效率不高，这里只是作为演示，代码如下：

```
foreach (Customer c in customers)
  foreach (Purchase p in purchases)
    if (c.ID == p.CustomerID)
      foreach (PurchaseItem pi in purchaseItems)
        if (p.ID == pi.PurchaseID)
          Console.WriteLine (c.Name + "," + p.Price + "," + pi.Detail);
```

使用查询表达式语法书写LINQ时，连接变量在定义之后，在整个查询语句中一直是有效的，这点类似于SelectMany查询中的外部范围变量。在不同的join子句之间允许引入where和let子句。

6. 基于多个键的连接

我们可以通过匿名类型在一个查询中基于多个键进行连接查询，如下所示：

```
from x in sequenceX
join y in sequenceY on new { K1 = x.Prop1, K2 = x.Prop2 }
                equals new { K1 = y.Prop3, K2 = y.Prop4 }
...
```

为了保证上面的连接方式能被编译器正确解释，需要将查询中的两个匿名类型定义成完全一样的格式、同样的属性名称、同样的属性个数。因此在编译的时候，会把这两个对象视作同一个对象的两个实例来处理，这样才能够保证可以使用这两个对象作为连接条件。

7. 连接操作在运算符流语法中的使用

下面是一个查询表达式语法的连接查询：

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
select new { c.Name, p.Description, p.Price };
```

改写成运算符流格式之后，查询代码如下：

```
customers.Join (                 // 外部集合
      purchases,                 // 内部集合
      c => c.ID,                 // 外部键选择器
      p => p.CustomerID,         // 内部键选择器
      (c, p) => new
         { c.Name, p.Description, p.Price }   // 结果选择器
);
```

在查询中，查询得到的元素c、p被映射到自定义的匿名类型中，以一个封装后的形式返回。如果想要在最后的映射之前对数据再做一些处理，例如使用orderby对数据进行排序，可以使用下面这种方式：

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
orderby p.Price
select c.Name + " bought a " + p.Description;
```

在运算符流语法中使用join时，必须定义一个临时的匿名类型用于存放查询结果。这样就可以将变量c和p保存下来，在随后的查询中继续使用，如下面代码所示：

```
customers.Join (                        // 外部集合
      purchases,                        // 内部集合
      c => c.ID,                        // 外部键选择器
      p => p.CustomerID,                // 内部键选择器
      (c, p) => new { c, p } )          // 结果选择器
  .OrderBy (x => x.p.Price)
  .Select (x => x.c.Name + " bought a " + x.p.Description);
```

当需要用到连接查询时，最好使用查询表达式语法，从上面的示例可以看出，查询表达式语法定义的查询更清晰、更容易理解。

8. GroupJoin

GroupJoin和前面讨论的Join运算符作用相同，只是返回的结果集的形式不同而已，GroupJoin返回的是嵌套类型的数据集，它的结构是按照外层元素将结果集进行分组，然后将相关数据放置在一个组中。

GroupJoin还支持左外连接。如果join和into关键字配合使用，那么作用和GroupJoin是完全相同的。

下面是一个简单的示例：

```
IEnumerable<IEnumerable<Purchase>> query =
  from c in customers
  join p in purchases on c.ID equals p.CustomerID
  into custPurchases
  select custPurchases; // custPurchases 是一个集合
```

提示：当into关键字出现在join后面的时候，编译器会将into关键字翻译成GroupJoin来执行。而当into出现在Select或者Group子句之后时，则翻译成扩展现有的查询。虽然都是into关键字，但是出现在不同地方，差别非常大。有一点它们是相同的，into关键字总是引入一个新的变量。

GroupJoin的返回结果实际上是集合的集合，也就是一个集合中的元素还是集合。可以使用下面的方式取出集合中的元素：

```
foreach (IEnumerable<Purchase> purchaseSequence in query)
  foreach (Purchase p in purchaseSequence)
    Console.WriteLine (p.Description);
```

上面这种取元素的方式并不是很好，因为在最内层循环中无法得到外层的变量信息。为了解决这个问题，通过数据映射，可以将Purchases和Customer信息封装到一起然后返回，如下面代码所示：

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
into custPurchases
select new { CustName = c.Name, custPurchases };
```

下面这个查询使用Select关键字可以完成与上面相同的功能，但是性能要差一点：

```
from c in customers
select new
{
  CustName = c.Name,
  custPurchases = purchases.Where (p => c.ID == p.CustomerID)
};
```

默认情况下，GroupJoin和左外连接的功能是完全相同的。如果想要执行一个内连接，也就是只查询包含消费记录的客户信息，可以使用下面这种方式：

```
from c in customers join p in purchases on c.ID equals p.CustomerID
into custPurchases
where custPurchases.Any()
select ...
```

由于在连接查询中，into关键字会将数据转换成一个集合的形式，而不是单个元素。因此如果需要对单个元素进行筛选的话，例如要求取出purchases中那些Price大于1000的记录，就必须在使用join之前先用where语句对元素进行筛选，示例代码如下：

```
from c in customers
join p in purchases.Where (p2 => p2.Price > 1000)
  on c.ID equals p.CustomerID
into custPurchases ...
```

从前面的示例我们知道，可以使用Join关键字来定义Lambda表达式，这一点同样适用于GroupJoin，也就是Lambda表达式中可以出现GroupJoin关键字。

9. 使用外连接得到非嵌套结果集

在目前的情况下，GroupJoin执行外连接返回嵌套类型结果集，Join执行内连接而返回一个非嵌套的结果集。那如果我们想使用外连接，并得到一个非嵌套的结果集，那么我们该怎么办呢？解决方案

是首先使用GroupJoin执行外连接，然后再在每个查询子集上使用DefaultEmpty关键字，然后再对最终的结果集使用SelectMany进行查询，下面是一段示例代码：

```
from c in customers
join p in purchases on c.ID equals p.CustomerID into custPurchases
from cp in custPurchases.DefaultIfEmpty()
select new
{
  CustName = c.Name,
  Price = cp == null ? (decimal?) null : cp.Price
};
```

DefaultEmpty会判断子集合（在这里是purchases）是否为空，如果为空，则返回一个null。第二个from子句会被翻译成SelectMany运算符。在这里，它的作用是展开purchase子集合，将这些子集合中的元素重新映射到一个非嵌套结构的结果集中。

10. 连接查询中的查找

在Enumberable类中，Join和GroupJoin运算符要完成连接工作，在内部分为两步来实现。首先，它会将内部集合（Join或GroupJoin关键字之后出现的那个集合）中的数据先加载到本地，存放在一个便于查询的集合中。

这个便于查询的集合实际上是一种以键值对结构存储的集合，可以通过对键的查找迅速找到需要的元素。可以将这种结构理解成字典的数据结构，字典这种结构允许一个键对应多个元素值。在连接中的这个查找表是只读类型的，它的接口定义如下：

```
public interface ILookup<TKey,TElement> :
   IEnumerable<IGrouping<TKey,TElement>>, IEnumerable
{
  int Count { get; }
  bool Contains (TKey key);
  IEnumerable<TElement> this [TKey key] { get; }
}
```

提示： 像其他运算符一样，上面这些连接运算符也支持延迟加载。也就是说，在没有使用连接查询的结果集之前，外部表和查找表中的数据都不会被加载。

我们既可以使用join运算符来自定义查找表，也可以手动地创建这种查找表，并手动地将查找表加载到本地，有以下两个好处：

- 可以在多个查询中重复使用这个查找表。
- 这种查找表可以让整个查询变得更容易理解。

可以使用扩展方法ToLookup来创建查找表，下面这段代码将所有的消费信息存放到一个查找表中，这个表以CustomerID作为键：

```
ILookup<int?,Purchase> purchLookup =
  purchases.ToLookup (p => p.CustomerID, p => p);
```

从上面的代码中可以看到，ToLookup方法有两个参数，第一个参数（p => p.CustomerID）用来定义查找表中的键，第二个参数用于定义将哪些数据加载到查找表中作为值。

遍历查找表的方式跟遍历字典集合的方式非常相似，不同的是，查找表中每个键所对应的是一个集合，也就是一组具有相同键的元素。而字典中每个键则对应一个特定的值。下面这个循环语句输出ID是1的客户的所有消费信息：

```
foreach (Purchase p in purchLookup [1])
  Console.WriteLine (p.Description);
```

如果在查询中使用了查找表，那么使用SelectMany/Select进行连接查询的话，可以达到和Join/GroupJoin相同的功能。更进一步说，在使用了查找表的查询中，Join和SelectMany的作用是等价的，下面是一个简单的示例：

```
from c in customers
from p in purchLookup [c.ID]
select new { c.Name, p.Description, p.Price };

查询结果：
Tom Bike 500
Tom Holiday 2000
Dick Bike 600
Dick Phone 300
...
```

如果在上面的查询中使用DefaultIfEmpty关键字，那么整个查询的连接方式就会变成外连接：

```
from c in customers
from p in purchLookup [c.ID].DefaultIfEmpty()
 select new {
              c.Name,
              Descript = p == null ? null : p.Description,
              Price = p == null ? (decimal?) null : p.Price
            };
```

GroupJoin的作用实际上相当于在一个映射操作中读取查找表中的内容：

```
from c in customers
select new {
             CustName = c.Name,
             CustPurchases = purchLookup [c.ID]
};
```

11. 在Enumerable类中连接的实现方式

下面是Enumerable.Join的最简单的实现方式，代码中没有包含非空验证的部分，代码如下：

```
public static IEnumerable <TResult> Join <TOuter,TInner,TKey,TResult> (
  this IEnumerable <TOuter> outer,
  IEnumerable <TInner> inner,
  Func <TOuter,TKey> outerKeySelector,
  Func <TInner,TKey> innerKeySelector,
  Func <TOuter,TInner,TResult> resultSelector)
{
  ILookup <TKey, TInner> lookup = inner.ToLookup (innerKeySelector);
  return
    from outerItem in outer
    from innerItem in lookup [outerKeySelector (outerItem)]
```

```
        select resultSelector (outerItem, innerItem);
    }
```

GroupJoin的实现方式和Join非常相似，只是略简单一些：

```
    public static IEnumerable <TResult> GroupJoin <TOuter,TInner,TKey,TResult> (
      this IEnumerable <TOuter> outer,
      IEnumerable <TInner> inner,
      Func <TOuter,TKey> outerKeySelector,
      Func <TInner,TKey> innerKeySelector,
      Func <TOuter,IEnumerable<TInner>,TResult> resultSelector)
    {
      ILookup <TKey, TInner> lookup = inner.ToLookup (innerKeySelector);
      return
        from outerItem in outer
        select resultSelector
         (outerItem, lookup [outerKeySelector (outerItem)]);
    }
```

9.5 Zip 运算符

IEnumerable<TFirst>, IEnumerable<TSecond>→IEnumerable<TResult>

Zip是在.NET Framework 4.0中新加入的一个运算符，它可以同时枚举两个集合中的元素（就像拉链的两边一样），返回的集合是经过处理的元素对，例如，下面这个查询的结果集综合了两个输入集合的元素：

```
int[] numbers = { 3, 5, 7 };
string[] words = { "three", "five", "seven", "ignored" };
IEnumerable<string> zip = numbers.Zip (words, (n, w) => n + "=" + w);
```

上面查询得到的结果集如下：

```
3=three
5=five
7=seven
```

两个集合中不能配对的元素会直接被忽略。需要注意的是，Zip运算符只能用于本地集合的查询，它不支持对数据库的查询。

9.6 排序

IEnumerable<TSource>→IOrderedEnumerable<TSource>

方法	描述	等价SQL
OrderBy、ThenBy	将一个集合按照升序排列	ORDER BY ...
OrderByDescending, ThenByDescending	将一个集合按照降序排列	ORDER BY ... DESC
Reverse	将一个集合中的元素反转	无对应的实现方式

经过排序的集合中的元素值和未排序之前是相同的，只是元素的顺序不同。

OrderBy、OrderByDescending、ThenBy和ThenBy Descending

1. OrderBy和OrderByDescending 参数

参数	类型
输入集合	`IEnumerable<TSource>`
键选择器	`TSource => TKey`

返回类型为 `IOrderedEnumerable<TSource>`

ThenBy和ThenByDescending 参数

参数	类型
输入集合	`IOrderedEnumerable<TSource>`
键选择器	`TSource => TKey`

2. 查询语法

```
orderby 表达式1 [descending] [, 表达式2 [descending] ... ]
```

3. 概述

OrderBy可以按照指定的方式对集合中的元素进行排序，具体的排序方式可以在keySelector表达式中定义。下面这个语句将names集合中的元素按照字母升序进行排序：

```
IEnumerable<string> query = names.OrderBy (s => s);
```

下面是将names集合中的元素按长度排序：

```
IEnumerable<string> query = names.OrderBy (s => s.Length);
//输出结果： { "Jay", "Tom", "Mary", "Dick", "Harry" };
```

如果通过Order By按照指定顺序进行排序后，集合中的元素的相对顺序仍无法确定时，可以使用Thenby：

```
IEnumerable<string> query = names.OrderBy (s => s.Length) ThenBy (s => s)
// Result: { "Jay", "Tom", "Dick", "Mary", "Harry" };
```

ThenBy关键字的作用是在前一次排序的基础上再进行一次排序。在一个查询中，可以使用任意多个ThenBy关键字。在下面这个查询中，对names集合进行了三次排序。第一次根据名字长度进行排序，再对名字长度相同的元素按照其第二个字母进行排序，最后对于那些名字长度和第二个字母都相同的元素再按照第一个字母排序，这种排序方式如下：

```
names.OrderBy (s => s.Length).ThenBy (s => s[1]).ThenBy (s => s[0]);
```

查询表达式语法中等价的表达方式如下：

```
from s in names
orderby s.Length, s[1], s[0]
select s;
```

警告：下面的修改是不正确的——它实际先按s[1]排序，然后再按s.Length排序（或者在数据库查询中，它只按s[1]排序，并抛弃前一个排序方式）：

```
from s in names
orderby s.Length
orderby s[1]
...
```

LINQ中还提供了OrderByDescending和ThenByDescending关键字，这两个关键字也是用于完成对集合的排序功能，它们的功能和OrderBy/ThenBy相同，用法也一样，只是它们排序后的集合中的元素是按指定字段的降序排列，下面是一个简单的示例：

```
dataContext.Purchases.OrderByDescending (p => p.Price)
                     .ThenBy (p => p.Description);
```

查询表达式语法的定义方式如下：

```
from p in dataContext.Purchases
orderby p.Price descending, p.Description
select p;
```

4. 比较与校对

在对本地集合的查询中，LINQ会根据默认的IComparable接口中的算法对集合中的元素进行排序。如果不想使用默认的排序方式，可以自己实现一个IComparable对象，然后将这个对象传递给查询LINQ，下面是示例对数组中的名字按照字母顺序排序，不区分大小写：

```
names.OrderBy (n => n, StringComparer.CurrentCultureIgnoreCase);
```

在查询表达式语法中我们没有办法将一个IComparable对象传递给查询语句，也就不能进行自定义的查询。实际上在LINQ to DB和EF中也不能这样做，在对远程数据源的查询中，排序算法是由参与查询的列决定的，排序算法事先没有办法知道根据哪个列进行排序，如果需要排序，必须显式地指定按照哪些列来排序。例如集合中的元素有大小写之分，但如果只想对这个集合中的元素按照字母顺序排序，不区分大小写，可以用ToUpper方法来实现：

```
from p in dataContext.Purchases
orderby p.Description.ToUpper()
select p;
```

5. IOrderedEnumerable和IOrderedQueryable

在使用了排序操作的查询中，排序运算符会将集合转换成IEnumerable<T>类型的一个特殊子类。具体来说，对Enumerable类型的集合查询时，返回IOrderedEnumerable类型的集合；在对Queryable类型的集合查询时，返回IOrderedQueryable类型的集合。这两种子类型是为排序专门设计的，在它们上面可以直接使用ThenBy运算符来进行多次排序。

IOrderedEnumerable和IOrderedQueryable并没有公开其他的成员，所以在对集合进行排序时，并不

能感觉到这两种类型的存在。但是如果对集合分步排序的话，就需要区分使用哪种类型，如下面代码所示：

```
IOrderedEnumerable<string> query1 = names.OrderBy (s => s.Length);
IOrderedEnumerable<string> query2 = query1.ThenBy (s => s);
```

如果将query1的类型定义成`IEnumerable<string>`，那么第二行代码就无法编译了，因为`ThenBy`关键字需要接收一个`IOrderedEnumerable<string>`类型的序列，如果是其他类型，它就无法处理了。这里可以使用隐式类型来定义query1和query2以避免这个问题：

```
var query1 = names.OrderBy (s => s.Length);
var query2 = query1.ThenBy (s => s);
```

但是如果使用不恰当，隐式类型也会出现问题，如下面这段代码就会出现编译时错误：

```
var query = names.OrderBy (s => s.Length);
query = query.Where (n => n.Length > 3);          // 编译时错误
```

之所以会出现编译时错误，是因为第一行查询语句中最后使用的是`OrderBy`关键字，变量`query`会被翻译成`IOrderedEnumerable<string>`类型。但是，第二行的`Where`关键字则返回一个`IEnumerable<string>`类型的集合，这种类型的集合不能给`query`赋值，显然它们的类型是不一致的。为了避免这个问题，可以在排序查询之后，对`query`调用`AsEnumerable()`方法，代码如下：

```
var query = names.OrderBy (s => s.Length).AsEnumerable();
query = query.Where (n => n.Length > 3);                // 没有错误
```

在对远程数据源的查询中，需要用`AsQueryable`代替AsEnumerable。

9.7 Grouping

IEnumerable<TSource>→IEnumerable<IGrouping<TSource,TElement>>

方法	描述	等价的SQL语句
GroupBy	将一个序列分组，形成多个子序列	GROUP BY

9.7.1 GroupBy

参数	类型
输出集合	IEnumzerable<TSource>
键选择器	TSource => TKey
元素选择器（可选）	TSource => TElement
比较方式（可选）	IEqualityComparer<TKey>

1. 查询语法

group *元素表达式* by *键表达式*

2. 概述

GroupBy可以将一个非嵌套的集合按某种条件分组，然后将得到的分组结果以组为单位封装到一个集合中。例如，在下面这个示例中，将*c:\temp*目录下的所有文件按照文件类型进行分组：

```
string[] files = Directory.GetFiles ("c:\\temp");

IEnumerable<IGrouping<string,string>> query =
  files.GroupBy (file => Path.GetExtension (file));
```

上面的类型太过麻烦，也可以使用隐式类型来定义：

```
var query = files.GroupBy (file => Path.GetExtension (file));
```

以下面这种方式遍历结果集：

```
foreach (IGrouping<string,string> grouping in query)
{
  Console.WriteLine ("扩展名： " + grouping.Key);
  foreach (string filename in grouping)
    Console.WriteLine (" - " + filename);
}

扩展名： .pdf
  -- chapter03.pdf
  -- chapter04.pdf
扩展名： .doc
  -- todo.doc
  -- menu.doc
  -- Copy of menu.doc
...
```

Enumerable.GroupBy的内部实现是，首先将集合中所有元素按照键值的关系存储到一个临时的字典类型的集合中。然后再将这个临时集合中的所有分组返回给调用者。这里一个分组就是一个键和它所对应的一个小集合，键的定义如下：

```
public interface IGrouping <TKey,TElement> : IEnumerable<TElement>,
                                             IEnumerable
{
  TKey Key { get; } // 使用键可以得到它对应的整个集合
}
```

默认情况下，分组之后的元素不会对原始元素做任何处理，如果需要在分组过程中对元素做某些处理的话，可以给元素选择器指定一个参数。下面这个示例演示了如何在分组的同时将集合中的元素转换成大写形式：

```
files.GroupBy (file => Path.GetExtension (file), file => file.ToUpper());
```

在结果集中，元素选择器并不依赖于键选择器，也就是说它们两个是互不影响的。具体到当前的示例，集合中元素的名字被转换成了大写形式，但键并没有改变，如果将上面查询的结果输出，会得到如下结果：

```
扩展名： .pdf
  -- CHAPTER03.PDF
  -- CHAPTER04.PDF
```

```
扩展名: .doc
  -- TODO.DOC
```

经过分组的集合，在输出的时候，并没有按照键的升序排列。这里需要注意，GroupBy只对集合进行分组，并不做任何排序操作，如果想要对集合进行排序的话，需要使用额外的OrderBy关键字，下面是一个示例：

```
files.GroupBy (file => Path.GetExtension (file), file => file.ToUpper())
    .OrderBy (grouping => grouping.Key);
```

在查询表达式语法中，GroupBy可以使用下面这个格式来创建：

```
group 元素表达式 by 键表达式
```

将上面的查询以查询表达式语法定义的话，代码如下：

```
from file in files
group file.ToUpper() by Path.GetExtension (file);
```

和其他的查询一样，当查询语句中出现了select或者group的时候，整个查询就结束了，如果不想让查询就此结束，那么就需要扩展整个查询，可以像下面这样使用into关键字：

```
from file in files
group file.ToUpper() by Path.GetExtension (file) into grouping
orderby grouping.Key
select grouping;
```

在group by查询中，经常需要扩展查询语句，因为需要对分组后的集合进一步进行处理。下面这个查询就会将少于5个元素的分组过滤掉：

```
from file in files
group file.ToUpper() by Path.GetExtension (file) into grouping
where grouping.Count() < 5
select grouping;
```

提示： 在LINQ中，group by后面跟着where查询相当于SQL中的HAVING关键字。这个where所作用的对象是整个集合或者集合中的每个分组，而不是单个元素。

有时候我们只需查询结果中元素的个数，那么就可以使用聚合函数得到需要的数值结果，例如下面这种查询，将整个集合中相同的元素分到一组，然后再按集合中元素个数降序排列，代码如下：

```
string[] votes = { "Bush", "Gore", "Gore", "Bush", "Bush" };

IEnumerable<string> query = from vote in votes
                    group vote by vote into g
                    orderby g.Count() descending
                    select g.Key;

string winner = query.First(); // Bush
```

3. LINQ to SQL和EF中使用GroupBy

分组操作同样适用于对数据库的查询。如果是在EF中，在使用了关联属性的情况下，分组操作并不

像在SQL中那样常用。例如，当想要找出消费记录超过两条的客户时，不需要使用group关键字，下面这种查询方式可以很好地实现：

```
from c in dataContext.Customers
where c.Purchases.Count >= 2
select c.Name + " has made " + c.Purchases.Count + " purchases";
```

当需要统计一年中的销售总额时，可能会用到分组操作，如下面这个查询所示：

```
from p in dataContext.Purchases
group p.Price by p.Date.Year into salesByYear
select new {
             Year = salesByYear.Key,
             TotalValue = salesByYear.Sum()
           };
```

LINQ中的分组功能对SQL中的“GROUP BY”进行了很大的扩展，可以认为LINQ中的分组是SQL中分组功能的一个超集。

和传统SQL查询不同点是，在LINQ中不需要对分组或者排序子句中的变量进行映射。像上面看到的，group子句将结果集暂存于salesByYear中，之后的查询可以从这个变量中直接取值。

4. 根据多个键进行分组

当需要使用集合中多个键来进行分组时，可以使用匿名类型将这几个键封装到一起，如下所示：

```
from n in names
group n by new { FirstLetter = n[0], Length = n.Length };
```

5. 自定义比较条件

在使用GroupBy对集合进行分组时，我们可以给GroupBy运算符传递一个自定义的相等比较器，这样就可以使用自定义的相等条件来覆盖默认的等价算法。实际上在查询中很少需要自定义一个等价判断条件。因为如果仅仅是想改变等价判断条件的话，通过改变键选择器已经足够用了。例如，下面这个语句就创建了一个不区分大小写的分组：

```
group name by name.ToUpper()
```

9.8 集合运算符

`IEnumerable<TSource>, IEnumerable<TSource>→IEnumerable<TSource>`

方法	描述	等价SQL
Concat	返回一个包含两个序列中所有元素的新序列	UNION ALL
Union	返回一个包含两个序列中所有元素，且移除重复元素之后的新序列	UNION
Intersect	返回在两个序列中都存在的元素	WHERE ... IN (...)
Except	返回存在于第一个序列中但不存在于第二个序列中的元素	EXCEPT或者 WHERE ... NOT IN (...)

9.8.1 Concat和Union

Concat运算符的作用是合并两个集合，合并方式是将第一个集合中所有元素放置到结果集中，然后再将第二个集合中的元素放在第一个结果集的后面。然后返回结果集。Union执行的也是这种合并操作，但是它最后会将结果集中重复的元素去除，以保证结果集中每个元素都是唯一的。下面是一个简单示例：

```
int[] seq1 = { 1, 2, 3 }, seq2 = { 3, 4, 5 };

IEnumerable<int>
  concat = seq1.Concat (seq2),    // { 1, 2, 3, 3, 4, 5 }
  union = seq1.Union (seq2);      // { 1, 2, 3, 4, 5 }
```

当对两个不同类型但基类型却相同的序列执行合并时，需要显式地指定这两个集合的类型以及合并之后的集合类型。例如，使用反射方法，所有的方法和属性的基类都是MethodInfo和PropertyInfo，当我们需要对两个集合进行合并时，就可以在Concat中显式地指定将基类型作为集合类型，如下面代码所示：

```
MethodInfo[] methods = typeof (string).GetMethods();
PropertyInfo[] props = typeof (string).GetProperties();
IEnumerable<MemberInfo> both = methods.Concat<MemberInfo> (props);
```

在下面这个示例中，在合并数组之前先对string类型中的方法进行了筛选：

```
var methods = typeof (string).GetMethods().Where (m => !m.IsSpecialName);
var props = typeof (string).GetProperties();
var both = methods.Concat<MemberInfo> (props);
```

但是，上面这段代码在C#4.0中可以正常编译，而在C#3.0中则会编译失败。这是由于接口不一致导致的，在C#4.0中，变量methods是IEnumerable<MethodInfo>类型的，这个类型从原始类型到IEnumerable<MethodInfo>需要编译器内部进行一系列的转换，但C#3.0并没有提供配套的转换方法，因此会编译出错。这也说明了C#正朝着使工作更高效的方向发展。

9.8.2 Intersect和Except

Intersect运算符用于取出两个集合中元素的交集。Except用于取出只出现在第一个集合中的元素，如果某个元素在两个集合中都存在，那么这个元素就不会包含在结果中。

```
int[] seq1 = { 1, 2, 3 }, seq2 = { 3, 4, 5 };

IEnumerable<int>
  commonality = seq1.Intersect (seq2),     // { 3 }
  difference1 = seq1.Except (seq2),        // { 1, 2 }
  difference2 = seq2.Except (seq1);        // { 4, 5 }
```

Enumerable.Except的内部实现方式是，首先将第一个集合中的所有元素加载到一个字典集合中，然后再比对第二个集合中的元素，如果字典中的某个元素在第二个集合中出现了，那么就将这个元素从字典中移除。在SQL中NOT EXISTS或者NOT IN可以完成相同的功能。下面是一个简单的示例：

```
SELECT number FROM numbers1Table
WHERE number NOT IN (SELECT number FROM numbers2Table)
```

9.9 转换方法

从根本上讲，LINQ处理的是IEnumerable<T>类型的集合，之所以现在众多的集合类型都可以使用LINQ进行处理，是因为编译器内部可以将其他类型的序列转换成IEnumerable<T>类型的。下表中列出了LINQ中的各种转换函数，用于完成不同类型序列之间的转换：

方法	描述
OfType	将IEnumerable类型的元素转换成IEnumerable<T>类型，如果转换过程中有错误类型的元素，则忽略并丢弃这个元素
Cast	将IEnumerable类型的元素转换成IEnumerable<T>类型，如果转换过程中有错误类型的元素，则抛出异常
ToArray	将IEnumerable<T>类型的集合转换成T[]类型
ToList	将IEnumerable<T>类型的集合转换成List<T>类型
ToDictionary	将IEnumerable<T>类型的集合转换成Dictionary<TKey,TValue>
ToLookup	将IEnumerable<T>类型的集合转换成ILookup<TKey,TElement>
AsEnumerable	将集合转换成IEnumerable<T>类型
AsQueryable	将集合转换成IQueryable<T>类型

9.9.1 OfType和Cast

OfType和Cast可以将非IEnumerable类型的集合转换成IEnumerable<T>类型的集合，下面是一个示例：

```
ArrayList classicList = new ArrayList();      // 在 System.Collections命名空间中
classicList.AddRange ( new int[] { 3, 4, 5 } );
IEnumerable<int> sequence1 = classicList.Cast<int>();
```

Cast和OfType运算符的唯一不同就是它们遇到不相容类型时的处理方式：Cast会抛出异常，而OfType则会忽略这个类型不相容的元素。使用之前的示例来演示这两个运算符的不同：

```
DateTime offender = DateTime.Now;
classicList.Add (offender);
IEnumerable<int>
  sequence2 = classicList.OfType<int>(),   // 忽略不相容的DateTime
  sequence3 = classicList.Cast<int>();     // 抛出异常
```

元素相容的规则与C#的is运算符完全相同，因此只能考虑引用转换和拆箱转换。我们可以通过查看OfType的内部实现来了解这个问题：

```
public static IEnumerable<TSource> OfType <TSource> (IEnumerable source)
{
  foreach (object element in source)
    if (element is TSource)
      yield return (TSource)element;
}
```

Cast运算符的内部实现与OfType完全相同，只是省略了类型检查那行代码：

```
public static IEnumerable<TSource> Cast <TSource> (IEnumerable source)
{
  foreach (object element in source)
    yield return (TSource)element;
}
```

受Cast运算符实现方式的限制，我们不能使用它来进行数字类型和自定义类型的强制转换，如果需要转换的话，可以使用Select运算符来实现。也就是说，LINQ中的Cast运算符并不像C#中的类型转换那样方便，在C#语法中，可以进行下面的转换：

```
int i = 3;
long l = i;        // 隐式类型转换
int i2 = (int) l;  // 显式类型转换
```

如果想要使用OfType或者Cast将一个int型的集合转换成long型的集合，是否可以像上面那样直接转换呢？如下面代码所示：

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

IEnumerable<long> test1 = integers.OfType<long>();
IEnumerable<long> test2 = integers.Cast<long>();
```

这样操作之后，test1中没有元素，而在遍历test2时会抛出异常。查看OfType的实现方式不难理解为什么test1中没有元素。在OfType的实现中，逻辑先会使用下面这个表达式检查每个元素的类型：

```
(element is long)
```

由于element是int型的，而int和long之间并没有继承关系，所以上面这个表达式始终是false。那么集合test2中也就不会有元素存在。

提示：在遍历test2中的元素时，为什么会抛出异常呢？这是因为在Cast的实现中，集合中的每个元素被定义成object类型的，当目标类型TSource（当前示例中是int）是值类型时，编译器任务要将一个object类型的数据转换成int型的数据，这是一个典型的拆箱操作。关于装箱和拆箱的详细内容可以参考本书第3章“装箱和拆箱”一节。

```
int value = 123;
object element = value;
long result = (long) element;    // 异常
```

由于Cast中元素的原始类型是object类型的，因此Cast内部执行的转换是从object型到long型，而我们期望的是从int型到long型的转换，这里明显出现了歧义，因此会抛出异常。

如前面提到的，可以使用Select关键字解决这个问题，如：

```
IEnumerable<long> castLong = integers.Select (s => (long) s);
```

OfType和Cast的另一个重要功能是：按类型从集合中取出元素。例如，IEnumerable<Fruit>类型的集合，这里Fruit是各种水果对象的基类，使用OfType<Apple>可以取出源集合中所有的Apple对象，这个功能在LINQ to XML中非常有用，这方面的具体内容会在第10章中做进一步介绍。

Cast还可以在查询表达式语法中使用，方式很简单，只需使用类型定义一个范围变量即可，如下面这行代码所示：

```
from TreeNode node in myTreeView.Nodes
...
```

9.9.2 ToArray、ToList、ToDictionary和ToLookup

ToArray和ToList可以分别将集合转换成数组和泛型集合。这两个运算符也会强制LINQ查询语句立即执行，也就是说当整个查询是延迟加载的时候，一旦遇到ToArray或者ToList，整个语句会被立即执行。这一点在第8章中“延迟加载”一节已经介绍过。

ToDictionary和ToLookup也是两个用于转换集合类型的方法，它们接收下表中的参数：

参数	类型
输入集合	IEnumerable<TSource>
键选择器	TSource => TKey
元素选择器（可选）	TSource => TElement
比较器（可选）	IEqualityComparer<TKey>

ToDictionary方法也会强制查询语句立即执行，然后将查询结果放在一个Dictionary类型的集合中。ToDictionary方法中的键选择器必须为每个元素提供一个唯一的键，也就是说不同元素的键是不能重复的，否则在查询的时候系统会抛出异常。而ToLookup方法的要求则不同，它允许多个元素共用相同的键。在本章的“连接查询中的查找”一节中对此有过详细介绍。

9.9.3 AsEnumerable和AsQueryable

AsEnumerable将一个其他类型的集合转换成IEnumerable<T>类型，这样可以强制编译器使用Enumerable类中的方法来解析查询中的运算符。在第8章的“综合使用解释型的查询和本地查询”一节中有相关的示例演示。

而AsQueryable方法则会将一个其他类型的集合转换成IQueryable<T>类型的集合，前提是被转换的集合实现了IQueryable<T>接口。否则IQueryable<T>会实例化一个对象，然后存储在本地数组外面，看起来是可以调用IQueryable中的方法，但实际上这些方法并没有真正的意义。

9.10 元素运算符

IEnumerable<TSource>→TSource

方法	描述	等价SQL
First、FirstOrDefault	返回集合中的第一个元素。可以以参数的形式为这两个方法添加筛选条件	SELECT TOP 1 ... ORDER BY ...
Last、LastOrDefault	返回集合中最后一个元素。可以以参数的形式为这两个方法添加筛选条件	SELECT TOP 1 ... ORDER BY ... DESC

方法	描述	等价SQL
Single、SingleOrDefault	功能和First/First-OrDefault相同，但是它要求集合中有且仅有一个元素，否则抛出异常	
ElementAt、ElementAtOrDefault	返回集合中指定位置上的元素	抛出异常
DefaultIfEmpty	当集合中没有元素的时候，返回null或者default(TSource)	外连接

所有以“OrDefault”结尾的方法有一个共同点，那就是当集合为空或者集合中没有符合要求的元素时，这些方法不抛出异常，而是返回一个默认类型的值default(TSource)。

对于引用类型的元素来说default(TSource)是null，而对于值类型的元素来说，这个默认值通常是0。

9.10.1 First、Last和Single

参数	类型
输入集合	IEnumerable<TSource>
筛选条件（可选）	TSource => bool

下面这个示例演示了First和Last的用法：

```
int[] numbers = { 1, 2, 3, 4, 5 };
int first      = numbers.First();                            // 1
int last       = numbers.Last();                             // 5
int firstEven  = numbers.First  (n => n % 2 == 0);           // 2
int lastEven   = numbers.Last   (n => n % 2 == 0);           // 4
```

下面这个示例演示了First和FirstOrDefault的区别：

```
int firstBigError  = numbers.First          (n => n > 10);   // 异常
int firstBigNumber = numbers.FirstOrDefault (n => n > 10);   // 0
```

为了避免出现异常，在使用Single运算符时必须保证集合中有且仅有一个元素；而SingleOrDefault运算符则要求集合中有一个或零个元素，如下面代码所示：

```
int onlyDivBy3 = numbers.Single (n => n % 3 == 0);   // 3
int divBy2Err  = numbers.Single (n => n % 2 == 0);   // 错误: 2 & 4 符合要求

int singleError = numbers.Single (n => n > 10);                    // 错误
int noMatches   = numbers.SingleOrDefault (n => n > 10);           // 0
int divBy2Error = numbers.SingleOrDefault (n => n % 2 == 0);       // 错误
```

Single是所有元素运算符中要求最多的，而FirstOrDefault和LastOrDefault则对集合中的元素没有什么要求。

在LINQ to SQL和EF中，Single运算符通常应用于使用主键到数据库中查找特定的单个元素，如下面的代码所示：

```
Customer cust = dataContext.Customers.Single (c => c.ID == 3);
```

9.10.2 ElementAt运算符

参数	类型
输入集合	IEnumerable<TSource>
元素的下标	Int

ElementAt运算符可以根据指定的下标取出集合中的元素，如下面这个示例所示：

```
int[] numbers  = { 1, 2, 3, 4, 5 };
int third      = numbers.ElementAt (2);            // 3
int tenthError = numbers.ElementAt (9);            // 异常
int tenth      = numbers.ElementAtOrDefault (9);   // 0
```

Enumerable.ElementAt的实现方式是，如果它所查询的集合实现了IList<T>接口，那么在取元素的时候，就使用IList<T>中的索引器，否则，就使用自定义的循环方法，在循环中依次向后查找元素，循环*n*次之后，返回下一个元素。ElementAt运算符不能在LINQ to SQL 和EF中使用。

9.10.3 DefaultIfEmpty

DefaultIfEmpty可以将一个空的集合转换成null或者default()类型。这个运算符一般用于定义外连接查询。我们在本章“使用SelectMany实现外连接”和“使用外连接得到非嵌套结果集”两节中使用过这个运算符。

9.11 聚合方法

IEnumerable<TSource>→ *标量值*

方法	描述	等价SQL
Count、LongCount	返回集合中元素的个数。可以添加筛选条件	COUNT (...)
Min、Max	返回集合中最大或最小的元素	MIN (...)、 MAX (...)
Sum、Average	统计集合中所有元素值的总和或者平均值	SUM (...),、AVG (...)
Aggregate	让用户可以自定义聚合方法	抛出异常

9.11.1 Count和LongCount

参数	类型
输入集合	IEnumerable<TSource>
筛选条件	TSource => bool

Count运算符的作用是返回集合中元素的个数：

```
int fullCount = new int[] { 5, 6, 7 }.Count();   // 3
```

Enumerable.Count方法的内部实现方式如下：首先判断输入集合有没有实现ICollection<T>接口，如果实现了，那么它就调用ICollection<T>.Count方法得到元素个数。否则就遍历整个集合中的元素，统计出元素的个数，然后返回。

还可以为这个方法中添加一个筛选条件，如下所示：

```
int digitCount = "pa55w0rd".Count (c => char.IsDigit (c));   // 3
```

LongCount运算符的作用和Count是相同的，只是它的返回值的类型是int64，也就是它能用于大数据量的统计，int64能统计大概20亿个元素的集合。

9.11.2 Min和Max

参数	类型
输入集合	IEnumerable<TSource>
结果选择器	TSource => TResult

Min和Max返回集合中最小和最大的元素：

```
int[] numbers = { 28, 32, 14 };
int smallest = numbers.Min(); // 14;
int largest  = numbers.Max(); // 32;
```

如果指定一个选择器表达式，那么结果会被映射成期望的格式：

```
int smallest = numbers.Max (n => n % 10);   // 8;
```

如果集合没有实现IComparable<T>接口的话，那么我们就必须为这两个运算符提供选择器，如下面的代码所示：

```
Purchase runtimeError = dataContext.Purchases.Min ();                // 错误
decimal? lowestPrice = dataContext.Purchases.Min (p => p.Price);     // 正确
```

选择器表达式不仅定义了元素的比较方式，还定义了最后的结果集的类型。在前面的示例中，查询的最终结果是decimal类型的，而不是Purchase类型的。如果想要得到钱数最少的消费记录，需要使用子查询，如下面代码所示：

```
Purchase cheapest = dataContext.Purchases
  .Where (p => p.Price == dataContext.Purchases.Min (p2 => p2.Price))
  .FirstOrDefault();
```

要满足上面的需求，还有一种不使用聚合函数的解决方案，那就是先使用OrderBy关键字按照Purchases中的价格排序，然后再使用FirstOrDefault关键字得到钱数最小的消费记录。

9.11.3 Sum和Average

参数	类型
输入集合	IEnumerable<TSource>
结果选择器（可选）	TSource => TResult

Sum和Average也是常用的聚合运算符，它们的用法与前面的Min和Max运算符非常相似：

```
decimal[] numbers = { 3, 4, 8 };
decimal sumTotal = numbers.Sum();                // 15
decimal average = numbers.Average();             // 5（平均值）
```

下面这个查询可以得到names集合中每个名字的长度：

```
int combinedLength = names.Sum (s => s.Length);      // 19
```

Sum和Average的返回值类型是有限的，它们内置了以下几种固定的返回值类型：int、long、 float、double、decimal以及这几种类型的可空类型。这里返回值都是值类型，也就是，Sum和Average的预期结果都是数字。而Min和Max则会返回所有实现了IComparable<T>接口的类型，如string类型。

更进一步讲，Average值返回两种类型：decimal和double，在查询中具体返回哪种类型，可以参考下表：

选择器类型	结果类型
decimal	decimal
Int、long float、double	double

根据上表中的规则，下面这行代码是不能编译的（因为不能将double类型转换成int类型）：

```
int avg = new int[] { 3, 4 }.Average();
```

但下面这行代码却可以编译成功：

```
double avg = new int[] { 3, 4 }.Average();   // 3.5
```

Average为了避免查询过程中数值的精度损失，会自动将返回值类型的精度升高一级，在上面这个示例中，我们对一组int型的元素求平均值，得到结果为3.5，并不需要像下面这样显式地将结果进行强制类型转换：

```
double avg = numbers.Average (n => (double) n);
```

在对数据库的查询中，Sum和Average会被转换成标准的SQL聚合语句。例如，当需要查出Purchases的平均值大于$500的客户信息时，可以使用下面这种查询方式：

```
from c in dataContext.Customers
where c.Purchases.Average (p => p.Price) > 500
select c.Name;
```

9.11.4 Aggregate

通过Aggregate运算符我们可以自定义聚合方法，这个运算符只能用于本地集合的查询中，不支持LINQ to SQL和EF。这个运算符的具体功能要根据它在特定情况下的定义来看，下面这个查询演示了如何使用Aggregate运算符完成和Sum相同的功能：

```
int[] numbers = { 2, 3, 4 };
int sum = numbers.Aggregate (0, (total, n) => total + n);    // 9
```

Aggregate运算符的第一个参数是一个种子，用于指示统计结果的初始值是多少；第二个参数是一个表达式，用于更新统计结果，并将统计结果赋值给新的变量；第三个参数是可选的，用于将统计结果映射成期望的形式。

提示：Aggregate运算符最大的问题是，它实现的功能通过foreach语句也可以实现，而且foreach语句的语法更清晰明了。Aggregate的主要用处在于处理比较大或者比较复杂的聚合操作。

---

1. 无初始值的聚合方法

在使用Aggregate运算符时，可以省略初始值。编译器会自动将集合的第一个元素作为初始值，然后集合从第二个元素开始统计。下面是一个无初始值的聚合查询：

```
int[] numbers = { 1, 2, 3 };
int sum = numbers.Aggregate ((total, n) => total + n);    // 6
```

虽然没有指定初始值，但上面这个示例的查询结果和前面的是一样的，但它们内部的统计过程是不同的。第一次的统计从0开始：0+1+2+3；第二次的内部计算则是：1+2+3。下面这个示例可以更好地演示这个问题：

```
int[] numbers = { 1, 2, 3 };
int x = numbers.Aggregate (0, (prod, n) => prod * n); // 0*1*2*3 = 0
int y = numbers.Aggregate ( (prod, n) => prod * n);  //   1*2*3 = 6
```

我们还会在本书第23章介绍这种没有初始值的聚合方法，它的好处是不需要进行重载就可以调用不同的方法。但是事物总是两面的，没有初始值的聚合方法在使用过程中经常会被误用，它有些使用上的陷阱需要我们特别注意。

2. 无初始值聚合函数的使用陷阱

无初始值的聚合函数应用于一组元素互相关联并且元素的顺序可以互换的查询中。如果用在其他情形中，那么结果集既不直观，也不一定准确。例如下面这个查询：

```
(total, n) => total + n * n
```

这个查询中，元素既不互相关联，元素位置也不能互换（例如，1+2*2 != 2+1*1）。如果使用上面这个聚合条件查询一个含有2、3、4三个元素的集合时：

```
int[] numbers = { 2, 3, 4 };
int sum = numbers.Aggregate ((total, n) => total + n * n);    // 27
```

预期的计算方式是：

```
2*2 + 3*3 + 4*4      // 29
```

而它实际的计算方式是：

```
2 + 3*3 + 4*4              // 27
```

有多种方法解决这个问题。首先可以将0包含到numbers数组中：

```
int[] numbers = { 0, 2, 3, 4 };
```

这种做法只是为了避免错误，看起来很不规范，而且如果使用PLINQ进行并行查询，那么它的查询结果也是不正确的，因为在PLINQ中，编译器会结合多个元素的值作为初始值。为了说明这个问题，使用下面这个示例进行说明：

```
f(total, n) => total + n * n
```

在LINQ to Object中执行时，它的计算方式如下：

```
f(f(f(0, 2),3),4)
```

而在PLINQ中，它的计算方式则是：

```
f(f(0,2),f(3,4))
```

查询的输出结果如下：

```
第1个分区：a = 0 + 2*2  (= 4)
第2个分区：b = 3 + 4*4  (= 19)
最终结果：      a + b*b  (= 365!)
或是：          b + a*a  (= 35!)
```

有两个比较好的解决方案。第一个解决方案是使用0作为聚合函数的初始值。使用这个方案时有一点需要注意，在PLINQ中，为了防止查询执行不必要的循环，需要使用一个特殊的重载方法来完成这个查询（参考第23章的“优化PLINQ”一节）。

第二个解决方案是将查询重新定义成关联的并且元素可交换的形式：

```
int sum = numbers.Select (n => n * n).Aggregate ((total, n) => total + n);
```

提示：当然，在上面这个情形下，应该使用Sum运算符而不是Aggregate：

```
int sum = numbers.Sum (n => n * n);
```

实际上使用Sum和Average我们可以实现很大的功能，例如使用Average可以用来求均方根：

```
Math.Sqrt (numbers.Average (n => n * n))
```

设置标准的除法也可以使用Averge来实现：

```
double mean = numbers.Average();
double sdev = Math.Sqrt (numbers.Average (n =>
              {
                double dif = n - mean;
                return dif * dif;
```

```
}));
```

上面的这种Average用法安全且高效。在第23章中我们将给出使用自定义聚合查询的实际示例，这些示例中演示的自定义聚合函数不能被Sum或者Avergae所取代。

9.12 数量词

IEnumerable<TSource>→*bool*

方法	描述	等价的SQL语句
Contains	如果集合中包含给定的元素，返回true	WHERE ... IN (...)
Any	如果集中有任何元素符合条件就返回true	WHERE ... IN (...)
All	如果集合中所有元素都符合给定条件，则返回true	WHERE (...)
SequenceEqual	如果两个集合中的元素完全一致，就返回true	

9.12.1 Contains和Any

Contains关键字接收一个TSource类型的参数；而Any的参数则定义了筛选条件，这个参数是可选的。

当集合中包含给定的元素时，Contains运算符返回true：

```
bool hasAThree = new int[] { 2, 3, 4 }.Contains (3);     // true;
```

Any关键字对集合中元素的要求要低一点，只要集合中有一个元素符合要求，就返回true。可以使用Any关键字重新定义上面的那个查询：

```
bool hasAThree = new int[] { 2, 3, 4 }.Any (n => n == 3);     // true;
```

Any包含了Contains关键字的所有功能，还有一些其他的功能，如：

```
bool hasABigNumber = new int[] { 2, 3, 4 }.Any (n => n > 10);  // false;
```

如果在使用Any关键字的时候不带参数，那么只要集合中有一个元素符合要求，就返回true，下面是之前的查询的另一种实现方式：

```
bool hasABigNumber = new int[] { 2, 3, 4 }.Where (n => n > 10).Any();
```

Any关键字在子查询中使用特别广泛，尤其是在对数据库的查询中，如下面这个示例：

```
from c in dataContext.Customers
where c.Purchases.Any (p => p.Price > 1000)
select c
```

9.12.2 All和SequenceEqual

当集合中的元素都符合给定的条件时，All运算符返回true。下面这个查询返回所有消费金额少于100的那些客户：

```
dataContext.Customers.Where (c => c.Purchases.All (p => p.Price < 100));
```

SequenceEqual用于比较两个集合中的元素是否相同，如果相同则返回true。它的筛选条件要求元素个数相同、元素内容相同而且元素在集合中的顺序也必须是相同的。

9.13 生成集合的方法

void→IEnumerable<TResult>

方法	描述
Empty	创建一个空的集合
Repeat	创建一个有重复元素的集合
Range	创建一个整型的集合

Empty、Repeat和Range都是静态的非扩展方法，它们只能用于本地集合中。

9.13.1 Empty运算符

Empty用于创建一个空的集合，它需要接收一个用于标识集合类型的参数：

```
foreach (string s in Enumerable.Empty<string>())
  Console.Write (s);                                    // <无>
```

和"??"运算符配合使用的话，Empty运算符可以实现DefaultEmpty的功能。例如，有一个整型的不规则数组，如果想要通过一个查询将这个不规则数组中的所有元素放到一个一维数组中，首先应该想到SelectMany运算符，但是如果不规则数组中有某行为空null的话，SelectMany运算符就不能正常工作了，如下面代码所示：

```
int[][] numbers =
{
  new int[] { 1, 2, 3 },
  new int[] { 4, 5, 6 },
  null                       // 这个空行导致无法查询
};

IEnumerable<int> flat = numbers.SelectMany (innerArray => innerArray);
```

Empty运算符配合"??"运算符可以解决这个问题：

```
IEnumerable<int> flat = numbers
  .SelectMany (innerArray => innerArray ?? Enumerable.Empty <int>());

foreach (int i in flat)
  Console.Write (i + " "); // 1 2 3 4 5 6
```

9.13.2 Range和Repeat

Range和Repeat运算符只能使用在整型集合中。Range接收两个参数，分别用于指示起始元素的下标和查询元素的个数：

```
foreach (int i in Enumerable.Range (5, 3))
  Console.Write (i + " ");                     // 5 6 7
```

Repeat接收两个参数，第一个参数是要创建的元素，第二个参数用于指示重复元素的个数：

```
foreach (int i in Enumerable.Repeat (5, 3))
  Console.Write (i + " ");                     // 5 5 5
```

第10章 LINQ to XML

在.NET Framework中提供了很多用于处理XML数据的API。从.NET Framework 3.5之后，*LINQ to XML*成为处理通用XML文档的首选工具。它提供了一个轻量的集成了LINQ友好的XML文档对象模型，当然还有相应的查询运算符。在大多数情况下，它完全可以替代之前W3C标准的DOM模型（又称为XmlDocument）。

在本章中将概要地介绍LINQ to XML。在后面的章节中，我们会结合具体的示例再详细地介绍特定的XML类型和API。例如前向的reader/writer，用于操作schema、样式表和Xpaths类以及兼容的W3C标准的DOM等内容都会在不同的章节中结合具体的示例来讲解。

提示： LINQ to XML中DOM的设计非常完善且高效。即使没有LINQ，单纯的LINQ to XML中DOM对底层`XmlReader`和`XmlWriter`类也进行了很好的封装，可以通过它来更简单地使用这两个类中的方法。

LINQ to XML中所有的类型定义都包含在`System.Xml.Linq`命名空间中。

10.1 架构概述

本节简要地介绍DOM的概念，然后分析LINQ to XML中DOM的基本工作原理。

10.1.1 什么是DOM

请看下面这个XML文件：

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customer id="123" status="archived">
  <firstname>Joe</firstname>
  <lastname>Bloggs</lastname>
</customer>
```

与所有XML文件一样，在文件开始都是声明部分，然后是根元素，也就是`customer`。这个根元素有两个属性，这两个属性分别由两部分组成：属性名（`id`和`status`）和属性值（`"123"`和`"archived"`）。在`customer`节点下，有两个子元素即`firstname`和`lastname`，每个节点都包含简单的文本内容（`"Joe"`和`"Bloggs"`）。

声明、元素、属性、值和文本内容这些结构都可以用类来表示。如果这种类有很多属性来存储子内容，我们可以用一个对象树来完全描述文档。这个树状结构就是文档对象模型（Document Object Model），简称DOM。

10.1.2 LINQ to XML中的DOM

LINQ to XML由两部分组成：

- 一个XML DOM，我们称之为*X-DOM*
- 约10个用于查询的运算符

可以想象，X-DOM是由诸如Xdocument、Xelement、Xattribute等类组成的。有意思的是，X-DOM类并没有和LINQ绑定在一起，也就是说，即使不使用LINQ查询，也可以加载、更新或存储X-DOM。

事实上，使用LINQ可以查询所有符合较旧W3C标准的DOM对象。当然这种查询是不完善的，使用起来有很多限制。这就是X-DOM与传统DOM模型的区别所在，X-DOM是集成了LINQ的模型：

- X-DOM中的一些方法可以返回IEnumerable类型的集合，使LINQ查询变得非常方便。
- X-DOM的构造方法更加灵活，可以通过LINQ将数据直接映射成X-DOM树。

10.2 X-DOM概述

图10-1显示了X-DOM中的核心类型，这些类型中最常用的是XElement。XObject是整个继承结构的根，XElement和XDocument则是平行结构的根。图10-2显示的是下面这段代码所创建的X-DOM树：

```
string xml = @"<customer id='123' status='archived'>
                 <firstname>Joe</firstname>
                 <lastname>Bloggs<!--nice name--></lastname>
               </customer>";

XElement customer = XElement.Parse (xml);
```

XObject是所有X-DOM内容的抽象基类。在这个类型中定义了一个指向Parent元素的链接，这样就可以确定节点之间的层次关系。另外这个类中还有一个XDocument类型的对象可供使用。

除了属性之外，XNode是其他大部分X-DOM内容的基类。XNode的一个重要特性是它可以被有顺序地存放在一个混合类型的XNodes集合中。例如，下面的XML代码：

```
<data>
  Hello world
  <subelement1/>
  <!--comment-->
  <subelement2/>
</data>
```

在父元素<data>下面，第一个元素（Hello world）是XText类型的，之后是一个XElement类型的节点，接下来是一个XComment类型的节点，再往后又是一个XElement类型的节点。与此相对的是XAttribute对象的存储方式，多个XAttribute对象必须成对存放。

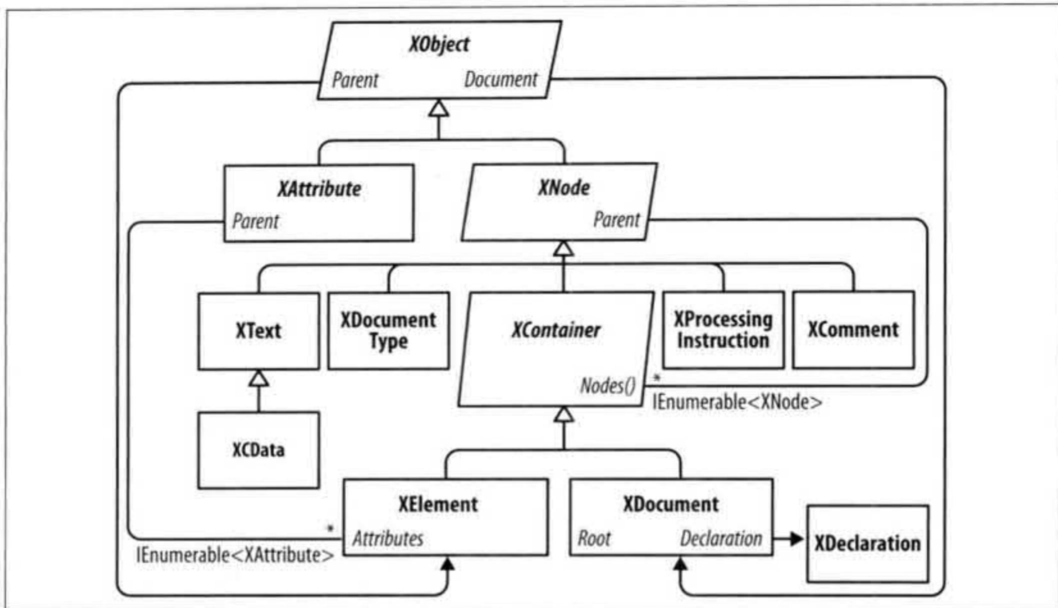


图10-1：X-DOM的核心类型

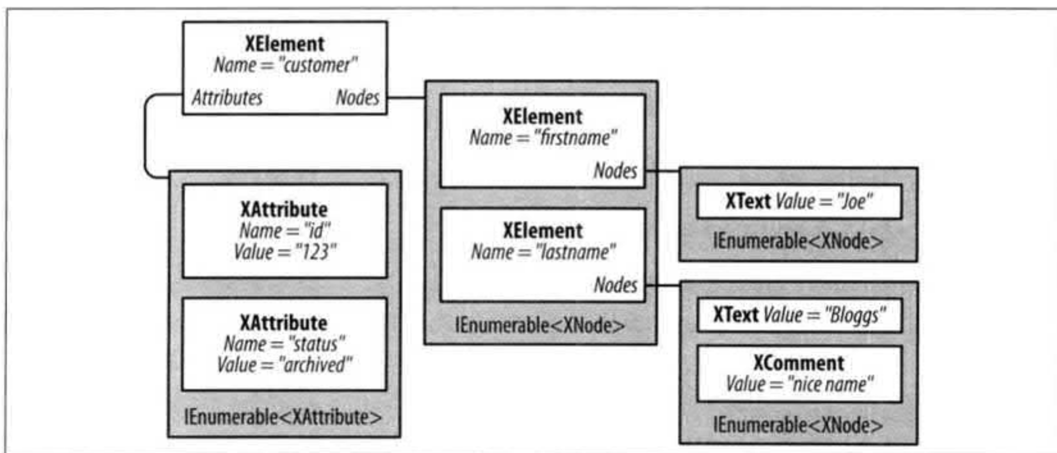


图10-2：一个简单的X-DOM树

虽然XNode可以访问它的父节点XElement，但是它却对自己的子节点一无所知，因为管理子节点的工作是由子类XContainer来做的。XContainer中定义了一系列成员和方法来管理它的子类，并且是XElement和XDocument的抽象基类。

除了Name和Value之外，XElement还定义了其他的成员来管理自己的属性，在绝大多数情况下，XElement会包含一个XText类型的子节点，XElement的Value属性同时包含了存取这个XText节点的get和set操作，这样可以更方便地设置节点值。由于Value属性的存在，我们可以不必直接使用XText对象，这使得对节点的赋值操作变得非常简单。

XML树的根节点是XDocument对象。更准确地说，它封装了根XElement，添加了XDelaration以及一些根节点需要执行的指令。与W3C标准的DOM有所不同，即使没有创建XDocument也可以加载、操作和保存X-DOM。这种对XDocument的不依赖性使得我们可以很容易将一个节点子树移到另一个X-DOM层次结构中。

10.2.1 加载和解析

XElement和XDocument都提供了静态Laod和Parse方法，使用这两个方法，开发者可以根据已有的数据创建X-DOM：

- Load可以根据文件、URI、Stream、TextReader或者XmlReader等构建X-DOM。
- Parse可以根据字符串构建X-DOM。

例如：

```
XDocument fromWeb = XDocument.Load ("http://albahari.com/sample.xml");

XElement fromFile = XElement.Load (@"e:\media\somefile.xml");

XElement config = XElement.Parse (
@"<configuration>
    <client enabled='true'>
      <timeout>30</timeout>
    </client>
  </configuration>");
```

在后面的各节中，我们会介绍如何遍历和更新X-DOM。为了使读者对此有个大概的印象，下面这段代码演示了如何操作上面刚生成的config元素：

```
foreach (XElement child in config.Elements())
  Console.WriteLine (child.Name);                         // 客户端

XElement client = config.Element ("client");

bool enabled = (bool) client.Attribute ("enabled");      // 读取属性
Console.WriteLine (enabled);                              // True
client.Attribute ("enabled").SetValue (!enabled);         // 更新属性

int timeout = (int) client.Element ("timeout");           // 读取元素
Console.WriteLine (timeout); // 30
client.Element ("timeout").SetValue (timeout * 2);        // 更新元素

client.Add (new XElement ("retries", 3));                 // 添加新元素

Console.WriteLine (config);          // 隐式调用config.ToString()
```

最后一句Console.WriteLine的输出结果如下：

```
<configuration>
  <client enabled="false">
    <timeout>60</timeout>
    <retries>3</retries>
  </client>
</configuration>
```

提示：XNode中还提供了一个静态ReadFrom方法，这个方法可以根据XmlReader中的节点来实例化和生成任何类型的XNode对象。与Load方法不同，ReadFrom读取一个完整的节点信息后就结束了，这样可以手动地控制创建节点的数量。

此外，还可以做相反的操作，使用XmlReader和XmlWriter类中的CreateReader和CreateWriter方法读写XNode中的信息。

我们会在第11章中介绍XmlReader和XmlWriter类的内容，以及如何在X-DOM中使用它们。

10.2.2 保存和序列化

在节点上调用ToString方法可将这个节点中的内容转换成XML字符串，默认情况下，转换后的XML字符串是经过格式化的，即使用换行和空格将XML字符串按层次结构逐行输出，且使用正确的缩进格式。如果不想让ToString方法格式化XML，那么可以指定SaveOptions.DisableFormatting参数。

XElement和XDocument还分别提供了Save方法，使用这个方法可将X-DOM写入文件、Stream、TextWriter或者XmlWriter中。如果选择将X-DOM写入到一个文件中，则会自动写入XML的声明部分。另外，XNode类还提供了一个WriteTo方法，这个方法只能向XmlWriter中写入数据。

在本章后面“文档和声明”一节中介绍使用Save方法时，会看到如何声明XML。

10.3 实例化X-DOM

创建X-DOM树常用的方法是手动实例化多个节点，然后通过XContainer的Add方法将所有节点拼装成XML树，而不是通过Load或者Parse方法。

要构建XElement和XAttribute，只需提供属性名和属性值：

```
XElement lastName = new XElement ("lastname", "Bloggs");
lastName.Add (new XComment ("nice name"));

XElement customer = new XElement ("customer");
customer.Add (new XAttribute ("id", 123));
customer.Add (new XElement ("firstname", "Joe"));
customer.Add (lastName);

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

上面这段代码的输出结果如下：

```
<customer id="123">
  <firstname>Joe</firstname>
  <lastname>Bloggs<!--nice name--></lastname>
</customer>
```

构建XElement时，属性值不是必须的，可以只提供一个元素名并在其后添加内容。注意，当需要为一个对象添加属性值时，只需设置一个字符串即可，不用显式创建并添加XText子节点，X-DOM的内部机制会自动完成这个操作，这使得活加属性值变得更加容易。

函数型构建

在前面的示例中，很难通过节点组成XML结构。X-DOM还支持另一种实例化方式：函数型构建。（源于函数式编程）。使用函数型构建的方式，可以用一个表达式构建整个树：

```
XElement customer =
  new XElement ("customer", new XAttribute ("id", 123),
    new XElement ("firstname", "joe"),
    new XElement ("lastname", "bloggs",
      new XComment ("nice name")
    )
  );
```

这种构建方式有两个优点。第一，代码可以体现出XML的结构；第二，这种表达式可以包含在LINQ查询的select子句中。例如，下面这段LINQ to SQL的查询将结果集直接映射成X-DOM格式：

```
XElement query =
  new XElement ("customers",
    from c in dataContext.Customers
    select
      new XElement ("customer", new XAttribute ("id", c.ID),
        new XElement ("firstname", c.FirstName),
          new XElement ("lastname", c.LastName,
            new XComment ("nice name")
          )
      )
  );
```

关于映射X-DOM的更多内容，在“将数据映射到X-DOM”一节中会有详细介绍。

10.4 指定内容

之所以以函数型构建的方式定义XML文件，是因为XElement（和XDocument）的构造方法都可重载，以接受params对象数组：

```
public XElement (XName name, params object[] content)
```

XContainer类的Add方法同样也接收这种类型的参数：

```
public void Add (params object[] content)
```

所以，我们可以在构建或添加X-DOM时指定任意数目、任意类型的子对象。这是因为任何内容都是合法的。那么在这些类的内部是如何处理每个对象呢？下面是XContainer类内部的解析方式：

（1）如果传入的对象是null，那么就忽略这个节点。

（2）如果传入对象是以XNode或者XStreamingElement作为基类，那么就将这个对象添加为Node对象，放到Nodes集合中。

（3）如果传入对象是XAttribute，那么就将这个对象作为Attribute集合来处理。

（4）如果对象是string，那么这个对象会被封装成一个XText节点，然后添加到Nodes集合中（注1）。

（5）如果对象实现了IEnumerable接口，则对其进行枚举，每个元素都按照上面的规则来处理。

（6）如果某个类型不符合上述任一条件，那么这个对象会被转换成string，然后被封装在XText节点中，并添加到Nodes集合中（注2）。

上述所有情况最终都是：nodes或Attributes。另外，所有对象都是有效的，因为最终肯定可以调用它的ToString方法并将其作为Xtext节点来处理。

提示： 在任意类型上调用ToString方法时，XContainer都会首先检查其类型，如果调用下面这些数据类型：

```
float、 double、 decimal、 bool、
DateTime、 DateTimeOffset、 TimeSpan
```

那么XContainer就调用XmlConvert类中合适类型的ToString方法，而不是直接调用这些对象自己的ToString方法。这样做的好处是确保数据是round-trippable并符合XML的格式规则的要求。

自动深层克隆

需要将一个节点或者属性添加给元素时（无论是函数型构建还是Add方法），新节点或属性的Parent属性都会被设置成该元素。一个节点只能拥有一个父节点；如果将一个已经拥有父节点的节点加入到第二个父节点中，该节点将会被自动深度克隆。下面这段代码中的每个customer都有一个单独的address副本：

```
var address = new XElement ("address",
                  new XElement ("street", "Lawley St"),
                  new XElement ("town", "North Beach")
              );
var customer1 = new XElement ("customer1", address);
var customer2 = new XElement ("customer2", address);

customer1.Element ("address").Element ("street").Value = "Another St";
Console.WriteLine (
  customer2.Element ("address").Element ("street").Value);    // Lawley St
```

这个自动复制操作对X-DOM对象实例化没有任何副作用，这也是函数式编程的一个标记。

10.5 导航和查询

你可能已经想到，XNode和XContainer类中肯定提供了方法和属性来遍历X-DOM树。与传统的DOM不同，这些函数并不会返回实现了IList<T>的集合，而是返回一个值或者一个实现了IEnumerable<T>的序列，取决于是执行一个LINQ查询还是使用foreach进行枚举。这同时也可以使用熟悉的LINQ查询语法来执行各种高级查询和简单的导航任务。

注1：实际上，X-DOM内部在处理string类型的对象时，会自动执行一些优化操作，也就是简单地将文本内容存放在字符串中。直到在XContainer上调用Nodes方法时，才会生成实际的XText节点。

注2：参见注1。

提示：与在XML中一样，X-DOM中的元素和属性名是区分大小写的。

10.5.1 子节点导航

返回值类型	成员	类
XNode	FirstNode { get; }	XContainer
	LastNode { get; }	XContainer
IEnumerable<XNode>	Nodes()	XContainer*
	DescendantNodes()	XContainer*
	DescendantNodesAndSelf()	XElement*
XElement	Element (XName)	XContainer
IEnumerable<XElement>	Elements()	XContainer*
	Elements (XName)	XContainer*
	Descendants()	XContainer*
	Descendants (XName)	XContainer*
	DescendantsAndSelf()	XElement*
	DescendantsAndSelf (XName)	XElement*
bool	HasElements { get; }	XElement

提示：上表的第三列及本章其他表中标注了星号的函数也可在同类型的序列上操作。例如，可以在XContainer或XContainer对象的序列上调用Nodes。之所以有这种功能，是因为System.Xml.Linq中定义了一些额外的扩展方法，这些扩展方法就是我们在概述中提到的那些额外的查询运算符。

1. FirstNode、LastNode和Nodes

使用FirstNode与LastNode可以直接访问第一个或最后一个子节点；Nodes返回所有的子节点并形成一个序列。这三个函数只用于直系的子节点。例如：

```
var bench = new XElement ("bench",
              new XElement ("toolbox",
                new XElement ("handtool", "Hammer"),
                new XElement ("handtool", "Rasp")
              ),
              new XElement ("toolbox",
                new XElement ("handtool", "Saw"),
                new XElement ("powertool", "Nail gun")
              ),
              new XComment ("Be careful with the nail gun")
            );
foreach (XNode node in bench.Nodes())
  Console.WriteLine (node.ToString (SaveOptions.DisableFormatting) + ".");
```

输出结果如下：

```
<toolbox><handtool>Hammer</handtool><handtool>Rasp</handtool></toolbox>.
<toolbox><handtool>Saw</handtool><powertool>Nailgun</powertool></toolbox>.
<!--Be careful with the nailgun-->.
```


2. 检索元素

Elements方法返回类型为XElement的子节点：

```
foreach (XElement e in bench.Elements())
Console.WriteLine (e.Name + "=" + e.Value);  // toolbox=HammerRasp
                                              // toolbox=SawNailgun
```

下面这个LINQ查询可以得到填写了“Nailgun”的工具框：

```
IEnumerable<string> query =
  from toolbox in bench.Elements()
  where toolbox.Elements().Any (tool => tool.Value == "Nailgun")
  select toolbox.Value;
```

查询结果：{ "SawNailgun" }

下面这个示例使用SelectMany查询得到所有名字是“handtool”的工具框中的值：

```
IEnumerable<string> query =
  from toolbox in bench.Elements()
  from tool in toolbox.Elements()
  where tool.Name == "handtool"
  select tool.Value;
```

查询结果：{ "Hammer", "Rasp", "Saw" }

提示： Elements方法实际上是对Nodes方法的返回值又进行了一次LINQ查询，前面使用了Elements方法的示例可以用下面这种方式开始：

```
from toolbox in bench.Nodes().OfType<XElement>()
where ...
```

Elements方法还可以只返回指定名字的元素。例如：

```
int x = bench.Elements ("toolbox").Count();     // 2
```

这等价于：

```
int x = bench.Elements().Where (e => e.Name == "toolbox").Count();     // 2
```

Elements还同时被定义成了扩展方法，可以接收IEnumerable<XContainer>类型或者更精确地说是：

```
IEnumerable<T> where T : XContainer
```

这使得Elements方法可以处理元素序列。使用该方法，我们可以重写上面从工具框中取值的查询：

```
from tool in bench.Elements ("toolbox").Elements ("handtool")
select tool.Value.ToUpper();
```

在上面的查询中，第一个Elements绑定到XContainer的实例方法，而第二个Elements方法则绑定到扩展方法。

3. 获取单个元素

Element方法返回匹配给定名称的第一个元素。Element对于简单的导航是非常有用的，例如：

```
XElement settings = XElement.Load ("databaseSettings.xml");
string cx = settings.Element ("database").Element ("connectString").Value;
```

Element的作用相当于调用Elements()，然后再应用LINQ的FirstOrDefault查询运算符给定一个名称作为匹配断言。如果没有找到所请求的元素，则Element返回null。

提示： 如果元素xyz不存在，那么Element("xyz").Value将会抛出一个NullReferenceException异常。如果倾向于使用null代替异常，可以将XElement转换成string而不是查询它的Value属性，如下：

```
string xyz = (string) settings.Element ("xyz");
```

可以这样做的原因是XElement专门为上面这种情况定义了显式string转换。

---

4. 递归功能

XContainer还定义了Descendants和DescendantNodes方法，它们递归地返回子元素或子节点。Descendant接受一个可选的元素名。在之前的例子中，我们可以使用Descendants查找所有的handtool：

```
Console.WriteLine (bench.Descendants ("handtool").Count()); // 3
```

不管是父节点还是叶节点都包括在内，如下面的例子所示：

```
foreach (XNode node in bench.DescendantNodes())
  Console.WriteLine (node.ToString (SaveOptions.DisableFormatting));

<toolbox><handtool>Hammer</handtool><handtool>Rasp</handtool></toolbox>
<handtool>Hammer</handtool>
Hammer
<handtool>Rasp</handtool>
Rasp
<toolbox><handtool>Saw</handtool><powertool>Nailgun</powertool></toolbox>
<handtool>Saw</handtool>
Saw
<powertool>Nailgun</powertool>
Nailgun
<!--Be careful with the nailgun-->
```

下面这个查询会得到X-DOM中所有包含了“careful”的内容：

```
IEnumerable<string> query =
  from c in bench.DescendantNodes().OfType<XComment>()
  where c.Value.Contains ("careful")
  orderby c.Value
  select c.Value;
```

10.5.2 查询父节点

所有的XNodes都包含一个Parent属性，另外还有一个AncestorXXX方法用来找到特定的父节点。一个父节点永远是一个XElement。

返回值类型	成员	类
XElement	Parent { get; }	XNode*
Enumerable<XElement>	Ancestors()	XNode*
	Ancestors (XName)	XNode*
	AncestorsAndSelf()	XElement*
	AncestorsAndSelf (XName)	XElement*

如果x是一个XElement，那么下面这段代码总是为true：

```
foreach (XNode child in x.Nodes())
  Console.WriteLine (child.Parent == x);
```

如果x是一个XDocument，则它可以有子节点，但永远不能作为任何节点的父节点。为了访问XDocument，应该使用Document属性，这对X-DOM树上的任意对象都适用。

Ancestors返回一个序列，其第一个元素是Parent，下一个元素则是Parent.Parent，依次类推，直到根元素。

提示： 还可以使用LINQ查询AncestorsAndSelf().Last()来取得根元素。

另外一种方法是调用Document.Root，但只有存在XDocument时才能执行。

10.5.3 对等节点的导航

返回值类型	成员	类
bool	IsBefore(XNdoe node)	XNode
	IsAfter(XNode node)	XNode
XNode	PreviousNode { get; }	XNode
	NextNode { get; }	XNode
IEnumerable<XNode>	NodesBeforeSelf()	XNode
	NodesAfterSelf()	XNode
IEnumerable<XElement>	ElementsBeforeSelf()	XNode
	ElementsBeforeSelf (XName 名称)	XNode
	ElementsAfterSelf()	XNode
	ElementsAfterSelf (XName 名称)	XNode

使用PreviousNode和NextNode（以及FirstNode/LastNode）方法查找节点时，相当于从一个链表中遍历所有节点。事实上XML中节点的存储结构确实是链表。

警告：XNode存储在一个单向链表中，所以PreviousNode并不是当前元素的前序元素。

10.5.4 属性导航

返回值类型	成员	类
bool	HasAttributes { get; }	XElement
XAttribute	Attribute (XName 名称)	XElement
	irstAttribute { get; }	XElement
	LastAttribute { get; }	XElement
IEnumerable<XAttribute>	Attributes()X	Element
	Attributes (XName 名称)	XElement

除了Parent属性之外，XAttribute中还定义了PreviousAttribute和NextAttribute属性。

Attributes方法接受一个名称并返回包含0或1个元素的序列；在XML中，元素不能包含重复的属性名。

10.6 更新X-DOM

可以使用下面这几种方式来更新XML中的元素和属性：

- 调用SetValue方法或者重新给Value属性赋值。
- 调用SetElementValue或SetAttributeValue方法。
- 调用某个RemoveXXX方法。
- 调用某个AddXXX或ReplaceXXX方法指定更新的内容。

也可以为XElement对象重新设置Name属性。

10.6.1 简单的值更新

成员	类
SetValue(Object value)	Xelement，XAttribute
Value { get; set }	Xelement，XAttribute

使用SetValue方法可以使用简单的值替换元素或者属性中原来的值。通过Value属性赋值会达到相同的效果，但只能使用string类型的数据。我们将会在本章“使用Value”一节中更详细地介绍这两种更新方式。

调用SetValue方法（或者为Value重新赋值）的结果就是它替换了所有的子节点：

```
XElement settings = new XElement ("settings",
                      new XElement ("timeout", 30)
                    );
settings.SetValue ("blah");
Console.WriteLine (settings.ToString()); // <settings>blah</settings>
```

10.6.2 更新子节点和属性

分类	成员	类
添加	Add (params object[] content)	XContainer
	AddFirst (params object[] content)	XContainer
移除	RemoveNodes()	XContainer
	RemoveAttributes()	XElement
	RemoveAll()	XElement
更新	ReplaceNodes (params object[] content)	XContainer
	ReplaceAttributes (params object[] content)	XElement
	ReplaceAll (params object[] content	XElement
	SetElementValue (XName name, object value)	XElement
	SetAttributeValue (XName name, object value)	XElement

在上表列出的各种方法中，最好的两个方法是：SetElementValue和SetAttributeValue。它们提供了一种非常便捷的方式来实例化XElement或XAttribute对象，然后调用父节点的Add方法，将新节点加入到父节点下面，从而替换相同名称的任何现有元素或属性：

```
XElement settings = new XElement ("settings");
settings.SetElementValue ("timeout", 30);    // 添加子节点
settings.SetElementValue ("timeout", 60);    // 将新节点的值更新为60
```

Add方法将一个子节点添加到一个元素或文档中。AddFirst也一样，但它将节点插入集合的开头而不是结尾。

我们也可以通过调用RemoveNodes或RemoveAttributes将所有的子节点或属性全部删除。RemoveAll相当于同时调用了这两个方法。

Replace*XXX*方法等价于调用Removing，然后再调用Adding。它们拥有输入参数的快照，因此e.ReplaceNodes(e.Nodes)可以正常执行。

10.6.3 通过父节点更新子节点

成员	类
AddBeforeSelf (params object[] content)	XNode
AddAfterSelf (params object[] content)	XNode
Remove()	XNode*、XAttribute*
ReplaceWith (params object[] content)	XNode

上表中的AddBeforeSelf、AddAfterSelf、Remove和ReplaceWith方法不能操作一个节点的子节点。它们只能操作当前节点所在的集合。这就要求当前节点都有父元素，否则在使用这些方法时就会抛出异常。此时AddBeforeSelf和AddAfterSelf方法非常有用，这两个方法可以将一个新节点插入到XML中的任意位置：

```
XElement items = new XElement ("items",
                   new XElement ("one"),
                   new XElement ("three")
                 );
items.FirstNode.AddAfterSelf (new XElement ("two"));
```

上面代码的输出结果如下：

```
<items><one /><two /><three /></items>
```

将很多元素组成的集合整体插入到任意位置的效率是很高的，因为在内部所有节点都存储在链表中。

Remove方法可以将当前节点从它的父节点中移除。ReplaceWith方法实现同样的操作，只是它在移除旧节点之后还会在同一位置插入其他内容。例如：

```
XElement items = XElement.Parse ("<items><one/><two/><three/></items>");
items.FirstNode.ReplaceWith (new XComment ("One was here"));
```

执行结果如下：

```
<items><!--one was here--><two /><three /></items>
```

整组地移除节点或者属性

通过System.Xml.Linq中的扩展方法，我们可以使用Remove方法整组地移除节点或者属性，首先看下面这段X-DOM的代码：

```
XElement contacts = XElement.Parse (
@"<contacts>
    <customer name='Mary'/>
    <customer name='Chris' archived='true'/>
    <supplier name='Susan'>
      <phone archived='true'>012345678<!--confidential--></phone>
    </supplier>
</contacts>");
```

下面这行代码可以移除所有的customer节点：

```
contacts.Elements ("customer").Remove();
```

下面这行代码可以移除所有已归档（archived）的联系人（contact）：

```
contacts.Elements().Where (e => (bool?) e.Attribute ("archived") == true)
                   .Remove();
```

如果使用Descendants()方法代替上面的Elements()方法，可以移除DOM中所有已归档的元素，结果是：

```
<contacts>
  <customer name="Mary" />
  <supplier name="Susan" />
</contacts>
```

下面这个示例会移除在树节点的任何地方包含“confidential”这个词的联系人：

```
contacts.Elements().Where (e => e.DescendantNodes()
```

```
                         .OfType<XComment>()
                         .Any (c => c.Value == "confidential")
                  ).Remove();
```

移除之后的结果如下：

```
<contacts>
  <customer name="Mary" />
  <customer name="Chris" archived="true" />
</contacts>
```

使用下面这个简单查询，可以移除整个树中的所有注释节点：

```
contacts.DescendantNodes().OfType<XComment>().Remove();
```

提示：Remove方法的内部实现机制是这样的：首先将所有匹配的元素读取到一个临时列表中，然后枚举该临时列表并执行删除操作。这避免了在删除的同时进行查询操作所引起的错误。

10.7 使用Value

XElement和XAttribute都有一个string类型的Value属性。如果一个元素有XText类型的子节点，那么XElement的Value属性就相当于访问此节点的快捷方式，对于XAttribute的Value属性就是指属性值。

尽管元素属性的存储方式不同，但是X-DOM还是提供了一致的操作方式用于操作元素和属性值。

10.7.1 设置Value

有两种方式可以设置Value属性值：调用SetValue方法或者直接给Value属性赋值。SetValue方法要复杂一些，因为它不仅可以接收string类型的参数，也可以设置其他简单的数据类型：

```
var e = new XElement ("date", DateTime.Now);
e.SetValue (DateTime.Now.AddDays(1));
Console.Write (e.Value); // 2007-03-02T16:39:10.734375+09:00
```

我们可以简单地设置元素的Value属性，但需要手动将DateTime转换成string类型。这比调用ToString更复杂，因为它需要使用XmlConvert来转换成一个XML兼容的结果。

向XElement或XAttribute的构造方法中传递值时，如果传递的值不是string类型的，那么也会自动进行相应的转换。这使得DateTime这种类型可以被格式化成XML能识别的格式；参数true以小写形式存储，参数double.NegativeInfinity可以指示参数以“-INF”的形式存储。

10.7.2 获取Value

为了将一个Value值转换成为其基础类型，我们可以简单地将XElement或XAttribute转换为我们期望的类型。这听起来似乎是不能执行的，但实际上它完全没有任何问题。例如：

```
XElement e = new XElement ("now", DateTime.Now);
DateTime dt = (DateTime) e;

XAttribute a = new XAttribute ("resolution", 1.234);
```

```
double res = (double) a;
```

XML中的元素或者属性不存储DateTime或数字，不论什么类型的数据，最终都是按照文本来存储的。它没有记住其原始类型，因为必须要将其转换为正确的类型，以免出现运行时错误。要让代码更加健壮，可以在try/catch块中执行类型转换，捕获FormatException异常。

通过显式的类型转换，可以将XElement和XAttribute解析成下面这些类型：

- 所有的标准数值类型
- string、bool、DateTime、DateTimeOffset、TimeSpan和Guid
- 上面所有类型的Nullable<>版本

如果请求的名称不存在，再结合使用Element和Attribute方法转换到可空类型，并且其转换应该可以顺利完成。例如，如果x没有timeout元素，第一行将会生成一个运行时错误，而第二行则不会：

```
int timeout = (int) x.Element ("timeout"); // 错误
int? timeout = (int?) x.Element ("timeout"); // 正确，timeout是null
```

我们可以使用??运算符去除最后结果中的可空类型。以下代码在resolution属性不存在的情况下将会返回1.0：

```
double resolution = (double?) x.Attribute ("resolution") ?? 1.0;
```

不过，如果元素或属性存在并且包含一个空值（或者格式不正确），转换到可空类型并不能顺利实现，此时会抛出一个FormatException异常。

我们也可以在LINQ查询中执行类型转换，如下面这个查询返回“John”：

```
var data = XElement.Parse (
  @"<data>
      <customer id='1' name='Mary' credit='100' />
      <customer id='2' name='John' credit='150' />
      <customer id='3' name='Anne' />
   </data>");

IEnumerable<string> query = from cust in data.Elements()
                            where (int?) cust.Attribute ("credit") > 100
                            select cust.Attribute ("name").Value;
```

由于Anne没有credit属性，将cust.Attribute ("credit")转换成可空的int型可以避免出现NullReferenceException异常。另一种解决方案是将一个断言添加到where子句中：

```
where cust.Attributes ("credit").Any() && (int) cust.Attribute...
```

上面这些原则同样适用于对元素值的查询。

10.7.3 值与混合内容的节点

由于有了Value的值，你可能会好奇什么时候才需要直接和XText节点打交道呢？答案是：当拥有混合内容时。例如：

```
<summary>An XAttribute is <bold>not</bold> an XNode</summary>
```

一个简单的Value属性并不能获取summary中的所有内容。Summary元素包含三项内容：一个XText类型的节点，之后是一个XElement类型的节点，最后又是一个XText类型的节点。下面是这三者之间的结构：

```
XElement summary = new XElement ("summary",
                     new XText ("An XAttribute is "),
                     new XElement ("bold", "not"),
                     new XText (" an XNode")
);
```

有意思的是，我们仍可以查询summary的Value，这不会出现异常，而且可以得到以下各个子节点值连接在一起的形式：

```
An XAttribute is not an XNode
```

为summary的Value属性重新赋值也是合法的，这个赋值操作相当于使用一个新的XText节点替换原来的所有子节点。

10.7.4 自动连接XText节点

向XElement添加简单的内容时，X-DOM会将新添加的内容附加到现有的XText节点后面，而不会新建一个XText节点。在下面这个示例中，e1和e2最后形成了一个Xtext子元素，值是HelloWorld：

```
var e1 = new XElement ("test", "Hello"); e1.Add ("World");
var e2 = new XElement ("test", "Hello", "World");
```

如果显式地指定创建新的XText节点，最终会得到多个子节点：

```
var e = new XElement ("test", new XText ("Hello"), new XText ("World"));
Console.WriteLine (e.Value);              // HelloWorld
Console.WriteLine (e.Nodes().Count());    // 2
```

XElement不会连接这两个XText节点，所以节点的对象标识被保留。

10.8 文档和声明

10.8.1 XDocument

如前所述，XDocument封装了根节点XElement，可以添加XDeclaration、处理指令、说明文档类型以及根级别的注释。XDocument是可选的，并且能够被忽略或者省略，这点与W3C DOM不同。

XDocument提供了和XElement相同的构造方法。另外由于它也继承了XContainer类，所以也支持Add*XXX*、Remove*XXX*和Replace*XXX*等方法。但与XElement不同，一个XDocument节点可添加的内容是有限的：

- 一个XElement对象（根节点）
- 一个XDeclaration对象
- 一个XDocumentType对象（引用一个DTD）
- 任意数目的XProcessingInstruction对象

- 任意数目的XComment对象

提示： 在上面提到的这些对象中，对于XDocument来说，只有根XElement对象是必须的。XDeclaration是可选的，如果省略，在序列化的过程中会应用默认设置。

最简单的合法XDocument只包含一个根元素：

```
var doc = new XDocument (
            new XElement ("test", "data")
          );
```

注意，这里并没有包括XDeclaration对象。但是调用doc.Save方法生成的文件中仍会包含XML声明，因为这是默认生成的。

下面这段代码创建了一个简单但正确的XHTML文件，展示了XDocument可以接受的所有结构：

```
var styleInstruction = new XProcessingInstruction (
  "xml-stylesheet", "href='styles.css' type='text/css'");

var docType = new XDocumentType ("html",
  "-//W3C//DTD XHTML 1.0 Strict//EN",
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd", null);

XNamespace ns = "http://www.w3.org/1999/xhtml";
var root =
  new XElement (ns + "html",
    new XElement (ns + "head",
      new XElement (ns + "title", "An XHTML page")),
    new XElement (ns + "body",
      new XElement (ns + "p", "This is the content"))
  );

var doc =
  new XDocument (
  new XDeclaration ("1.0", "utf-8", "no"),
  new XComment ("Reference a stylesheet"),
  styleInstruction,
  docType,
  root);

doc.Save ("test.html");
```

所生成的*test.html*的内容如下：

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!--Reference a stylesheet-->
<?xml-stylesheet href='styles.css' type='text/css'?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
                      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>An XHTML page</title>
  </head>
  <body>
    <p>This is the content</p>
  </body>
```

```
</html>
```

XDocument有一个Root属性，这个属性是取得当前XDocument对象单个XElement的快捷方式。其反向的链接是由XObject的Document属性提供的，并且可以应用于树中的所有对象：

```
Console.WriteLine (doc.Root.Name.LocalName);        // html
XElement bodyNode = doc.Root.Element (ns + "body");
Console.WriteLine (bodyNode.Document == doc);       // True
```

前面提到过，Document对象的子节点是没有Parent信息的：

```
Console.WriteLine (doc.Root.Parent == null);        // True
foreach (XNode node in doc.Nodes())
Console.Write (node.Parent == null);                // TrueTrueTrueTrue
```

提示：XDeclaration并不是XNode类型的，因此它不会出现在文档的Nodes集合中，而注释、处理指令和根元素等都会出现在Nodes集合中。XDeclaration对象专门存放在一个Declaration属性中。这就是为什么上面那段代码的最后一行只输出四个True，而不是五个。

10.8.2 XML声明

一个标准的XML文件由下面这样的声明开始：

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
```

XML声明是为了保证整个文件被XML阅读器正确解析并理解。XElement和XDocument都遵循下面这些XML声明的规则：

- 在一个文件名上调用Save方法时，总是自动写入XML声明。
- 在XmlWriter对象上调用Save方法时，除非XmlWriter特别指出，否则都会写入XML声明。
- ToString方法从来都不返回XML声明。

提示：如果不想让XmlWriter创建XML声明，可以在构建XmlWriter对象时，通过设置XmlWriterSettings对象的OmitXmlDeclaration和ConformanceLevel属性来实现。我们会在第11章中介绍该内容。

是否有XDeclaration对象对是否写入XML声明没有任何影响。XDeclaration的目的是提示进行XML序列化进程，方式有两种：

- 使用的文本编码标准。
- 定义XML声明中encoding和standalone两个属性的值（如果写入声明）。

XDeclaration的构造方法接受三个参数，分别用于设置version、encoding和standalone属性。下面的示例中*test.xml*使用UTF-16对XML文件进行编码：

```
var doc = new XDocument (
            new XDeclaration ("1.0", "utf-16", "yes"),
            new XElement ("test", "data")
          );
```

```
doc.Save ("test.xml");
```

提示：XML编写器会忽略所指定的XML版本信息，始终写入“1.0”。

需要注意的是，XML声明中指定的必须是诸如“utf-16”这样的IETF编码方式。

将XML声明输出为字符串

假如想要将XDocument中的内容以及XML的声明输出为一个字符串，应该怎么办呢？由于ToString方法不能直接输出声明中的内容，因此需要使用XmlWriter来实现：

```
var doc = new XDocument (
            new XDeclaration ("1.0", "utf-8", "yes"),
            new XElement ("test", "data")
          );
var output = new StringBuilder();
var settings = new XmlWriterSettings { Indent = true };
using (XmlWriter xw = XmlWriter.Create (output, settings))
  doc.Save (xw);
Console.WriteLine (output.ToString());
```

下面是输出结果：

```
<?xml version="1.0" encoding="utf-16" standalone="yes"?>
<test>data</test>
```

注意上面的代码，在定义XDocument的声明时指定的编码方式是UTF-8，但在输出的声明中的编码方式却是UTF-16。这似乎是一个bug，但实际上不是，而是XmlWriter自动纠正了一些内在的错误。由于我们想要最终将XML中的内容以一个string的形式输出，而不是文件或流，所以无法应用除UTF-16以外的任何编码方式，只有这种格式才能在内部存储字符串。因此XmlWriter最后在输出时选择了UTF-16，因为它使用的确实是这个编码方式。

这也可以解释为什么ToString方法不能输出XML声明了。假如不使用Save方法，而是使用下面的方法将XDocument的内容写入一个文件：

```
File.WriteAllText ("data.xml", doc.ToString());
```

按照上面这种写法，*data.xml*会缺少XML声明，虽然不完整，但仍然能够被解析（可以推断出文本编码）。但是如果ToString()方法也同样输出声明中的部分，*data.xml*中实际上包含一个错误的声明（encoding="utf-16"），这将导致整个XML文件不能被正确读取，因为WriteAllText使用UTF-8进行编码。

10.9 名称和命名空间

我们都知道，.NET类型都有自己的命名空间，实际上XML元素和属性也是如此。

XML命名空间有两个功能。首先，与C#的命名空间一样，它们可以帮助避免命名冲突。当要合并来自两个不同XML文件的数据时，这可能会成为一个问题。其次，命名空间赋予了名称一个绝对的意义。例如名字“nil”可以有各种各样的意义。但这个名字在*http://www.w3.org/2001/xmlschema-instance*命名空间下，所表达的意思是某个元素是否等于C#中的null，当然这种比较还有其他的规则。

由于XML中的命名空间特别容易混淆，这里先概要地介绍一下它的概念，在后面的章节中再结合示例介绍如何在LINQ to XML中使用它。

LINQ to XML

10.9.1 XML中的命名空间

假如想在OReilly.Nutshell.CSharp命名空间下定义一个customer元素。有两种实现方式，第一种是使用xmlns属性，代码如下：

```
<customer xmlns="OReilly.Nutshell.CSharp"/>
```

xmlns是一个特殊的保留属性，以上用法使它执行下面两种功能：

- 它为有疑问的元素指定了一个命名空间。
- 它为所有后代元素指定了一个默认的命名空间。

这意味着在下面的示例中，address和postcode节点也同样在OReilly.Nutshell.CSharp命名空间下：

```
<customer xmlns="OReilly.Nutshell.CSharp">
  <address>
    <postcode>02138</postcode>
  </address>
</customer>
```

如果不想让address和postcode继承父节点的命名空间，我们可以像下面这样设置：

```
<customer xmlns="OReilly.Nutshell.CSharp"s>
  <address xmlns="">
    <postcode>02138</postcode> <!-- postcode now inherits empty ns -->
  </address>
</customer>
```

1. 前缀

还可以使用前缀来指定命名空间。前缀是一个命名空间的别名，使用它指定命名空间时可以输入较少的字符。使用前缀有两个步骤：定义前缀和使用前缀。可以用下面这种方式同时完成这两个步骤：

```
<nut:customer xmlns:nut="OReilly.Nutshell.CSharp"/>
```

此处完成了两个不同的操作。在右侧，xmlns:nut="..."定义了名为nut的前缀，这个nut可用于该元素及其所有子元素。左边的nut:customer就是将新分配的前缀指定给customer元素。

有前缀的元素不会为它的后代元素定义默认的命名空间。在下面的XML中，firstname包含一个空的命名空间：

```
<nut:customer nut:xmlns="OReilly.Nutshell.CSharp">
  <firstname>Joe</firstname>
</customer>
```

要想将OReilly.NutShell.CSharp前缀赋予firstname，可以这样定义：

```
<nut:customer xmlns:nut="OReilly.Nutshell.CSharp">
  <nut:firstname>Joe</firstname>
</customer>
```

为了方便子元素的使用，还可以定义一个或多个元素，但这些前缀不应用于父元素。在下面的示例中定义了两个前缀i和z，但customer元素却没有命名空间：

```
<customer xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/">
  ...
</customer>
```

如果customer节点是根节点，那么整个文档的节点就都可以使用i和z了。需要对不同的节点使用不同的命名空间时，这种前缀的方式使用起来非常方便。

注意，上面用到的命名空间都是一些URI。使用URI（自定义的URI）作为命名空间是一种通用的做法，这可以有效地保证名称空间的唯一性。所以在现实的开发中，customer元素应该是这样的：

```
<customer xmlns="http://oreilly.com/schemas/nutshell/csharp"/>
```

或：

```
<nut:customer xmlns:nut="http://oreilly.com/schemas/nutshell/csharp"/>
```

2. 属性

也可以为属性指定命名空间。不同之处在于它总是要求必须有一个前缀，例如：

```
<customer xmlns:nut="OReilly.Nutshell.CSharp" nut:id="123" />
```

另一个不同之处在于，非限定属性总是包含一个空的命名空间：它永远不会从父元素继承一个默认的命名空间。

实际上对于属性来说，最好不使用命名空间，因为属性往往是对本地元素起作用的。那些通用属性或者元数据属性除外，例如前面提到的W3C中的nil属性：

```
<customer xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <firstname>Joe</firstname>
  <lastname xsi:nil="true"/>
</customer>
```

上面的nil属性明确指出lastname是nil类型的（C#中的null），但不可以是一个空字符串。由于我们使用标准的命名空间，因此通用的解析器都可以正确地解析。

10.9.2 在X-DOM中指定命名空间

本章到目前为止，使用的示例中都是用一个简单的字符串作为XElement和XAttribute的名字。一个简单的字符串对应一个具有空命名空间的XML名字，这就像在全局命名空间中定义的.NET类。

有多种方式可以指定XML命名空间。第一种方式是在本地名字前面使用大括号来指定：

```
var e = new XElement ("{http://domain.com/xmlspace}customer", "Bloggs");
Console.WriteLine (e.ToString());
```

下面是设置后的XML代码：

```
<customer xmlns="http://domain.com/xmlspace">Bloggs</customer>
```

第二种方式（也是更好的一种方式）是通过XNamespace和XName为XML设置命名空间。它们的定义方式如下：

```
public sealed class XNamespace
{
  public string NamespaceName { get; }
}

public sealed class XName // 使用可选命名空间的本地名称
{
  public string LocalName { get; }
  public XNamespace Namespace { get; } // 可选
}
```

两种类型都定义了从string到它们的隐式转换，所以下面这种操作方式也是合法的：

```
XNamespace ns = "http://domain.com/xmlspace";
XName localName = "customer";
XName fullName = "{http://domain.com/xmlspace}customer";
```

XName还重载了+运算符，这样无需使用大括号即可直接将命名空间和元素名组合在一起：

```
XNamespace ns = "http://domain.com/xmlspace";
XName fullName = ns + "customer";
Console.WriteLine (fullName); // {http://domain.com/xmlspace}customer
```

在X-DOM中有很多构造方法和方法都能接受元素名或者属性名作为参数，但它们实际上接受XName对象，而不是字符串。到目前为止我们都是在用字符串作参数，之所以可以这么用，是因为字符串可以被隐式转换成XName对象。

可以为属性或元素指定相同的命名空间：

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data",
             new XAttribute (ns + "id", 123)
           );
```

10.9.3 X-DOM和默认命名空间

除非需要输出XML，否则X-DOM会忽略默认命名空间的概念。这意味着，如果要构建子XElement，必须显式地指定命名空间，因为子元素不会从父元素继承命名空间：

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data",
             new XElement (ns + "customer", "Bloggs"),
             new XElement (ns + "purchase", "Bicycle")
           );
```

但是，当X-DOM读取和输入XML时会应用默认命名空间：

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

输出内容：

```
<data xmlns="http://domain.com/xmlspace">
  <customer>Bloggs</customer>
```

```
    <purchase>Bicycle</purchase>
  </data>
```

```
Console.WriteLine (data.Element (ns + "customer").ToString());
```

输出内容：

```
  <customer xmlns="http://domain.com/xmlspace">Bloggs</customer>
```

如果没有在构造XElement子元素时指定命名空间，如下面这段代码所示：

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data",
             new XElement ("customer", "Bloggs"),
             new XElement ("purchase", "Bicycle")
);
Console.WriteLine (data.ToString());
```

得到的结果如下：

```
<data xmlns="http://domain.com/xmlspace">
  <customer xmlns="">Bloggs</customer>
  <purchase xmlns="">Bicycle</purchase>
</data>
```

在使用命名空间时，一个很容易犯的错误是在查找XML的元素时没有指定它所属的命名空间：

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data",
             new XElement (ns + "customer", "Bloggs"),
             new XElement (ns + "purchase", "Bicycle")
           );
XElement x = data.Element (ns + "customer");// 查询正确
XElement y = data.Element ("customer"); // 得到null
```

如果在构建X-DOM树时没有指定命名空间，可以在随后的代码中为每个元素分配一个命名空间：

```
foreach (XElement e in data.DescendantsAndSelf())
  if (e.Name.Namespace == "")
    e.Name = ns + e.Name.LocalName;
```

10.9.4 前缀

X-DOM操作前缀就像操作命名空间一样：主要是为了实现序列化功能。这意味着选择完全忽略前缀也是可行的。唯一的理由是为了输出XML文件时更加高效。例如，考虑如下代码：

```
XNamespace ns1 = "http://domain.com/space1";
XNamespace ns2 = "http://domain.com/space2";

var mix = new XElement (ns1 + "data",
            new XElement (ns2 + "element", "value"),
            new XElement (ns2 + "element", "value"),
            new XElement (ns2 + "element", "value")
);
```

默认情况下，XElement会被序列化成下面这种形式：

```
<data xmlns="http://domain.com/space1">
  <element xmlns="http://domain.com/space2">value</element>
  <element xmlns="http://domain.com/space2">value</element>
  <element xmlns="http://domain.com/space2">value</element>
</data>
```

可以看到的，有一些不必要的重复代码。解决方案是：不需要更改X-DOM的构建方式，只需要在写XML之前提示序列化器，可通过添加需要的属性定义前缀来实现。这种操作一般在根元素中完成：

```
mix.SetAttributeValue (XNamespace.Xmlns + "ns1", ns1);
mix.SetAttributeValue (XNamespace.Xmlns + "ns2", ns2);
```

上面这两行代码会将前缀“ns1”赋给XNamespace变量ns1，“ns2”赋给ns2。序列化时X-DOM会自动使用这些属性去缩短所得到的XML。下面是在mix上调用ToString的结果：

```
<ns1:data xmlns:ns1="http://domain.com/space1"
          xmlns:ns2="http://domain.com/space2">
  <ns2:element>value</ns2:element>
  <ns2:element>value</ns2:element>
  <ns2:element>value</ns2:element>
</ns1:data>
```

前缀的引入并不会影响X-DOM的构建、查询、更新等操作，实际上，在X-DOM的这几种操作中，可以完全忽略前缀，直接使用完整的名字即可。只有当序列化到文件或者流（反之亦然）的时候才需要使用前缀。

前缀对于序列化属性同样有用。在下面的示例中，我们使用W3C标准属性将一个客户的生日和信用卡信息设置成nil。加粗的代码行确保序列化的时候不会有不必要的重复命名空间：

```
XNamespace xsi = "http://www.w3.org/2001/XMLSchema-instance";
var nil = new XAttribute (xsi + "nil", true);

var cust = new XElement ("customers",
             new XAttribute (XNamespace.Xmlns + "xsi", xsi),
             new XElement ("customer",
               new XElement ("lastname", "Bloggs"),
               new XElement ("dob", nil),
               new XElement ("credit", nil)
             )
           );
```

上面代码生成的XML如下：

```
<customers xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <customer>
    <lastname>Bloggs</lastname>
    <dob xsi:nil="true" />
    <credit xsi:nil="true" />
  </customer>
</customers>
```

为了让代码更简短，在前面的代码中我们预先定义了一个变量nil，它被用于XAttribute对象。在构建DOM中这个变量被使用了两次。可以使用两次相同的属性是因为它会像我们要求的那样自动复制。

10.10 注解

使用注解可以将任何自定义数据附加到XObject对象上。注解是为存放个人私有数据而设计的，X-DOM将其视为黑盒，内部的结构不可见。如果用过Windows Forms或WPF控件上的Tag属性，可能对注解这个对象不会觉得陌生，它们的不同之处在于，一个XObject可以有多个注解，而且可以将注解设置成私有的。可以创建其他数据类型根本看不到的注解，更不用说修改其中的数据了。

通过下面的方法可以向XObject上添加或移除注解：

```
public void AddAnnotation (object annotation)
public void RemoveAnnotations<T>() where T : class
```

下面这两个方法用于查找注解：

```
public T Annotation<T>() where T : class
public IEnumerable<T> Annotations<T>() where T : class
```

每个注解都使用它的数据类型作为键，而且这个键必须是引用类型。下面这段代码首先添加一个注解，然后又检索这个String类型的注解：

```
XElement e = new XElement ("test");
e.AddAnnotation ("Hello");
Console.WriteLine (e.Annotation<string>());   // Hello
```

我们还可以添加多个相同类型的注解，然后调用Annotations方法得到所有符合条件的注解对象。

使用string这种功能类型作为键不是一个很好的选择，因为其他类中的代码会干扰注解。一个比较好的做法是使用内部或嵌套的私有类：

```
class X
{
  class CustomData { internal string Message; }    // 私有嵌套类

  static void Test()
  {
    XElement e = new XElement ("test");
    e.AddAnnotation (new CustomData { Message = "Hello" } );
    Console.Write (e.Annotations<CustomData>().First().Message);    // Hello
  }
}
```

要移除一个注解，也必须通过键找到它：

```
e.RemoveAnnotations<CustomData>();
```

10.11 将数据映射到X-DOM

我们讨论过如何使用LINQ从X-DOM中查询数据。实际上，使用LINQ还可以将数据直接映射到X-DOM中。只要映射的数据源支持LINQ查询，都可以进行这种操作，如：

- LINQ to SQL或者EF查询结果
- 本地集合

- 另一个X-DOM

无论使用何种数据源，使用LINQ映射一个X-DOM的实现方式是相同的：首先使用函数型构建表达式定义出所期望的X-DOM结构，然后针对该表达式进行相应的LINQ查询。

例如，假设我们从一个数据库中查询客户并产生相应的XML：

```
<customers>
  <customer id="1">
    <name>Sue</name>
    <buys>3</buys>
  </customer>
  ...
</customers>
```

我们首先使用简单的字面值编写X-DOM函数型构建表达式：

```
var customers =
  new XElement ("customers",
    new XElement ("customer", new XAttribute ("id", 1),
    new XElement ("name", "Sue"),
    new XElement ("buys", 3)
  )
);
```

然后将其变为映射并围绕它构建LINQ查询：

```
var customers =
  new XElement ("customers",
    from c in dataContext.Customers
    select
      new XElement ("customer", new XAttribute ("id", c.ID),
        new XElement ("name", c.Name),
        new XElement ("buys", c.Purchases.Count)
      )
    );
```

提示： 在Entity Framework中，查询到客户信息后必须调用ToList()，所以将第三行改成如下形式：

```
from c in objectContext.Customers.ToList()
```

上述代码生成的XML如下：

```
<customers>
  <customer id="1">
    <name>Tom</name>
    <buys>3</buys>
  </customer>
  <customer id="2">
    <name>Harry</name>
    <buys>2</buys>
  </customer>
...
</customers>
```

可以用另一种更清晰的方式来实现相同的查询，分为两步：

第一步：

```
IEnumerable<XElement> sqlQuery =
  from c in dataContext.Customers
  select
    new XElement ("customer", new XAttribute ("id", c.ID),
      new XElement ("name", c.Name),
      new XElement ("buys", c.Purchases.Count)
    );
```

内层是一个普通的LINQ to SQL查询，将数据源映射成用户自定义类型（从LINQ to SQL的角度看）。下面是第二步操作：

```
var customers = new XElement ("customers", sqlQuery);
```

这一步操作构建了根节点XElement。这里比较特殊的地方是节点的内容sqlQuery，它不是单个的XElement对象，而是一个实现了IEnumerable<XElement>的IQueryable<XElement>。注意，在处理XML内容时，会自动枚举集合中的元素，所以会将每个XElement添加为子节点。

在这个例子中外部的查询使得整个查询从LINQ to SQL数据库查询转换成了可枚举查询。XElement的构造方法并不知道IQueryable<>，因此它将强制执行数据库查询枚举，从而立即执行SQL语句。

10.11.1 排除空元素

假如在前面的示例中，我们希望得到客户最近的高价值采购单的更具体的数据，可以使用下面这种方式来获取：

```
var customers =
  new XElement ("customers",
    from c in dataContext.Customers
    let lastBigBuy = (from p in c.Purchases
                      where p.Price > 1000
                      orderby p.Date descending
                      select p).FirstOrDefault()
    select
      new XElement ("customer", new XAttribute ("id", c.ID),
        new XElement ("name", c.Name),
        new XElement ("buys", c.Purchases.Count),
        new XElement ("lastBigBuy",
          new XElement ("description",
            lastBigBuy == null ? null : lastBigBuy.Description),
          new XElement ("price",
            lastBigBuy == null ? 0m : lastBigBuy.Price)
        )
      )
  );
```

这种方式去掉空的元素，也就是那些没有高价值采购单的客户（如果它是一个本地查询，而不是数据库查询，会抛出NullReferenceException异常）。在这个示例中，完全省略lastBigBuy节点会更好。我们可以通过在条件运算符里面封装一个lastBigBuy的构造方法来完成这个目标：

```
select
```

```
    new XElement ("customer", new XAttribute ("id", c.ID),
      new XElement ("name", c.Name),
      new XElement ("buys", c.Purchases.Count),
      lastBigBuy == null ? null :
        new XElement ("lastBigBuy",
          new XElement ("description", lastBigBuy.Description),
          new XElement ("price", lastBigBuy.Price)
```


对于那些没有lastBigBuy的客户，不再返回空的XElement元素，而是一个null。这是期望的结果，因为处理XML时null会被自动忽略掉。

10.11.2 流化一个映射

将数据映射到X-DOM中，如果只是为了调用Save方法存储它（或者对它调用ToString方法），那么可以使用XStreamingElement提高内存效率。一个XStreamingElement对象实际上是简化版的XElement，对其子元素使用了延迟加载。可使用XStreamingElement替换外围的Xelement：

```
var customers =
  new XStreamingElement ("customers",
    from c in dataContext.Customers
    select
      new XStreamingElement ("customer", new XAttribute ("id", c.ID),
        new XElement ("name", c.Name),
        new XElement ("buys", c.Purchases.Count)
      )
    );
customers.Save ("data.xml");
```

传入到XStreamingElement构造方法中的查询语句并不会立即执行，只有当调用Save、ToString或WriteTo方法时，里面的元素才会被真正从数据库中读取出来，这可以避免将整个X-DOM一次加载到内存中。这种查询的不足之处在于，如果多次调用Save方法，整个查询语句会被多次解释；另外，也不能遍历XStreamingElement的子元素，因为它没有提供诸如Elements或者Attributes之类的方法。

XStreamingElement并不基于XObject，也不基于其他类，因为它只有有限个数的成员。除了Save、ToString和WriteTo，它的成员仅限于：

- Add方法，接受的内容与构造方法相似
- Name属性

另外，不能以流的方式从XStreamingElement中读取内容，如果要从中读取内容，必须使用XmlReader配合X-DOM来实现。我们会在第11章的“XmlReader/XmlWriter使用模式”一节中介绍如何实现这一点。

10.11.3 转换X-DOM

可以使用映射的方式改变一个X-DOM的格式。例如，假设要转换一个*msbuild*的XML文件（C#编译器和Visual Studio使用该文件描述一个项目），使其格式更简单，适合生成一个报表。下面就是一个*msbuild*文件：

```
<Project DefaultTargets="Build" xmlns="http://schemas.microsoft.com/dev...>
  <PropertyGroup>
    <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
```

```
    <ProductVersion>9.0.11209</ProductVersion>
    ...
  </PropertyGroup>
  <ItemGroup>
    <Compile Include="ObjectGraph.cs" />
    <Compile Include="Program.cs" />
    <Compile Include="Properties\AssemblyInfo.cs" />
    <Compile Include="Tests\Aggregation.cs" />
    <Compile Include="Tests\Advanced\RecursiveXml.cs" />
  </ItemGroup>
  <ItemGroup>
    ...
  </ItemGroup>
    ...
</Project>
```

如果只想得到XML中有关文件的信息，即下面这些内容：

```
<ProjectReport>
  <File>ObjectGraph.cs</File>
  <File>Program.cs</File>
  <File>Properties\AssemblyInfo.cs</File>
  <File>Tests\Aggregation.cs</File>
  <File>Tests\Advanced\RecursiveXml.cs</File>
</ProjectReport>
```

下面的查询可以完成这种转换：

```
XElement project = XElement.Load ("myProjectFile.csproj");
XNamespace ns = project.Name.Namespace;
var query =
  new XElement ("ProjectReport",
    from compileItem in
      project.Elements (ns + "ItemGroup").Elements (ns + "Compile")
    let include = compileItem.Attribute ("Include")
    where include != null
    select new XElement ("File", include.Value)
);
```

在上面这个查询中，首先取出所有的ItemGroup元素，然后使用扩展方法Elements取出ItemGroup中所有的Compile子元素。注意，在查找元素时我们指定了XML命名空间，这是因为在原始文件中，所有的元素都继承了Project元素中的命名空间。如果在这里仅仅使用一个本地的元素名（ItemGroup），那么将不能找到我们期望的元素。最后取出Include属性值并把这些属性值映射成新的元素。

高级转换

对诸如X-DOM之类的本地集合进行查询时，可以自定义需要的查询运算符，以协助完成比较复杂的查询操作。

设想在前面的示例中，如果想得到一个文件夹和其中文件的这种分层的结果集：

```
<Project>
  <File>ObjectGraph.cs</File>
  <File>Program.cs</File>
  <Folder name="Properties">
```

```
      <File>AssemblyInfo.cs</File>
    </Folder>
    <Folder name="Tests">
      <File>Aggregation.cs</File>
      <Folder name="Advanced">
        <File>RecursiveXml.cs</File>
      </Folder>
     </Folder>
</Project>
```

为了实现这个需求，需要递归处理路径字符串*Tests\Advanced\RecursiveXml.cs*。下面这个方法就实现了这种处理：它接收一组路径字符串，经过处理之后返回我们所期望的那种文件夹和文件分层显示的XML：

```
static IEnumerable<XElement> ExpandPaths (IEnumerable<string> paths)
{
  var brokenUp = from path in paths
                 let split = path.Split (new char[] { '\\' }, 2)
                 orderby split[0]
                 select new
                 {
                   name = split[0],
                   remainder = split.ElementAtOrDefault (1)
                 };

  IEnumerable<XElement> files = from b in brokenUp
                                where b.remainder == null
                                select new XElement ("file", b.name);

  IEnumerable<XElement> folders = from b in brokenUp
                                  where b.remainder != null
                                  group b.remainder by b.name into grp
                                  select new XElement ("folder",
                                    new XAttribute ("name", grp.Key),
                                    ExpandPaths (grp)
                                  );
  return files.Concat (folders);
}
```

第一个查询语句将每个路径字符串拆分成name+remainder的形式，拆分点是第一个反斜杠处：

```
Tests\Advanced\RecursiveXml.cs -> Tests + Advanced\RecursiveXml.cs
```

如果remainder是null，那么直接处理文件名即可。files会查询处理这种情况。

如果remainder不是null，就意味是一个文件夹。folders查询会处理这种情况。由于其他的文件也可能在这个文件夹中，所以在查询中必须使用group语句将同属一个文件夹的文件放到同一组中。对于每个组，接下来的处理都是一样的。

最终的结果还需要将files和folders中的数据结合到一起。Concat运算符保留了元素的顺序，所以按照字母顺序先列出文件，然后列出文件夹。

有了这个方法，就可以分两步完成上面的查询。首先，要得到简单的路径字符串序列：

```
IEnumerable<string> paths =
  from compileItem in
```

```
    project.Elements (ns + "ItemGroup").Elements (ns + "Compile")
  let include = compileItem.Attribute ("Include")
  where include != null
  select include.Value;
```

然后将上面得到的路径字符串传给ExpandPaths方法，就可以得到最终的结果：

```
var query = new XElement ("Project", ExpandPaths (paths));
```

第11章 其他XML技术

System.Xml命名空间由以下命名空间和核心类组成：

System.Xml.*

XmlReader和XmlWriter

高性能、只向前地读写XML流。

XmlDocument

代表基于W3C标准的文档对象模型（DOM）的XML文档。

System.Xml.XPath

为XPath（一种基于字符串的查询XML的语言）提供基础结构和API（XPathNavigator类）。

System.Xml.XmlSchema

为（W3C）XSD提供基础结构和API。

System.Xml.Xsl

为使用（W3C）XSLT对XML进行解析提供基础结构和API。

System.Xml.Serialization

提供类和XML之间的序列化（见第16章）。

System.Xml.XLinq

先进的、简化的、LINQ版本的XmlDocument（见第10章）。

W3C是World Wide Web Consortium（万维网联盟）的缩写，定义了XML标准。

静态类XmlConvert是解析和格式化XML字符串的类，这个类在第6章涉及到。

11.1 XmlReader

XmlReader是一个高性能的类，能够以低级别、只向前的方式读取XML流。

考虑下面的XML文件：

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customer id="123" status="archived">
  <firstname>Jim</firstname>
```

```
    <lastname>Bo</lastname>
  </customer>
```

通过调用静态方法XmlReader.Create来实例化一个XmlReader对象，可以向这个方法传递一个Stream、TextReader或者URI字符串，例如：

```
using (XmlReader reader = XmlReader.Create ("customer.xml"))
...
```

提示：因为XmlReader可以读取一些可能速度较慢的数据源（Stream和URI），所以它为大多数方法提供了异步版本，这样我们可以方便编写非阻塞代码。

我们将在第14章详细介绍异步编程。

以下代码通过读取一个字符串来创建一个XmlReader：

```
XmlReader reader = XmlReader.Create(new System.IO.StringReader(myString));
```

也可以通过传递一个XmlReaderSettings对象来设置解析和验证选项。XmlReaderSettings的以下三个属性在忽略不需要的内容时特别有用：

```
bool IgnoreComments                    // 是否忽略注释节点?
bool IgnoreProcessingInstructions      // 是否忽略处理指令?
bool IgnoreWhitespace                  // 是否忽略空白?
```

在某些情况下，我们可以忽略空白节点，实现方式如下面的示例所示：

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;

using (XmlReader reader = XmlReader.Create ("customer.xml", settings))
```

XmlReaderSettings另一个重要的属性是ConformanceLevel。其默认值是Document，指示读取器验证一个有效的XML文档是否只包含一个根节点，但是如果读取一个内部包含多个节点的XML文档，就会有问题。例如要读取以下内容，并且避免抛出异常，必须设置ConformanceLevel为Fragment。

```
<firstname>Jim</firstname>
<lastname>Bo</lastname>
```

在Fragment上设置ConformanceLevel，即可以在读取时不抛出异常。

XmlReaderSettings还包含一个CloseInput属性，它指示当关闭读取器时，是否关闭基础流。在XmlWriterSettings上也有类似的CloseOutput属性，它们的默认值都是true。

11.1.1 读取节点

XML流以XML节点为单位。读取器按文本顺序（深度优先）来遍历XML流，Depth属性返回游标的当前深度。

从XmlReader读取节点的最基本的方法是调用Read方法。它指向XML流的下一个节点，相当于IEnumerator的MoveNext方法。第一次调用Read会把游标放置在第一个节点，当Read方法返回false时，说明游标已经到达最后一个节点，在这个时候XmlReader应该被关闭。

在下面这个示例中，我们读取XML流中的每一个节点，同时输出每个节点的类型：

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;

using (XmlReader reader = XmlReader.Create("customer.xml", settings))
  while (reader.Read())
  {
    Console.Write(new string(' ', reader.Depth * 2)); // 缩进
    Console.WriteLine(reader.NodeType);
  }
```


输出如下：

```
XmlDeclaration
Element
  Element
    Text
  EndElement
  Element
    Text
  EndElement
EndElement
```

提示： 属性没有包含在基于Read的遍历中（在本章的“读取属性”中详细介绍）。

NodeType属性的类型是XmlNodeType，它包含以下成员：

None	Comment	Document
XmlDeclaration	Entity	DocumentType
Element	EndEntity	DocumentFragment
EndElement	EntityReference	Notation
Text	ProcessingInstruction	Whitespace
Attribute	CDATA	SignificantWhitespace

XmlReader提供了Name和Value这两个string类型的属性来访问节点的内容。根据节点类型，内容可能定义在Name或Value上，或者两者都有。

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;
settings.ProhibitDtd = false;        // 必须符合DTD规则

using (XmlReader r = XmlReader.Create("customer.xml", settings)) while (r.Read())
{
  Console.Write(r.NodeType.ToString().PadRight(17, '-'));
  Console.Write("> ".PadRight(r.Depth * 3));

  switch (r.NodeType)
  {
    case XmlNodeType.Element:
    case XmlNodeType.EndElement:
      Console.WriteLine(r.Name); break;

    case XmlNodeType.Text:
```

```
      case XmlNodeType.CDATA:
      case XmlNodeType.Comment:
      case XmlNodeType.XmlDeclaration:
        Console.WriteLine(r.Value); break;

      case XmlNodeType.DocumentType:
        Console.WriteLine(r.Name + " - " + r.Value); break;

      default: break;
    }
  }
```

为演示这个，我们扩展XML文件使其包含文档类型、实体、CDATA和注释：

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE customer [ <!ENTITY tc "Top Customer"> ]>
<customer id="123" status="archived">
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
  <quote><![CDATA[C#'s operators include: < > &]]></quote>
  <notes>Jim Bo is a &tc;</notes>
  <!--  That wasn't so bad!  -->
</customer>
```

一个实体类似一个宏，一个CDATA类似一个在C#中只取原始文本值的字符串（@"..."）。结果如下：

```
XmlDeclaration---> version="1.0" encoding="utf-8"
DocumentType-----> customer - <!ENTITY tc "Top Customer">
Element----------> customer
Element----------> firstname
Text------------->   Jim
EndElement-------> firstname
Element----------> lastname
Text------------->    Bo
EndElement-------> lastname
Element----------> quote
CDATA------------>    C#'s operators include: < > &
EndElement-------> quote
Element----------> notes
Text------------->    Jim Bo is a Top Customer
EndElement-------> notes
Comment---------->   That wasn't so bad!
EndElement-------> customer
```

XmlReader自动地解析实体，所以在这个示例中实体引用“&tc;”展开为“Top Customer”。

11.1.2 读元素

通常已知读取的XML文档的结构，同样的，XmlReader提供了一些方法可以在读取的时候推断文档的特定结构。这样会简化代码，同时会进行某些验证。

提示： 验证失败会导致XmlReader抛出XmlException，这个异常包含错误发生的行号（LineNumber）和位置（LinePosition）。当XML文件很大时记录这些信息会比较关键。

ReadStartElement方法验证当前NodeType是否为StartElement，然后调用Read。如果指定一个名字，就会验证当前节点是否符合这个名称。

ReadEndElement方法验证当前NodeType是否为EndElement，然后调用Read。

例如我们可以读取：

```
<firstname>Jim</firstname>
```

通过如下片段：

```
reader.ReadStartElement ("firstname");
Console.WriteLine (reader.Value);
reader.ReadEndElement();
```

ReadElementContentAsString方法会一次完成以上所有操作：它读取一个开始节点、一个文本节点和一个结束节点，最后以字符串形式返回内容：

```
string firstName = reader.ReadElementContentAsString("firstname", "");
```

第二个值为空的参数是命名空间。这个方法也有类型化的版本，例如ReadElementContentAsInt。回到我们刚开始的XML文档：

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customer id="123" status="archived">
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
  <creditlimit>500.00</creditlimit>   <!-- OK, we sneaked this in! -->
</customer>
```

可以按下面的代码读取：

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;

using (XmlReader r = XmlReader.Create("customer.xml", settings))
{
  r.MoveToContent();                            // 跳过XML声明
  r.ReadStartElement("customer");
  string firstName= r.ReadElementContentAsString("firstname", "");
  string lastName  = r.ReadElementContentAsString("lastname", "");
  decimal creditLimit = r.ReadElementContentAsDecimal("creditlimit", "");

  r.MoveToContent();                  // 跳过注释
  r.ReadEndElement();                 // 读取关闭customer节点
}
```

提示：MoveToContent方法非常有用，它可以让读取器跳过不必要的：XML声明、空白节点、注释、处理指令。也可以设置XmlReaderSettings的属性使读取器忽略这些内容。

---

1. 可选元素

在前一个示例，如果设置<lastname>是可选的，一个简单的方法是：

```
r.ReadStartElement ("customer");
string firstName    = r. ReadElementContentAsString ("firstname", "");
string lastName     = r.Name == "lastname"? r.ReadElementContentAsString(): null;
decimal creditLimit = r.ReadElementContentAsDecimal ("creditlimit", "");
```

2. 随机元素顺序

这部分的示例取决于元素在XML文件出现的顺序。如果需要拷贝以任何顺序出现的元素，最简单的方法是读取XML片段到一个X-DOM。在“使用XmlReader/XmlWriter的模式”中介绍如何实现。

3. 空元素

XmlReader处理空元素的方式会是一个可怕的陷阱，例如下面的元素：

```
<customerList></customerList>
```

在XML 中等同于：

```
<customerList/>
```

但是XmlReader会以不同的方式进行处理，对于第一种情况，下面的代码可以按预期的方式执行：

```
reader.ReadStartElement("customerList");
reader.ReadEndElement();
```

但对于第二种情况，ReadEndElement会抛出一个异常，因为XmlReader不能读取到指定的结束节点。解决方法是按下面代码检查这个空节点：

```
bool isEmpty = reader.IsEmptyElement;
reader.ReadStartElement("customerList");
if (!isEmpty) reader.ReadEndElement();
```

实际应用中，当元素中包含子元素时（例如客户列表）会有点累赘。对包含简单文本的元素（如firstname），可以调用类似ReadElementContentAsString的方法来避免这个问题。ReadElement*XXX*这些方法都可以正确地处理这两种空元素。

4. 其他ReadXXX方法

表11-1汇总了XmlReader的所有Read*XXX*方法。它们大多数用来处理元素。

表11-1：读取方法

成员	作用的节点	XML片断示例	输入参数	返回的数据
ReadContentAsXXX	文本	<a>x		x
ReadString	文本	<a>x		x
ReadElementString	元素	<a>x		x
ReadElementContentAsXXX	元素	<a>x		x
ReadInnerXml	元素	<a>x		x
ReadOuterXml	元素	<a>x		<a>x
ReadStartElement	元素	<a>x		

表11-1：读取方法（续）

成员	作用的节点	XML片断示例	输入参数	返回的数据
ReadEndElement	元素	<a>x		
ReadSubtree	元素	<a>x		<a>x
ReadToDescendent	元素	<a>x	"b"	
ReadToFollowing	元素	<a>x	"b"	
ReadToNextSibling	元素	<a>x	"b"	
ReadAttributeValue	属性	见“读取属性”一节		

方法ReadContentAs*XXX*把一个文本节点解析为*XXX*类型，而在其内部，使用了XMLConvert类来实现字符串的类型转换。文本节点可以被包含在一个元素或者一个属性中。

方法ReadElementContentAs*XXX*对应于ReadContentAs*XXX*方法的封装，它用来读取元素节点，而不是元素内的文本节点。

提示： 类型化的Read*XXX*方法也包含对应的版本来读取基于64和BinHex格式数据到一个字节数组。

ReadInnerXml应用于一个元素，它读取并返回一个元素及其所有后代节点。当应用于一个属性时，返回这个属性的值。

ReadOuterXml和ReadInnerXml类似，但是它包含而不是排除游标所在位置的元素。

ReadSubtree返回一个代理读取器用以查看当前节点（包含子孙后代节点）。代理读取器必须在原始读取器进行下一次读取之前被关闭。在代理读取器关闭时，原始读取器的的游标必须要移动到子树的末尾。

ReadToDescendent移动游标到指定名称或命名空间的第一后代节点的开始处。

ReadToFollowing移动游标到指定名称或命名空间的第一个节点的开始处（不管深度大小）。

ReadToNextSibling移动游标到指定名称或命名空间的第一个兄弟节点的开始处。

ReadString和ReadElementString类似于ReadContentAsString和ReadElementContentAsString，只不过当节点包含多于一个文本节点时会抛出一个异常。通常来说，应避免使用这些方法，因为当元素中包含注释时会抛出异常。

11.1.3 读取属性

XmlReader提供了一个索引器以直接（随机）地通过名字或位置来访问一个节点的属性，使用索引器等同于调用GetAttribute方法。

如下面的XML片段：

```
<customer id="123" status="archived"/>
```

我们可以按下面的方式来读取属性值：

```
Console.WriteLine (reader ["id"]);              // 123
```

```
Console.WriteLine (reader ["status"]);            // archived
Console.WriteLine (reader ["bogus"] == null);     // True
```

警告：XmlReader必须在元素的开始位置才能读取属性。在调用ReadStartElement后，再也无法访问这些属性。

可以通过排列位置来访问属性，虽然它们的顺序在语义上是不相关的，先前的示例可以被重写为：

```
Console.WriteLine(reader[0]);             // 123
Console.WriteLine(reader[1]);             // archived
```

也可以通过索引器指定属性的命名空间（如果有的话）。

AttributeCount属性返回当前节点的个数。

属性节点

为了显式地遍历属性节点，必须进行一种特别的转移，而非只是简单地调用Read。这样做的理由是，当把属性值解析成其他的类型时，可以使用ReadContentAs*XXX*方法。

这种转移必须由一个元素起始节点开始。为了处理过程更简单，在遍历属性时不考虑只向前的原则：可以调用MoveToAttribute方法来跳转（向前或者向后）到任何的结点。

提示：MoveToElement可以从属性节点转移的任何地方返回其所在元素的开始处。

例如先前的示例：

```
<customer id="123" status="archived"/>
```

我们可以这样实现：

```
reader.MoveToAttribute("status");
string status = ReadContentAsString();

reader.MoveToAttribute("id");
int id = ReadContentAsInt();
```

如果指定的属性不存在，MoveToAttribute返回false。

也可以通过调用方法MoveToFirstAttribute和MoveToNextAttribute来依次遍历每个属性：

```
if (reader.MoveToFirstAttribute())
  do
  {
     Console.WriteLine(reader.Name + "=" + reader.Value);
  }
  while (reader.MoveToNextAttribute());

// 输出结果:
id=123
status=archived
```

11.1.4 命名空间和前缀

XmlReader为引用元素和属性名称提供了两个并行系统：

- Name
- NameSpaceURI和LocalName

当读取元素的Name属性或者调用只接受一个name参数的方法时，你用的就是第一个系统。这个系统在没有命名空间和前缀的情况下可以工作得很好；否则命名空间将被忽略，而前缀也只是作为name的一部分传递进来。例如：

片段示例	Name
<customer ...>	customer
<customer xmlns='blah' ...>	customer
<x:customer ...>	x:customer

下面的代码对前两种情况正常处理：

```
reader.ReadStartElement ("customer");
```

第三种情况需要按下面的代码处理：

```
reader.ReadStartElement("x:customer");
```

第二个系统基于命名空间的属性：NamespaceURI和LocalName。这些属性包含前缀以及父元素所定义的默认命名空间。前缀会被自动地展开，这意味着NamespaceURI总能反映当前元素的语义上正确的命名空间，而LocalName总是不包含前缀。

当调用需要传递两个参数的方法（如ReadStartElement）时，使用第二种系统。例如，考虑下面的XML：

```
<customer xmlns="DefaultNamespace" xmlns:other="OtherNamespace">
  <address>
    <other:city>
    ...
```

我们可以这样读取：

```
reader.ReadStartElement("customer", "DefaultNamespace");
reader.ReadStartElement("address", "DefaultNamespace");
reader.ReadStartElement("city", "OtherNamespace");
```

如果需要读取前缀，可以通过Prefix属性来查看什么前缀在被使用，并且可以通过调用LookupNamespace把它转换为命名空间。

11.2 XmlWriter

XmlWriter是一个XML流的只向前的编写器。XmlWriter的设计和XmlReader是对称的。

和XmlTextReader一样，可以通过调用静态方法Create来构建一个XmlWriter，在下面的示例中，

我们打开缩进选项使输出更可读，然后写一个简单的XML文件：

```
XmlWriterSettings settings = new XmlWriterSettings();
settings.Indent = true;

using (XmlWriter writer = XmlWriter.Create("..\\..\\foo.xml", settings))
{
  writer.WriteStartElement("customer");
  writer.WriteElementString("firstname", "Jim");
  writer.WriteElementString("lastname", "Bo");
  writer.WriteEndElement();
}
```

运行代码会产生下面的文档（和我们在第一个示例中用XmlReader读取的文件一样）：

```
<?xml version="1.0" encoding="utf-8" ?>
<customer>
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
</customer>
```

除非使用XmlWriterSettings，并设置其OmitXmlDeclaration为true或者ConformanceLevel为Fragment，否则XmlWriter会自动地在顶部写上声明。并且后者允许写多个根节点，如果不设置的话会抛出异常。

WriteValue方法写一个文本节点。它不仅接受string类型的参数，还可以接受像bool、DateTime类型的参数，实际在内部调用了XmlConvert来实现符合XML的字符串解析：

```
writer.WriteStartElement("birthdate");
writer.WriteValue(DateTime.Now);
writer.WriteEndElement();
```

相比，如果我们调用：

```
WriteElementString("birthdate", DateTime.Now.ToString());
```

结果可能会是既不符合XML又容易导致不正确的解析。

WriteString和调用WriteValue传递一个string参数实现的操作是等价的。 XmlWriter会自动地解析那些在属性或者元素之中不合法的字符，例如&、<、>以及扩展Unicode字符。

11.2.1 写属性

在写完开始节点后可以立即写属性：

```
writer.WriteStartElement("customer");
writer.WriteAttributeString("id", "1");
writer.WriteAttributeString("status", "archived");
```

而对于写非字符串类型的值，可以调用WriteStartAttribute、WriteValue然后再调用WriteEndAttribute。

11.2.2 其他类型节点

XmlWriter也为写其他类型的节点定义了以下方法：

```
WriteBase64         // 二进制类型数据
WriteBinHex         // 二进制类型数据
WriteCData
WriteComment
WriteDocType
WriteEntityRef
WriteProcessingInstruction
WriteRaw
WriteWhitespace
```

WriteRaw直接向输出流注入一个字符串。也可以通过接受XmlReader的WriteNode方法，把XmlReader中的所有内容写入输出流。

11.2.3 命名空间和前缀

对Write*方法的重载可以把命名空间和元素或属性关联起来。我们重写上个示例中的XML文件，将所有的元素关联到*http://oreilly.com*这个命名空间，在customer元素上声明前缀o：

```
writer.WriteStartElement("o", "customer", "http://oreilly.com");
writer.WriteElementString("o", "firstname", "http://oreilly.com", "Jim");
writer.WriteElementString("o", "lastname", "http://oreilly.com", "Bo");
writer.WriteEndElement();
```

现在输出的XML文件内容如下：

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<o:customer xmlns:o='http://oreilly.com'>
  <o:firstname>Jim</o:firstname>
  <o:lastname>Bo</o:lastname>
</o:customer>
```

XmlWriter使代码非常简洁，如果相同的命名空间在父元素上已声明，它会自动地省略子元素上命名空间的声明。

11.3 使用XmlReader/XmlWriter的模式

11.3.1 处理多层次数据

考虑下面的类：

```
public class Contacts
{
  public IList<Customer> Customers = new List<Customer>();
  public IList<Supplier> Suppliers = new List<Supplier>();
}

public class Customer { public string FirstName, LastName; }
public class Supplier { public string Name; }
```

如果想要使用XmlReader和XmlWriter按下面的结果来序列化Contacts对象到XML文档：

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<contacts>
```

```
    <customer id="1">
      <firstname>Jay</firstname>
      <lastname>Dee</lastname>
    </customer>
    <customer>                     <!-- we'll assume id is optional -->
      <firstname>Kay</firstname>
      <lastname>Gee</lastname>
    </customer>
    <supplier>
      <name>X Technologies Ltd</name>
    </supplier>
  </contacts>
```

最好的办法不是定义一个总体方法，而是通过在类Customer和Supplier上分别定义WriteXml和ReadXml方法封装这些XML功能，这种做法很容易理解：

- ReadXml和WriteXml保证当他们退出的时候读取器或者写入器保持同样的深度。
- ReadXml只读取外部元素，而WriteXml只写元素内部内容。

下面是定义Customer类的示例：

```
public class Customer
{
  public const string XmlName = "customer";
  public int? ID;
  public string FirstName, LastName;

  public Customer() { }
  public Customer(XmlReader r) { ReadXml(r); }

  public void ReadXml(XmlReader r)
  {
    if (r.MoveToAttribute("id")) ID = r.ReadContentAsInt();
    r.ReadStartElement();
    FirstName = r.ReadElementContentAsString("firstname", "");
    LastName = r.ReadElementContentAsString("lastname", "");
    r.ReadEndElement();
  }

  public void WriteXml(XmlWriter w)
  {
    if (ID.HasValue) w.WriteAttributeString("id", "", ID.ToString());
    w.WriteElementString("firstname", FirstName);
    w.WriteElementString("lastname", LastName);
  }
}
```

注意ReadXml读取对应节点及其子节点时，如果这个方法的调用者读过了这部分内容，将会导致Customer不能读到它的属性。而WriteXml没有采用对称的写法是由于考虑到以下两点：

- 调用者也许需要选择外部元素的命名方式。
- 调用者可能需要去写其他XML属性，例如元素的子类型（在读回到这个元素后用来决定到底应该实例化哪个类）。

使用这种模式的另一个好处就是它可以让你的实现与IXmlSerializable（见第17章）兼容。

Supplier类可以按同样的方式定义：

```
public class Supplier
{
  public const string XmlName = "supplier";
  public string Name;

  public Supplier() { }
  public Supplier(XmlReader r) { ReadXml(r); }

  public void ReadXml(XmlReader r)
  {
    r.ReadStartElement();
    Name = r.ReadElementContentAsString("name", "");
    r.ReadEndElement();
  }

  public void WriteXml(XmlWriter w)
  {
    w.WriteElementString("name", Name);
  }
}
```

对于Contacts类，必须在ReadXml方法里遍历customers元素，以检测子元素是一个customer还是一个supplier，同时还要避开那些空元素，代码如下：

```
public void ReadXml(XmlReader r)
{
  bool isEmpty = r.IsEmptyElement;   // 这样可以保证我们不会被一个空元素<contacts/> 干扰。
  r.ReadStartElement();
  if (isEmpty) return;
  while (r.NodeType == XmlNodeType.Element)
  {
    if (r.Name == Customer.XmlName)      Customers.Add(new Customer(r));
    else if (r.Name == Supplier.XmlName) Suppliers.Add(new Supplier(r));
    else
      throw new XmlException("Unexpected node: " + r.Name);
  }
  r.ReadEndElement();
}

public void WriteXml(XmlWriter w)
{
  foreach (Customer c in Customers)
  {
    w.WriteStartElement(Customer.XmlName);
    c.WriteXml(w);
    w.WriteEndElement();
  }
  foreach (Supplier s in Suppliers)
  {
    w.WriteStartElement(Supplier.XmlName);
    s.WriteXml(w);
    w.WriteEndElement();
  }
}
```

11.3.2 混合XmlReader/XmlWriter和X-DOM

可以在使用XmlReader或XmlWriter使代码复杂时使用X-DOM，使用X-DOM是处理内部元素的最佳方式，这样就可以兼并X-DOM的易用性和XmlReader、XmlWriter低内存消耗的优点。

1. 使用XmlReader和XElement

可以在调用XNode.ReadFrom方法时给它传递XmlReader来读取当前节点到一个X-DOM，不像XElement.Load方法那样复杂，它不会去查看整个文档，而是只读到当前子树的末尾。

例如，有一个XML日志文件，结构如下：

```
<log>
  <logentry id="1">
    <date>...</date>
    <source>...</source>
    ...
  </logentry>
  ...
</log>
```

假如有一百万个logentry元素，读取这个文档到X-DOM将会占用大量内存，解决方法就是用一个XmlReader来遍历每个logentry，然后用XElement去处理这些元素：

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;

using (XmlReader r = XmlReader.Create("logfile.xml", settings))
{
  r.ReadStartElement("log");
  while (r.Name == "logentry")
  {
    XElement logEntry = (XElement)XNode.ReadFrom(r);
    int id = (int)logEntry.Attribute("id");
    DateTime date = (DateTime)logEntry.Element("date");
    string source = (string)logEntry.Element("source");
    ...
  }
  r.ReadEndElement();
}
```

如果使用前面示例中的模式，可以在ReadXml或WriteXml方法中使用XElement，而不用让调用者知道这些！例如我们可以按下面的方式重写Customer的ReadXml方法：

```
public void ReadXml(XmlReader r)
{
  XElement x =(XElement)XNode.ReadFrom(r);
  FirstName =(string)x.Element("firstname");
  LastName =(string)x.Element("lastname");
}
```

XElement可以和XmlReader共同保证命名空间被正确处理并且前缀被恰当地展开——即使它被定义在级别外。所以如果我们按下面的方式来读取XML文档：

```
<log xmlns="http://loggingspace">
```

```
<logentry id="1">
...
```

我们在logentry级别构造的这些XElement会正确地继承外部的命名空间。

2. 使用XmlWriter 和XElement

可以使用一个XElement去写内部元素到一个XmlWriter。下面的代码使用XElement写一百万个logentry元素到一个XML文件，而不必把整体存储在内存中：

```
using (XmlWriter w = XmlWriter.Create("log.xml"))
{
  w.WriteStartElement("log");
  for (int i = 0; i < 1000000; i++)
  {
    XElement e = new XElement("logentry",
                   new XAttribute("id", i),
                   new XElement("date", DateTime.Today.AddDays(-1)),
                   new XElement("source", "test"));
    e.WriteTo(w);
  }
  w.WriteEndElement();
}
```

使用一个XElement会影响执行时间最小化，但是如果我们这个示例修改为使用XmlWriter，在执行时间上几乎没有差别。

11.4 XmlDocument

XmlDocument是一个XML文档的内存表示，这个类型的对象模型和方法与W3C所定义的模式一致。如果你熟悉其他符合W3C的XML DOM技术（例如 Java），就会同样熟悉XmlDocument类。但是如果和X-DOM相比的话，W3C模型就显得过于复杂。

提示：XmlDocument不适用于Metro配置。然而，WinRT命名空间Windows.Data.Xml.Dom中有一个类似的DOM。

在XmlDocument树中的所有对象的基类型是XmlNode。下面的类型的父类是XmlNode：

```
XmlNode
  XmlDocument
  XmlDocumentFragment
  XmlEntity
  XmlNotation
  XmlLinkedNode
```

XmlLinkedNode定义了NextSibling和PreviousSibling属性，是下列类的基类：

```
XmlLinkedNode
  XmlCharacterData
  XmlDeclaration
  XmlDocumentType
  XmlElement
```

```
XmlEntityReference
XmlProcesingInstruction
```

11.4.1 加载和保存XmlDocument

可以实例化一个XmlDocument，然后调用Load或LoadXm来从一个已知的源加载一个XmlDocument：

- Load接受一个文件名（filename）、流（Stream）、文本读取器（TextReader）或者XML读取器（XmlReader）。
- LoadXml接受一个XML字符串。

相对应的，通过调用Save方法时，传递filename、Stream、TextWriter或者XmlWriter参数来保存一个文档。

```
XmlDocument doc = new XmlDocument();
doc.Load("customer1.xml");
doc.Save("customer2.xml");
```

11.4.2 遍历XmlDocument

为了演示遍历一个XmlDocument，我们使用下面的XML文件：

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customer id="123" status="archived">
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
</customer>
```

通过定义在XNode上的ChildNodes属性可以深入到此节点的下层树型结构（descend into the tree structure）。它返回一个可索引的集合：

```
XmlDocument doc = new XmlDocument();
doc.Load("customer.xml");

Console.WriteLine(doc.DocumentElement.ChildNodes[0].InnerText); // Jim
Console.WriteLine(doc.DocumentElement.ChildNodes[1].InnerText); // Bo
```

而使用ParentNode属性，可以返回其父节点：

```
Console.WriteLine(
doc.DocumentElement.ChildNodes[1].ParentNode.Name);  // customer
```

下面的属性用来帮助我们遍历文档，如果相应节点不存在，返回null：

```
FirstChild、LastChild、NextSibling、PreviousSibling
```

下面两句也都输出firstname：

```
Console.WriteLine(doc.DocumentElement.FirstChild.Name);
Console.WriteLine(doc.DocumentElement.LastChild.PreviousSibling.Name);
```

XmlNode定义了一个Attributes属性用来通过名字或命名空间或顺序位置来访问属性，例如：

```
Console.WriteLine(doc.DocumentElement.Attributes["id"].Value);
```

11.4.3 InnerText和InnerXml

InnerText属性代表所有子文本节点的联合，下面两行都输出Jim，因为我们的XML文档仅包含一个文本节点：

```
Console.WriteLine(doc.DocumentElement.ChildNodes[0].InnerText);
Console.WriteLine(doc.DocumentElement.ChildNodes[0].FirstChild.Value);
```

设置InnerText属性会用一个文本节点替换所有子节点，所以在设置这个属性时要谨慎以防止不小心覆盖了所有子节点。例如：

```
doc.DocumentElement.ChildNodes[0].InnerText = "Jo";                 // 错误
doc.DocumentElement.ChildNodes[0].FirstChild.InnerText = "Jo";      // 正确
```

InnerXml属性表示当前节点中的XML片段。下面的代码使用元素上的InnerXml：

```
Console.WriteLine (doc.DocumentElement.InnerXml);

// OUTPUT:
<firstname>Jim</firstname><lastname>Bo</lastname>
```

如果节点类型不能有子节点，InnerXml会抛出一个异常。

11.4.4 创建和操作节点

创建和添加新节点：

1. 调用XmlDocument其中一个Create*XXX*方法，例如CreateElement。
2. 在父节点上调用AppendChild、PrependChild、InsertBefore或者InsertAfter来添加新节点到树上。

提示： 要创建节点，首先要有一个XmlDocument，不能像X-DOM那样简单地实例化一个XmlElement。节点需要“寄生”在一个XmlDocument宿主上。

例如：

```
XmlDocument doc = new XmlDocument();
XmlElement customer = doc.CreateElement("customer");
doc.AppendChild(customer);
```

下面创建一个与前面“XmlReader”中介绍XML对应的文档：

```
XmlDocument doc = new XmlDocument();
doc.AppendChild(doc.CreateXmlDeclaration("1.0", null, "yes"));

XmlAttribute id = doc.CreateAttribute("id");
XmlAttribute status = doc.CreateAttribute("status");
id.Value = "123";
status.Value = "archived";

XmlElement firstname = doc.CreateElement("firstname");
XmlElement lastname = doc.CreateElement("lastname");
firstname.AppendChild(doc.CreateTextNode("Jim"));
lastname.AppendChild(doc.CreateTextNode("Bo"));
```

```
XmlElement customer = doc.CreateElement("customer");

customer.Attributes.Append(id);
customer.Attributes.Append(status);
customer.AppendChild(lastname);
customer.AppendChild(firstname);

doc.AppendChild(customer);
```

可以以任何顺序来构建这棵树，即便重新排列添加子节点后的语句顺序，对此也没有影响。

也可以调用RemoveChild、ReplaceChild或者RemoveAll来移除节点。

11.4.5 命名空间

提示： 见第10章关于XML命名空间和前缀的介绍。

使用CreateElement和CreateAttribute的重载方法可以指定命名空间和前缀：

```
CreateXXX (string name);
CreateXXX (string name, string namespaceURI);
CreateXXX (string prefix, string localName, string namespaceURI);
```

参数name既可以是本地名称（没有前缀），也可以是带前缀的名称。参数namespaceURI用在当且仅当声明（而不是仅在引用）一个命名空间时。

下面是在创建一个元素时声明一个带前缀的命名空间：

```
XmlElement customer = doc.CreateElement ("o", "customer", "http://oreilly.com");
```

下面是在创建一个元时使用前缀引用一个命名空间的例子：

```
XmlElement customer = doc.CreateElement ("o:firstname");
```

在下一节，我们将解释如何在写XPath时处理命名空间。

11.5 XPath

XPath是XML查询的W3C标准。在.NET Framework中，XPath可以查询一个XmlDocument，就像用LINQ查询X-DOM。然而XPath应用更广泛，它也在其他XML技术中被使用，例如XML Schema、XLST和XAML。

提示： XPath查询按照XPath 2.0 数据模型（XPath Data Model）来表示。DOM和XPath数据模型都表示一个XML文档树。区别是XPath数据模型纯粹以数据为中心，采取了XML文本的格式。例如在XPath数据模型中，CDATA部分不是必需的，因为CDATA存在的唯一原因是可以在文本中包含XML的一些标识符。可以在*http://www.w3.org/tr/xpath20*找到XPath的详细说明。

这一部分的所有示例都采用下面的这个XML文件：

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
```

```
<customers>
  <customer id="123" status="archived">
    <firstname>Jim</firstname>
    <lastname>Bo</lastname>
   </customer>
   <customer>
     <firstname>Thomas</firstname>
     <lastname>Jefferson</lastname>
   </customer>
</customers>
```

可以使用下面的方式在代码中实现XPath查询：

- 在一个XmlDocument或XmlNode上调用Select*XXX*方法。
- 从一个XmlDocument或者XPathDocument上生成一个XPathNavigator。
- 在XNode上调用一个XPath*XXX*扩展方法。

Select*XXX*方法接受一个XPath查询字符串。下面的代码用来查找XmlDocument的firstname节点：

```
XmlDocument doc = new XmlDocument();
doc.Load ("customers.xml");
XmlNode n = doc.SelectSingleNode("customers/customer[firstname='Jim']");
Console.WriteLine (n.InnerText);     // JimBo
```

Select*XXX*方法委托了它们对XPathNavigator的实现，也可以在一个XmlDocument或一个只读的XPathDocument上直接使用它。

也可以通过调用System.Xml.XPath命名空间下定义的扩展方法在一个X-DOM上执行XPath查询：

```
XDocument doc = XDocument.Load(@"Customers.xml");
XElement e = e.XPathSelectElement("customers/customer[firstname='Jim']");
Console.WriteLine(e.Value);  // JimBo
```

为执行XPath查询，XNode定义了以下扩展方法：

```
CreateNavigator
XPathEvaluate
XPathSelectElement
XPathSelectElements
```

11.5.1 常用XPath运算符

XPath规范说明非常庞大。但是可以通过以下这些运算符来了解它，就像即使只知道三个和弦，就可以演奏好多歌曲。

表11-2：常用XPath运算符

运算符	介绍
/	子节点
//	递归的子节点
.	当前节点（通常暗指）
..	父节点

表11-2：常用XPath运算符（续）

运算符	介绍
*	通配符
@	属性
[]	筛选器
:	命名控件分隔符

例如查找customers节点的代码如下：

```
XmlNode node = doc.SelectSingleNode("customers");
```

运算符“/”用来查询所有子节点。例如查找customer节点：

```
XmlNode node = doc.SelectSingleNode("customers/customer");
```

运算符“//”用来查询所有子节点，不考虑嵌套深度。例如查询所有的lastname节点：

```
XmlNodeList nodes = doc.SelectNodes("//lastname");
```

运算符“..”用于查询父节点。这个示例虽然有点多余，因为我们总是从根节点开始，但是能很好地解释它的功能：

```
XmlNodeList nodes = doc.SelectNodes("customers/customer..customers");
```

运算符“*”是一个通配符，用于选择所有元素，与元素名无关。下面的代码选取customer的所有子节点：

```
XmlNodeList nodes = doc.SelectNodes("customers/customer/*");
```

运算符“@”选取属性，“*”可以作通配符。例如选取“id”属性：

```
XmlNode node = doc.SelectSingleNode("customers/customer/@id");
```

运算符“[]”可以对选择结果进行筛选，可以联合运算符=、!=、<、>、not()、and、or一起使用，例如我们筛选firstname：

```
XmlNode n = doc.SelectSingleNode("customers/customer[firstname='Jim']");
```

运算符“:”用来限定命名空间。例如验证customer元素是否带有x命名空间，我们可以这么写：

```
XmlNode node = doc.SelectSingleNode("x:customers");
```

11.5.2 XPathNavigator

XPathNavigator是XML文档的XPath数据模型上的一个游标，它被加载并提供了一些基本方法以在文档树上移动光标（例如：移动到父节点、子节点等）。XPathNavigator的Select*方法可以使用一个XPath字符串来表达更复杂的导航或查询以返回多个节点。

可以从一个XmlDocument、XPathDocument或者另一个XPathNavigator上来生成XPathNavigator实例。下面是从一个XmlDocument上生成一个XPathNavigator的示例：

```
XPathNavigator nav = doc.CreateNavigator();
XPathNavigator jim = nav.SelectSingleNode
  (
    "customers/customer[firstname='Jim']"
  );

Console.WriteLine(jim.Value);                // JimBo
```

在XPath数据模型中，一个节点的值是文本元素的连接，等同于XmlDocument的InnerText属性。

SelectSingleNode方法返回一个XPathNavigator。Select方法返回一个XPathNodeIterator以在多个XPathNavigator上进行简便地遍历。例如：

```
XPathNavigator nav = doc.CreateNavigator();
string xPath = "customers/customer/firstname/text()";
foreach (XPathNavigator navC in nav.Select (xPath))
Console.WriteLine (navC.Value);

输出：
Jim
Thomas
```

为了更快地查询，可以把XPath编译成一个XPathExpression，然后传递给Select*方法，例如：

```
XPathNavigator nav = doc.CreateNavigator();
XPathExpression expr = nav.Compile ("customers/customer/firstname");
foreach (XPathNavigator a in nav.Select (expr))
  Console.WriteLine (a.Value);

输出：
Jim
Thomas
```

11.5.3 查询中处理命名空间

查询包含命名空间的元素和属性时需要额外的步骤。考虑下面的XML文件：

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>

<o:customers xmlns:o='http://oreilly.com'>
  <o:customer id="123" status="archived">
    <firstname>Jim</firstname>
    <lastname>Bo</lastname>
  </o:customer>
  <o:customer>
    <firstname>Thomas</firstname>
    <lastname>Jefferson</lastname>
  </o:customer>
</o:customers>
```

虽然在查询节点中限定了“o”前缀，但是下面的查询会失败：

```
XmlDocument doc = new XmlDocument();
doc.Load("customers.xml");
XmlNode n = doc.SelectSingleNode("o:customers/o:customer");
Console.WriteLine(n.InnerText);       // JimBo
```

为了使查询正常执行，必须首先创建一个XmlNamespaceManager实例：

```
XmlNamespaceManager xnm = new XmlNamespaceManager(doc.NameTable);
```

可以把NameTable看作一个黑盒（XmlNamespaceManager在内部使用它来缓存和重用字符串）。一旦我们创建了NamespaceManager，就可以像下面这样添加前缀和命名空间：

```
xnm.AddNamespace("o", "http://oreilly.com");
```

XmlDocument和XPathNavigator的Select*方法有对应的重载函数来接受一个XmlNamespaceManager。我们可以把上一个查询重写为：

```
XmlNode n = doc.SelectSingleNode("o:customers/o:customer", xnm);
```

11.5.4 XPathDocument

XPathDocument是符合W3C XPath数据模型的只读的XML文档。使用XPathDocument后跟一个XPathNavigator要比一个单纯的XmlDocument快，但是不能对底层的文档进行更改。

```
XPathDocument doc = new XPathDocument("customers.xml");
XPathNavigator nav = doc.CreateNavigator();
foreach (XPathNavigator a in nav.Select("customers/customer/firstname"))
  Console.WriteLine(a.Value);
```

```
输出：
Jim
Thomas
```

11.6 XSD和模式验证

特定XML文档的内容几乎总是特定领域的，例如一个Microsoft Word 文档、一个应用程序配置文档、或者一个Web service。在每个领域，XML文件总是遵守特定的模式。有几个标准来介绍这样的架构模式，以能够标准化和自动地解释和验证XML文档。最被广泛接受的一个标准是XSD（XML Schema Definition，XML结构定义），它的前身是DTD和XDR，不过在System.Xml中也支持它们。

考虑下面的XML文档：

```
<?xml version="1.0"?>
<customers>
  <customer id="1" status="active">
    <firstname>Jim</firstname>
    <lastname>Bo</lastname>
  </customer>
  <customer id="1" status="archived">
    <firstname>Thomas</firstname>
    <lastname>Jefferson</lastname>
  </customer>
</customers>
```

我们可以为此文档写一个XSD，如下：

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified"
```

```
            elementFormDefault="qualified"
            xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="customers">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="customer">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="firstname" type="xs:string" />
              <xs:element name="lastname" type="xs:string" />
            </xs:sequence>
            <xs:attribute name="id" type="xs:int" use="required" />
            <xs:attribute name="status" type="xs:string" use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

可以看到，XSD文档本身就是用XML来写的，并且XSD文档也是用XSD来介绍的，可以在"*http://www.w3.org/2001/xmlschema.xsd*"找到它的定义。

11.6.1 模式验证

可以在读或处理XML文件或文档时用一个或多个模式来验证它，这样做有以下几个理由：

- 可以避免更少的错误检查和异常处理。
- 模式验证可以查出注意不到的错误。
- 错误信息比较详细重要。

为进行验证，可以把模式加入到XmlReader、XmlDocument或者X-DOM对象中，然后像通常那样读取或加载XML文档。模式验证会在内容被读的时候自动进行，所以输入流没有被读取两次。

1. 用一个XmlReader进行验证

下面把*customers.xsd*模式加入到一个XmlReader中：

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationType = ValidationType.Schema;
settings.Schemas.Add null, "customers.xsd");

using (XmlReader r = XmlReader.Create ("customers.xml", settings))
  ...
```

如果模式是内联的，应设置下面的标识，而不是添加schema：

```
settings.ValidationFlags |= XmlSchemaValidationFlags.ProcessInlineSchema;
```

然后正常进行读取。如果模式验证在任何一个地方失败了，将抛出一个XmlSchemaValicationException。

提示： 调用它的Read方法既会验证元素又会验证属性，因此不需要为了验证它而导航到每个单独的属性。

如果只验证文档，可以这样做：

```
using (XmlReader r = XmlReader.Create ("customers.xml", settings))
  try { while (r.Read()) ; }
  catch (XmlSchemaValidationException ex)
  {
    ...
  }
```

XmlSchemaValidationException包含错误的Message，LineNumber和LinePosition。在这个示例里，它只包含文档的第一个错误。如果想报告这个文档上的所有错误，就必须处理它的ValicationEventHandler事件：

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationType = ValidationType.Schema;
settings.Schemas.Add(null, "customers.xsd");
settings.ValidationEventHandler += ValidationHandler;
using (XmlReader r = XmlReader.Create("customers.xml", settings))
    while (r.Read()) ;
```

一旦处理了这个事件，模式错误就不会抛出异常，而是事件处理代码被触发：

```
static void ValidationHandler(object sender, ValidationEventArgs e)
{
   Console.WriteLine("Error: " + e.Exception.Message);
}
```

ValidationEventArgs的Exception属性包含XmlSchemaValidationException信息。

提示：在System.Xml命名空间下包含一个XmlValidatingReader类，这个类存于.NET Framework 2.0之前的版本中，用来进行模式验证，现在已经不再使用。

---

2. 验证一个X-DOM或者一个XmlDocument

为了验证一个正在读入到X-DOM或XmlDocument的XML文件或流，可以创建一个XmlReader并加入模式中，然后用读取器加载DOM：

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationType = ValidationType.Schema;
settings.Schemas.Add (null, "customers.xsd");
XDocument doc;
using (XmlReader r = XmlReader.Create ("customers.xml", settings))
  try { doc = XDocument.Load(r); }
  catch (XmlSchemaValidationException ex) { ... }

XmlDocument xmlDoc = new XmlDocument();
using (XmlReader r = XmlReader.Create ("customers.xml", settings))
  try { xmlDoc.Load(r); }
  catch (XmlSchemaValidationException ex) { ... }
```

也可以通过调用System.Xml.Schema中定义的扩展方法来验证一个已加载到内存中的XDocument或XElement。这些方法接受一个XmlSchemaSet（模式的集合）和一个验证事件处理器：

```
XDocument doc = XDocument.Load (@"customers.xml");
XmlSchemaSet set = new XmlSchemaSet ();
set.Add (null, @"customers.xsd");
StringBuilder errors = new StringBuilder ();
doc.Validate (set, (sender, args) => { errors.AppendLine(args.Exception.Message); });
Console.WriteLine (errors. ToString());
```

验证一个已驻内存的XmlDocument，可以添加模式到XmlDocument的Schema(s)属性（一个集合）中，然后调用其Validate方法，同时传入一个ValidationEventHandler来处理验证错误。

其他XML技术

11.7 XSLT

XSLT（Extensible Stylesheet Language Transformations，扩展样式表转换语言）是一种XML语言，它介绍了如何把一种XML语言转化为另一种。这种转化的典型就是把一个（描述数据的）XML文档转化为一个（描述格式化文档的）XHTML文档。

考虑下面的XML文件：

```
<customer>
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
</customer>
```

下面的这个XSLT文件演示这种转化：

```
<?xml version="1.0" encoding="UTF-8"?>
  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:template match="/">
    <html>
      <p><xsl:value-of select="//firstname"/></p>
      <p><xsl:value-of select="//lastname"/></p>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

输出如下：

```
<html>
  <p>Jim</p>
  <p>Bo</p>
</html>
```

System.Xml.Xsl.XslCompiledTransform转化类能有效地进行XLST转化，它舍弃了过去的XmlTransform类。用法非常简单：

```
XslCompiledTransform transform = new XslCompiledTransform();
transform.Load("test.xslt");
transform.Transform("input.xml", "output.xml");
```

一般说来，使用接受一个XmlWriter而不是一个输出文件的Transform重载方法会更有用，可以用来控制格式。

第12章

销毁和垃圾回收

有些对象要求显式地卸载代码来释放资源，如打开的文件、锁、执行中的系统句柄和非托管对象。在.NET的术语中，这叫做销毁（disposal），它由IDisposable接口来实现。那些占用托管内存的未使用对象必须在某些时候被回收，这个功能被称为垃圾回收，它由CLR执行。

销毁不同于垃圾回收的是，销毁通常是显式调用，而垃圾回收则完全自动进行。换言之，程序员要关心释放文件句柄、锁和操作系统资源等，而CLR则关心释放内存。

本章将介绍销毁和垃圾回收，还介绍C#的终止器和为销毁提供备份的模式。最后，我们将介绍垃圾回收器的复杂性和其他内存管理选项。

12.1 IDisposable接口、Dispose方法和Close方法

.NET Framework定义了特殊的接口，它为类型提供了卸载方法：

```
public interface IDisposable
{
  void Dispose();
}
```

C#的using语句从语法上提供了对实现IDisposable接口的对象调用Dispose方法的捷径，它还使用了try/finally块。例如：

```
using (FileStream fs = new FileStream ("myFile.txt", FileMode.Open))
{
  // ... 写入文件 ...
}
```

编译器把它转换成：

```
FileStream fs = new FileStream ("myFile.txt", FileMode.Open);
try
{
  // ... 写入文件 ...
}
finally
{
```

```
    if (fs != null) ((IDisposable)fs).Dispose();
  }
```

Finally语句块保证Dispose方法一定被调用，即使是抛出异常（注1）或代码提前离开这个语句块。

在简单的情况下，编写自定义的可销毁类型只需要实现IDisposable接口并编写Dispose方法：

```
sealed class Demo : IDisposable
{
  public void Dispose()
  {
    // 执行清理/释放资源
    ...
  }
}
```


提示：这种模式在简单的情况下工作正常，也适用于密封类。我们将在“通过终止器调用Dispose方法”中介绍一个更好的模式，它能够提供在使用者忘记调用Dispose方法时的备用方法。对于非密封类，强烈建议从外层遵守这种模式，否则，如果子类型想自己添加这样的功能，它将变得非常混乱。

12.1.1 标准的销毁语义

在销毁的逻辑中，.NET Framework遵循了一系列实际存在的规则。这些规则并不是硬编码在.NET Framework或C#语言中；它们的目的是为使用者定义一致的协议。它们是：

1. 一旦被销毁，对象无法恢复。对象也不能重新被激活，调用它的方法或属性将抛出ObjectDisposedException异常。
2. 重复调用对象的Dispose方法不会产生异常。
3. 如果可销毁对象*x*包含或“封装”或“占有”可释放资源对象*y*，*x*的Dispose方法自动调用*y*的Dispose方法——除非接收到其他指令。

这些规则对于编写自定义的类也是很有帮助的，尽管不是强制的。没有什么能阻止你编写“不能销毁”的方法，也许除了遭到同事的指责之外。

根据第3条规则，容器对象自动销毁它的子对象。一个典型示例是像Form或Panel控件之类的Windows容器控件。容器控件也许包含很多子控件，尽管没有显式地销毁每一个子控件：关闭或销毁父控件或Form控件将负责这一切。另一个示例是当你将一个FileStream对象包含在一个DeflateStream对象中时。销毁DeflateStream对象的同时也将FileStream对象销毁——除非在构造方法中接到其他指令。

Close方法和Stop方法

除了Dispose方法，一些类还定义了Close方法。.NET Framework对Close方法的语义并不是完全一致，尽管几乎所有的情况都是下面中的一种：

- 从功能上等同于Dispose方法

注1：在第22章“中断和终止”中，我们将介绍终止一个线程是如何破坏这个模式的安全性的。在实践中这通常不是问题，为了这个（或其他的）原因而终止线程是不提倡的。

- 从功能上是Dispose方法的子集

第二种的示例是IDbConnection接口：一个关闭的连接可以被重新打开；一个被销毁的连接则不能。另一个示例是由ShowDialog激活的Windows Form控件：Close隐藏它；Dispose则释放它的资源。

一些类定义了Stop方法（例如Timer或HttpListener），它们可以像Dispose方法一样释放非托管资源，但不同于Dispose方法的是，它允许重新开始。

在WinRT中，Close可以认为与Dispose相同，事实上，运行时会将Close方法映射到Dispose方法上，使它们的类型同样可以在using语句中使用。

12.1.2 何时销毁

要遵守的安全规则（几乎所有的情况下）是"如果有疑问，则销毁"。一个可销毁的对象（如果它能说话）会说：

> 当你不再使用我，请告诉我。如果简单地抛弃我，我也许会对其他的对象实例、应用程序域、计算机、网络或数据库造成麻烦。

包含非托管资源句柄的对象几乎总是要求销毁，目的是为了释放这些句柄。Windows Form控件对象、文件或网络流对象、网络接口对象、GDI+画笔对象、画刷对象和位图对象中都有这样的示例。相反的，如果一个类型是可销毁的，它经常（而非总是）直接或间接地引用非托管句柄。这是因为非托管句柄提供了到操作系统资源、网络连接、数据库锁等"外面世界"的方法，这主要是说如果对象没有适当地被释放，它将在对象之外造成麻烦。

然而，这里有3种情况不能释放：

- 当通过静态字段或属性获得共享对象时
- 当对象的Dispose方法执行不需要的操作时
- 当对象的方法在设计时不是必须的，而且释放那个对象将增加程序的复杂性时

第一种情况很少见。主要的案例就在System.Drawing命名空间中：通过静态字段或属性（像Brushes.Blue）获得的GDI+对象永远不能被销毁，这是因为同样的实例在整个应用程序生命周期中都可能被用到。通过构造方法获得的实例（例如new SolidBrush）应该被销毁。

第二种情况很常见。在System.IO和System.Data命名空间中有一些很好的示例：

种类	销毁的功能	何时不用销毁
MemoryStream	防止进一步输入/输出	当后面还要读/写这个流
StreamReader、StreamWriter	清空读取器/编写器，关闭相关的流	当需要保持相关的流处于打开状态（必须在完成操作之后立即调用StreamWriter的Flush方法）
IDbConnection	释放数据库连接，清空连接字	如果需要重新打开数据库连接，要调用Close方法而不是Dispose方法
DataContext (LINQ to SQL)	防止进一步使用	当有连接到上下文的延后求值查询时

MemoryStream的Dispose方法禁用的只是对象；它没有执行任何重要的清理工作，因为Memory Stream不包含非托管句柄或其他类似的资源。

第三种情况包含下面的类：WebClient、StringReader、StringWriter和BackgroundWorker（在System.ComponentModel中）。这些类型是可释放类型，这是由它们的基类决定的，而不是真的需要执行必要的清理。如果要在整个方法中实例化并使用这样的对象，把它放到using块中将增加一些不方便之处。但是如果这个对象会长时间使用，持续追踪并在不再使用它的时候销毁它将增加不必要的复杂性。在这种情况下，你可以简单地忽略销毁对象。

12.1.3 选择性销毁

因为IDisposable接口使类型在使用C#的using结构时变得更方便，也因此导致扩展IDisposable接口实现不需要的功能。例如：

```
public sealed class HouseManager : IDisposable
{
  public void Dispose()
  {
    CheckTheMail();
  }
  ...
}
```

这个想法是这个类的使用者可以选择规避不必要的清理，只是不调用Dispose方法即可。然而，这依赖于使用者知道在Demo的Dispose方法中都做了什么，这也破坏了需要在后面添加的必要清理工作：

```
public void Dispose()
{
  CheckTheMail();      // 不必要
  LockTheHouse();      // 必要
}
```

这个问题的解决方案就是选择性销毁模式：

```
public sealed class HouseManager : IDisposable
{
  public readonly bool CheckMailOnDispose;

  public Demo (bool checkMailOnDispose)
  {
    CheckMailOnDispose = checkMailOnDispose;
  }

  public void Dispose()
  {
    if (CheckMailOnDispose) CheckTheMail();
    LockTheHouse();
  }
  ...
}
```

使用者总是可以调用Dispose方法，因为它提供了简单性和避免特殊文档或反射的需求。实现这个模式的示例是在System.IO.Compression中的DeflateStream类。下面是它的构造方法：

```
public DeflateStream (Stream stream, CompressionMode mode, bool leaveOpen)
```

不必要的工作是在销毁中关闭内部流（第一个参数）。有时候希望保持内部流处于打开状态，同时仍然销毁DeflateStream来执行它的必要清理工作（清空缓存数据）。

这个模式看起来简单，但是StreamReader和StreamWriter（在System.IO命名空间中）并没有这么做。原因比较复杂：SteamWriter必须公开另一个方法（Flush方法）来保证使用者不调用Dispose方法也能执行必要的清理工作。System.Security.Cryptography中的CryptoStream类也有类似的问题，它要求在保持内部流打开的情况下调用FlushFinalBlock来释放资源。

提示：可以把这个问题描述成所属权问题。对可销毁对象的问题是：我真的拥有我使用的那些潜在资源吗？或者我只是从管理潜在资源的生命周期和通过某些未记录条约来管理我的生命周期的人那里借到那些潜在资源？

下面的选择性模式通过明确地记录所属权条约来避免这个问题。

12.1.4 在销毁时清理字段

总的来说，不需要在对象的Dispose方法中清理它的字段。然而，实践是取消订阅那些对象已经订阅的事件来实现在内部结束它的生命周期（见“托管内存泄露”的示例）。取消订阅这种事件能避免接收不需要的事件通知，并能避免在垃圾回收器（GC）中保持对象存活。

提示：Dispose方法本身并没有释放内存，只有垃圾回收时才释放内存。

设置字段表示对象是否被销毁也是必要的，当试图在使用者之后调用对象的成员时抛出ObjectDisposedException异常。最佳模式是对它使用公开的只读的自动属性：

```
public bool IsDisposed { get; private set; }
```

尽管从技术上并不需要，但是在Dispose方法中清除对象拥有的事件句柄（把它们设置成null）也是有用的。这消除了在销毁时或销毁后激活事件的可能性。

对象有时拥有像密钥等高价值的数据。在这种情况下，在销毁的时候清理这种来自字段的数据就很有意义（避免被没有权限的程序集和恶意软件发现）。System.Security.Cryptography中的SymmetricAlgorithm类就是这么做的，需要调用保存密钥字节数组的Array.Clear方法。

12.2 自动垃圾回收

无论对象是否要求使用Dispose方法来自定义清理逻辑，某些情况下在堆上被占用的内存必须被释放。CLR通过垃圾回收器完全自动地处理这方面工作。永远不能自动释放托管内存。例如，考虑下面的方法：

```
public void Test()
{
  byte[] myArray = new byte[1000];
  ...
}
```

当执行Test方法时，拥有1000个字节的数组在内存堆上被创建。这个数组被变量myArray引用，保存在局部变量栈上。当离开方法时，局部变量myArray跳出区域，意味着没有留下什么东西来引用堆上内存的数组。未引用的数组之后就被垃圾回收器回收。

提示： 在禁用优化的调试模式中，被局部变量引用的对象的生命周期被延长到代码段的结束，这是为了方便调试。否则，它将在最早不再使用之后被垃圾回收器选中回收。

垃圾回收并不是在对象没有引用之后立即执行。与街道上垃圾回收很像，它也是定期进行的，尽管（与街道上垃圾回收不同）回收的计划安排不是固定的。CLR基于一些因素来决定何时收集，这些因素包括可用内存、内存分配的数量和最后一次回收的时间间隔。这意味在对象没有引用和内存释放之间有不确定的延时。理论上说，这段间隔可能从纳秒到几天。

提示： 垃圾回收器在每次回收时并没有回收所有的垃圾。相反的，内存管理器将对象分为不同的代，垃圾回收器收集新代（最近分配的对象）的垃圾比旧代（长时间存活的对象）的垃圾更频繁。我们将在“垃圾回收器如何工作”中详细介绍。

垃圾回收和内存使用

垃圾回收器试图在垃圾回收所花费的时间和应用程序内存使用（工作区）上保持平衡。因此，应用程序会使用比实际需要更多的内存，特别是构造大的的临时数组。

如果你用Windows XP任务管理器的“内存使用率”特性来衡量内存使用，问题会更复杂。不同于最新版本的Windows（只报告个人工作区），XP计算包括进程内部释放的和它即将释放给操作系统而其他进程需要的内存。（内存并没有立即返回给操作系统，这是为避免之后没多久又把它们要回来的总开销。这是因为：如果计算机有足够的空闲内存，为什么不用它减少分配/释放的总开销）

可以通过查询性能计数器（System.Diagnostics）来确定进程的实时内存使用：

```
string procName = Process.GetCurrentProcess().ProcessName;
using (PerformanceCounter pc = new PerformanceCounter
      ("Process", "Private Bytes", procName))
  Console.WriteLine (pc.NextValue());
```

读取性能计数器需要管理员权限。

12.2.1 根

根保持对象存活。如果对象没有直接或间接地由根引用，那么它将被垃圾回收器选中。

根有以下三种：

- 局部变量或执行方法中的参数（或在调用它的栈的方法中）
- 静态变量
- 准备运行终止器的对象（见下一节）

代码不可能执行已删除的对象，因此如果（实例）方法有任何可能被执行，那么它的对象无论如何也会通过上面几种方法中的一种被引用。

注意，相互循环引用的对象组在没有根裁定时被认为是无用的（见图12-1）。换言之，不能被从根对象跟随的指针（引用）访问的对象被认为是不可到达的，因此将会被回收。

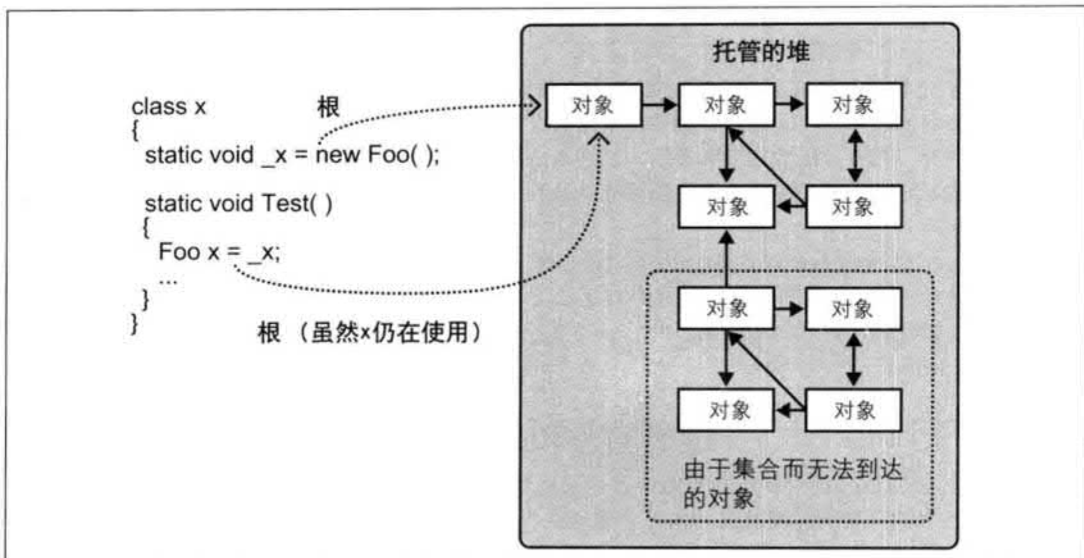


图12-1：根

12.2.2 垃圾回收与WinRT

Windows Runtime依靠COM的引用计数机制来释放内存，而非依靠自动化的垃圾回收器。尽管这样，但是从C#实例化的WinRT对象的生命周期仍然由CLR的垃圾回收器管理，因为CLR通过一个在后台创建的运行时可调用包装（runtime callable wrapper）对象控制COM对象的访问（参见第24章）。

12.3 终止器

在对象从内存中被释放之前，它的终止器将运行（如果它有终止器的话）。终止器像构造方法一样声明，但是它有~符号作前缀：

```
class Test
{
   Test()
  {
    // 终止器逻辑...
  }
}
```

（虽然与构造函数的声明相似，但是析构器无法声明为public或static，不能有参数，而且不能调用基类。）

终止器之所以可行是因为垃圾回收分不同阶段进行。首先，垃圾回收器确认适合删除的不可用对象。没有终止器的对象立刻被删除。准备运行（但还没有运行）终止器的对象保持存活（对于现在来说）并进入特殊队列。

那时，垃圾回收已经完成，程序将继续执行。终止器线程开始并行执行程序，在特殊队列中挑选对象并执行它们的终结方法。在每个对象的终止器执行之前，对象仍然完好地存在，那个队列扮演着根对象的角色。一旦它离开队列，终止器被执行，对象就变成未被引用，并且将在下一次垃圾回收（属于那个对象的一代）中删除。

终止器很有用，但是它有一些附带条件：

- 终止器使分配和内存回收变得缓慢（垃圾回收器将对执行的终止器保持追踪）
- 终止器延长了对象和任意引用对象的生命周期（它们必须等待下一次垃圾回收来实际删除）
- 无法预测终止器以什么顺序调用一系列的对象
- 对对象的终止器何时被调用只有有限的控制
- 如果终止器的代码被阻碍，其他对象也不能被终结
- 如果应用程序没有被完全地卸载，终止器也许会被规避

总之，终止器有点像律师——尽管在有些时候你确实需要它，通常你不想使用它，除非绝对必要。如果确实要使用它，需要100%确定理解它所做的一切。

下面有一些实现终止器的准则：

- 保证终止器执行得很快
- 永远不要在终止器中中断（第21章）
- 不要引用其他可终结对象
- 不要抛出异常

警告： 对象的终止器即使在构造时抛出异常也可以被调用。正因如此，在编写终止器时，要注意不要认为对象的字段一定会被正确地初始化。

12.3.1 在终止器中调用Dispose方法

终止器的一个很好的用途是当忘记对可销毁对象调用Dispose方法的时候提供一个备份；对象迟一点被销毁通常比没有被销毁好。

提示： 记住，在这种模式下，内存和资源的重新分配将同时进行——这是两种可能具有相反作用的情况（除非资源就是内存）。此外，这也会增加析构线程的负载。
当使用者忘记调用Dispose时，这个模式还可以作为后备方法。然而，最好要记录这个错误，以便将来可以修复这个bug。

下面是实现这种用途的常规模式：

```
class Test : IDisposable
{
  public void Dispose()             // 不是虚拟方法
  {
    Dispose (true);
    GC.SuppressFinalize (this);     // 防止GC运行终止器
  }
  protected virtual void Dispose (bool disposing)
  {
    if (disposing)
    {
      // 调用这个实例拥有的其他对象的Dispose()
      // 可以在这里引用其他可终结的对象
      // ...
    }
    // 释放只有这个对象拥有的非托管资源
    // ...
  }
   Test()
  {
    Dispose (false);
  }
}
```

重载Dispose方法接受一个布尔类型的disposing标识。无参数的版本没有被声明成虚拟（virtual）方法，它只是简单地用true作为参数调用的增强版本。

增强版本包含实际的销毁逻辑，它是受保护的（protected）和虚拟的（virtual），这为子类添加它们自己的销毁逻辑提供了安全的方法。Disposing标识意味着Dispose方法被正确地调用而不是终止器的最后方法模式。通常来说，当调用这个方法时设置disposing为true，它不应该引用其他有终止器的对象（因为这种对象也许会被终结并处于不可预测的状态）。这个规则排除了很多情况。下面是当disposing为false时，一些工作仍然执行最终模式的示例：

- 释放任何直接引用操作系统的资源（也许是通过P/Invoke调用Win32 API获得的）
- 删除构造时创建的临时文件

为使这个方法更健壮，任何能够抛出异常的代码都应该在try/catch块之中，异常最好被记录。任何记录都应该尽可能地简单、健壮。

请注意我们在没有参数的Dispose方法中调用了GC.SuppressFinalize方法，这防止当垃圾回收器在之后捕捉这个对象时终止器也同时运行的情况。从技术上讲这并不必要，因为Dispose方法能够接受重复调用。但是，这样可以提高效率，因为允许对象（和它引用的对象）在一个周期中被回收。

12.3.2 复活

假设终止器修改了引用即将销毁对象的活动对象。当下一次垃圾回收发生时（属于那个对象的一代），CLR将认为先前即将销毁的对象不再需要销毁，因此它将避开垃圾回收。这是一种极端的情况，被称为复活。

要解释这种情况，假设我们要编写管理临时文件的类。当这个类的实例被垃圾回收时，我们希望终止

器来删除临时文件。这看起来很简单：

```
public class TempFileRef
{
  public readonly string FilePath;
  public TempFileRef (string filePath) { FilePath = filePath; }

  ~TempFileRef() { File.Delete (FilePath); }
}
```

但是，这里有一个bug：File.Delete方法也许会抛出异常（由于缺少权限或者文件正在被使用）。这种异常能使整个应用程序停止也会阻止其他的终止器运行。我们只能简单地用空的catch块"吞下"异常，但是之后我们永远无法知道有什么地方出错了。调用一些复杂的错误报告API不是我们希望的，因为这会加重终止器线程的负担，阻碍其他对象的回收。我们希望限制终结的过程，使它简单、可靠和快速。

最佳选择是用如下的静态集合记录错误：

```
public class TempFileRef
{
  static ConcurrentQueue<TempFileRef> _failedDeletions
    = new ConcurrentQueue<TempFileRef>();

  public readonly string FilePath;
  public Exception DeletionError { get; private set; }

  public TempFileRef (string filePath) { FilePath = filePath; }

  ~TempFileRef()
  {
    try { File.Delete (FilePath); }
    catch (Exception ex)
    {
      DeletionError = ex;
      _failedDeletions.Enqueue (this);   // 复活
    }
  }
}
```

将对象加到静态_failedDeletions集合中提供给对象另一个裁判（referee），保证对象最终离开集合之前仍然处于活动状态。

提示： ConcurrentQueue<T>是Queue<T>的线程安全版本，它在System.Collections.Concurrent中被定义（见第23章）。以下是一些使用线程安全集合的原因：首先，CLR保留在多于一个线程上并行运行终止器的权利。这意味着当访问像静态集合这种共享资源时，我们必须考虑两个对象一起被终结的可能性。其次，在某些情况下我们希望将一些项目从_failedDeletions退队以便我们可以对它们进行操作。这也是以一种线程安全方式完成的，因为它能够在终止器让其他对象离开集合同时发生。

GC.ReRegisterForFinalize方法

复活对象的终止器将不会第二次执行，除非调用GC.ReRegisterForFinalize方法。

在下面的示例中，我们试图在终止器中删除临时文件（就像上一个示例一样）。但是如果删除失败，

我们重新注册对象使它在下次垃圾回收中再试一次：

```
public class TempFileRef
{
  public readonly string FilePath;
  int _deleteAttempt;

  public TempFileRef (string filePath) { FilePath = filePath; }

  ~TempFileRef()
  {
    try { File.Delete (FilePath); }
    catch
    {
      if (_deleteAttempt++ < 3) GC.ReRegisterForFinalize (this);
    }
  }
}
```

在第三次尝试失败之后，我们的终止器默默地放弃了删除文件。我们通过结合这个示例与之前的示例来增强它的功能，换言之，在第三次失败的时候将它添加到_failedDeletions集合中。

提示： 请注意在终止器方法中只调用一次ReRegisterForFinalize方法。如果调用了两次，对象将会被注册两次并且经历两次终结过程。

12.4 垃圾回收器如何工作

CRL使用分代式标记－紧缩型垃圾回收器来执行存储在托管堆上对象的自动内存管理。垃圾回收器被认为是追踪型垃圾回收器，因为它不会干涉每次对对象的访问，而是立刻激活并追踪存储在托管堆上对象的记录，以此来决定哪些对象被认为是垃圾并被回收。

垃圾回收器通过执行内存分配（通过new关键字）开始一次垃圾回收，在内存分配或者某个内存起始点被分配之后，或者在其他减少应用程序内存的时候。这个过程也可以通过调用System.GC.Collect方法手动开始。在垃圾回收时，所有的线程也许都会被冻结（下一节有更详细的介绍）。

垃圾回收器从根对象引用开始，按对象记录前进，标记它所有接触的对象为可到达的。一旦这个过程结束，所有没有被标记的对象被认为是无用的，将会被垃圾回收器回收。

没有终止器的无用对象将立刻被删除；有终止器的对象将在垃圾回收结束之后在终止器中排队进行处理。这些对象将在下一次对这代对象的垃圾回收过程中被选中回收（除非复活）。

然后将剩余的“活动”对象移到堆的开头（紧缩），释放出更多的对象空间。这种压紧操作有两个目的：避免出现内存片断，允许垃圾回收器在分配新对象时始终在堆的末尾分配内存。这可避免为可能非常耗时的任务维护剩余内存片断的列表。

如果在垃圾回收之后没有足够的内存来分配新的对象，操作系统将无法分配更多的内存，这时将抛出OutOfMemoryException异常。

12.4.1 优化技术

垃圾回收器包含多种优化技术来减少垃圾回收的时间。

1. 分代回收

最重要的优化是垃圾回收器是分代的。这是因为尽管许多对象快速地被分配和删除，但是某些对象长时间活动，并不需要在每次回收时都被追踪。

基本上讲，垃圾回收器将托管堆分为三代。刚刚被分配的对象在*Gen0*里，在一轮回收幸存下来的对象在*Gen1*里，其他所有对象都在*Gen2*里。

CLR将Gen0部分保持在相对较小的空间内（在32位工作站CLR上最大是16MB，典型的大小是几百KB到几MB）。当Gen0部分被填满之后，垃圾回收器引发Gen0的回收，这经常发生。垃圾回收器对Gen1执行相似的内存限制（Gen1扮演着Gen2的缓存角色），因此Gen1的回收也相对地快速和频繁。然而，包括Gen2的完全回收花费更长的时间，发生得不那么频繁。图12-2表明了完全回收的效果。

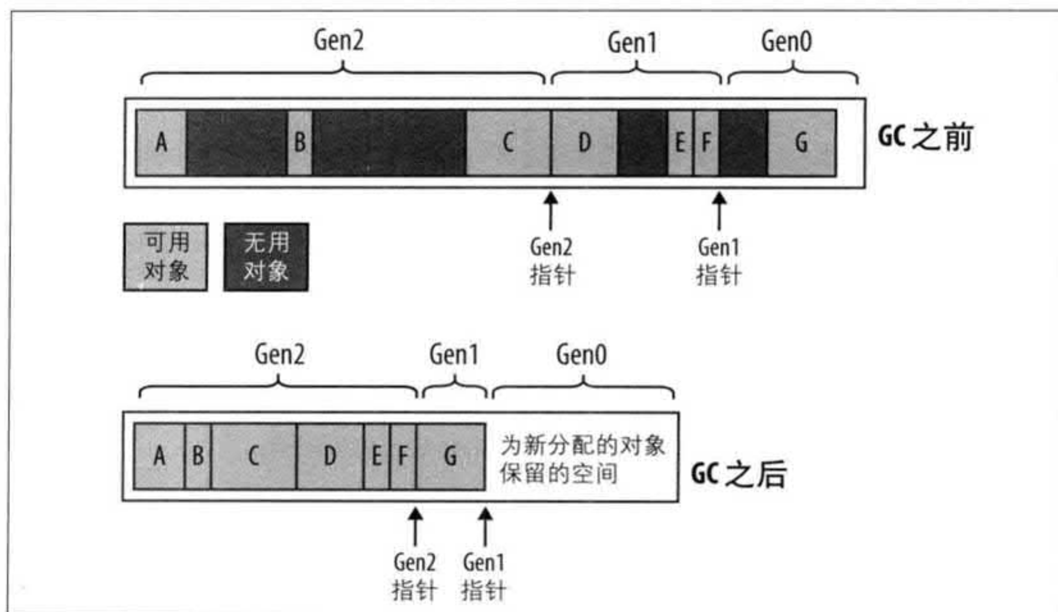


图12-2：堆上的代

给一些非常粗略的估计值，一次Gen0回收大概花费不到1ms的时间，这不足以引起典型应用程序的注意。然而，在有大对象图的程序上，一次完全回收也许要花费100ms。这些数字由很多因素决定，因此可能有明显的差别，特别是在Gen2的大小是无限的情况下（不同于Gen0和Gen1）。

结果是存活周期短的对象非常有效地被垃圾回收器使用。像下面方法创建的`StringBuilders`几乎肯定在快速的Gen0中被回收：

```
string Foo()
{
  var sb1 = new StringBuilder ("test");
  sb1.Append ("...");
  var sb2 = new StringBuilder ("test");
  sb2.Append (sb1.ToString());
  return sb2.ToString();
}
```

2. 大对象堆

对大于某一限度（当前是85,000字节）的对象，垃圾回收器使用特殊的堆即大对象堆。这避免了过多的Gen0回收，分配一系列16MB的对象也许会在每次分配之后引起一次Gen0的回收。

大对象堆不可以被压缩，因为在垃圾回收的时候移动大内存块将花费大量时间。这将会有两种后果：

- 分配将变得更慢，因为垃圾回收器不能总是简单地在堆的末尾分配对象，它也必须查看中间的间隔，这要求维护空闲内存块的连接列表（注2）。
- 大对象堆将导致碎片。这意味着释放对象能在大对象堆上创建很难再被填充的空隙。例如，由86,000字节的对象所留下的空隙只能被85,000字节和86,000字节之间的对象来填充（除非与其他空隙相连）。

大对象堆并不是分代的：所有对象都按Gen2来处理。

3. 并行和后台回收

垃圾回收器在回收的时候必定会冻结（阻止）执行线程一段时间，这包括Gen0和Gen1回收发生的整个时间。

垃圾回收器在Gen2回收时允许线程运行的特殊尝试，这是因为它不希望冻结应用程序潜在的一大段时间。这种优化只应用于工作站版本的CLR，它们用在Windows的桌面版本上（也用在所有有独立应用程序的Windows版本上）。依据是阻止回收的等待时间对于没有用户界面的服务器应用程序来说似乎不是个问题。

提示： 一个减轻的因素是服务器的CLR通过所有可用的内核来执行垃圾回收。因此8核服务器执行完全回收将快好多倍。实际上，服务器的CLR为工作量而进行调试，而不是等待时间。

工作站优化之前被称为并行回收。从CLR 4.0开始，它被修改并从命名成后台回收。后台回收移除了当运行Gen2回收时如果Gen0部分被填充，并行回收将会停止的限制。这意味着从CLR 4.0开始，持续地分配内存的应用程序将更加快速。

4. 垃圾回收通知（服务器CLR）

从.NET Framework 3.5 SP1开始，服务器版本的CLR在完全垃圾回收发生之前通知你。这是为服务器设计的：这种想法是在回收之前将请求转换到其他服务器上。然后立即开始回收，在把请求转移回这台服务器之前完成回收。

要开始通知，请调用`GC.RegisterForFullGCNotification`方法。之后开始另一个线程（见第14章）并首先调用`GC.WaitForFullGCApproach`方法。当这个方法返回`GCNotificationStatus`时表示回收在接近，可以将请求转移到其他服务器上并强制手动回收（见下一节）。之后调用`GC.WaitForFullGCComplete`方法：当这个方法返回结果时，垃圾回收就完成了。可以再次接受请求，重复整个过程。

12.4.2 强制垃圾回收

可以在任何时间通过调用`GC.Collect`方法强制垃圾回收。调用`GC.Collect`方法而没有参数将发起完

注2：由于内存块固定，相同的事情也会在分代的堆上偶然发生（见第4章的“fixed语句”）。

全回收。如果传入一个整数值，只有整数值的那一代将被回收，因此GC.Collect(0)只执行一次快速的Gen0回收。

总的来说，通过允许垃圾回收器来决定何时回收来获得最好的性能。强制回收不必要地将Gen0对象提升到Gen1中，这将降低性能，也将影响垃圾回收器的自我调节能力，即垃圾回收器动态调整每一代回收的开始时间，以保证在应用程序执行的时候性能最大化。

然而，这里有些例外。最常见的干预情况是在应用程序要休眠一段时间的时候：很好的示例是执行日常活动的Windows服务（也许是检查更新）。这样的应用程序也许使用了System.Timers.Timer对象来每24小时初始化某种活动。在完成这种活动之后的24小时内将没有代码执行，意味着在这段时间内，没有内存分配发生，因此垃圾回收没有机会被激活。无论这个服务使用多少内存来执行它的活动，它将在接下来的24小时中继续消费这些内存，即使是空的对象图。解决方案是在每天的活动结束之后立即调用GC.Collect方法。

为避免对象的回收由于终止器而延长，可以采取调用WaitForPendingFinalizers方法和重新回收的额外步骤：

```
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();
```

另一种调用GC.Collect的情况是测试一个拥有析构器的类。

12.4.3 内存压力

运行时基于一些因素来决定何时初始化回收，这些因素包括机器负载的总内存。如果程序分配非托管内存（见第25章），运行时将收到内存使用的不真实的乐观感觉，因为CLR只知道托管内存。可以通过调用GC.AddMemoryPressure方法告诉CLR假设指定数量的非托管内存来减轻这种感觉，（当非托管内存被释放）调用GC.RemoveMemoryPressure方法来取消这种感觉。

12.5 托管内存泄露

在C++这种非托管语言中，必须记得当对象不再被使用时手动释放内存，否则将导致内存泄露。在托管语言中，由于CLR的自动垃圾回收机制，这种类型的错误不可能发生。

尽管如此，大型的复杂的.NET应用程序也存在同样的问题：应用程序在它的生命周期中使用的内存越来越多，直到它最终被重新启动。好消息是托管内存泄露通常更容易诊断和预防。

托管内存泄露是由于不使用的对象因为不使用的或已被遗忘的引用仍然存活着。常见的是事件处理程序，它们保存着目标对象的引用（除非目标是静态方法）。例如，考虑下面的类：

```
class Host
{
  public event EventHandler Click;
}

class Client
{
  Host _host;
  public Client (Host host)
  {
```

```
    _host = host;
    _host.Click += HostClicked;
  }
  void HostClicked (object sender, EventArgs e) { ... }
}
```

下面的测试类包含了实例化1000个clients的方法：

```
class Test
{
  static Host _host = new Host();

  public static void CreateClients()
  {
    Client[] clients = Enumerable.Range (0, 1000)
      .Select (i => new Client (_host))
      .ToArray();

    // 对clients进行某种操作
  }
}
```

你也许希望在CreateClients方法结束执行之后，1000个Client对象将被垃圾回收器选中。但是，每一个client都有一个受托者：_host对象，它的Click事件现在引用了每一个Client实例。这也许在Click事件没有被激活时被忽视，或者HostClicked方法没有执行任何操作。

一种解决方法是使Client实现IDisposable接口，并在Dispose方法中取消事件处理程序的注册：

```
public void Dispose() { _host.Click -= HostClicked; }
```

Client的使用者在完成之后销毁这些实例：

```
Array.ForEach (clients, c => c.Dispose());
```

提示：在“弱引用”一节中，我们将介绍这个问题的另一种解决方法，它在不使用可销毁对象的环境中（WPF就是一个示例）很有用。实际上，WPF框架提供了WeakEventManager类来充分利用那些使用弱引用的模式。

在WPF的主题中，数据绑定是另一个导致内存泄露的常见情况。

12.5.1 定时器

忘记定时器也能造成内存泄露（我们将在第22章介绍定时器）。根据不同类型的定时器，也有两种不同的情况。我们首先来看System.Timers命名空间中的定时器。在下面的示例中，Foo类（当初始化之后）每秒调用一次tmr_Elapsed方法：

```
using System.Timers;

class Foo
{
  Timer _timer;

  Foo()
```

```
  {
    _timer = new System.Timers.Timer { Interval = 1000 };
    _timer.Elapsed += tmr_Elapsed;
    _timer.Start();
  }
  void tmr_Elapsed (object sender, ElapsedEventArgs e) { ... }
}
```

但是Foo实例永远都不会被回收！问题是.NET Framework自己保存引用来激活定时器，因此定时器可以激活它们的Elapsed事件。因此：

- .NET Framework将保持_timer存活
- _timer将通过tmr_Elapsed事件处理器保持Foo实例存活

当你明白Timer类实现了IDisposable接口时，解决方法就很明显了。销毁定时器将停止并保证.NET Fremework不再引用定时器对象：

```
class Foo : IDisposable
{
  ...
  public void Dispose() { _timer.Dispose(); }
}
```

提示：一个很好的准则是如果类中的任何字段被赋值给实现IDisposable接口的对象，类也应该实现IDisposable接口。

WPF和Windows窗体定时器也严格按照我们刚刚介绍的方式工作。

然而，在System.Threading命名空间中的定时器是很特殊的。.NET Framework并不保存引用来激活线程定时器，而是直接用回调委托代替引用。这意味着如果忘记销毁线程定时器，终止器将会被激活，定时器将自动停止并销毁。然而，这将出现一个不同的问题，这个问题能用下面的示例来说明：

```
static void Main()
{
  var tmr = new System.Threading.Timer (TimerTick, null, 1000, 1000);
  GC.Collect();
  System.Threading.Thread.Sleep (10000);    // 等待10秒
}
static void TimerTick (object notUsed) { Console.WriteLine ("tick"); }
```

如果这个示例按照“发布”禁止调试和开启优化的模式编译，定时器将在它甚至只有一次激活机会之前被回收和释放！再次说明，我们可以在完成下面的语句之后销毁定时器来解决这个问题：

```
using (var tmr = new System.Threading.Timer (TimerTick, null, 1000, 1000))
{
  GC.Collect();
  System.Threading.Thread.Sleep (10000);    // 等待10秒
}
```

在using块的结尾隐式调用tmr.Dispose方法将保证tmr变量被“使用”，在这个块结束之前不会被垃圾回收器认为无用。有趣的是，调用这个Dispose方法实际上使对象存活了更长时间！

12.5.2 诊断内存泄露

最容易避免托管内存泄露的方法是当应用程序写内存时主动监视内存使用。你能用下面的语句获得程序对象当前的内存使用（参数true告诉垃圾回收器先执行一次垃圾回收）：

```
long memoryUsed = GC.GetTotalMemory (true);
```

如果从事测试驱动开发，一种可能是用单元测试来断定内存是否按照预想的被回收。如果这种断定失败，之后你只要检查最近的更改。

如果已经有包含托管代码漏洞的大型应用程序，*windbg.exe*工具可以找到它。这里也有一些友好的图形工具，像是Microsoft的CLR Profiler、SciTech的Memory Profiler和Red Gate的ANTS Memory Profiler。

CLR也公开了一些Windows WMI计数器为资源监视提供帮助。

12.6 弱引用

有时，为了使对象存活，保持对象的引用对垃圾回收器“不可见”是很有用的，这叫做弱引用，它是由System.WeakReference类实现的。

要使用WeakReference类，像下面一样用目标对象构造：

```
var sb = new StringBuilder ("this is a test");
var weak = new WeakReference (sb);
Console.WriteLine (weak.Target);     // This is a test
```

如果目标只被一个或多个弱引用引用，垃圾回收器将考虑将目标选中回收。当目标被回收之后，WeakReference类的Target属性将是null：

```
var weak = new WeakReference (new StringBuilder ("weak"));
Console.WriteLine (weak.Target);   // weak
GC.Collect();
Console.WriteLine (weak.Target);   // （空）
```

为避免目标在测试它是否为null和使用它之间被回收，将目标赋值给局部变量：

```
var weak = new WeakReference (new StringBuilder ("weak"));
var sb = (StringBuilder) weak.Target;
if (sb != null) { /* 对sb进行些操作 */ }
```

一旦目标被赋值给局部变量，它将有强根并且在变量被使用时不能被回收。

下面的类使用弱引用来持续追踪所有被实例化的Widget对象，而且不阻止对象被回收：

```
class Widget
{
  static List<WeakReference> _allWidgets = new List<WeakReference>();

  public readonly string Name;

  public Widget (string name)
  {
    Name = name;
```

```
      _allWidgets.Add (new WeakReference (this));
    }
    public static void ListAllWidgets()
    {
      foreach (WeakReference weak in _allWidgets)
      {
        Widget w = (Widget)weak.Target;
        if (w != null) Console.WriteLine (w.Name);
      }
    }
}
```

这种系统的唯一附加条件是静态列表将一直增长，增加有空目标的弱引用。因此需要实现一些清理的策略。

销毁和垃圾回收

12.6.1 弱引用和缓存

使用WeakReference类将缓存大对象图。这允许内存密集数据简洁地被缓存而不引起过多的内存使用：

```
_weakCache = new WeakReference (...);   // _weakCache是一个字段
...
var cache = _weakCache.Target;
if (cache == null) { /* 重新创建缓存并赋给_weakCache */ }
```

这种策略在实践中也许只有有限的效果，因为只对垃圾回收器何时激活和哪一代被选中回收有很少的控制权。特别是，如果缓存仍然在Gen0中，它可能在几毫秒之内就被回收了（记住垃圾回收器只在没有多少内存时才回收，它在正常内存条件下有规律地回收）。因此，至少要使用2级缓存来保存从弱引用转换过来的强引用。

12.6.2 弱引用和事件

在之前我们看到事件是如何导致托管代码泄露的。最简单的解决方案是：或者在这种条件下避免订阅事件或者实现Dispose方法取消订阅。弱引用提供了另一种解决方案。

想象一个委托只保存弱引用它自己的目标。这样的委托不会保持它的目标存活蜒除非这些目标有独立的受托人。当然，这并不会防止激活的委托选中未引用的对象，在目标被选中回收和垃圾回收器清理它之间。要使这种解决方案生效，代码必须在那种情况中依然健壮。假设在那种情况下，弱引用类要按照下面的方式实现：

```
class WeakDelegate<TDelegate> where TDelegate : class
{
  List<WeakReference> _targets = new List<WeakReference>();
  public WeakDelegate()
  {
    if (!typeof (TDelegate).IsSubclassOf (typeof (Delegate)))
      throw new InvalidOperationException
       ("TDelegate must be a delegate type");
  }
  public void Combine (TDelegate target)
  {
    if (target == null) return;
```

```
    foreach (Delegate d in (target as Delegate).GetInvocationList())
      _targets.Add (new WeakReference (d));
  }

  public void Remove (TDelegate target)
  {
    if (target == null) return;
    foreach (Delegate d in (target as Delegate).GetInvocationList())
    {
      WeakReference weak = _targets.Find (w => d.Equals (w.Target));
      if (weak != null) _targets.Remove (weak);
    }
  }

  public TDelegate Target
  {
    get
    {
      var deadRefs = new List<WeakReference>();
      Delegate combinedTarget = null;

      foreach (WeakReference weak in _targets)
      {
        Delegate target = (Delegate)weak.Target;
        if (target != null)
          combinedTarget = Delegate.Combine (combinedTarget, target);
        else
          deadRefs.Add (weak);
      }
      foreach (WeakReference weak in deadRefs)   // 从_targets中删除无目标的引用
        _targets.Remove (weak);

      return combinedTarget as TDelegate;
    }
    set
    {
      _targets.Clear();
      Combine (value);
    }
  }
}
```

这段代码说明了C#和CLR的一些有趣的特点。首先，请注意到我们在构造方法中检查了TDelegate是否是委托类型。这是因为C#中的一个限制——下面的类型限制是不合法的，因为C#认为System.Delegate不支持这些限制的特殊类型：

```
... where TDelegate : Delegate   // 编译器不允许这样做
```

因此，我们必须选择类的限制，并且在构造方法中执行运行时检查。

在Combine和Remove方法中，我们通过as运算符进行从target到Delegate的引用转换，而不是cast运算符。这是因为C#不允许对这种类型参数使用cast运算符，因为在自定义转换和引用转换中存在潜在的多义性。

我们之后调用GetInvocationList方法，这是因为这些方法也许会被多播委托（指多于一个方法接受者的委托）调用。

在Target属性中，我们创建了一个多播委托，这个多播委托包含了所有被目标存活的弱引用所引用的委托。我们之后从列表中清理了剩下的无目标的引用来避免_targets列表无限的增长。我们应该通过在Combine方法中用同样的方式来改进我们的类；另一个改进的方法是为了线程安全而添加锁（见第22章）。

下面的代码说明了如何通过实现事件来使用这个委托：

```
public class Foo
{
  WeakDelegate<EventHandler> _click = new WeakDelegate<EventHandler>();

  public event EventHandler Click
  {
    add { _click.Combine (value); } remove { _click.Remove (value); }
  }

  protected virtual void OnClick (EventArgs e)
  {
    EventHandler target = _click.Target;
    if (target != null) target (this, e);
  }
}
```

注意在激活事件时，我们在检查和调用它之前将_click.Target赋值给一个临时变量。这避免了目标事件在此期间被回收的可能。

第13章

诊断和代码契约

有错误发生时，有相关的信息能够帮助诊断问题是很重要的。IDE或调试器能够极大地帮助诊断问题，但是通常只在开发阶段可用。一旦应用程序发布，应用程序必须收集和记录诊断信息。为了满足这个要求，.NET Framework提供了一系列的工具来记录诊断信息、监视应用程序的运行状态、检测运行时错误、并与可用的调试工具进行集成。

从.NET Framework 4.0开始，还有一组新的类来执行代码契约。这允许方法与一系列契约进行交互，如果违反这些契约，则提前报错。

本章的类主要都在System.Diagnostics和System.Diagnostics.Contracts命名空间中定义。

13.1 条件编译

可以使用预处理器指令有条件地编译C#中的任何代码段。预处理器指令是以#符号开头特殊的编译器指令。不同于其他C#结构体的是，它必须出现在单独的一行。条件编译的预处理指令有#if、#else、#endif和#elif。

#if指令表示编译器将忽略一段代码，除非定义了特定的符号。可以用#define指令或编译开关来定义一个符号。#define指令应用于特定的文件；编译开关应用于整个程序集：

```
#define TESTMODE            // #define指令必须在文件顶端
                            // 按照约定符号名要大写
using System;

class Program
{
  static void Main()
  {
#if TESTMODE
    Console.WriteLine ("in test mode!");     // 输出：in test mode!
#endif
  }
}
```

如果我们删除第一行，程序将从可执行代码中完全忽略Console.WriteLine语句来编译。

#else语句和C#的else语句很类似，#elif等同于#if其后的#else。||、&&和!运算符用于执行或、与和非运算：

```
#if TESTMODE && !PLAYMODE    // 如果是TESTMODE 并且非PLAYMODE
  ...
```

但是要注意，不是在构建普通的C#表达式，操作的符号绝对与变量没有关系，无论是静态的还是其他种类的。

要在程序集范围内定义符号，可在编译时指定/define开关：

```
csc Program.cs /define:TESTMODE,PLAYMODE
```

Visual Studio在“项目属性”中提供了输入条件编译符号的选项。

如果在程序集级别定义了符号，之后想在某些特定文件中取消定义，可使用#undef指令。

13.1.1 条件编辑和静态变量标志位

之前的程序也可以用简单的静态字段来实现：

```
static internal bool TestMode = true;

static void Main()
{
  if (TestMode) Console.WriteLine ("in test mode!");
}
```

它的优点是允许运行时配置。所以，选择条件编译的原因是有些地方条件编译可以，而变量标志位不可以，例如：

- 有条件的包含属性
- 改变变量的声明类型
- 在using指令中的命名空间或类型别名之间转换——例如：

```
using TestType =
  #if V2
     MyCompany.Widgets.GadgetV2;
  #else
     MyCompany.Widgets.Gadget;
  #endif
```

也可以用条件编译指令执行大型的重构，因此可以立即在新旧版本之间转换，编写能在不同Framework版本中编译的库，可以利用最新的Framework特性。

条件编译另一个优点是调试的代码能引用在程序集开发阶段中不包含的类型。

13.1.2 Conditional属性

如果指定的符号没有被定义，Conditional属性指示编译器忽略任何特定类或方法的调用。

来看看这种属性如何应用，假设编写了像下面这样记录状态信息的方法：

```
static void LogStatus (string msg)
{
  string logFilePath = ...
  System.IO.File.AppendAllText (logFilePath, msg + "\r\n");
}
```

现在假设希望只在LOGGINGMODE符号被定义时执行这个方法。第一种解决方案是在所有调用LogStatus方法的前后加上#if指令：

```
#if LOGGINGMODE
LogStatus ("Message Headers: " + GetMsgHeaders());
#endif
```

这提供了理想的结果，但比较单调。第二种解决方案是在LogStatus方法内包含#if指令。然而，按照下面的方式调用LogStatus方法时会出现问题：

```
LogStatus ("Message Headers: " + GetComplexMessageHeaders());
```

GetComplexMessageHeaders方法总是被调用，这可能引起性能问题。

通过给LogStatus方法附加Conditional属性（在System.Diagnostics中定义）来结合第一种解决方案的功能性和第二种解决方案的方便性：

```
[Conditional ("LOGGINGMODE")]
static void LogStatus (string msg)
{
  ...
}
```

这个属性指示编译器隐式地在所有调用LogStatus方法的地方加上#if LOGGINGMODE指令。如果LOGGINGMODE符号没有被定义，所有LogStatus方法的调用都在编译时被忽略——包括它们的参数计算表达式。即使LogStatus和调用方在不同的程序集中也有效。

提示：[Conditional]的另一个好处是条件性检测在调用方被编译时执行，而不是在调用的方法被编译时。并且这种方式允许编写包含像LogStatus方法的库，并只创建一种版本。

Conditional属性在运行时被忽略，因为它仅仅是给编译器的指令而已。

Conditional属性的替代方法

如果需要在运行时动态地启用或禁用某种功能，Conditional属性将毫无用处，而是必须使用基于变量的方法。这就将问题转换成如何在调用条件性日志方法时巧妙地规避参数计算。函数型方法解决了这个问题：

```
using System;
using System.Linq;

class Program
{
  public static bool EnableLogging;

  static void LogStatus (Func<string> message)
  {
```

```
        string logFilePath = ...
        if (EnableLogging)
          System.IO.File.AppendAllText (logFilePath, message() + "\r\n");
      }
    }
```

Lambda表达式可以在调用这个方法时避免语法臃肿：

```
LogStatus ( () => "Message Headers: " + GetComplexMessageHeaders() );
```

如果EnableLogging为false，GetComplexMessageHeaders永远也不会被计算。

13.2 Debug和Trace类

Debug和Trace是提供基本日志和断言功能的静态类。这两个类很类似，主要的不同是它们的特定用途。Debug类用于调试版本；Trace类用于调试和发布版本。为了实现相应的功能：

所有Debug类的方法都用[Conditional("DEBUG")]定义；

所有Trace类的方法都用[Conditional("TRACE")]定义。

这意味着所有调用标记为DEBUG或TRACE的方法都会被编译器忽略，除非定义了DEBUG或TRACE符号。默认情况下，Visual Studio在项目的调试配置中定义了DEBUG和TRACE符号，同时只在发布配置中定义了TRACE符号。

Debug和Trace类都提供了Write、WriteLine和WriteIf方法。默认情况下，这些方法向调试器的输出窗口发送消息：

```
Debug.Write     ("Data");
Debug.WriteLine (23 * 34);
int x = 5, y = 3;
Debug.WriteIf   (x > y, "x is greater than y");
```

Trace类也提供了TraceInformation、TraceWarning和TraceError方法。这些方法和Write方法在行为上的不同取决于TraceListeners类（我们将在“TraceListener”中介绍）。

13.2.1 Fail和Assert方法

Debug和Trace类都提供了Fail和Assert方法。Fail方法给每一个在Debug或Trace类的Listeners集合中的TraceListener发送消息（见下一节），默认在调试输出窗口和对话框中显示消息：

```
Debug.Fail ("File data.txt does not exist!");
```

弹出的对话框询问是忽略、终止还是重试。后者允许附加一个调试器，这在立即诊断问题时很有用。

Assert方法在布尔参数为false时仅仅调用Fail方法，这叫做使用断言。指定错误消息也是可选的：

```
Debug.Assert (File.Exists ("data.txt"), "File data.txt does not exist!");
var result = ...
Debug.Assert (result != null);
```

Write、Fail和Assert方法也被重载来接受字符串类型的额外消息，这在处理输出时很有用。

另一种使用断言的方法是在相反的条件成立时抛出异常。这是验证方法参数的通用示例：

```
public void ShowMessage (string message)
{
  if (message != null) throw new ArgumentNullException ("message");
  ...
}
```

这些“断言”会无条件地编译，而且不够灵活，因为无法通过TraceListeners控制错误断言的输出结果。在技术上，它们也不是真正的断言。断言指的是，如果出错，那么当前方法的代码存在bug。根据参数验证抛出一个异常，可以反映调用者代码所存在的bug。

提示： 我们将在后面看到代码契约扩展Fail和Assert方法的原则，并提供更多的功能和灵活性。

13.2.2 TraceListener类

Debug和Trace类都有Listeners属性，包含了TraceListener实例的静态集合。它们负责处理由Write、Fail和Trace方法发起的内容。

默认情况下，每一个Listeners集合包含一个单独的监听器（DefaultTrace-Listener）。默认的监听器有两个关键特性：

- 当连接到像Visual Stuido这种调试器时，消息将被输出到调试输出窗口中；否则，消息内容将被忽略。
- 当Fail方法被调用（或断言失败）时，弹出对话框询问用户是继续、忽略还是重试（附加/调试），无论是否已经附加了调试器。

可以通过移除默认监听器（可选的），然后添加一个或多个监听器来改变这种行为。可以编写追踪监听器（通过继承TraceListener类）或使用一个预定义的类型：

- TextWriterTraceListener写入Stream实例或TextWriter实例或附加到某个文件
- EventLogTraceListener写入Windows事件日志中
- EventProviderTraceListener写入操作系统的Windows事件追踪（Event Tracing for Windows，ETW）子系统中
- WebPageTraceListener写入某个ASP.NET网页

TextWriterTraceListener类进一步划分成ConsoleTraceListener类、DelimitedList-TraceListener类、XmlWriterTraceListener类和EventSchemaTraceListener类。

提示： 这些监听器除了DefaultTraceListener在Fail方法被调用时都不显示对话框。

下面的示例清除Trace的默认监听器，之后添加了3个监听器——其中一个附加到文件，另一个输出到控制台，最后一个输出到Windows事件日志：

```
// 清除默认监听器：
Trace.Listeners.Clear();
```

```
// 添加附加到trace.txt文件的编写器：
Trace.Listeners.Add (new TextWriterTraceListener ("trace.txt"));

// 获得控制台的输出流，之后把它添加为监听器：
System.IO.TextWriter tw = Console.Out;
Trace.Listeners.Add (new TextWriterTraceListener (tw));

// 设置Windows事件日志的源，之后创建/添加监听器
// CreateEventSource需要管理员权限，通常都在应用程序设置中完成
if (!EventLog.SourceExists ("DemoApp"))
    EventLog.CreateEventSource ("DemoApp", "Application");

Trace.Listeners.Add (new EventLogTraceListener ("DemoApp"));
```

（另外，还可以通过应用配置文件添加监视器；这样测试者可以很方便地在应用创建之后配置跟踪。详情参见MSDN文章：http://albahari.com/traceconfig）。

对于Windows事件日志，通过Write、Fail或Assert方法输出的消息在Windows事件查看器中总是显示为"消息"。但是，通过TraceWarning和TraceError方法输出的消息，则显示为"警告"或"错误"。

TraceListener也有一个TraceFilter类型的筛选器，能设置消息是否输出给监听器。必须通过实例化其中一种预定义子类（EventTypeFilter或SourceFilter）或继承TraceFilter类并重写ShouldTrace方法来实现。例如，可以用它来按种类进行筛选。

TraceListener类也定义了IndentLevel和IndentSize属性来控制缩进，TraceOutput Options属性来写入额外的数据：

```
TextWriterTraceListener tl = new TextWriterTraceListener (Console.Out);
tl.TraceOutputOptions = TraceOptions.DateTime | TraceOptions.Callstack;
```

TraceOutputOptions应用在使用Trace方法时：

```
Trace.TraceWarning ("Orange alert");
```

DiagTest.vshost.exe Warning: 0 : Orange alert
DateTime=2007-03-08T05:57:13.6250000Z
Callstack= at System.Environment.GetStackTrace(Exception e, Boolean needFileInfo)
at System.Environment.get_StackTrace() at ...

13.2.3 清除和关闭监听器

一些像TextWriterTraceListener这样的监听器最终写入流中被缓存管理。这有两种含义：

- 消息也许不会在输出流或文件中立即出现。
- 在应用程序结束之前，必须关闭或至少清除监听器；否则，将失去缓存中的内容（如果输出到文件的话，默认最多4K）。

Trace和Debug类提供了静态的Close和Flush方法来调用所有监听器的Close和Flush方法（依次调用它所属的编写器和流的Close或Flush方法）。Close方法隐式地调用Flush方法，关闭文件句柄，防止数据进一步被写入。

作为一般的规则，要在应用程序结束之前调用Close方法，随时调用Flush方法来保证当前的消息数据被写入。这适用于使用流或基于文件的监听器。

Trace和Debug类也提供了AutoFlush属性，如果它为true，则在每条消息之后强制执行Flush方法。

警告： 如果使用任何文件或基于流的监听器，将AutoFlush设为true是很好的方法。否则，如果任何未处理的异常或关键的错误发生，最后4KB的诊断信息也许会丢失。

13.3 代码契约概述

我们之前提到断言的概念，通过它检查某些条件在整个程序中是否满足。如果条件不满足，它表示一个bug，通常应该请求调试器处理（在调试版本下）或抛出异常（在发布版本下）。

断言遵守的准则如果出现问题，最好尽早结束并关闭错误源。通常这比试图继续使用不合法的数据导致产生不正确的结果或之后在程序中产生异常结束（很难诊断错误）要好得多。

曾经有两种方法来强制断言：

- 通过调用Debug或Trace类的Assert方法
- 通过抛出异常（如ArgumentNullException）

Framework 4.0提供了叫做代码契约的新特性，用统一的系统代替了这些方法。这种系统不但支持简单的断言，也支持更加强大的基于契约的断言。

代码契约由Eiffel编程语言中的契约式设计原则而来，函数之间通过相互有义务和好处的系统进行交互。本质上讲，客户端（调用方）必须满足函数指定的先决条件和保证当函数返回时客户端能够依赖的后置条件。

代码契约的类型存在于System.Diagnostics.Contracts命名空间中。

提示： 尽管支持代码契约的类型内建于.NET Framework 4.0之中，但是二进制重写器和静态检查工具可以在Microsoft DevLabs站点（http://msdn.microsoft.com/devlabs）下载。在Visual Studio 2010中使用代码契约之前必须下载这些工具。

13.3.1 为什么使用代码契约

要解释这个问题，我们编写将不存在的项目添加到列表中的方法，它包含2个先决条件和1个后置条件：

```
public static bool AddIfNotPresent<T> (IList<T> list, T item)
{
  Contract.Requires (list != null);          // 先决条件
  Contract.Requires (!list.IsReadOnly);      // 先决条件
  Contract.Ensures (list.Contains (item));   // 后置条件

  if (list.Contains(item)) return false;
  list.Add (item);
  return true;
}
```

先决条件由Contract.Requires定义，它在方法开始时被验证。后置条件由Contract.Ensures定义，它并不在它出现的地方被验证，而是当方法结束时被验证。

在这种情况下，先决条件和后置条件像断言和检测可以发现下面的错误：

- 用null或只读列表调用方法
- 我们忘记将项目添加到列表时方法中的bug

提示： 先决条件和后置条件必须出现在方法的开始。优点是如果没有在按顺序编写的方法中实现契约，错误就会被检测出来。

此外，这些条件将为这个方法形成一个可发现的契约。AddIfNotPresent方法向用户公布：

- 必须用非空可写的列表调用方法
- 当方法返回时，那个列表将包含指定的项目

这些情况能添加到程序集的XML文档文件中（能通过在Visual Studio中项目属性窗口中的代码契约标签中，开启契约引用程序集模式，勾选"Emit Contracts into XML doc file"）。像SandCastle这种工具能将契约的详细信息加入到文档文件中。

通过静态契约验证工具，契约也能够进行程序正确性分析。例如，当用值可能为空的列表调用AddIfNotPresent方法时，静态验证工具能够在运行程序之前发出警告。

契约的另一个优点是容易使用。在我们的示例中，直接编写后置条件比在所有离开点编写更容易。契约也支持对象不变式，这将进一步减少重复编写代码和更可靠的执行。

条件也能被接口成员和抽象方法所替代，有些不可能用标准验证方法来实现。虚方法中的条件不能通过子类有意规避。

代码契约另一个好处是契约验证行为能够被轻松地自定义，有很多方法而不是仅仅依赖于调用Debug.Assert方法或抛出异常。它也能够保证契约验证总是被记录——即使契约验证异常在调用堆中被更高级的异常处理器所掩盖。

使用代码契约的缺点是.NET的实现代码契约依赖于二进制重写器这个在编译之后改变程序集的工具。这将减慢编译过程和使依赖于调用C#编译器的服务更复杂（无论是显式的或通过CSharpCodeProvider类）。

实现代码契约也会影响运行时性能，尽管在发布模式通过减小契约检查的规模能轻松减少性能损失。

警告： 代码契约的另一个限制是不能用它们来执行安全性检查，因为它们在运行时被规避（通过处理ContractFailed事件）。

13.3.2 契约原则

代码契约由先决条件、后置条件、断言和对象不变式组成。这些都是可发现的断言。不同之处是它们何时被验证：

- 先决条件在函数开始时被验证。
- 后置条件在函数结束之前被验证。

- 断言在它出现的地方被验证。
- 对象不变式在每个类中的公有函数之后被验证。

代码契约完全通过调用Contract类中的（静态）方法来定义，这与契约语言无关。

契约不仅在方法中出现，也可以在其他函数中出现，例如构造方法、属性、索引器和运算符。

1. 编译

Contract类中几乎所有的方法都定义了[Conditional("CONTRACTS_FULL")]属性。这意味着除非定义了CONTRACTS_FULL符号，否则（大多数）契约代码将被消除。如果在项目属性页中的代码契约标签中开启了契约检查，Visual Studio将自定义CONTRACTS_FULL符号（要是这个标签出现，必须从微软DevLabs站点下载和安装契约工具）。

警告： 移除CONTRACTS_FULL符号关闭所有的契约检查看起来很容易。然而，它并不适用于Requires<TException>条件（我们将在后面详细介绍）。

使用Requires<TException>的代码关闭契约的唯一方法是开启CONTRACTS_FULL符号，之后通过选择强制级别为“none”来使二进制重写器消除契约代码。

---

2. 二进制重写器

在编译好包含契约的代码之后，必须调用二进制重写器工具，*ccrewrite.exe*（如果开启契约检查，Visual Studio将自动调用它）。二进制重写器将后置条件（和对象不变式）移到正确的位置，在重写的方法中调用其他条件和对象不变式，用调用契约运行时类替换调用Contract类。这里是我们之前的示例在重写之后的简化版本：

```
static bool AddIfNotPresent<T> (IList<T> list, T item)
{
  _ContractsRuntime.Requires (list != null);
  _ContractsRuntime.Requires (!list.IsReadOnly);
  bool result;
  if (list.Contains (item))
    result = false;
  else
  {
    list.Add (item);
    result = true;
  }
  _ContractsRuntime.Ensures (list.Contains (item));   // 后置条件
  return result;
}
```

如果调用二进制重写器失败，Contract将不会被_ContractsRuntime替代，Contract也将以抛出异常而结束。

提示： _ContractsRuntime类型是默认的契约运行时类。在高级情况下，能通过/rw开关或Visual Studio的项目属性页中的代码契约标签来指定契约运行时类。

因为_ContractsRuntime是由二进制重写器包装的（并不是.NET Framework的标准部分），实际上二

进制重写器将_ContractsRuntime类嵌入到编译好的程序集中。可以反编译任何开启代码契约的程序集来检查它的代码。

二进制重写器也提供开关来剥夺一些或所有的契约检查：我们将在“选择实行契约”中介绍。通常在调试配置时开启了完全契约检查，而在发布配置时开启了一部分契约检查。

3. 断言和失败时抛出异常的对比

当契约失败时，二进制重写器选择是显示对话框还是抛出ContractException异常。第一种情况通常适用于调试模式；第二种适用于发行模式。要实行第二种模式，在调用二进制重写器时指定/throwonfailure，或者在Visual Studio项目属性中的代码契约标签中取消“Assert on contract failure”复选框。见“处理契约失败”。

我们将在第13章“处理契约错误”中更深入介绍这个话题。

4. 单纯性

调用的所有作为参数传递给契约方法（Requires、Assumes、Assert等方法）的函数必须是单纯的，只能单方面起作用（不能改变字段的值）。指定二进制重写器调用的函数必须是单纯的，要使用[Pure]属性：

```
[Pure]
public static bool IsValidUri (string uri) { ... }
```

这使下面的代码合法：

```
Contract.Requires (IsValidUri (uri));
```

契约工具隐含假设所有能访问的属性是单纯的，例如所有的C#运算符（+、*、%等）和选中的Framework类型成员，包括string、Contract、Type、System.IO.Path和LINQ查询运算符。它也假设通过委托调用的有[Pure]属性的方法是单纯的（Comparison<T>和Predicate<T>都标记了这个属性）。

13.4 先决条件

可以通过调用Contract.Requires、Contract.Requires<TException>或Contract.EndContractBlock方法来定义代码契约先决条件。

13.4.1 Contract.Requires方法

在函数开始调用Contract.Requires方法时将实行先决条件：

```
static string ToProperCase (string s)
{
  Contract.Requires (!string.IsNullOrEmpty(s));
  ...
}
```

这就像使用一个断言，除了先决条件会对函数形成可发现的信息外。这些信息能从编译好的代码中取得，并能被文档和静态检测工具使用（因此它能警告你在程序的某个地方试图用null或空字符串来调

用ToProperCase方法）。

先决条件的优点是包含重写先决条件虚方法的子类无法阻止基类成员的先决条件被检测。在接口成员中定义的先决条件将被隐式地植入到具体实现之中（见“接口和抽象方法中的契约”）。

警告： 先决条件只应该访问至少函数自身能访问的成员，这保证了调用方能够理解契约。如果需要读取或调用访问权更少的成员，这看起来像验证内部状态而不是实现调用契约。在这种情况下最好使用断言。

可以在方法开始时调用需要次数的Contract.Requires方法来实现不同的条件。

先决条件应该放什么？

代码契约团队的指导方针是先决条件应该：

- 尽可能地让调用方容易验证
- 只依赖至少方法本身能访问的数据和函数
- 如果违反验证，总是指出一个bug

最后一点的结果是客户永远不能明确地捕捉到契约失败（事实上，ContractException类型在内部来帮助实现这个原则）。相反的，客户应该正确地调用目标；如果它失败了，这表示一个应该通过异常保障来处理的bug（也许会是应用程序终止）。换句话说，如果决定控制流程或执行其他基于先决条件失败的操作，它并不是真的契约，因为它在失败之后可以继续执行。

当选择先决条件和抛出异常时，有以下建议：

- 如果失败总是指出客户端bug，则倾向先决条件。
- 如果失败指出不正常的条件，也许意味着客户端bug，则抛出（可捕捉的）异常。

要解释它，假设我们正在编写Int32.Parse函数。假设空输入字符串总是指出调用方的bug是可以理解的，因此我们执行一个先决条件：

```
public static int Parse (string s)
{
  Contract.Requires (s != null);
}
```

接下来，我们要检测字符串是否包含数字和像+和-这样的符号（在正确的位置）。调用方验证这些合成为不可理解的负担，因此不能作为先决条件来实行，只能手动检测，如果违反则抛出（可捕捉的）FormatException异常。

要解释成员访问权的问题，考虑一下下面的代码，它经常出现在实现IDisposable接口的类型中：

```
public void Foo()
{
  if (_isDisposed)  // _isDisposed是私有字段
    throw new ObjectDisposedException();
  ...
}
```

这种检查是不能作为先决条件的，除非我们让调用方能访问_isDisposed（例如，使它重构为公开可读的属性）。

最后，考虑File.ReadAllText方法。下面是先决条件的不适当的用法：

```
public static string ReadAllText (string path)
{
    Contract.Requires (File.Exists (path));
    ...
}
```

调用方在调用这个方法之前不能完全知道文件是否存在（它能在检测和调用方法之间被删除）。因此我们要用旧的方式即通过抛出可捕捉的FileNotFoundException异常来代替。

13.4.2 Contract.Requires<TException>

代码契约的引入质疑了下面牢固确立的模式，这种模式从.NET Framework 1.0就被建立了：

```
static void SetProgress (string message, int percent)  // 传统的方法
{
  if (message == null)
    throw new ArgumentNullException ("message");
  if (percent < 0 || percent > 100)
    throw new ArgumentOutOfRangeException ("percent");
  ...
}

static void SetProgress (string message, int percent)  // 现代的方法
{
  Contract.Requires (message != null);
  Contract.Requires (percent >= 0 && percent <= 100);
  ...
}
```

如果有很大的程序集执行传统代码检查，用先决条件编写新的方法将创建不一致的库：一些方法将抛出参数异常而另一些将抛出ContractException异常。一种解决方案是更新所有的方法使它们使用契约，但是存在两个问题：

- 花费时间长。
- 调用方已经开始依赖抛出某种异常类型，像ArgumentNullException异常（这几乎肯定表示不好的设计，但是也许是事实）。

解决方案是调用Contract.Requires的泛型版本，在失败时能指定异常类型：

```
Contract.Requires<ArgumentNullException> (message != null, "message");
Contract.Requires<ArgumentOutOfRangeException>
 (percent >= 0 && percent <= 100, "percent");
```

（第二个参数传递给异常类的构造方法）。

这样的结果是它和旧方式的参数检查有同样的行为，同时也引用了契约的好处（简明、支持接口、隐式的文档、静态检查和运行时自定义）。

提示：只有在指定/throwonfailure（或者在Visual Studio中取消“Assert on Contract Failure”复选框）时，才会抛出指定的异常。否则，将出现对话框。

也可以在二进制重写器中指定ReleaseRequires的契约检测级别（见“选择实行契约”）。当所有其他检测被剥夺之后，调用的泛型Contract.Requires<TException>方法仍然存在，这使程序集同过去一样运转。

13.4.3 Contract.EndContractBlock方法

Contract.EndContractBlock方法使用传统的参数检查代码来获得代码契约，是为了避免在Framework 4.0中重构之前写的代码。所要做的就是在执行完手动参数检测之后执行这个方法：

```
static void Foo (string name)
{
  if (name == null) throw new ArgumentNullException ("name");
  Contract.EndContractBlock();
  ...
}
```

二进制重写器之后将代码转换成：

```
static void Foo (string name)
{
  Contract.Requires<ArgumentNullException> (name != null, "name");
  ...
}
```

EndContractBlock方法之前的代码必须包含下面形式的简单语句：

```
if 条件 throw 表达式;
```

能混合使用传统的参数检查和代码契约调用，只需简单地把后者放到前者之后：

```
static void Foo (string name)
{
  if (name == null) throw new ArgumentNullException ("name");
  Contract.Requires (name.Length >= 2);
  ...
}
```

调用任何执行契约的方法都将隐式结束契约块。

重点是在方法开始定义一块区域让契约重写器知道每个if语句是契约的一部分。调用任何执行契约方法隐式延长契约块，因此如果使用像Contract.Ensures等其他方法，就不需要使用EndContractBlock方法了。

13.4.4 先决条件和重写的方法

重写的方法不能添加先决条件，因为这样做将改变契约（使它有更多限制），违反了多态性原则。

（从技术上讲，设计者能允许重写方法来减弱先决条件；他们反对这样做是因为这种情况没有足够让人信服的理由来抵消增加的复杂性）。

提示：无论重写的方法是否调用了基方法，二进制重写器能保证基方法的先决条件总是在子类中被执行。

13.5 后置条件

13.5.1 Contract.Ensures方法

Contract.Ensures方法执行后置条件：在方法结束时某些条件必须为true。我们在之前就看到了一个示例：

```
static bool AddIfNotPresent<T> (IList<T> list, T item)
{
  Contract.Requires (list != null);          // 先决条件
  Contract.Ensures (list.Contains (item));    // 后置条件
  if (list.Contains(item)) return false;
  list.Add (item);
  return true;
}
```

二进制重写器将后置条件转移到方法结束时。后置条件是检查是否从方法提前返回（就像刚才的示例），而不是是否通过未处理的异常而提前返回。

不同于检查调用方错误使用的先决条件，后置条件检查函数本身的错误（就像断言）。因此，后置条件也许会访问私有状态（简而言之就是警告状态，将在“后置条件和重写的方法”中介绍）。

后置条件和线程安全

多线程（见第14章）的情况质疑了后置条件的可用性。例如，假设我们写了下面包含List<T>的线程安全方法：

```
public class ThreadSafeList<T>
{
  List<T> _list = new List<T>();
  object _locker = new object();

  public bool AddIfNotPresent (T item)
  {
    Contract.Ensures (_list.Contains (item));
    lock (_locker)
    {
      if (_list.Contains(item)) return false;
      _list.Add (item);
      return true;
    }
  }

  public void Remove (T item)
  {
    lock (_locker)
      _list.Remove (item);
  }
}
```

AddIfNotPresent方法中的后置条件在释放锁之后才被检查，如果其他的线程之后调用了Remove方法，此时这个项目也许不存在于列表中了。对于这个问题目前没有解决方案，不能使用后置条件，只能按断言（见下一节）执行这种条件。

13.5.2 Contract.EnsuresOnThrow<TException>

有时候，保证某些条件下特定类型异常被抛出是很有用的。可以通过EnsuresOnThrow方法来实现：

```
Contract.EnsuresOnThrow<WebException> (this.ErrorMessage != null);
```

13.5.3 Contract.Resule<T>和Contract.ValueAeRetum<T>

因为后置条件在函数结束之后才被检查，访问方法的返回值是合理的。可以通过Contract.Result<T>方法来实现：

```
Random _random = new Random();
int GetOddRandomNumber()
{
  Contract.Ensures (Contract.Result<int>() % 2 == 1);
  return _random.Next (100) * 2 + 1;
}
```

Contract.ValueAtReturn<T>方法实现了同样的函数，但适用于ref和out参数。

13.5.4 Contract.OldValue<T>方法

Contract.OldValue<T>方法返回方法参数的原始值。这对后置条件很有用，因为它在函数结束后才被检查。因此后置条件中的任何包含参数的表达式都将读取修改之后的参数值。

例如，下面方法的后置条件总是失败的：

```
static string Middle (string s)
{
  Contract.Requires (s != null && s.Length >= 2);
  Contract.Ensures (Contract.Result<string>().Length < s.Length);
  s = s.Substring (1, s.Length - 2);
  return s.Trim();
}
```

这里我们改写为：

```
static string Middle (string s)
{
  Contract.Requires (s != null && s.Length >= 2);
  Contract.Ensures (Contract.Result<string>().Length <Contract.OldValue (s).Length);
  s = s.Substring (1, s.Length - 2);
  return s.Trim();
}
```

13.5.5 后置条件和重写的方法

重写的方法不能回避基类定义的后置条件，但是它能够添加新的后置条件。二进制重写器保证基类方

法的后置条件总是被检查，即使重写方法并没有调用基类的实现。

警告： 由于上述原因，虚方法中的后置条件不能访问私有成员。如果二进制重写器改变代码让子类试图访问基类中的私有成员，这将导致运行时错误。

13.6 断言和对象不变式

除了先决条件和后置条件，代码契约API还可以使用断言和定义对象不变式。

13.6.1 断言

1. Contract.Assert方法

可以通过调用Contract.Assert方法在函数中任意地方使用断言。如果断言失败，也可以选择性地指定错误信息：

```
...
int x = 3;
...
Contract.Assert (x == 3);                // 除非x等于3，否则失败
Contract.Assert (x == 3, "x must be 3");
...
```

二进制重写器并不会移动断言。以下两个原因使Contract.Assert比Debug.Assert更受欢迎：

- 通过代码契约提供的失败处理机制能获得更多的灵活性。
- 静态检测工具能尝试验证Contract.Asserts。

2. Contract.Assume方法

Contract.Assume方法在运行时执行与Contract.Assert方法一样的功能，对于静态检测工具来说实现方式稍有不同。本质上讲，静态检测工具不会质疑假设，然而也许会质疑断言。这对于总有静态检测工具无法证明的方法来说很有用，也许会导致合法的断言失败。将断言换成假设将使静态检测工具不进行质疑。

13.6.2 对象不变式

对于类，可以指定一个或多个对象不变式方法。这些方法在类中每个公有函数之后自动执行，并且允许断言对象处于内部的一致状态。

提示： 为了支持多对象不变式方法，对象不变式在局部类中工作正常。

要定义对象不变式方法，需要编写没有参数和返回值的方法，并标识[Contract-InvariantMethod]属性。在方法中调用Contract.Invariant方法来强制每个条件都保持为真，如果对象是合法状态的话：

```
class Test
```

诊断和代码契约

```
{
  int _x, _y;

  [ContractInvariantMethod]
  void ObjectInvariant()
  {
    Contract.Invariant (_x >= 0);
    Contract.Invariant (_y >= _x);
  }

  public int X { get { return _x; } set { _x = value; } }
  public void Test1() { _x = -3; }
  void Test2()        { _x = -3; }
}
```

二进制重写器将X属性，Test1方法和Test2方法转变成等同于下面的代码：

```
public void X { get { return _x; } set { _x = value; ObjectInvariant(); } }
public void Test1() { _x = -3; ObjectInvariant(); }
void Test2()        { _x = -3; }    // 不能改变，因为它是私有的
```

提示： 对象不变式并没有防止对象进入不合法状态，只是检测条件何时发生。

Contract.Invariant方法和Contract.Assert方法类似，不同的是它只能出现在标识着[ContractInvariantMethod]属性的方法中。相反的，契约不变式方法只能包含Contract.Invariant方法的调用。

子类也能引入它自己的对象不变式方法，除了基类的不变式方法，子类的也会被检测。当然，要注意对象不变式的检测只在公有方法被调用之后执行。

13.7 接口和抽象方法中的契约

代码契约的一个强大特性是可以将条件加入到接口成员和抽象方法中。二进制重写器之后自动地将这些条件添加到成员的具体实现中。

一种特殊的机制使接口和抽象方法指定独立的契约类，这样就可以编写包含契约条件的方法体。下面就是它的工作实现：

```
[ContractClass (typeof (ContractForITest))]
interface ITest
{
  int Process (string s);
}

[ContractClassFor (typeof (ITest))]
sealed class ContractForITest : ITest
{
  int ITest.Process (string s)      // 必须显式调用
  {
    Contract.Requires (s != null);
    return 0;                        // 返回任意的值来满足编译器
  }
}
```

注意，实现ITest.Process方法必须返回一个值来满足编译器。然而，返回0的代码将不会执行。相反，二进制重写器从方法中提取这个条件，添加到ITest.Process的具体实现中。这意味着契约类永远不会被实例化（编写的任何构造方法也不会被执行）。

可以通过在契约块中给临时变量赋值来更容易地引用接口中的其他成员。例如，如果我们的ITest接口也定义了string类型的Message属性，将在ITest.Process方法中写入下面的代码：

```
int ITest.Process (string s)
{
  ITest test = this;
  Contract.Requires (s != test.Message);
  ...
}
```

更简单的写法是：

```
Contract.Requires (s != ((ITest)this).Message);
```

简单地使用this.Message并不会成功，因为Message必须显式调用。定义抽象方法的契约类的流程也是如此，不同的是契约类要标识为abstract而不是sealed。

诊断和代码契约

13.8 处理契约错误

通过/throwonfailure开关（或Visual Studiao项目属性中契约页上的“Assert on Contract Failure”复选框），二进制重写器允许指定契约条件失败时执行的操作。

如果不指定/throwonfailure或选中“Assert on Contract Failure”复选框，在契约失败时将弹出对话框，可以选择终止、调试或忽略错误。

提示：需要知道的一些细微差别：

- 如果是基于CLR的应用程序（例如SQL Server或Exchange），宿主的高级策略将被触发，而不是弹出对话框。
- 否则，如果现在的进程无法对用户弹出对话框，Environment.FailFast方法将被调用。

在调试模式中对话框很有用：

- 在出错时诊断和调试契约错误很容易，不需要重写运行程序。无论Visual Studio是否设置了在首次通知异常中断，它都能工作。不同于一般的异常，契约失败意味着代码中一定存在bug。
- 提供契约失败的信息，即使栈中更高级的调用像下面一样“吞”掉了异常：

```
try
{
  // 调用契约失败的某些方法
}
catch { }
```

提示：在大多数情况下，上面的代码被认为违反了模式，因为它隐藏了失败，也包括从没预料到的条件失败。

如果指定/throwonfailure开关或者在Visual Studio中取消选中“Assert on Contract Failure”复选框，在失败时将会抛出ContractException异常。这对下面的情况很有利：

- 发布模式——使异常上升到栈中并像其他未预料异常一样被处理（可以通过顶级异常处理程序记录错误或邀请用户报告错误）。
- 单元测试环境——自动记录错误的流程。

提示： ContractException不能出现在catch块中，因为这个类型不是公有的。根据是没有特别的原因要捕捉到ContractException异常，它只是作为一般异常的一部分来。

13.8.1 ContractFailed事件

当契约失败时，静态的Contract.ContractFailed事件在任何进一步的操作执行前被触发。如果处理了这个事件，能查询事件参数对象来获得错误的详细信息。也可以调用SetHandled方法来防止接下来要抛出的ContractException异常（或出现的对话框）。

在/throwonfailure被指定时，处理这个事件特别有用，因为它能记录所有的契约错误，即使像我们之前介绍的调用栈中被高级异常吞掉的异常。下面是包含自动单元测试的示例：

```
Contract.ContractFailed += (sender, args) =>
{
  string failureMessage = args.FailureKind + ": " + args.Message;
  // 用单元测试框架记录错误信息:
  // ...
  args.SetUnwind();
};
```

当在事件处理程序结束后允许运行ContractExcepton异常（或契约失败对话框）时，这个处理程序将记录所有契约的失败。请注意我们也调用了SetUnwind方法，它将使其他事件订阅者时SetHandled方法的调用失效。换句话说，它保证ContractException异常（或对话框）总是在所有的事件处理程序运行之后出现。

如果在这个处理程序中抛出异常，任何其他的事件处理程序仍继续执行。抛出的异常包含最终抛出ContractException异常的InnerException属性。

13.8.2 契约条件中的异常

契约条件自身抛出的异常，无论是否指定/throwon-failure，将像其他的异常一样传播。在用null字符串调用时下面的方法抛出NullReferenceException异常。

```
string Test (string s)
{
  Contract.Requires (s.Length > 0);
  ...
}
```

先决条件是有错误的。应该用下面的代替：

```
Contract.Requires (!string.IsNullOrEmpty (s));
```

13.9 选择性执行契约

二进制重写器提供了两个开关来取消部分或所有的契约检查：/publicsurface和/level。可以通过Visual Studio中的项目属性中的代码契约页来控制这些选项。/publicsurface开关指示重写器只检查公有方法的契约。/level开关有下面的选项：

None（*Level 0*）
取消所有的契约验证。

ReleaseRequires（*Level 1*）
只允许调用Contract.Requires<TExceptioin>的泛型版本。

Preconditions（*Level 2*）
允许所有的先决条件（Level 1加上通常的先决条件）。

Pre and Post（*Level 3*）
允许Level 2检查加上后置条件。

Full（*Level 4*）
允许Level 3检查加上对象不变式和断言（换句话说，所有的代码契约）。

在调试配置中通常允许全部契约检查。

13.9.1 发布模式中的契约

当进入到发布模式时，通常有以下两个原则：

- 注重安全性则允许所有的契约检查
- 注重性能则关闭所有的契约检查

如果要创建公用的库，尽管第二种方法会带来一些问题。想象一下要在发布模式禁止契约检查的情况下编译和发布库L，客户之后在调试模式下创建项目C并引用库L，之后程序集C不能用契约检查来错误地调用L的成员。在这种情况下，执行L的一部分契约来保证L的正确使用，换句话说，即L公有成员中的先决条件。

最简单的解决这个问题的办法是开启/publicsurface开关进行L中的Preconditions或ReleaseRequires级别的检查。这保证基本的先决条件因对客户有利而被执行，同时导致只有这些先决条件才有性能消耗。

在极端情况下，也许不希望造成这种小的性能损失。这时，可以用调用端检查这种更复杂的方法。

13.9.2 调用端检查

调用端检查将先决条件检查从调用过的方法移到调用中的方法（调用端）。通过允许库L的用户在调试配置下自己执行L的先决条件检查解决了上述问题。

要允许调用端检查，必须首先创建独立的契约应用程序集——一个补充的程序集并且只包含引用程序集的先决条件。要这么做，或者使用*ccrefgen*命令行工具，或者在Visual Studio中执行：

1. 在引用库L的发布配置中，到代码属性中的代码契约页，勾选“Build a Contract Reference Assembly”来禁止运行时契约检查。之后生成补充的契约引用程序集（带*.contracts.dll*后缀）。

2. 在引用的程序集的发布配置中，禁止所有的契约检查。

3. 在引用的程序集的调试配置中，勾选“Call-site Requires Checking”。

第三步等同于用`/callsiterequires`开关调用*ccrewrite*。它从契约引用程序集中读取先决条件并把它们添加到引用程序集中的调用端。

13.10 静态契约检查

代码契约允许静态契约检查，通过工具分析契约条件在程序运行之前发现潜在的bug。例如，静态检查下面的代码将生成一个警告：

```
static void Main()
{
  string message = null;
  WriteLine (message);      // 静态检查工具将生成警告
}

static void WriteLine (string s)
{
  Contract.Requires (s != null);
  Console.WriteLine (s);
}
```

可以通过*cccheck*命令行来调用Microsoft的静态契约工具，或者在Visual Studio中项目属性对话框中允许静态契约检查（商业版本只支持Visual Studio的Premium和Ultimate版本）。

要使静态检查工作，需要给成员添加先决条件和后置条件。提供一个简单的示例，下面的代码将生成一个警告：

```
static void WriteLine (string s, bool b)
{
  if (b)
    WriteLine (s);     // 警告：条件没有被证实
}

static void WriteLine (string s)
{
  Contract.Requires (s != null);
  Console.WriteLine (s);
}
```

因为我们调用的方法要求参数非空，必须先证明参数是非空的。要这么做，我们能够按照下面的代码为第一个方法添加先决条件：

```
static void WriteLine (string s, bool b)
{
  Contract.Requires (s != null);
  if (b)
    WriteLine (s);     // OK
}
```

3.10.1 ContractVerification属性

如果从项目生命周期一开始就进行静态检查是最简单的，否则很可能会生成大量的警告。

如果希望在现存代码库中应用静态契约检查，它将首先通过ContractVerification属性（在System.Diagnostics.Contracts中）应用于程序被选择的部分。这个属性能应用在程序集、类型和成员这些级别上。如果应用在不同级别上，更细化的优先级高。因此，要只对特别的类开启静态契约检查，按照下面的代码开始在程序集级别禁止检查：

```
[assembly: ContractVerification (false)]
```

之后在需要的类上开启它：

```
[ContractVerification (true)]
class Foo { ... }
```

13.10.2 Baseline选项

另一个在现有代码库中执行静态契约检查的手段是在Visual Studio中执行选中Baseline选项的静态检查工具。之后发生的所有警告都被写到特定的XML文件中。下一次运行静态检查，那个文件中的所有警告都将被忽略，因此只能看到新代码生成的信息。

13.10.3 SuppressMessage属性

也可以通过SuppressMessage属性（在System.Diagnostics.CodeAnalysis中）指示静态检查工具忽略某些特定类型的警告：

```
[SuppressMessage ("Microsoft.Contracts", warningFamily)]
```

*warningFamily*取下面的值中的一个：

```
Requires Ensures Invariant NonNull DivByZero MinValueNegation
ArrayCreation ArrayLowerBound ArrayUpperBound
```

可以在程序集或类型级别应用这个属性。

13.11 调试器集成

有时候应用程序和调试器（如果有的话）交互是很有用的。在开发阶段，调试器是集成开发环境（例如，Visual Studio）；在部署阶段，调试器很可能是：

- DbgCLR
- 低级调试工具，例如WinDbg、Cordbg或Mdbg

DbgCLR是Visual Studio中的调试器，和.NET Framework SDK一起免费下载。它是当没有IDE时最简单的调试选择，尽管必须下载整个SDK。

13.11.1 附加和中断

在System.Diagnostics中的静态Debugger类提供了和调试器交互的基本函数，也就是Break、Launch、Log和IsAttached。

调试器必须首先附加到应用程序上进行调试。如果从IDE（集成开发环境）运行应用程序，将自动调试，除非选择不这么做（通过选择“Start without debugging”）。然而，有时候在IDE的调试模式不

方便或不能启动应用程序。一个示例就是Windows 服务应用程序或Visual Studio设计器。一种解决方案是正常启动应用程序，之后在IDE中选择Debug Process。然而这不允许在程序执行之前设置断点。

应对方案是在应用程序中调用Debugger.Break方法。这个方法启动一个调试器，附加到它上面，在那点终止执行（Launch方法也是这么操作，但是不会终止执行）。一旦附加成功，可以通过Log方法直接记录信息到调试器的输出窗口中。可以用IsAttached属性指示程序是否已经附加到调试器中。

13.11.2 Debugger属性

DebuggerStepThrough和DebuggerHidden属性为调试器提供了如何处理特定方法、构造方法和类的单步调试的建议。

DebuggerStepThrough要求调试器进入函数内部而不需要任何用户交互。这个属性在自动生成方法和转收实际工作到其他的代理方法时非常有用。在后一种情况下，如果在实际的方法中设置了断点，调试器在调用栈中仍然显示代理方法，除非添加了DebuggerHidden属性。这两个方法可以组成代理来帮助用户专注于调试应用程序的逻辑：

```
[DebuggerStepThrough, DebuggerHidden]
void DoWorkProxy()
{
  // 开始...
  DoWork();
  // 结束...
}

void DoWork() {...}   // 实际的方法...
```

13.12 进程和处理线程

在第6章的最后一节，我们介绍了如何用Process.Start方法开启新的进程。Process类也可以查询并和其他相同或不同计算机上运行的进程交互。

13.12.1 检查运行中的进程

Process.GetProcess*XXX*方法通过名称或进程ID检索指定进程，或检索所有运行在当前或指定名称计算机中的进程，包括所有托管和非托管的进程。每一个Process实例都有很多属性映射到各种统计数据上，例如名称、ID、优先级、内存和处理器利用率、窗口句柄等。下面的示例列举了当前计算机上所有运行的进程：

```
foreach (Process p in Process.GetProcesses())
using (p)
{
  Console.WriteLine (p.ProcessName);
  Console.WriteLine ("   PID:      " + p.Id);
  Console.WriteLine ("   Memory:   " + p.WorkingSet64);
  Console.WriteLine ("   Threads:  " + p.Threads.Count);
}
```

Process.GetCurrentProcess方法返回当前的进程。如果创建了额外的应用程序域，它们将共享同一个进程。

可以通过调用Kill方法来终止一个进程。

13.12.2 在进程中检查线程

也可以用Process.Threads属性遍历其他进程的所有线程。然而，获得的对象并不是System.Threading.Thread对象，而是ProcessThread对象，它用于管理而不是同步任务。ProcessThread对象提供了潜在线程的诊断信息，并允许控制它的一些属性，例如优先级和处理器亲和度：

```
public void EnumerateThreads (Process p)
{
  foreach (ProcessThread pt in p.Threads)
  {
    Console.WriteLine (pt.Id);
    Console.WriteLine ("   State:    " + pt.ThreadState);
    Console.WriteLine ("   Priority: " + pt.PriorityLevel);
    Console.WriteLine ("   Started:  " + pt.StartTime);
    Console.WriteLine ("   CPU time: " + pt.TotalProcessorTime);
  }
}
```

13.13 StackTrace和StackFrame类

StackTrace和StackFrame类提供了执行调用栈的只读视图。可以获得当前线程、同一进程中的其他线程或异常对象的栈追踪信息。这些信息对于诊断都很有用，也可以用于编程（破解）。StackTrace代表一个完整的调用栈；StackFrame代表栈中调用的一个单独方法。

如果不提供参数或提供一个bool参数来实例化StackTrace对象，将获得当然线程的调用栈的快照。Bool参数如果为true，将指示StackTrack读取程序集的*.pdb*（项目调试）文件，如果该文件存在的话，可以对文件名、行号、列偏移数据进行访问。

一旦获得StackTrace对象，可以通过调用GetFrame方法检查特定的结构或者用GetFrames方法获得所有结构：

```
static void Main() { A (); }
static void A()    { B (); }
static void B()    { C (); }
static void C()
{
  StackTrace s = new StackTrace (true);

  Console.WriteLine ("Total frames:   " + s.FrameCount);
  Console.WriteLine ("Current method: " + s.GetFrame(0).GetMethod().Name);
  Console.WriteLine ("Calling method: " + s.GetFrame(1).GetMethod().Name);
  Console.WriteLine ("Entry method:   " + s.GetFrame(s.FrameCount-1).GetMethod().Name);
  Console.WriteLine ("Call Stack:");
  foreach (StackFrame f in s.GetFrames())
    Console.WriteLine (
      "  File: "   + f.GetFileName() +
      "  Line: "   + f.GetFileLineNumber() +
      "  Col: "    + f.GetFileColumnNumber() +
      "  Offset: " + f.GetILOffset() +
      "  Method: " + f.GetMethod().Name);
}
```

下面是输出：

```
Total frames:    4
Current method: C
Calling method: B
Entry method: Main
Call Stack:
  File: C:\Test\Program.cs  Line: 15  Col: 4  Offset: 7  Method: C
  File: C:\Test\Program.cs  Line: 12  Col: 22  Offset: 6  Method: B
  File: C:\Test\Program.cs  Line: 11  Col: 22  Offset: 6  Method: A
  File: C:\Test\Program.cs  Line: 10  Col: 25  Offset: 6  Method: Main
```

提示： IL偏移量表示下一条执行的指令偏离量，而不是当前执行的指令。但是，行号和列号（如果有一个.pdb文件）通常表示实际的执行点。

这是因为，在根据IL偏移量计算行号和列号时，CLR会尽力推断实际的执行点。编译器会提供IL，帮助它完成计算——其中包括在IL流中插入nop（无操作）指令。

然而，在启用优化的前提下进行编译，禁止插入nop指令，这样堆跟踪就可以显示下一条执行指令的行号和列号。由于优化可能使用了其他技术，如收起全部方法，因此会进一步影响堆跟踪的获取。

对于整个StackTrace来说，获得基本信息的简单方式是调用ToString方法。它的结果看起来和下面的一样：

```
at DebugTest.Program.C() in C:\Test\Program.cs:line 16
at DebugTest.Program.B() in C:\Test\Program.cs:line 12
at DebugTest.Program.A() in C:\Test\Program.cs:line 11
at DebugTest.Program.Main() in C:\Test\Program.cs:line 10
```

要获得另一线程的栈追踪信息，可以将其他的Thread对象传入StackTrace的构造方法中。这是分析一个程序的有效方式。附带条件是必须首先挂起线程，通过调用Suspend方法（当结束之后调用Resume方法）。这是线程不宜用的Suspend和Resume方法的一种合法使用。

也可以通过将Exception对象传入StackTrace的构造方法中获得Exception对象的栈跟踪信息（显示导致抛出异常的信息）。

提示： Exception已经有StackTrace属性，但是这个属性返回的是简单的字符串不是StackTrace对象。StackTrace对象在记录部署之后发生的异常时很有用，这时没有*.pdb*文件，因为能记录IL的偏移量来代替行和列号。使用IL偏移量和*ildasm.exe*，能精确定位方法中发生错误的地方。

13.14 Windows事件日志

Win32平台提供了Windows事件日志形式的集中日志机制。

如果注册了EventLogTraceListener类，之前使用的Debug和Trace类可以写入Windows事件日志。但是，可以使用EventLog类直接写入Windows事件日志而不使用Trace或Debug类。也可以使用这个类来读取和监视事件数据。

提示： 写入事件日志对Windows服务应用程序来说很有意义，因为如果出错了，不能弹出用户界面来提供给

用户一些包含诊断信息的特殊文件。也因为Windows服务通常都写入Windows事件日志，如果服务出现问题，Windows事件日志几乎是管理员首先要查看的地方。

有三种标准的Windows事件日志，按名称分类：

- 应用程序
- 系统
- 安全

应用程序日志是大多数应用程序通常写入的地方。

13.14.1 写入事件日志

要写入Windows事件日志：

1. 选择三种事件日志中的一种（通常是应用程序日志）。
2. 决定源名称，必要时创建。
3. 用日志名称、源名称和消息数据来调用用EventLog.WriteEntry方法。

源名称使应用程序更容易分类。必须在使用它之前注册源名称，使用CreateEventSource方法可以实现这个功能。之后可以调用WriteEntry方法：

```
const string SourceName = "MyCompany.WidgetServer";

// CreateEventSource需要管理员权限，所以它通常都在应用程序安装时进行
if (!EventLog.SourceExists (SourceName))
  EventLog.CreateEventSource (SourceName, "Application");

EventLog.WriteEntry (SourceName,
  "Service started; using configuration file=...",EventLogEntryType.Information);
```

EventLogEntryType可以是Information、Warning、Error、SuccessAudit或FailureAudit。每一个在Windows事件查看器中都显示不同的图标。也可以选择性地指定目录和事件ID（每一个都是自己选择的数字）并提供可选的二进制数据。

CreateEventSource也允许指定计算机名：这可以写入其他计算机的事件日志，如果有足够的权限。

13.14.2 读取事件日志

要读取事件日志，用想访问的日志名来实例化EventLog类，并选择性地使用日志存在的其他计算机名。每一个日志项目能够通过Entries集合属性来读取：

```
EventLog log = new EventLog ("Application");

Console.WriteLine ("Total entries: " + log.Entries.Count);

EventLogEntry last = log.Entries [log.Entries.Count - 1];
Console.WriteLine ("Index:   " + last.Index);
Console.WriteLine ("Source:  " + last.Source);
Console.WriteLine ("Type:    " + last.EntryType);
Console.WriteLine ("Time:    " + last.TimeWritten);
Console.WriteLine ("Message: " + last.Message);
```

可以通过静态方法EventLog.GetEventLogs来遍历当前（或其他）计算机上的所有日志（这需要管理员权限）：

```
foreach (EventLog log in EventLog.GetEventLogs())
  Console.WriteLine (log.LogDisplayName);
```

通常这至少会打印应用程序日志、安全日志和系统日志。

13.14.3 监视事件日志

通过EntryWritten事件，一条项目被写入到Windows事件日志时，将获得通知。对工作在本机的事件日志，无论什么应用程序记录日志都会被触发。

要开启日志监视：

（1）实例化EventLog并设置它的EnableRaisingEvents属性为true。

（2）处理EntryWritten事件。

例如：

```
static void Main()
{
  using (var log = new EventLog ("Application"))
  {
    log.EnableRaisingEvents = true;
    log.EntryWritten += DisplayEntry;
    Console.ReadLine();
  }
}

static void DisplayEntry (object sender, EntryWrittenEventArgs e)
{
  EventLogEntry entry = e.Entry;
  Console.WriteLine (entry.Message);
}
```

13.15 性能计数器

我们之前讨论的日志机制对获得信息进行事后分析很有用。然而，要获得应用程序（或整个系统）的当前状态，需要更实时的方法。这种需求的Win32方案是性能监视结构，它包含一系列系统和应用程序公开的性能计数器和用于实时监视这些计数器的Microsoft管理控制台（MMC）插件。

性能计数器按照“系统”、“处理器”、“.NET CLR内存”等类别分组。在GUI工具中这些种类有时也指性能对象。每个种类都包含一系列相关的监视系统或应用程序某方面的性能计数器。“.NET CLR内存”中的性能计数器的示例包括“% Time in GC（GC中时间的百分比）”、“# Bytes in All Heaps（所有堆中的字节数）”和“Allocated bytes/sec（每秒分配的字节数）”。

每个种类都可以选择性地拥有能被独立监视的一个或多个实例。例如，在“处理器”种类中的“% Processor Time（处理器时间百分比）”性能计数器很有用，它允许一个实例监视CPU利用率。在多处理器机器上，这个计数器支持每个CPU的实例，允许每个实例独立监视每个CPU的利用率。

下面一节将解释如何执行常用的任务，像确定公开的计数器、监视计数器和创建自定义的计数器来公

开应用程序状态信息。

警告：读取性能计数器或种类有时需要本机或目标机器的管理员权限，这取决于要读取的信息。

13.15.1 遍历可用的计数器

下面的示例遍历计算机上所有可用的性能计数器。对于那些有实例的，它遍历每个实例的计数器：

```
PerformanceCounterCategory[] cats = PerformanceCounterCategory.GetCategories();
foreach (PerformanceCounterCategory cat in cats)
{
  Console.WriteLine ("Category: " + cat.CategoryName);

  string[] instances = cat.GetInstanceNames();
  if (instances.Length == 0)
  {
    foreach (PerformanceCounter ctr in cat.GetCounters())
      Console.WriteLine ("  Counter: " + ctr.CounterName);
  }
  else   // 转储计数器实例
  {
    foreach (string instance in instances)
    {
      Console.WriteLine ("  Instance: " + instance);
      if (cat.InstanceExists (instance))
        foreach (PerformanceCounter ctr in cat.GetCounters (instance))
          Console.WriteLine ("    Counter: " + ctr.CounterName);
    }
  }
}
```

警告：结果将多于10,000行。要花费一段时间来执行，因为PerformanceCounterCategory.InstanceExists有一些低效率的实现。在实际的系统中，只在需要时才获得更加详细的信息。

下一个示例使用LINQ查询来获得.NET性能计数器，并将结果写到XML文件中：

```
var x =
  new XElement ("counters",
    from PerformanceCounterCategory cat in
         PerformanceCounterCategory.GetCategories()
    where cat.CategoryName.StartsWith (".NET")
    let instances = cat.GetInstanceNames()
    select new XElement ("category",
      new XAttribute ("name", cat.CategoryName),
      instances.Length == 0
      ?
        from c in cat.GetCounters()
        select new XElement ("counter",
          new XAttribute ("name", c.CounterName))
      :
        from i in instances
        select new XElement ("instance", new XAttribute ("name", i),
          !cat.InstanceExists (i)
```

```
            ?
              null
            :
              from c in cat.GetCounters (i)
              select new XElement ("counter",
                new XAttribute ("name", c.CounterName))
          )
      )
  );
x.Save ("counters.xml");
```

13.15.2 读取性能计数器数据

要获得性能计数器的值，实例化PerformanceCounter对象并调用NextValue或NextSample方法。NextValue返回简单的float值；NextSample返回公开了更高级属性集的CounterSample对象，这些属性包括CounterFrequency、TimeStamp、BaseValue和RawValue。

PerformanceCounter的构造方法接受种类名、计数器名和可选的实例。因此，要显示所有CPU的当前处理器利用率，要按照下面的代码操作：

```
using (PerformanceCounter pc = new PerformanceCounter ("Processor",
                                                       "% Processor Time",
                                                       "_Total"))
  Console.WriteLine (pc.NextValue());
```

或者要显示当前进程的私有内存消耗：

```
string procName = Process.GetCurrentProcess().ProcessName;
using (PerformanceCounter pc = new PerformanceCounter ("Process",
                                                       "Private Bytes",
                                                       procName))
  Console.WriteLine (pc.NextValue());
```

PerformanceCounter并没有公开ValueChanged事件，因此如果想监视各种改变，必须进行轮询。在下面的示例中，我们每200ms轮询一次，直到EventWaitHandle发出退出的信号：

```
// 需要引入System.Threading和System.Diagnostics

static void Monitor (string category, string counter, string instance,
                     EventWaitHandle stopper)
{
  if (!PerformanceCounterCategory.Exists (category))
    throw new InvalidOperationException ("Category does not exist");

  if (!PerformanceCounterCategory.CounterExists (counter, category))
    throw new InvalidOperationException ("Counter does not exist");

  if (instance == null) instance = "";    // "" == 没有实例 (不是null!)
  if (instance != "" &&
      !PerformanceCounterCategory.InstanceExists (instance, category))
    throw new InvalidOperationException ("Instance does not exist");

  float lastValue = 0f;
  using (PerformanceCounter pc = new PerformanceCounter (category,counter, instance))
    while (!stopper.WaitOne (200, false))
    {
```

```
        float value = pc.NextValue();
        if (value != lastValue)          // 只输出改变的值
        {
          Console.WriteLine (value);
          lastValue = value;
        }
      }
    }
```

下面是我们如何使用这个方法同步监视处理器和硬盘活动：

```
static void Main()
{
  EventWaitHandle stopper = new ManualResetEvent (false);

  new Thread (() =>
    Monitor ("Processor", "% Processor Time", "_Total", stopper)
  ).Start();

  new Thread (() =>
    Monitor ("LogicalDisk", "% Idle Time", "C:", stopper)
  ).Start();

  Console.WriteLine ("Monitoring - press any key to quit");
  Console.ReadKey();
  stopper.Set();
}
```

诊断和代码契约

13.15.3 创建计数器并写入性能数据

在写入性能计数器数据之前，需要创建性能种类和计数器。例如下面的代码同时创建性能种类和属于该种类的所有计数器：

```
string category = "Nutshell Monitoring";

// 我们将在这个Categorg中创建两个计数器：

string eatenPerMin = "Macadamias eaten so far";
string tooHard = "Macadamias deemed too hard";

if (!PerformanceCounterCategory.Exists (category))
{
  CounterCreationDataCollection cd = new CounterCreationDataCollection();

  cd.Add (new CounterCreationData (eatenPerMin,
          "Number of macadamias consumed, including shelling time",
          PerformanceCounterType.NumberOfItems32));

  cd.Add (new CounterCreationData (tooHard,
          "Number of macadamias that will not crack, despite much effort",
          PerformanceCounterType.NumberOfItems32));

  PerformanceCounterCategory.Create (category, "Test Category",
    PerformanceCounterCategoryType.SingleInstance, cd);
}
```

如图13-1所示，当选择添加计数器之后，新的计数器将在Windows性能监视工具中显示出来。

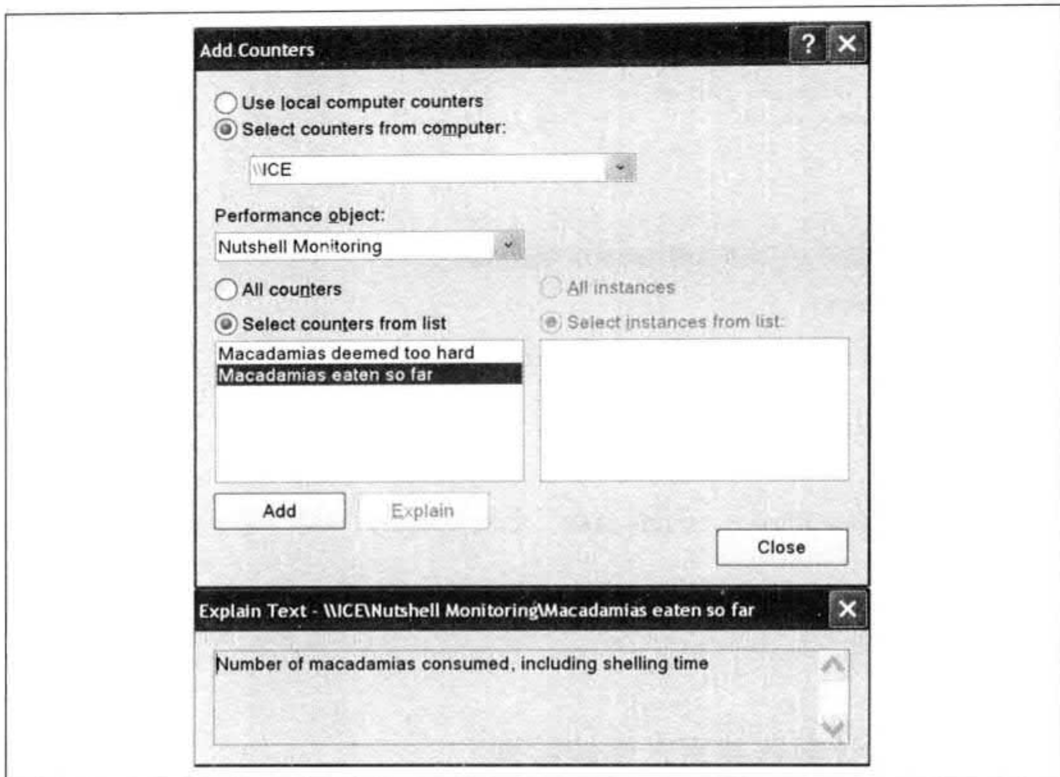


图13-1：自定义性能计数器

如果希望在同一个种类中定义更多的计数器，必须首先调用PerformanceCoun-terCategory.Delete方法删除旧的种类。

提示： 创建和删除性能计数器需要管理员权限。因此，它通常都作为应用程序安装的一部分进行。

一旦计数器被创建，可以实例化PerformanceCounter，设置ReadOnly为false，并设置RawValue来更新它的值。也可以使用Increment和IncrementBy方法来更新现在的值：

```
string category = "Nutshell Monitoring";
string eatenPerMin = "Macadamias eaten so far";

using (PerformanceCounter pc = new PerformanceCounter (category,eatenPerMin, ""))
{
  pc.ReadOnly = false;
  pc.RawValue = 1000;
  pc.Increment();
  pc.IncrementBy (10);
  Console.WriteLine (pc.NextValue());    // 1011
}
```

13.16 Stopwatch类

Stopwatch类提供了一种方便的机制来衡量执行时间。Stopwatch使用了操作系统和硬件提供的最高分辨率机制，通常少于1ms（对比一下，DateTime.Now和Environment.TickCount有大约15ms的分辨率）。

要使用Stopwatch调用StartNew方法，它实例化Stopwatch对象并开始计时（换句话说，可以手动实例化并在之后调用Start方法）。Elapsed返回表示过去的时间间隔的TimeSpan对象：

```
Stopwatch s = Stopwatch.StartNew();
System.IO.File.WriteAllText ("test.txt", new string ('*', 30000000));
Console.WriteLine (s.Elapsed);       // 00:00:01.4322661
```

Stopwatch也公开了ElapsedTicks属性，它返回表示过去时间的long类型的数字。要将时间转换成秒，请除以StopWatch.Frequency。Stopwatch也有ElapsedMilliseconds属性，这通常是最方便的。

调用Stop方法将终止Elapsed和ElapsedTicks。运行的Stopwatch并不会引起任何后台活动，因此调用Stop方法是可选的。

第14章

并发与异步

大多数应用程序都需要同时（并发）处理多个任务。在本章中，我们先介绍一些基础知识，即线程和任务的基本概念，然后再详细介绍异步处理的原理和C# 5.0的异步函数。

第22章将更深入地介绍多线程处理，而第23章则介绍并行编程的相关知识。

14.1 简介

最常见的并发场景包括：

（1）*编写快速响应的用户界面*

WPF、Metro和Windows窗体应用程序都必须在代码中并发运行耗费时间的任务，才能保证实现用户界面的快速响应。

（2）*允许并发请求处理*

在服务器上，客户端请求可能同时到达，所以必须通过并行处理才能够保证可伸缩性。如果使用ASP.NET、WCF或Web Services，那么.NET框架会自动执行并行处理。然而，程序员仍然需要知道共享的状态（例如，使用静态变量进行缓存）。

（3）*并行编程*

如果将负载划分到多个核心上，那么多核/多处理器计算机可以提升密集计算代码的执行速度（第23章将专门介绍这方面内容）。

（4）*预测执行*

在多核主机上，有时候可以通过预测必须执行的任务，然后提前执行些任务，以此提高程序的运行性能。LINQPad使用这种方法来提高查询的创建速度。另一种方法是并行运行多个可解决同一任务的不同算法。无论使用哪种方法，最先完成的算法更优——如果事先不知道哪一种算法的执行速度最快，则可以使用这种比较方式。

程序并发执行代码的通用机制是多线程（multithreading）。CLR和操作系统都支持多线程，它是一种基础并发概念。因此，最基本的要求是理解线程的基本概念，特别是线程的共享状态。

14.2 线程处理

线程（thread）是一个独立处理的执行路径。

每一个线程都运行在一个操作系统进程中，这个进程是程序执行的独立环境。在单线程（single-threaded）程序中，在进程的独立环境中只有一个线程运行，所以该线程具有独立使用进程资源的权利。在多线程（multithreaded）程序中，在进程中有多个线程运行，它们共享同一个执行环境（特别是内存）。这在一定程度上反映了多线程处理的作用：例如，一个线程在后台获取数据，同时另一个线程显示所获得的数据，这些数据就是所谓的共享状态（shared state）。

14.2.1 创建一个线程

提示：Windows Metro配置文件不允许直接创建和启动线程；相反，必须通过任务来操作线程（参见第14.3节的“任务”）。任务增加了间接创建线程的方法，这种方法增加了学习复杂性，所以最好从控制台应用程序（或LINQPad）开始，熟悉它们的使用方法，然后再直接创建线程。

客户端程序（Console、WPF、Metro或Windows窗体）都从操作系统自动创建的一个线程（主线程）开始。除非创建更多的线程（直接或间接），否则这就是单线程应用程序的运行环境（注1）。

实例化一个Thread对象，然后调用它的Start方法，就可以创建和启动一个新的线程。最简单的Thread构造方法是接受一个ThreadStart代理：一个无参数方法，表示执行开始位置。例如：

```
// 注意：本章所有示例都会导入以下命名空间：
using System;
using System.Threading;

class ThreadTest
{
  static void Main()
  {
    Thread t = new Thread (WriteY); // 创建一个新线程
    t.Start(); // 启动线程WriteY()
    // 同时，主线程也会执行。
    for (int i = 0; i < 1000; i++) Console.Write ("x");
  }
  static void WriteY()
  {
    for (int i = 0; i < 1000; i++) Console.Write ("y");
  }
}
// 一般输出：
xxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
...
```

主线程会创建一个新线程t，然后在上面运行一个方法，重复打印出字符y。同时，主线程也会重复打印出字符x，如图14-1所示。在单核计算机上，操作系统会给每一个线程分配一些“时间片”

注1：在垃圾收集和终止化的背后，CLR创建其他线程。

（Windows一般为20毫秒），用于模拟并发性，因此这段代码会出现连续的x和y。在多核/多处理器主机上执行时，虽然这个例子仍然会出现重复的x和y（受控制台处理并发请求的机制影响），但是线程却能够真正实现并行执行（分别由计算机上其他激活处理器完成）。

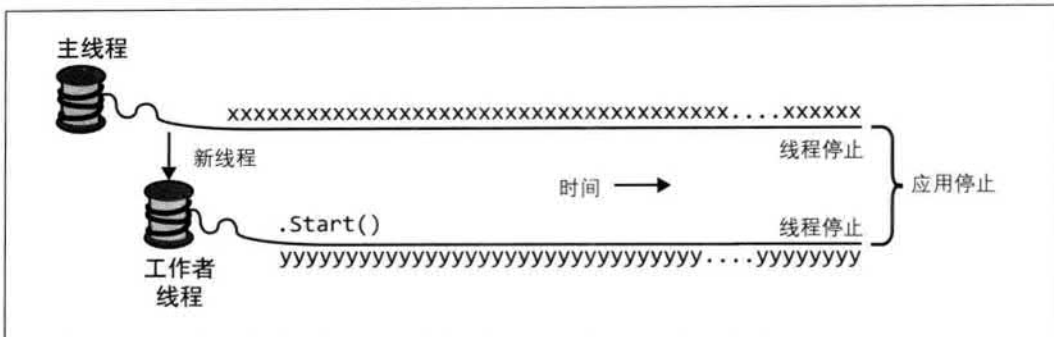


图14-1 启动一个新线程

提示： 线程被认为是优先占用（*preempted*）它在执行过程与其他线程代码交叉执行的位置。这个术语通常可以解释出现的问题！

在线程启动之后，线程的`IsAlive`属性就会变成`true`，直到线程停止。当`Thread`的构造函数接收的代理执行完毕时，线程就会停止。在停止之后，线程无法再次启动。

每个线程都有一个`Name`属性，它可用于调试程序。它在Visual Studio中特别有用，因为线程的名称会显示在Threads窗口和Debug Location工具栏上。线程名称只能设置一次；修改线程名称会抛出异常。

静态属性`Thread.CurrentThread`可以返回当前执行的线程：

```
Console.WriteLine (Thread.CurrentThread.Name);
```

14.2.2 联合与休眠

在等待另一个线程结束时，可以调用另一个线程的`Join`方法：

```
static void Main()
{
  Thread t = new Thread (Go);
  t.Start();
  t.Join();
  Console.WriteLine ("Thread t has ended!");
}

static void Go() { for (int i = 0; i < 1000; i++) Console.Write ("y"); }
```

这段代码会打印1,000 次“y”，然后再接着打印“Thread t has ended!”。在调用`Join`时，可以指定一个超时时间，可以以毫秒为单位，或者是一个`TimeSpan`。然后，它会在线程结束时返回`true`，或者在超时时返回`false`。

`Thread.Sleep`会将当前线程暂停执行一定的时间：

```
Thread.Sleep (TimeSpan.FromHours (1)); // 休眠1小时
```

```
Thread.Sleep (500); // 休眠500毫秒
```

调用Thread.Sleep(0)，会马上放弃线程的当前时间片，自动将CPU交给其他线程。Thread.Yield()方法也有相同的效果，但是它只会将资源交给在同一个处理器上运行的线程。

提示： 有时候，在生产代码中使用Sleep(0)或Yield，可以优化性能。它还是一种很好的诊断工具，可以帮助开发者发现线程安全问题：如果在代码任意位置插入Thread.Yield()会破坏程序，那么代码肯定存在bug。

在等待线程Sleep或Join的过程中，还可以阻塞线程。

14.2.3 阻塞

线程阻塞是指线程由于特定原因暂停执行，如Sleeping或执行Join后等待另一个线程停止。阻塞的线程会立刻交出（yield）它的处理器时间片，然后从这时开始不再消耗处理器时间，直至阻塞条件结束。使用线程的ThreadState属性，可以测试线程的阻塞状态：

```
bool blocked = (someThread.ThreadState & ThreadState.WaitSleepJoin) != 0;
```

提示： ThreadState是一个标记枚举量，它由三“层”二进制位数据组成。然而，这其中大多数值都是冗余、无用或废弃的。下面的扩展方法将ThreadState限定为以下四个值之一：Unstarted、Running、WaitSleepJoin和Stopped：

```
public static ThreadState Simplify (this ThreadState ts)
{
    return ts & (ThreadState.Unstarted |
                 ThreadState.WaitSleepJoin |
                 ThreadState.Stopped);
}
```

ThreadState属性可用于诊断程序，但是不适用于实现同步，因为线程状态可能在测试ThreadState和获取这个信息的时间段内发生变化。

当线程阻塞或未阻塞时，操作系统会执行环境切换（*context switch*）。这个操作会稍微增加负载，幅度一般在1～2毫秒左右。

1. I/O密集与计算密集

如果一个操作将大部分时间用于等待一个条件的发生，那么它就称为I/O密集（I/O-bound）操作——在网页上下载一个例子或者调用Console.ReadLine。（I/O密集操作一般都会涉及输入或输出，但是这不是硬性要求：Thread.Sleep也是一种I/O密集操作。）相反，如果一个操作将大部分时间用于执行CPU密集操作，那么它就称为计算密集（*compute-bound*）操作。

2. 阻塞与自旋

I/O密集操作可以以两种方式执行：同步等待当前线程的操作完成（如Console.ReadLine、Thread.Sleep或Thread.Join），或者异步执行，然后在将来操作完成时触发一个回调函数（后面将深入介绍这些内容）。

异步等待的I/O密集操作会将大部分时间花费在线程阻塞上。它们也可能在一个定期循环中自旋：

```
while (DateTime.Now < nextStartTime)
Thread.Sleep (100);
```

除此之外，还有一些更好的方法（如计时器或信号结构）可以实现这种效果，另一种实现线程持续自旋的方法是：

```
while (DateTime.Now < nextStartTime);
```

通常，这非常浪费处理器时间：只要CLR和操作系统可用，线程就会执行一些重要计算，因此会消耗分配的相应资源。实际上，我们已经将一种I/O密集变成了计算密集。

提示：自旋与阻塞有一些细微差别。首先，非常短暂的自旋可能非常适用于设置很快能满足的条件（也许是几毫秒之内），因为它可以避免过载和环境切换延迟。.NET框架提供了一些特殊的辅助方法和类——参见http://albahari.com/threading/的"SpinLock与SpinWait"。

其次，阻塞并非零开销。这是因为，只要线程执行，它就会占用1MB内存，并且会对CLR和操作系统产生持续管理负载。因此，阻塞在一些繁重I/O密集程序环境中可能会造成问题，这些程序需要处理成百上千的并发操作。相反，这些程序使用基于回调的方法，在等待时完全解除它们的线程。我们将在后面讨论异步模式的用例时介绍这种方法。

14.2.4 本地状态与共享状态

CLR会给每一个线程分配独立的内存堆，从而保证本地变量的隔离。下例定义了一个方法，其中包含一个局部（本地）变量，然后同时在主线程和新创建的线程上调用这个方法：

```
static void Main()
{
  new Thread (Go).Start();     // 在新线程上调用Go()
  Go();                        // 在主线程上调用Go()
}
static void Go()
{
  // 声明和使用局部变量——'cycles'
  for (int cycles = 0; cycles < 5; cycles++) Console.Write ('?');
}
```

每一个线程的内存堆都会创建cycles变量副本，所以（可以推断）输出结果是10个问号。

如果线程拥有同一个对象实例的通用引用，那么这些线程就共享相同数据：

```
class ThreadTest
{
  bool _done;

  static void Main()
  {
    ThreadTest tt = new ThreadTest(); // 创建一个通用实例
    new Thread (tt.Go).Start();
    tt.Go();
  }
  void Go() // 注意这是一个实例方法
  {
```

```
      if (!_done) { _done = true; Console.WriteLine ("Done"); }
    }
  }
```

因为这两个线程都在同一个ThreadTest实例上调用Go()，所以它们共享_done域。因此，"Done"只会打印一次，而不会打印两次。

编译器会将Lambda表达式或匿名代理捕获的局部变量转换为域，所以它们也可以共享：

```
  class ThreadTest
  {
    static void Main()
    {
      bool done = false;
      ThreadStart action = () =>
    {
      if (!done) { done = true; Console.WriteLine ("Done"); }
    };
    new Thread (action).Start();
    action();
   }
  }
```

静态域是在线程之间共享数据的另一种方法：

```
  class ThreadTest
  {
    static bool _done; // 静态域在同一个应用域的所有线程之间共享
    static void Main()
    {
      new Thread (Go).Start();
      Go();
    }
    static void Go()
    {
      if (!_done) { _done = true; Console.WriteLine ("Done"); }
    }
  }
```

这三个例子都演示了另一个重要概念：实现线程安全性（或者缺少安全性）。它的输出其实是不确定的："Done"可能会打印两次（实际上不可能）。然而，如果对换Go方法的语句顺序，"Done"打印两次的可能性会大大增加：

```
  static void Go()
  {
    if (!_done) { Console.WriteLine ("Done"); _done = true; }
  }
```

问题是，当一个线程在判断if语句值时，另一个线程可能正在执行WriteLine语句——它还来不及将done设置为true。

提示：我们的例子演示了其中一种共享可写状态可能引起间歇性错误的方式，这正是多线程经常被诟病的问题。我们将介绍如何通过锁机制来优化程序；然而，最好是避免使用共享状态。我们将在后面介绍如何通过异步编程模式解决这个问题。

14.2.5 锁与线程安全

提示：锁与线程安全性都是重要问题。第22.2节“排他锁”和第22.3节“锁与线程安全”将全面介绍这个问题。

在读写共享域时，先获取一个排他锁（*exclusive lock*），就可以解决前面例子的问题。使用C#的lock语句，就可以实现这个目标：

```
class ThreadSafe
{
  static bool _done;
  static readonly object _locker = new object();

  static void Main()
  {
    new Thread (Go).Start();
    Go();
  }
  static void Go()
  {
    lock (_locker)
    {
      if (!_done) { Console.WriteLine ("Done"); _done = true; }
    }
  }
}
```

当两个线程同时争夺同一个锁时（它可以是任意引用类型的对象，这里是_locker），其中一个线程会等待（或阻塞），直到锁释放。这个例子保证一次只有一个线程能够进入它的代码块，因此“Done”只打印一次。在复杂的多线程环境中，采用这种方式来保护的代码就是具有线程安全性（*thread-safe*）。

提示：即使变量自增操作不是线程安全操作，底层处理器也采用清晰的读－增－写操作来执行表达式x++。所以，如果两个线程同时在锁之外执行x++，那么这个变量最终只会增加1次，而不会增加2次（更坏的情况是，x可能会破坏，最终在特定情况下得到新旧数值的混合二进制数）。

锁并不是解决线程安全的万能法宝——人们很容易在访问域时忘记锁，而且锁本身也存在一些问题（如死锁）。

在ASP.NET应用程序中，锁的一个常见用途是访问一个共享内存缓存，以避免频繁访问数据库对象。这种应用很容易实现，而且也不可能出现死锁问题。第22.3.2节“应用服务器的线程安全性”将给出这样的例子。

14.2.6 传递数据到线程

有时候，我们需要给线程的启动方法传递一些参数。最简单的方法是使用一个Lambda表达式，然后用指定的参数调用这个方法：

```
static void Main()
{
  Thread t = new Thread ( () => Print ("Hello from t!") );
```

```
    t.Start();
}
static void Print (string message) { Console.WriteLine (message); }
```

使用这种方法，可以给这个方法传递任意数量的参数。甚至，可以将整个实现过程封装在一个多语句Lambda 表达式中：

```
new Thread (() =>
{
  Console.WriteLine ("I'm running on another thread!");
  Console.WriteLine ("This is so easy!");
}).Start();
```

C# 3.0之前不支持Lambda表达式。所以，还可以使用旧方法，即将参数传递给Thread的Start方法：

```
static void Main()
{
  Thread t = new Thread (Print);
  t.Start ("Hello from t!");
}

static void Print (object messageObj)
{
  string message = (string) messageObj; // 这里需要强制转换
  Console.WriteLine (message);
}
```

这种方法同样有效，因为Thread的构造函数已经重载，可以接受下面的其中一个代理：

```
public delegate void ThreadStart();
public delegate void ParameterizedThreadStart (object obj);
```

ParameterizedThreadStart的局限性在于：它只接受一个参数。而且因为参数属于类型object，所以它通常需要进行强制转换。

Lambda表达式与捕获的变量

正如我们所看到的，Lambda表达式是向线程传递数据的最方便且最强大的方法。然而，在线程启动之后，一定要注意小心修改捕获的变量（*captured variables*）。例如，假如有以下语句：

```
for (int i = 0; i < 10; i++)
new Thread (() => Console.Write (i)).Start();
```

这段代码的输出结果是不确定的！下面是一种常见结果：

```
0223557799
```

问题是，在整个循环过程中，变量i都指向同一块内存地址。因此，每次线程调用Console.Write处理变量时，这个变量的值可能发生了变化！解决方法是使用临时变量：

```
for (int i = 0; i < 10; i++)
{
  int temp = i;
  new Thread (() => Console.Write (temp)).Start();
}
```

这样，数字0至9都只会出现一次。（各个数字出现的顺序仍然是不确定的，因为线程的启动时间不确定。）

提示：这个问题与第8.4.2节“捕获的变量”所描述的问题类似。关于C#捕获变量的规则，在循环与多线程中，问题都非常多。

这个问题同样存在于C# 5.0之前的foreach循环中。

现在，变量temp是每一个循环过程的局部变量。因此，每一个线程都会获得完全不同的内存地址，因此不会出现前面提到的问题。下面的例子更能够说明前一段代码的问题：

```
string text = "t1";
Thread t1 = new Thread ( () => Console.WriteLine (text) );

text = "t2";
Thread t2 = new Thread ( () => Console.WriteLine (text) );
t1.Start(); t2.Start();
```

由于两个Lambda表达式都捕获同一个text变量，所以t2会打印两次。

14.2.7 异常处理

在线程创建时任何生效的try/catch/finally语句块在线程开始执行后都与线程无关。假设有以下程序：

```
public static void Main()
{
  try
  {
    new Thread (Go).Start();
  }
  catch (Exception ex)
  {
    // 代码永远不会运行到这里!
    Console.WriteLine ("Exception!");
  }
}

static void Go() { throw null; } // 抛出异常NullReferenceException
```

这个例子的try/catch语句是无效的，而且新创建的线程被一个未处理的异常NullReferenceException阻塞。如果将每一个线程看作独立的执行路径，那么就可以理解这种行为。

解决方法是将异常处理移到Go方法之内：

```
public static void Main()
{
   new Thread (Go).Start();
}

static void Go()
{
  try
  {
    ...
```

```
        throw null; // 下面会捕获到异常 NullReferenceException
        ...
      }
      catch (Exception ex)
      {
        通常是记录异常，并且/或者发信号给另一个线程，告诉它我们捕捉到了异常
        ...
      }
    }
```

在运行环境中，应用程序的所有线程入口方法都需要添加一个异常处理方法——就和主线程一样（通常位于更高一级的执行堆栈中）。未处理的异常可能导致整个应用程序崩溃，然后弹出一个错误对话框！

提示：在编写这样的异常处理语句块时，一般是不能忽略错误的：通常需要记录异常明细，然后显示一个对话框，允许用户将这些明细自动提交到Web服务器。然后，还可以选择重启应用程序，因为意外异常可能会使程序处于无效状态。

集中式异常处理

WPF、Metro和Windows窗体应用程序都支持订阅全局异常处理事件，分别是`Application.DispatcherUnhandledException`和`Application.ThreadException`。如果通过消息循环调用的程序中出现未处理异常（相当于在`Application`激活时运行在主线程上的所有代码），就会触发这些异常。这非常适合于记录日志和报告缺陷（但是它不会触发非UI线程的未处理异常）。处理这些事件可以防止程序意外关闭，但是必须选择重启应用程序，避免在未处理异常发生之后（或导致）出现可能已经破坏的状态。

`AppDomain.CurrentDomain.UnhandledException`可以触发任意线程的任意未处理异常，但是从CLR 2.0开始，CLR会在事件处理器执行完毕之后强制关闭应用程序。然而，在应用程序配置文件中添加以下代码，就可以防止应用程序关闭：

```
<configuration>
  <runtime>
    <legacyUnhandledExceptionPolicy enabled="1" />
  </runtime>
</configuration>
```

在一些运行在多个应用程序域的程序上，这是非常有用的（第24章）：如果非默认应用程序域中出现未处理异常，则可以销毁并重新创建备用域，而不要重启整个应用程序。

14.2.8 前台线程与后台线程

默认情况下，显式创建的线程都是前台线程（*foreground thread*）。无论是否还有后台线程（background thread）运行，只要有一个前台线程仍在运行，整个应用程序就会保持运行状态。当所有前台线程结束时，应用程序就会停止，而且所有仍在运行的后台线程也会随之中止。

提示：线程的前台/后台状态与线程的优先级（执行时间分配）无关。

使用线程的`IsBackground`属性，可以查询或修改线程的后台状态：

```
static void Main (string[] args)
{
Thread worker = new Thread ( () => Console.ReadLine() );
if (args.Length > 0) worker.IsBackground = true;
worker.Start();
}
```

如果程序调用时不带任何参数，那么工作者线程会假定线程处于前台状态，然后一直在ReadLine语句上等待按下Enter键。同时，主线程退出，但是应用程序仍然继续运行，因为还有一个前台线程仍在运行。另一方面，如果给Main()传递一个参数，那么工作者线程就会转变为后台状态，因此当主线程停止时，整个程序也会立即退出（中止ReadLine）。

如果进程按照这种方式中止，那么在后台线程执行堆中的所有finally语句块都不会执行。如果程序使用finally语句块执行一些清理操作（如删除临时文件），那么可以通过联合线程或发送信号结构（参见第14.2.10节“发送信号”），显式等待这些后台线程停止，然后再退出应用程序，就可以避免这个问题。无论采用哪一种方法，都必须指定一个超时时间，这样才能够抛弃一个无法按时结束的问题线程，否则应用程序必须通过用户干预（使用TaskManager）才能正常关闭。

前台线程不需要进行这样的处理，但是必须注意，一定不能出现可能导致线程无法结束的缺陷。导致应用程序无法正常退出的常见原因之一就是出现了活跃的前台线程。

14.2.9 线程优先级

线程的Priority属性可以确定它与其他激活线程在操作系统中的相对执行时间长短，具体的优先级包括：

```
enum ThreadPriority { Lowest, BelowNormal, Normal, AboveNormal, Highest }
```

如果同时激活多个线程，优先级就会变得很重要。提高一个线程的优先级时，要注意不要过度抢占其他线程的执行时间。如果希望一个线程拥有比其他进程的线程更高的优先级，那么还必须使用System.Diagnostics的Process类，提高进程本身的优先级：

```
using (Process p = Process.GetCurrentProcess())
  p.PriorityClass = ProcessPriorityClass.High;
```

这种方法非常适合于一些工作量较少但要求较低延迟时间（能够快速响应）的UI进程中。在计算密集特别是带有用户界面的应用程序中，提高进程优先级可能会抢占其他进程的执行时间，从而影响整个计算机的运行速度。

14.2.10 发送信号

有时候，一个线程需要等待来自其他线程的通知，这就是所谓的发送信号（*signaling*）。最简单的发送信号结构是ManualResetEvent。在一个ManualResetEvent上调用WaitOne，可以阻塞当前线程，使之一直等待另一个线程通过调用Set “打开”信号。下面的例子启动了一个线程，一直等待ManualResetEvent到达。它会保持阻塞2秒钟，直至主线程发送信号：

```
var signal = new ManualResetEvent (false);

new Thread (() =>
{
  Console.WriteLine ("Waiting for signal...");
```

```
    signal.WaitOne();
    signal.Dispose();
    Console.WriteLine ("Got signal!");
  }).Start();

  Thread.Sleep(2000);
  signal.Set(); // "打开"信号
```

在调用Set之后，信号仍然保持打开；调用Reset，就可以再次将它关闭。ManualResetEvent是CLR提供的多个信号发送结构之一，第22章将详细介绍所有的结构。

14.2.11 富客户端应用程序的线程

在WPF、Metro和Windows窗体应用程序中，在主线程上执行长时间操作可能会影响应用程序响应速度，因为主线程还需要处理消息循环，用于执行渲染和处理键盘与鼠标事件。

常用的方法是启动一个工作者线程，执行耗费时间的操作。工作者线程的代码负责执行耗费时间的操作，然后在完成之后更新UI。然而，所有富客户端应用程序采用的线程模型规定，只有创建UI元素和控件的线程可以访问它们（通常是主UI线程）。如果违反这个要求，则可能导致不可预见的行为或者抛出异常。

因此，如果想要在工作者线程上更新UI，那么必须将请求转发给UI线程，技术上称为编列（marshal）。实现这个操作的底层方法有（稍后，我们将讨论基于这些技术创建的其他解决方案）：

- 在WPF中，调用元素上Dispatcher对象的BeginInvoke或Invoke。
- 在Metro应用中，调用Dispatcher对象的RunAsync或Invoke。
- 在Windows窗体中，调用控件的BeginInvoke或Invoke。

所有这些方法都接受一个引用所执行方法的代理。BeginInvoke/RunAsync会将这个代理添加到UI线程的*消息队列*上（也是处理键盘、鼠标和计时器事件的队列）。Invoke也会执行相同的操作，但是它会阻塞，直至UI线程读取或处理了这些消息。因此，使用Invoke，可以从方法得到一个返回值。如果不需要返回值，则适合使用BeginInvoke/RunAsync，因为它们不会阻塞调用者，也不会造成死锁（参见第22.2.7节“死锁”）。

提示：可以想象，当调用Application.Run时，就会执行下面的伪代码：

```
while (!thisApplication.Ended)
{
  wait for something to appear in message queue
  Got something: what kind of message is it?
  Keyboard/mouse message -> fire an event handler
  User BeginInvoke message -> execute delegate
  User Invoke message -> execute delegate & post result
}
```

这种循环使工作者线程将执行代理编列到UI线程上。

例如，有一个WPF窗口包含一个文本框txtMessage，我们希望用一个工作者线程在执行一个耗费时间的任务之后（这里通过调用Thread.Sleep来模拟）更新它的内容。下面是具体的实现过程：

并发与异步

```
partial class MyWindow : Window
{
  public MyWindow()
  {
    InitializeComponent();
    new Thread (Work).Start();
  }

  void Work()
  {
    Thread.Sleep (5000); // 模拟耗费时间的任务
    UpdateMessage ("The answer");
  }

  void UpdateMessage (string message)
  {
    Action action = () => txtMessage.Text = message;
    Dispatcher.BeginInvoke (action);
  }
}
```

运行这段代码，立刻会出现一个响应窗口。在5秒之后，它会更新文本框。这段代码与Windows窗体相似，唯一不同的是换成调用窗体的BeginInvoke方法：

```
void UpdateMessage (string message)
{
  Action action = () => txtMessage.Text = message;
  this.BeginInvoke (action);
}
```

> **多个UI线程**
>
> UI线程也可以有多个，但是每一个线程要对应不同的窗口。最有代表性的场景是，如果一个应用程序有多个顶级窗口，通常称为单文档界面（Single Document Interface, SDI）应用程序，如Microsoft Word。每一个SDI窗口通常会在任务栏将自己显示为独立的应用程序，而且与其他SDI窗口在功能上完全独立。为每一个此类窗口指定独立的UI线程，那么每一个窗口都具有更好的独立响应性能。

14.2.12 同步上下文

System.ComponentModel命名空间中有一个抽象类SynchronizationContext，它实现了线程编列的一般化。

WPF、Metro和Windows窗体都定义和实例化了SynchronizationContext的子类，当运行在UI线程上时，它可以通过静态属性SynchronizationContext.Current获得。捕获这个属性，将来就可以在工作者线程上提交数据到UI控件：

```
partial class MyWindow : Window
{
  SynchronizationContext _uiSyncContext;

  public MyWindow()
```

```
    {
      InitializeComponent();
      //为UI线程获得同步上下文:
      _uiSyncContext = SynchronizationContext.Current;
      new Thread (Work).Start();
    }

    void Work()
    {
      Thread.Sleep (5000); // 模拟耗时的任务
      UpdateMessage ("The answer");
    }

    void UpdateMessage (string message)
    {
      // 将代理编列到UI线程上:
      _uiSyncContext.Post (_ => txtMessage.Text = message);
    }
  }
```

这是很有用的，因为同样的方法适用于WPF、Metro和Windows窗体。（SynchronizationContext还有一个专门用在ASP.NET的子类，它这时作为一个更微妙的角色，保证按照异步操作方式处理页面处理事件，并且保留HttpContext。）

在Dispatcher或Control上调用Post与调用BeginInvoke的效果相同；另外Send方法与Invoke的效果相同。

提示： Framework 2.0引入了BackgroundWorker类，它使用SynchronizationContext类简化富客户端应用程序的工作者线程。BackgroundWorker 增加了相同的Tasks和异步功能（后面会介绍），它也使用SynchronizationContext。

14.2.13 线程池

无论何时启动一个线程，都需要一定时间（几百毫秒）用于创建新的局部变量堆。线程池（thread pool）预先创建了一组可回收线程，因此可以缩短这段过载时间。要实现高效的并行编程和细致的并发性，必须使用线程池；它可用于运行一些短暂操作，而不会受到线程启动过载的影响。

在使用线程池中的线程（池化线程）时，还需要考虑下面这些问题：

- 由于不能设置池化线程的Name，因此会增加代码调试难度（虽然在Visual Studio的Threads窗口中调试时，可以附加一些描述）。
- 池化线程通常都是后台线程。
- 池化线程阻塞会影响性能（参见本节的“线程池整洁性”）。

池化线程的优先级可以随意修改——在释放回线程池时，优先级会恢复为普通级别。

使用属性Thread.CurrentThread.IsThreadPoolThread，可以确定当前是否运行在一个池化线程上。

1. 进入线程池

在池化线程上运行代码的最简单方法是使用Task.Run（下一节将深入介绍这个方法）：

```
// Task位于System.Threading.Tasks之中
Task.Run (() => Console.WriteLine ("Hello from the thread pool"));
```

由于Framework 4.0之前不支持任务，所以可以改为调用ThreadPool.QueueUserWorkItem：

```
ThreadPool.QueueUserWorkItem (notUsed => Console.WriteLine ("Hello"));
```

提示：使用线程池的情况有：

- WCF、远程处理（Remoting）、ASP.NET和ASMX Web Services应用服务器
- `System.Timers.Timer`和`System.Threading.Timer`
- 第23章介绍的并行编程结构
- `BackgroundWorker`类（现在是多余的）
- 异步代理（现在也是多余的）

---

2. 线程池整洁性

线程池还有另一个功能，即保证计算密集作业的临时过载不会引起CPU超负荷（*oversubscription*）。超负荷是指激活的线程数量多于CPU内核数量，因此操作系统必须按时间片执行线程调度。超负荷会影响性能，因为划分时间片需要大量的上下文切换开销，并且可能使CPU缓存失效，而这是现代处理器实现高性能的必要条件。

CLR能够将任务进行排序，并且控制任务启动数量，从而避免线程池超负荷。它首先运行与硬件内核数量一样多的并发任务，然后通过爬山算法调整并发数量，在一个方向上不停调整工作负载。如果吞吐量提升，那么它会在这个方向上继续调整（否则换到另一个方向）。这样就保证能够发现最优性能曲线——即使是计算机上同时发生的活动。

如果满足以下两个条件，则适合使用CLR的策略：

- 大多数工作项目的运行时间都非常短（小于250ms，最理想情况是小于100ms），这样CLR就有大量的机会可以测量和调整。
- 线程池不会出现大量将大部分时间都浪费在阻塞上的作业。

阻塞是很麻烦的，因为它会让CLR错误地认为它占用了大量CPU。CLR能够检测并补偿（往池中注入更多的线程），但是这可能使线程池受到后续超负荷的影响。此外，这样也会增加延迟，因为CLR会限制注入新线程的速度，特别是应用程序生命周期的前期（在客户端操作系统上更严重，因为它有严格的低资源消耗要求）。

如果想要提高CPU的利用率（例如，通过第23章的并行编程API），那么一定要保持线程池的整洁性。

14.3 任务

线程是创建并发的底层工具，因此它具有一定的局限性。特别是：

- 虽然很容易向启动的线程传入数据，但是并没有简单的方法可以从联合（`Join`）线程得到“返回值”。因此，必须创建一些共享域。当操作抛出一个异常时，捕捉和处理异常也是非常麻烦的。

- 在线程完成之后，无法再次启动该线程；相反，只能够联合（Join）它（在进程中阻塞当前线程）。

这些局限性会影响并发性的实现；换而言之，不容易通过组合较小的并发操作实现较大的并发操作（这对于异步编程而言非常重要，后面的章节将对此进行介绍）。因此，这会增加对手工同步处理（加锁、发送信号等）的依赖，而且很容易出现问题。

直接使用线程也会对性能产生影响，具体见第14.2.13节“线程池”。而且，如果需要运行大量并发I/O密集操作，那么基于线程的方法仅仅在线程过载方面就会消耗大量的内存。

Task类可以解决所有这些问题。与线程相比，Task是一个更高级的抽象概念，它表示一个通过或不通过线程实现的并发操作。任务是可组合的（*compositional*）——使用延续（*continuation*）将它们串联在一起。它们可以使用线程池减少启动延迟，而且它们可以通过TaskCompletionSource使用回调方法，避免多个线程同时等待I/O密集操作。

Task类型是Framework 4.0引入的，作为并行编程库的组成部分。然而，它们后来（通过使用等待者awaiter）进行了很多改进，从而在常见并发场景中发挥越来越大的作用，并且也是C# 5.0异步功能的基础类型。

提示：专门用于并行编程的任务特性将在第23章中介绍。

14.3.1 启动任务

从Framework 4.5开始，启动一个由后台线程实现的Task，最简单的方法是使用静态方法Task.Run（Task类位于System.Threading.Tasks命名空间）。调用时只需要传入一个Action代理：

```
Task.Run (() => Console.WriteLine ("Foo"));
```

Task.Run是Framework 4.5新引入的方法。在Framework 4.0中，调用Task.Factory.StartNew，可以实现相同的效果。前者相当于是后者的快捷方式。

提示：Task默认使用池化线程，它们都是后台线程。这意味着当主线程结束时，所有任务也会随之停止。因此，要在控制台应用程序中运行这些例子，必须在启动任务之后阻塞主线程。例如，挂起（Waiting）该任务，或者调用Console.ReadLine：

```
static void Main()
{
  Task.Run (() => Console.WriteLine ("Foo"));
  Console.ReadLine();
}
```

书中的LINQPad相关示例省略了Console.ReadLine，因为LINQPad进程会保持后台线程的运行。

采用这种方式调用Task.Run，其效果与下面启动线程的方式类似（唯一不同的是没有隐含使用线程池，下面我们将介绍这个问题）：

```
new Thread (() => Console.WriteLine ("Foo")).Start();
```

Task.Run会返回一个Task对象，它可用于监控任务执行过程，这一点与Thread对象不同。（然而，注意这里没有调用Start，因为Task.Run创建的是“热”任务；相反，如果想要创建“冷”任务，则必须使用Task的构造函数，但是这种用法在实践中很少使用。）

提示：任务的Status属性可用于跟踪任务的执行状态。

---

1. 等待（Wait）

调用任务的Wait方法，可以阻塞任务，直至任务完成，其效果等同于调用线程的Join：

```
Task task = Task.Run (() =>
{
  Thread.Sleep (2000);
  Console.WriteLine ("Foo");
});
Console.WriteLine (task.IsCompleted); // False
task.Wait(); // 阻塞，直至任务完成
```

可以在Wait中指定一个超时时间和一个取消令牌（用于提前中止等待状态，参见第14.6.1节“取消”）。

2. 长任务

在默认情况下，CLR会运行在池化线程上，这种线程非常适合执行短计算密集作业。如果要执行长阻塞操作（如上面的例子），则可以按以下方式避免使用池化线程：

```
Task task = Task.Factory.StartNew (() => ...,
                                   TaskCreationOptions.LongRunning);
```

提示：在池化线程上运行一个长任务问题并不大；但是如果要同时运行多个长任务（特别是会阻塞的任务），则会对性能产生影响。在这种情况下，通常更好的方法是使用TaskCreationOptions.LongRunning：

- 如果是运行I/O密集任务，则可以使用TaskCompletionSource和异步函数（*asynchronous functions*），通过回调函数（延续）实现并发性，而不通过线程实现。
- 如果是运行计算密集任务，则可以使用一个生产者/消费者队列，控制这些任务的并发数量，避免出现线程和进程阻塞的问题（参见第23.7.1“编写生产者/消费者队列”）。

14.3.2 返回值

Task有一个泛型子类Task<TResult>，它允许任务返回一个值。调用Task.Run，传入一个Func<TResult>代理（或者兼容的Lambda表达式），代替Action，就可以获得一个Task<TResult>：

```
Task<int> task = Task.Run (() => { Console.WriteLine ("Foo"); return 3; });
// ...
```

然后，查询Result属性，就可以获得结果。如果任务还没有完成，那么访问这个属性会阻塞当前线

程，直至任务完成：

```
int result = task.Result; // 如果任务未完成，则阻塞
Console.WriteLine (result); // 3
```

下面的例子创建了一个任务，它使用LINQ计算前3百万个整数（从2开始）中的素数个数：

```
Task<int> primeNumberTask = Task.Run (() =>
  Enumerable.Range (2, 3000000).Count (n =>
    Enumerable.Range (2, (int)Math.Sqrt(n)-1).All (i => n % i > 0)));

Console.WriteLine ("Task running...");
Console.WriteLine ("The answer is " + primeNumberTask.Result);
```

这段代码会打印“Task running...”，然后在几秒钟之后打印结果216815。

提示：Task<TResult>可以看作是“将来”，其中封装了后面很快生效的Result。

有趣的是，当Task和Task<TResult>第一次出现在早期的CTP时，后者实际上是Future<TResult>。

14.3.3 异常

与线程不同，任务可以随时抛出异常。所以，如果任务中的代码抛出一个未处理异常（换而言之，任务出错），那么这个异常会自动传递到调用Wait()的任务上或者访问Task<TResult>的Result属性的代码上：

```
// 启动一个会抛出NullReferenceException的Task：
Task task = Task.Run (() => { throw null; });
try
{
  task.Wait();
}
catch (AggregateException aex)
{
  if (aex.InnerException is NullReferenceException)
    Console.WriteLine ("Null!");
  else
    throw;
}
```

（CLR会将异常封装在AggregateException中，从而更适合并行编程场景；第23章将介绍这些内容。）

使用Task的IsFaulted和IsCanceled属性，就可以不重新抛出异常而检测出错的任务。如果这两个属性都返回false，则表示没有错误发生；如果IsCanceled为true，则任务抛出了OperationCanceledOperation（参见第14.6.1节“取消”）；如果IsFaulted为true，则任务抛出了另一种异常，而Exception属性包含了该错误。

1. 异常和自主任务

如果使用了自主的“设置后忘记的”任务（不通过Wait()或Result控制的任务，或者实现相同效果的延续），那么最好在任务代码中显式声明异常处理，避免出现静默错误，就像线程的异常处理一样。

自主任务上的未处理异常称为未监控异常（*unobserved exception*），在CLR 4.0中，它们实际上会中止程序（当任务跳出运行范围并被垃圾回收器回收时，CLR会在终结线程上重新抛出异常）。这种方式有利于提醒一些悄悄发生的问题；然而，错误发生时间可能并不准确，因为垃圾回收器可能会明显滞后于发生问题的任务。因此，在发现这种行为具有复杂的不同步性模式时（参见第14.5.2节“并行性”和第14.6.4节“WhenAll”），CLR 4.5删除了这个特性。

提示：如果异常仅仅表示无法获得一些不重要的结果，那么忽略异常是最好的处理方式。例如，如果用户取消了一个网页下载操作，那么我们就不需要关心网页是否存在。

如果异常反映了程序的重大缺陷，那么忽略异常是很有问题的。这其中的原因有两个：

- 这个缺陷可能使程序处于无效状态。
- 这个缺陷可能导致更多的异常发生，而且无法记录初始错误也会增加诊断难度。

使用静态事件`TaskScheduler.UnobservedTaskException`，可以在全局范围订阅未监控的异常；处理这个事件，然后记录发生的错误，是一个很好的异常处理方法。

未监控异常有一些有趣的细微差别：

- 如果在超时周期之后发生错误，那么等待超时的任务将生成一个未监控异常。
- 在错误发生之后检查任务的`Exception`属性，会使异常变成“已监控异常”。

14.3.4 延续

延续（continuation）会告诉任务在完成后继续执行下面的操作。”延续通常由一个回调方法实现，它会在操作完成之后执行一次。给一个任务附加延续的方法有两种。第一种方法是Framework 4.5新增加的，它非常重要，因为C# 5.0的异步功能使用了这种方法，我们马上会介绍这种方法。第14.3.2节“返回值”的素数计算例子可以说明延续的用法：

```
Task<int> primeNumberTask = Task.Run (() =>
  Enumerable.Range (2, 3000000).Count (n =>
    Enumerable.Range (2, (int)Math.Sqrt(n)-1).All (i => n % i > 0)));

var awaiter = primeNumberTask.GetAwaiter();
awaiter.OnCompleted (() =>
{
  int result = awaiter.GetResult();
  Console.WriteLine (result); // 打印结果
});
```

调用`GetAwaiter`会返回一个等待者（*awaiter*）对象，它的方法会让先导（*antecedent*）任务（`primeNumberTask`）在完成（或出错）之后执行一个代理。已经完成的任务也可以附加一个延续，这时延续就马上执行。

提示：等待者（*awaiter*）可以是任意对象，但是它必须包含前面所示两个方法（`OnCompleted`和`GetResult`）和一个Boolean类型属性`IsCompleted`的对象，它不需要实现包含所有这些成员的特定接口或继承特定基类（但是`OnCompleted`属于接口`INotifyCompletion`）。我们将在第14.5节“C# 5.0的异步函数”中介绍这种模式的重要性。

如果先导任务出现错误，那么当延续代码调用awaiter.GetResult()时就会重新抛出异常。我们不需要调用GetResult，而是直接访问先导任务的Result属性。调用GetResult的好处是，当先导任务出现错误时，异常可以直接抛出，而不会封装在AggregateException之中，从而可以实现更简单且更清晰的异常捕捉代码。

对于非泛型任务，GetResult()会返回空值（void），然后它的实用函数会单独重新抛出异常。

如果出现同步上下文，那么会自动捕捉它，然后将延续提交到这个上下文中。这对于富客户端应用程序而言非常实用，因为会将延续弹回UI线程。然而，在编写库时，通常不采用这种方法，因为开销相对较大的UI线程只会在离开库时运行一次，而不会在方法调用期间运行。因此，我们可以使用ConfigureAwait代替它：

```
var awaiter = primeNumberTask.ConfigureAwait(false).GetAwaiter();
```

如果不出现同步上下文或者使用ConfigureAwait(false)，那么通常延续会运行在先导任务所在的线程上，从而避免不必要的过载。

另一种附加延续的方法是调用任务的ContinueWith方法：

```
primeNumberTask.ContinueWith (antecedent =>
{
  int result = antecedent.Result;
  Console.WriteLine (result);                // 打印出123
});
```

ContinueWith本身会返回一个Task，它非常适用于添加更多的延续。然而，如果任务出现错误，我们必须直接处理AggregateException，然后编写额外代码，将延续编列到UI应用程序中（参见第23.4.5节“任务调度器”）。而在非UI上下文中，如果想要让延续运行在同一个线程上，则必须指定TaskContinuationOptions.ExecuteSynchronously；否则它会弹回线程池。ContinueWith特别适用于并行编程场景；我们将在第23.4.4节“延续任务”中介绍。

14.3.5 TaskCompletionSource

前面介绍了Task.Run如何创建一个在池化（或非池化）线程上运行代理的任务。另一种创建任务的方法是使用TaskCompletionSource。

TaskCompletionSource可以创建一个任务，它不包含任何必须在后面启动和结束的操作。它的实现原理是提供一个可以手工操作的“附属”任务——用于指示操作完成或出错的时间。这种方法非常适合于I/O密集作业：可以利用所有任务的优点（它们能够生成返回值、异常和延续），但不会在操作执行期间阻塞线程。

TaskCompletionSource用法很简单，直接初始化就可以。它包含一个Task属性，它返回一个可以等待和附加延续的任务——和其他任务一样。然而，这个任务完全通过下面的方法由TaskCompletionSource对象进行控制：

```
public class TaskCompletionSource<TResult>
{
  public void SetResult (TResult result);
  public void SetException (Exception exception);
  public void SetCanceled();
  public bool TrySetResult (TResult result);
```

```
    public bool TrySetException (Exception exception);
    public bool TrySetCanceled();
    ...
  }
```

调用这些方法，就可以给任务发送信号，将任务修改为完成、异常或取消状态（我们将在第23.2.7节“取消”中介绍）。这些方法都只能调用一次：如果多次调用SetResult、SetException或SetCanceled，它们就会抛出异常，而Try*等方法则会返回false。

下面的例子会在等待5秒钟之后打印出42：

```
var tcs = new TaskCompletionSource<int>();

new Thread (() => { Thread.Sleep (5000); tcs.SetResult (42); })
  .Start();

Task<int> task = tcs.Task;
Console.WriteLine (task.Result); // 42
```

使用TaskCompletionSource，可以编写出自定义的Run方法：

```
Task<TResult> Run<TResult> (Func<TResult> function)
{
  var tcs = new TaskCompletionSource<TResult>();
  new Thread (() =>
  {
    try { tcs.SetResult (function()); }
    catch (Exception ex) { tcs.SetException (ex); }
    }).Start();
    return tcs.Task;
}
...
Task<int> task = Run (() => { Thread.Sleep (5000); return 42; });
```

调用这个方法等同于使用TaskCreationOptions.LongRunning选项调用Task.Factory.StartNew，请求生成一个非池化线程。

TaskCompletionSource的真正作用是创建一个不绑定线程的任务。例如，假设一个任务需要等待5秒钟，然后再返回数字42。我们可以使用Timer类实现，而不需要使用线程，由CLR（以及操作系统）在x毫秒之后触发一个事件（定时器将在第22章中介绍）：

```
Task<int> GetAnswerToLife()
{
  var tcs = new TaskCompletionSource<int>();
  // 创建一个每隔5000毫秒触发一次的定时器：
  var timer = new System.Timers.Timer (5000) { AutoReset = false };
  timer.Elapsed += delegate { timer.Dispose(); tcs.SetResult (42); };
  timer.Start();
  return tcs.Task;
}
```

因此，我们的方法会返回一个在5秒钟后完成的任务，其结果是42。通过给任务附加一个延续，就可以在不阻塞任何线程的前提下打印出这个结果：

```
var awaiter = GetAnswerToLife().GetAwaiter();
```

```
awaiter.OnCompleted (() => Console.WriteLine (awaiter.GetResult()));
```

将延迟时间参数化，并且删除返回值，可以优化这段代码，并将它变成一个通用的Delay方法。这意味着，要让它返回一个Task，而不是Task<int>。然而，TaskCompletionSource并没有泛型版本，因此我们无法直接创建一个非泛型任务。但变通方法很简单：因为Task<TResult>派生自Task，所以创建一个TaskCompletionSource<*anything*>，然后将它隐式转换为Task<*anything*>，就可以得到一个Task，具体方法如下：

```
var tcs = new TaskCompletionSource<object>();
Task task = tcs.Task;
```

现在，我们可以编写出通用的Delay方法：

```
Task Delay (int milliseconds)
{
  var tcs = new TaskCompletionSource<object>();
  var timer = new System.Timers.Timer (milliseconds) { AutoReset = false };
  timer.Elapsed += delegate { timer.Dispose(); tcs.SetResult (null); };
  timer.Start();
  return tcs.Task;
}
```

然后，让它5秒钟之后打印出“42”：

```
Delay (5000).GetAwaiter().OnCompleted (() => Console.WriteLine (42));
```

不在线程上使用TaskCompletionSource，意味着只有在延续启动时（5秒钟之后）才创建线程。同时启动10,000个这种操作，而不会出错或超出资源限制，就可以说明这种方法：

```
for (int i = 0; i < 10000; i++)
  Delay (5000).GetAwaiter().OnCompleted (() => Console.WriteLine (42));
```

提示： 定时器会在池化线程上触发它们的回调方法，所以在5秒钟之后，线程池会接收到10,000个在TaskCompletionSource上调用SetResult(null)的请求。如果请求到达速度比处理速度快，那么线程池就会进行排除，然后按照最优的CPU并行原则处理这些请求。这种方法最适合于处理短执行时间的线程阻塞作业，例如：线程阻塞作业仅仅是调用SetResult，然后将延续提交到异步上下文（在UI应用程序中），或者操作延续本身（Console.WriteLine(42)）。

14.3.6 Task.Delay

Delay方法非常实用，因此它成为Task类的一个静态方法：

```
Task.Delay (5000).GetAwaiter().OnCompleted (() => Console.WriteLine (42));
```

或者：

```
Task.Delay (5000).ContinueWith (ant => Console.WriteLine (42));
```

Task.Delay是Thread.Sleep的异步版本。

14.4 异步原则

在演示TaskCompletionSource的过程中，最终编写的是异步方法。在这一节中，我们将具体说明异步操作的定义，并且说明如何实现异步编程。

14.4.1 同步操作与异步操作

同步操作（*synchronous operation*）在返回调用者之前才完成它的工作。

在大多数情况下，异步操作（*asynchronous operation*）则在返回调用者之后才完成它的工作。

我们平常编写和调用的大多数方法都是同步方法。例如，Console.WriteLine或Thread.Sleep。异步方法使用频率较小，并且需要初始化并发编程，因为它的作业会继续与调用者并行处理。异步方法一般会快速（或立刻）返回给调用者；因此，它们也称为非阻塞方法。

到目前为止，我们学习的异步方法都可以认为是通用方法：

- Thread.Start
- Task.Run
- 给任务附加延续的方法

此外，在第14.2.12节“同步上下文”中介绍的一些方法（Dispatcher.BeginInvoke、Control.BeginInvoke和SynchronizationContext.Post）也是异步的，之前介绍的TaskCompletionSource也是，其中包括Delay。

14.4.2 什么是异步编程

异步编程的原则是以异步方式编写运行时间很长（或可能很长）的函数。这与编写长运行时间函数的传统同步方法相反，它会在一个新线程或任务上调用这些函数，从而实现所需要的并发性。

异步方法的不同点是它会在长运行时间函数之中而非在函数之外初始化并发性。这样做有两个优点：

- I/O密集并发性的实现不需要绑定线程（具体参见“TaskCompletionSource”的演示），因此可以提高可伸缩性和效率。
- 富客户端应用程序可以减少工作者线程的代码，因此可以简化线程的安全实现。

这样也产生了两种异步编程特殊用法。第一种是编写高效处理并发I/O的应用程序（通常位于服务器端）。关键不在于线程安全性（因为这里很少使用共享状态），而在于线程效率；特别是，每一个网络请求不会独自消耗一个线程。因此，在这种环境中，只有I/O密集操作可以受益于异步性。

第二种是简化富客户端应用程序的线程安全性。这与程序的增长速度密切相关，因为为了降低复杂性，通常我们需要将大函数重构为多个小方法，从而产生一连串相互调用的方法（调用图）。

在传统的同步调用图中，如果图中出现一个运行时间很长的操作，我们就必须将整个调用图转移到一个工作者线程中，以保证UI的高速响应。因此，我们最终会得到一个跨越许多方法的并发操作（过程级并发性），而且这时需要考虑图中每一个方法的线程安全性。

使用异步调用图，就可以在真正需要时才启动线程，因此可以降低调用图中线程的使用频率（或者在特定操作中完全不需要使用线程，如I/O密集操作）。其他方法则可以在UI线程上运行，从而可以大

大简化线程安全性的实现。

这样就可以实现细致的并发性——它由一系列并发操作构成，这些操作之间再插入一些UI线程执行过程。

提示： 如果要利用这种实现的优点，I/O密集和计算密集操作都必须采用异步方式实现；常用的经验法则是以异步方式处理所有运行时间超过50毫秒的操作。

（另一方面，过度使用细致异步性可能会对性能产生影响，因为异步操作会引起一定的超载——参见第14.5.6节“优化”。）

在本章中，我们主要关注于富客户端场景，因为这是两种情况中较复杂的一种。在第16章中，我们将举例说明I/O密集场景（参见第16.10.1节“TCP并发性”和第16.6节“编写HTTP服务器”）。

提示： Metro和Silverlight .NET鼓励使用异步编程，甚至一些运行时间较长的方法完全不会出现同步执行版本。相反，它们使用一些可以返回任务（或者可以通过扩展方法转换为任务的对象）的异步方法。

14.4.3 异步编程与延续

任务非常适合异步编程，因为它们支持异步编程所需要的延续（例如前面TaskCompletionSource所使用的Delay方法）。编写Delay时使用了TaskCompletionSource，它是一种实现“底层”I/O密集异步方法的标准方法。

在计算密集方法中，我们使用Task.Run创建线程阻塞并发性。只需要直接给调用者返回一个任务，就可以创建一个异步方法。异步编程的不同点在于，它的目的是在调用图的最底层实现异步操作，这样富客户端应用程序的上层方法可以运行在UI线程上，自由访问控件和共享状态，而不会发生线程安全问题。为了说明这一点，可以参考下面这个计算素数个数的方法，它使用了所有的可用内存（ParallelEnumerable将在第23章介绍）：

```
int GetPrimesCount (int start, int count)
{
  return
    ParallelEnumerable.Range (start, count).Count (n =>
      Enumerable.Range (2, (int)Math.Sqrt(n)-1).All (i => n % i > 0));
}
```

它的实现细节并不重要，我们主要关心它的运行时间。结合下面这个方法，可以对比它们的运行时间：

```
void DisplayPrimeCounts()
{
  for (int i = 0; i < 10; i++)
    Console.WriteLine (GetPrimesCount (i*1000000 + 2, 1000000) +
      " primes between " + (i*1000000) + " and " + ((i+1)*1000000-1));
  Console.WriteLine ("Done!");
}
```

输出结果是：

```
78498 primes between 0 and 999999
```

并发与异步

```
70435 primes between 1000000 and 1999999
67883 primes between 2000000 and 2999999
66330 primes between 3000000 and 3999999
65367 primes between 4000000 and 4999999
64336 primes between 5000000 and 5999999
63799 primes between 6000000 and 6999999
63129 primes between 7000000 and 7999999
62712 primes between 8000000 and 8999999
62090 primes between 9000000 and 9999999
```

这样我们就可以查看整个调用图，其中DisplayPrimeCounts调用GetPrimesCount。前者使用Console.WriteLine来简化实现，但是在现实中它很可能会用于更新富客户端应用程序的UI控件（后面会介绍这一点）。采用下面的方法，就可以为这个调用图创建过程级并发性：

```
Task.Run (() => DisplayPrimeCounts());
```

相反，如何采用细致的异步方法实现，则需要编写异步的GetPrimesCount：

```
Task<int> GetPrimesCountAsync (int start, int count)
{
return Task.Run (() =>
ParallelEnumerable.Range (start, count).Count (n =>
Enumerable.Range (2, (int) Math.Sqrt(n)-1).All (i => n % i > 0)));
}
```

14.4.4 语言支持的重要性

由于现在必须修改DisplayPrimeCounts，所以它会调用GetPrimesCountAsync。这时需要用到C#新增加的await和async关键字，因为这是最简单的实现方法。如果直接将循环修改为：

```
for (int i = 0; i < 10; i++)
{
  var awaiter = GetPrimesCountAsync (i*1000000 + 2, 1000000).GetAwaiter();
  awaiter.OnCompleted (() =>
    Console.WriteLine (awaiter.GetResult() + " primes between... "));
}
Console.WriteLine ("Done");
```

那么这个循环将快速完成10次循环（这些方法都是非阻塞的），然后全部10个操作都会并行执行（后面是提前输出的“Done”）。

提示：在这种情况中，并行执行这些任务是不可取的，因为它们的内部实现已经是并行的；它只有让我们等待更长时间才能看到第一个结果（而且会搞乱顺序）。

然而，更常见的用法是用于产生串行执行的任务，即Task B依赖于Task A。例如，在抓取网页时，DNS查询发生在HTTP请求发送之前。

如果要让它们按顺序执行，则必须在延续上触发下一次循环。这意味着，必须抛弃for循环，将它变成延续的递归调用：

```
void DisplayPrimeCounts()
{
```

```
    DisplayPrimeCountsFrom (0);
  }

  void DisplayPrimeCountsFrom (int i)
  {
    var awaiter = GetPrimesCountAsync (i*1000000 + 2, 1000000).GetAwaiter();
    awaiter.OnCompleted (() =>
    {
      Console.WriteLine (awaiter.GetResult() + " primes between...");
      if (i++ < 10) DisplayPrimeCountsFrom (i);
      else Console.WriteLine ("Done");
    });
  }
```

如果想要使DisplayPrimesCount本身变成异步处理，返回一个任务并发送完成信号，那么情况会变得更加复杂。这时，必须创建一个TaskCompletionSource：

```
  Task DisplayPrimeCountsAsync()
  {
    var machine = new PrimesStateMachine();
    machine.DisplayPrimeCountsFrom (0);
    return machine.Task;
  }

  class PrimesStateMachine
  {
    TaskCompletionSource<object> _tcs = new TaskCompletionSource<object>();
    public Task Task { get { return _tcs.Task; } }

    public void DisplayPrimeCountsFrom (int i)
    {
      var awaiter = GetPrimesCountAsync (i*1000000+2, 1000000).GetAwaiter();
      awaiter.OnCompleted (() =>
      {
        Console.WriteLine (awaiter.GetResult());
        if (i++ < 10) DisplayPrimeCountsFrom (i);
        else { Console.WriteLine ("Done"); _tcs.SetResult (null); }
      });
    }
  }
```

幸好，C# 5.0的异步函数（asynchronous functions）可以帮我们完成这些操作。使用async和await关键字，可以将程序变成：

```
  async Task DisplayPrimeCounts()
  {
    for (int i = 0; i < 10; i++)
      Console.WriteLine (await GetPrimesCountAsync (i*1000000 + 2, 1000000) +
        " primes between " + (i*1000000) + " and " + ((i+1)*1000000-1));
    Console.WriteLine ("Done!");
  }
```

因此，如果不想增加程序复杂性，那么必须使用async和await关键字实现异步性。下面，让我们了解一下这些关键字的用法。

提示：解决这个问题的另一种方法是命令式循环结构（for、foreach等），但不要使用延续，因为它们依赖于方法的当前本地状态（“循环将多运行多少次？”）

虽然async和await关键字可以解决问题，但是有时候还可以使用另一种方法，即将命令式循环结构替换为函数式等价操作（即LINQ查询）。这是*Reactive Framework* (Rx)的基础，也适合于执行查询操作符处理结果或者组合多个序列。这样做的代价是，为了避免出现阻塞，Rx必须运行在基于推送的序列上，这种方法在理论上比较复杂。

14.5 C# 5.0的异步函数

C# 5.0引入了async和await关键字。这两个关键字可用于编写异步代码，它具有与同步代码相当的结构和简单性，并且摒弃了异步编程的复杂结构。

14.5.1 等待处理

使用await关键字，可以简化延续的附加过程。首先看一个基本过程：

```
var result = await expression;
statement(s);
```

编译器会将它转换为下面具有相同功能的代码：

```
var awaiter = expression.GetAwaiter();
awaiter.OnCompleted (() =>
{
  var result = awaiter.GetResult();
  statement(s);
);
```

提示：编译器还增加了在异步操作完成之后停止延续的代码（参见第14.5.6“优化”），以及用于处理后面章节将会介绍的各种函数的代码。

为了演示它们的用法，让我们先回顾一下前面用于计算素数个数的异步方法：

```
Task<int> GetPrimesCountAsync (int start, int count)
{
  return Task.Run (() =>
    ParallelEnumerable.Range (start, count).Count (n =>
      Enumerable.Range (2, (int)Math.Sqrt(n)-1).All (i => n % i > 0)));
}
```

使用await关键字，就可以将代码变成：

```
int result = await GetPrimesCountAsync (2, 1000000);
Console.WriteLine (result);
```

为了完成编译，我们必须在包含的方法上添加async修饰符：

```
async void DisplayPrimesCount()
{
```

```
    int result = await GetPrimesCountAsync (2, 1000000);
    Console.WriteLine (result);
  }
```

修饰符async会指示编译器将await视为一个关键字，而非在方法中随意添加的修饰符（这样可以保证C# 5.0之前编写并使用await作为修改符的代码不会出现编译错误）。async修饰符只能应用到返回void、（后面介绍的）Task或Task<TResult>的方法（和lambda表达式）上。

提示：async修饰符与unsafe修饰符类似，都不会对方法签名或公共元数据产生影响；它只影响方法内部的执行细节。因此，在接口上添加async是没有意义的。例如，可以在重写的非async虚方法上添加async，但前提是方法签名保持不变。

添加async修饰符的方法就是所谓的异步函数，因为它们通常本身也是异步的。为了说明这一点，我们需要了解异步函数的执行过程。

当遇到await表达式时，通常执行过程会返回调用者，效果就像在循环只添加了yield return。但是，在返回之前，运行时环境会给等待的任务附加一个延续，保证在任务完成时执行过程会跳回方法，然后继续执行方法的剩余代码。如果任务出错，那么它的异常就会重新抛出；如果正常完成，它的返回值则会赋值给await表达式。分析上面的异步方法业务逻辑，就可以总结前面所介绍的整个过程：

```
void DisplayPrimesCount()
{
  var awaiter = GetPrimesCountAsync (2, 1000000).GetAwaiter();
  awaiter.OnCompleted (() =>
  {
    int result = awaiter.GetResult();
    Console.WriteLine (result);
  });
}
```

await之前的表达式通常是一个任务；然而，编译器允许这里使用任意对象，只要它包含能够返回可等待对象的GetAwaiter方法。具体地，这个对象要实现INotifyCompletion.OnCompleted，带有一个正确的类型化GetResult方法和一个bool类型的IsCompleted属性。

注意，这个await表达式返回一个int类型值；因为我们等待的表达式是一个Task<int>（它的GetAwaiter().GetResult()方法返回一个int类型值）。

等待一个非泛型任务也是合法的，它会生成一个void表达式：

```
await Task.Delay (5000);
Console.WriteLine ("Five seconds passed!");
```

1. 获取本地状态

await表达式的最大特点在于它们可以出现在代码的任意位置。具体地，await表达式可以出现在异步方法中除catch或finally语句块、lock表达式、unsafe上下文或可执行入口（main方法）之外的任意位置。

在下面的例子中，await出现在循环结构中：

```
async void DisplayPrimeCounts()
```

```
{
  for (int i = 0; i < 10; i++)
    Console.WriteLine (await GetPrimesCountAsync (i*1000000+2, 1000000));
}
```

在第一次执行GetPrimesCount时，由于出现await表达式，所以执行过程会立刻返回调用者。当方法完成（或出错）时，执行过程会从停止之处恢复运行，同时保留本地变量和循环计数的值。

如果不使用await关键字，最简单的等价价码就是“语言支持的重要性”这节的示例代码。然而，编译器会采用一些更为通用的重构策略，将这些方法转换为状态机（而非迭代器）。

编译器会使用延续（通过等待者模式）在await表达式之后恢复执行。这意味着，如果运行在富客户端的UI线程上，那么同步上下文可以保证将执行恢复到同一个线程上。否则，执行过程会恢复到任务所在的任意线程上。线程更换不会影响执行顺序，而且除非使用了线程亲和操作（如通过使用线程本地存储，参见第22.8节“线程本地存储”），否则它的结果也不会发生变化。这个过程就像在乘坐一辆出租车游览某个城市一样。在同步上下文中，代码总是使用同一辆出租车（线程）；而在非同步上下文中，每一次都会使用不同的出租车。但是，无论是哪一种情况，旅程（结果）都是相同的。

2. UI上的等待处理

通过编写一个简单的UI，保持响应速度同时调用一个计算密集方法，就可以演示异步函数的实际用法。下面是一个同步实现：

```
class TestUI : Window
{
  Button _button = new Button { Content = "Go" };
  TextBlock _results = new TextBlock();

  public TestUI()
  {
    var panel = new StackPanel();
    panel.Children.Add (_button);
    panel.Children.Add (_results);
    Content = panel;
    _button.Click += (sender, args) => Go();
  }

  void Go()
  {
    for (int i = 1; i < 5; i++)
    _results.Text += GetPrimesCount (i * 1000000, 1000000) +
      " primes between " + (i*1000000) + " and " + ((i+1)*1000000-1) +
      Environment.NewLine;
  }

  int GetPrimesCount (int start, int count)
  {
    return ParallelEnumerable.Range (start, count).Count (n =>
      Enumerable.Range (2, (int) Math.Sqrt(n)-1).All (i => n % i > 0));
  }
}
```

当按下“Go”按钮时，由于执行计算密集代码的时间较长，应用程序的响应速度会很慢。实现相同效果的异步方法包含两个步骤：第一步是切换到前一个例子所使用的异步版本GetPrimesCount：

```
Task<int> GetPrimesCountAsync (int start, int count)
{
  return Task.Run (() =>
    ParallelEnumerable.Range (start, count).Count (n =>
    Enumerable.Range (2, (int) Math.Sqrt(n)-1).All (i => n % i > 0)));
}
```

第二步是将Go方法修改为调用GetPrimesCountAsync：

```
async void Go()
{
  _button.IsEnabled = false;
  for (int i = 1; i < 5; i++)
    _results.Text += await GetPrimesCountAsync (i * 1000000, 1000000) +
      " primes between " + (i*1000000) + " and " + ((i+1)*1000000-1) +
      Environment.NewLine;
  _button.IsEnabled = true;
}
```

这段代码可以说明异步函数的编程简单性：这里采用同步编程方法，但是调用的是异步函数，而非阻塞函数，然后等待（await）它们。只有之中的代码会运行在工作者线程上；Go的代码会“租得”UI线程的时间。可以说，Go在消息循环上以伪并发方式执行（它的执行会被UI线程处理的其他事件打断）。在这个伪并发过程中，只有在await过程才可能出现优先占用。这样就简化了线程安全性：在这个例子中，唯一的问题是它可能发生重入（reentrancy）——在运行时再次单击按钮，但是我们可以通过禁用按钮来避免这个问题。真正的并发发生在调用堆的底层，位于Task.Run调用的代码中。为了利用这种模式的优点，直接并发的代码要避免访问共享状态或UI控件。

另一个例子中，假设不是计算素数个数，而是要下载几个网页，然后计算它们的总大小。Framework 4.5提供了许多返回任务的异步方法，其中一个是System.Net的WebClient类。DownloadDataTaskAsync方法会以异步方式将一个URI下载到一个字节数组中，然后返回一个Task<byte[]>，所以在等待时，我们会得到一个byte[]。

现在，将Go方法重写为：

```
async void Go()
{
  _button.IsEnabled = false;
  string[] urls = "www.albahari.com www.oreilly.com www.linqpad.net".Split();
  int totalLength = 0;
  try
  {
    foreach (string url in urls)
    {
      var uri = new Uri ("http://" + url);
      byte[] data = await new WebClient().DownloadDataTaskAsync (uri);
      _results.Text += "Length of " + url + " is " + data.Length +
                       Environment.NewLine;
      totalLength += data.Length;
    }
    _results.Text += "Total length: " + totalLength;
  }
  catch (WebException ex)
  {
  _results.Text += "Error: " + ex.Message;
```

并发与异步

```
    }
    finally { _button.IsEnabled = true; }
  }
```

同样，这段代码也说明了如何以同步方式实现相同效果——其中包括使用catch和finally语句块。即使在第一次await之后执行过程返回调用者，finally语句块也只能在该方法逻辑完成之后才会执行（由于它的代码执行——或者提前返回或未处理异常）。

仔细考虑将会生成什么样的结果，是很有帮助的。首先，我们需要重新考虑在UI线程上运行消息循环的伪代码：

```
Set synchronization context for this thread to WPF sync context
while (!thisApplication.Ended)
{
  wait for something to appear in message queue
  Got something: what kind of message is it?
    Keyboard/mouse message -> fire an event handler
    User BeginInvoke/Invoke message -> execute delegate
}
```

附加给UI元素的事件处理器将通过这个消息循环执行。当Go方法运行时，执行过程会处理到await表达式为止，然后再返回该消息循环（使UI能够响应后续事件）。然而，编译器的await扩展可以在返回之前创建一个延续，使任务完成时执行过程在原位置恢复。而且，因为我们在一个UI线程上执行等待操作，所以延续会通过消息循环提交到同步上下文中，使我们的Go方法能够以伪并发方式在UI线程上运行。真正的（I/O密集）并发执行发生在DownloadDataTaskAsync的实现过程中。

3. 与过程级并发的比较

在C# 5.0之前，异步编程很难实现，原因不仅仅在于缺少语言支持，还因为.NET框架是通过EAP和APM等模式（参见第14.7节“废弃模式”）实现异步功能，而非通过任务返回方法。

常用变通方法是采用过程级并发实现（事实上，还有一个BackgroundWorker类型可以帮助解决这个问题）。同样是前一个使用GetPrimesCount实现的同步例子，按照下面的方法修改按钮的事件，就可以演示过程级异步性：

```
...
_button.Click += (sender, args) =>
{
  _button.IsEnabled = false;
  Task.Run (() => Go());
};
```

（我们选择使用Task.Run，而不是BackgroundWorker，因为后者没有对这个例子进行任何简化。）在这两种情况中，最终的结果是我们整个同步调用图（Go和GetPrimesCount）都运行在工作者线程上。而且，因为Go会更新UI元素，我们现在必须抛弃使用Dispatcher.BeginInvoke的代码：

```
void Go()
{
  for (int i = 1; i < 5; i++)
  {
    int result = GetPrimesCount (i * 1000000, 1000000);
    Dispatcher.BeginInvoke (new Action (() =>
      _results.Text += result + " primes between " + (i*1000000) +
```

```
      " and " + ((i+1)*1000000-1) + Environment.NewLine));
    }
    Dispatcher.BeginInvoke (new Action (() => _button.IsEnabled = true));
  }
```

与异步实现不同，循环本身运行在工作者线程上。这似乎并无大问题，但即便是在这个简单的例子中，使用多线程也会引入竞争条件。（您是否发现？如果还没有，可以尝试运行这个程序，那么问题很快就会出现。）

实现取消和进度报告，就更可能发生线程安全错误，这一点和方法的其他代码一样。例如，假设循环的上限并不固定，而是来自一个方法调用：

```
for (int i = 1; i < GetUpperBound(); i++)
```

现在，假设GetUpperBound()从加载配置文件中读取上限值，那么它会先执行一个硬盘读取操作。现在，所有代码都运行在工作者线程上，所以这些代码很可能不具备线程安全性。在调用图上层启动工作者线程是很冒险的做法。

14.5.2 编写异步函数

如果使用异步函数，则可以将返回类型void修改为Task，使方法本身适合采用异步实现（即可等待的）。其他方面都不需要修改：

```
async Task PrintAnswerToLife() // 可以返回Task，代替void
{
  await Task.Delay (5000);
  int answer = 21 * 2;
  Console.WriteLine (answer);
}
```

注意，方法体内不需要显式返回一个任务。编译器会负责生成任务，它会在方法完成或者出现未处理异常时发出信号。这样就很容易创建异步调用链：

```
async Task Go()
{
  await PrintAnswerToLife();
  Console.WriteLine ("Done");
}
```

而且，因为Go函数声明中已经添加了Task返回类型，所以Go本身是可等待的。

编译器会扩展异步函数，它会将任务返回给使用TaskCompletionSource的代码，用于创建任务，然后再发送信号或异常中止。

提示： 实际上，编译器会通过System.CompilerServices命名空间的Async*MethodBuilder类型，间接调用TaskCompletionSource。这些类型会处理一些边界情况，如在发生OperationCanceledException时将任务修改为取消状态，以及实现第14.5.5节“异步与同步上下文”所介绍的细微区别。

除了这些细微区别，我们还可以将PrintAnswerToLife扩展为下面的等价功能实现：

```
Task PrintAnswerToLife()
```

```
{
  var tcs = new TaskCompletionSource<object>();
  var awaiter = Task.Delay (5000).GetAwaiter();
  awaiter.OnCompleted (() =>
  {
    try
    {
      awaiter.GetResult(); // 重新抛出异常
      int answer = 21 * 2;
      Console.WriteLine (answer);
      tcs.SetResult (null);
    }
    catch (Exception ex) { tcs.SetException (ex); }
  });
  return tcs.Task;
}
```

因此，当一个返回任务的异步方法结束时，执行过程会返回等待它的程序（通过一个延续）。

提示： 在富客户端场景中，执行过程会在这一点跳回UI线程（如果之前未在UI线程上）；否则，它会继续在延续所在的线程上运行。这意味着，跳出异步调用图不会发生任何延迟开销，这与通过UI线程创建的第一个“回弹”不同。

---

1. 返回Task<TResult>

如果方法体返回TResult，则可以返回一个Task<TResult>：

```
async Task<int> GetAnswerToLife()
{
  await Task.Delay (5000);
  int answer = 21 * 2;
  return answer; // 方法的返回类型是Task<int>，所以返回int
}
```

在内部，这段代码会向TaskCompletionSource发送一个值，而非null。在PrintAnswerToLife上调用GetAnswerToLife（实际上源自Go调用），就可以演示它的用法：

```
async Task Go()
{
  await PrintAnswerToLife();
  Console.WriteLine ("Done");
}

async Task PrintAnswerToLife()
{
  int answer = await GetAnswerToLife();
  Console.WriteLine (answer);
}

async Task<int> GetAnswerToLife()
{
  await Task.Delay (5000);
  int answer = 21 * 2;
  return answer;
```

```
}
```

实际上，我们将原始的PrintAnswerToLife重构为两个方法——如果采用同步编程，那么也一样简单。它的内部实现与同步编程类似；这里是调用图的同步等价实现，其中在阻塞5秒钟之后，调用Go()可以返回相同的结果：

```
void Go()
{
  PrintAnswerToLife();
  Console.WriteLine ("Done");
}

void PrintAnswerToLife()
{
  int answer = GetAnswerToLife();
  Console.WriteLine (answer);
}

int GetAnswerToLife()
{
  Thread.Sleep (5000);
  int answer = 21 * 2;
  return answer;
}
```

提示：这也说明了使用C#异步函数进行程序设计的基本原则：

（1）以同步方式编写方法。

（2）使用异步方法调用替换同步方法，然后等待它们。

（3）除了“最顶级的”方法（一般是UI控件的事件处理器），将异步方法的返回类型修改为Task或Task<TResult>，使它们变成可等待的方法。

编译能够为异步函数创建任务，意味着在很大程度上，我们只需要在创建I/O密集并发性的底层方法中显式创建一个TaskCompletionSource实例。（而对于创建计算密集并发性的方法，则可以使用Task.Run创建任务。）

2. 异步调用图的执行

为了确切理解它的执行过程，最好将代码重新排列为：

```
async Task Go()
{
  var task = PrintAnswerToLife();
  await task; Console.WriteLine ("Done");
}

async Task PrintAnswerToLife()
{
  var task = GetAnswerToLife();
  int answer = await task; Console.WriteLine (answer);
}

async Task<int> GetAnswerToLife()
```

```
{
  var task = Task.Delay (5000);
  await task; int answer = 21 * 2; return answer;
}
```

Go调用PrintAnswerToLife，PrintAnswerToLife又调用GetAnswerToLife，GetAnswerToLife又调用Delay，然后再等待。await会使执行过程返回它所等待的PrintAnswerToLife，然后再返回Go，它同样会等待并返回调用者。所有这些方法调用都在调用Go的线程上以同步方式执行；这就是执行过程的主要同步阶段。

5秒钟之后，Delay上的延续就会触发，然后执行过程会返回一个池化线程的GetAnswerToLife。（如果在一个UI线程上启动，那么执行过程现在会返回该线程。）然后，GetAnswerToLife的其他语句就会执行，然后方法的Task<int>会完成执行并生成结果42，执行PrintAnswerToLife的延续，然后它再执行方法的其余语句。这个过程会继续执行，直至Go的任务执行完成。

整个执行流程与前面介绍的同步调用图完全匹配，因为我们采用了一种模式，其中在每一个异步调用之后都会等待。这样就可以在调用图中形成一个无并发或重叠执行的串行流。每一个await表达式都会在执行中创建一个“缺口”，之后程序都可以在原处恢复执行。

3. 并行性

调用一个异步方法，但是等待它，就可以使代码并行执行。在前面的例子中，有一个按钮添加了一个像下面这样的事件处理器Go：

```
_button.Click += (sender, args) => Go();
```

尽管Go是一个异步方法，但是我们并没有等待它，事实上它正是利用并发性来实现快速响应的UI：

我们可以使用相同的法则以并行方式执行两个异步操作：

```
var task1 = PrintAnswerToLife();
var task2 = PrintAnswerToLife();
await task1; await task2;
```

（通过在后面等待这两个操作，我们就可以中止此处的并行性。在后面，我们将介绍如何使用WhenAll任务连接符优化这个模式。）

以这种方式创建的并发性可以支持UI线程或非UI线程上执行的操作，但是它们的实现方式有所区别。这两种情况都可以在底层操作上（如Task.Delay或Task.Run生成的代码）实现真正的并发性。在调用堆中，只有操作不通过同步上下文创建，在这之上的方法才可能实现真正的并发性；否则，它们就是前面介绍的伪并发性和简化的线程安全性，其中我们唯一能够优先使用的是await语句。例如，它允许我们定义一个共享域_x，然后不需要使用锁就可以在增加它的值：

```
async Task<int> GetAnswerToLife()
{
  _x++;
  await Task.Delay (5000);
  return 21 * 2;
}
```

（但是，这里不能假定_x在await前后均保持相同的值。）

14.5.3 异步lambda表达式

就像普通的命名（*named*）方法可以采用异步方式执行一样：

```
async Task NamedMethod()
{
  await Task.Delay (1000);
  Console.WriteLine ("Foo");
}
```

只要添加async关键字，未命名（*unnamed*）方法（lambda表达式和匿名方法）也一样可以采用异步方式执行：

```
Func<Task> unnamed = async () =>
{
  await Task.Delay (1000);
  Console.WriteLine ("Foo");
};
```

它们可以采用相同的方式进行调用和等待：

```
await NamedMethod();
await unnamed();
```

异步lambda表达式可用于附加事件处理器：

```
myButton.Click += async (sender, args) =>
{
  await Task.Delay (1000);
  myButton.Content = "Done";
};
```

下面的代码更为简洁，但是可以实现相同的效果：

```
myButton.Click += ButtonHandler;
...
async void ButtonHander (object sender, EventArgs args)
{
  await Task.Delay (1000);
  myButton.Content = "Done";
};
```

异步lambda表达式也可以返回Task<TResult>：

```
Func<Task<int>> unnamed = async () =>
{
  await Task.Delay (1000);
  return 123;
};
int answer = await unnamed();
```

14.5.4 WinRT的异步方法

在WinRT中，与Task等价的是IAsyncAction，而与Task<TResult>等价的是IAsyncOperation<TResult>（位于Windows.Foundation命名空间）。

这两个类都可以通过System.Runtime.WindowsRuntime.dll程序集的AsTask扩展方法转换为Task或Task<TResult>。这个程序集也定义了一个GetAwaiter方法，它可以操作IAsyncAction和IAsyncOperation<TResult>类型，它们可以直接执行等待操作。例如：

```
Task<StorageFile> fileTask = KnownFolders.DocumentsLibrary.CreateFileAsync
                             ("test.txt").AsTask();
```

或者：

```
StorageFile file = await KnownFolders.DocumentsLibrary.CreateFileAsync
                   ("test.txt");
```

提示：由于COM类型系统的限制，IAsyncOperation<TResult>并不是基于IAsyncAction，它们继承一个通用基本类型IAsyncInfo。

AsTask方法也有重载方法，可以接受一个取消令牌（参见第14.6.1节“取消”）和一个对象IProgress<T>（参见第14.6.2节“进度报告”）。

14.5.5 异步与同步上下文

前面已经介绍了同步上下文对于提交延续的重要性。但是在另外一些更复杂的情况中，这种同步上下文也能够在无返回值的异步函数中发挥作用。这并不属于C#编译器扩展的直接结果，而属于System.CompilerServices命名空间中Async*MethodBuilder类型的功能，编译器会使用它们扩展异步函数。

1．异常提交

在富客户端应用程序中，常用方法是利用集中异常处理事件（WPF的Application.DispatcherUnhandledException）来处理UI线程抛出的未处理异常。而在ASP.NET应用程序中，*global.asax*的Application_Error可以完成类似的工作。在内部，它们采用的方式是在自己的语句块中（或者在ASP.NET的页面处理方法管道中）调用UI事件。

上层异步函数会更复杂。假设使用下面的事件处理器处理按钮单击事件：

```
async void ButtonClick (object sender, RoutedEventArgs args)
{
  await Task.Delay(1000);
  throw new Exception ("Will this be ignored?");
}
```

当单击按钮时，事件处理器就会执行，正常情况下它的执行过程会返回到await语句之后的消息循环体，但是消息循环体中的catch语句块无法捕捉到1秒钟之后抛出的异常。

为了解决这个问题，AsyncVoidMethodBuilder会捕捉未处理异常（在无返回值的异步函数中），然后将它们提交到同步上下文中（如果有），以保证触发全局异常处理事件。

提示：编译器只能将这个逻辑应用到无返回值的异步函数中。所以，如果我们将ButtonClick的返回值从void改为Task，那么未处理的异常可能会破坏最终产生的Task，而现在它是无法处理的（结果产生了一个未监控异常）。

注意，在await之前或之后抛出异常并没有任何区别。所以，在下面的例子中，异常会提交到同步上下文中（如果有），但是绝不会提交给调用者：

```
async void Foo() { throw null; await Task.Delay(1000); }
```

如果没有同步上下文，那么异常就成为未监视异常。没有将异常抛出到调用者，这种行为似乎有一些奇怪，但是这与迭代器的实现方式有一定的相似性：

```
IEnumerable<int> Foo() { throw null; yield return 123; }
```

在这个例子中，异常不会直接抛出到调用者，直到枚举序列结束之后才会抛出异常。

2. OperationStarted和OperationCompleted

如果有同步上下文，那么无返回值的异步函数也会在进入这个函数时调用它的OperationStarted方法，然后在函数结束时调用它的OperationCompleted方法。ASP.NET的同步上下文会使用这些方法保证页面处理管道的顺序执行过程。

如果要编写一个自定义同步上下文，对这些无返回值的异步方法执行单元测试，那么可能重写这些方法。Microsoft的并行编程博客有这方面的介绍：http://blogs.msdn.com/b/pfxteam。

14.5.6 优化

1. 同步完成

异步方法可能会在等待之前返回。假设有下面这样一个方法，它会缓存下载的网页：

```
static Dictionary<string,string> _cache = new Dictionary<string,string>();
async Task<string> GetWebPageAsync (string uri)
{
  string html;
  if (_cache.TryGetValue (uri, out html)) return html;
  return _cache [uri] =
  await new WebClient().DownloadStringTaskAsync (uri);
}
```

假设某个URI已经存在于缓存之中，那么执行过程会在等待发生之前返回调用者，同时这个方法会返回一个已发送信号的任务，这称为同步完成。

如果等待一个同步完成任务，那么执行过程不会返回调用者并通过一个延续弹回——相反，它会马上进入下一条语句。编译器会通过检查等待者的IsCompleted属性来实现这种优化；换而言之，无论何时执行等待：

```
Console.WriteLine (await GetWebPageAsync ("http://oreilly.com"));
```

在同步完成时，编译器会生成中止延续的代码：

```
var awaiter = GetWebPageAsync().GetAwaiter();
if (awaiter.IsCompleted)
  Console.WriteLine (awaiter.GetResult());
else
  awaiter.OnCompleted (() => Console.WriteLine (awaiter.GetResult());
```

提示： 等待一个同步返回的异步函数仍然存在一些过载——在2012年的PC机上可能是50～100纳秒。

相反，弹回线程池也会带来上下文切换开销——大约是1～2毫秒，而弹回UI消息循环的开销至少是10倍（当UI线程繁忙时会更长）。

编写从不等待的异步方法也是允许的，但是编译器会发出一条警告信息：

```
async Task<string> Foo() { return "abc"; }
```

在重写虚方法/抽象方法时，如果不需要实现异步处理，那么很适合使用这种方法。（例如，MemoryStream的读/写方法——参见第15章）。实现相同结果的另一种方法是使用Task.FromResult，它会返回一个已发送信号的任务：

```
Task<string> Foo() { return Task.FromResult ("abc"); }
```

如果从UI线程调用，那么GetWebPageAsync方法本身就具有线程安全性，即可以在成功执行之后多次调用这个方法（从而初始化多个并发下载），而且不需要使用锁来保证缓存。但是，如果多次处理同一个URI，则会生成多个冗余下载，它们最终都会更新同一个缓存记录（最后一个下载会覆盖前面的下载）。如果没有错误，那么更高效的方式是让同一个URI的后续调用（异步）等待正在处理的请求。

还有一个简单方法可以实现这种效果——不需要使用锁或信号结构。这里创建一个“未来”缓存（Task<string>），代替字符串缓存：

```
static Dictionary<string,Task<string>> _cache =
  new Dictionary<string,Task<string>>();

Task<string> GetWebPageAsync (string uri)
{
  Task<string> downloadTask;
  if (_cache.TryGetValue (uri, out downloadTask)) return downloadTask;
  return _cache [uri] = new WebClient().DownloadStringTaskAsync (uri);
}
```

注意，这里并没有在方法上添加关键字，因为我们直接返回通过从WebClient方法获得的任务。

如果重复调用GetWebPageAsync处理同一个URI，就可以保证能够获得同一个Task<string>对象。（这样做有另一个好处：降低GC负载。）如果任务完成，那么由于有前面介绍的编译器优化，所以等待它的开销是很低的。

我们可以进一步扩展这个例子，使用锁处理整个方法体，就可以使它在不需要同步上下文的保证下仍然具有线程安全性：

```
lock (_cache)
{
  Task<string> downloadTask;
  if (_cache.TryGetValue (uri, out downloadTask)) return downloadTask;
  return _cache [uri] = new WebClient().DownloadStringTaskAsync (uri);
}
```

这种方法之所以有效，是因为我们并没有在网页下载过程中添加锁（这样会破坏并发性）；我们只在检查缓存的一小段时间中添加了锁，在需要时还可以启动一个新任务，然后使用这个任务更新缓存。

2. 避免过度回弹

对于在循环中多次调用的方法，通过调用ConfigureAwait，可以避免重复回弹UI消息循环带来的开销。这样会阻止任务将延续回弹至同步上下文，从而使过载降低到接近上下文切换的开销（或者远远小于等待同步完成的方法开销）：

```
async void A() { ... await B(); ... }

async Task B()
{
  for (int i = 0; i < 1000; i++)
  await C().ConfigureAwait (false);
}

async Task C() { ... }
```

这意味着，我们在B方法和C方法中撤销了UI应用所使用的简单线程安全模型，其中代码运行在UI线程上，而且只能在await语句中优先占用。然而，A方法不受影响，它在启动之后就一直停留在UI线程上。

这种优化特别适用于编写程序库：这时不需要简化线程安全性的好处，因为这些代码不会与调用者共享状态——它不会访问UI控件。（在我们的例子中，如果已知操作很可能短时间完成，那么这种方法也适用于同步完成的C方法上。）

14.6 异步模式

14.6.1 取消

通常，一定要能够在并发操作启动之后取消这个操作（可能是出于用户请求）。实现这个操作的一个简单方法是使用取消令牌，编写一个像下面这样的封装类：

```
class CancellationToken
{
  public bool IsCancellationRequested { get; private set; }
  public void Cancel() { IsCancellationRequested = true; }
  public void ThrowIfCancellationRequested()
  {
    if (IsCancellationRequested)
      throw new OperationCanceledException();
  }
}
```

然后，再按照下面的方式编写一个可取消的异步方法：

```
async Task Foo (CancellationToken cancellationToken)
{
  for (int i = 0; i < 10; i++)
  {
    Console.WriteLine (i);
    await Task.Delay (1000);
    cancellationToken.ThrowIfCancellationRequested();
  }
}
```

当调用者想要取消操作时，它会调用传递给Foo的取消令牌上的Cancel。这样会将IsCancellationRequested设置为true，因此在出现OperationCanceledException异常（System命名空间中用于实现这个设计的一个预定义异常）时，Foo很快就会出错。

如果不考虑线程安全性（可以为读/写IsCancellationRequested添加锁），那么这种模式是很高效的，而且CLR还提供了一个实现类似功能的类型CancellationToken。然而，它没有Cancel方法；但是这个方法提供了另一个类型CancellationTokenSource。这种分离也具有一定的安全性：只能访问CancellationToken对象的方法可以检查取消操作，但是不能初始化取消操作。

为了获得一个取消令牌，我们首先需要初始化一个CancellationTokenSource：

```
var cancelSource = new CancellationTokenSource();
```

它有一个Token属性，可以返回一个CancellationToken。因此，要按照以下方式调用Foo方法：

```
var cancelSource = new CancellationTokenSource();
Task foo = Foo (cancelSource.Token);
...
... (some time later)
cancelSource.Cancel();
```

在CLR中，大多数异步方法都提供了取消令牌，其中包括Delay。如果修改Foo，让它将令牌传递到Delay方法，那么请求到达后任务会马上停止（而不会等到1秒钟之后）：

```
async Task Foo (CancellationToken cancellationToken)
{
  for (int i = 0; i < 10; i++)
  {
    Console.WriteLine (i);
    await Task.Delay (1000, cancellationToken);
  }
}
```

注意，我们不再需要调用ThrowIfCancellationRequested，因为Task.Delay已经包含这个操作。取消令牌会顺利向下传递到调用栈底部（就像取得请求通过异常向上传递到调用栈顶部一样）。

提示： WinRT的异步方法采用一种下级协议处理取消操作，其中它会接收一个CancellationToken，调用IAsyncInfo类型的Cancel方法。然而，AsTask扩展方法会重载并接受一个取消令牌，以解决这个问题。

同步方法也可以支持取消操作（如Task的Wait方法）。在这些情况中，取消指令必须以异步方式执行（例如，在另一个任务中执行）。例如：

```
var cancelSource = new CancellationTokenSource();
Task.Delay (5000).ContinueWith (ant => cancelSource.Cancel());
...
```

事实上，从Framework 4.5开始，在创建CancellationTokenSource时，可以指定一个时间间隔，表示在一定时间段之后初始化取消操作（正如我们所演示的）。无论是同步或是异步，最好指定一个超时时间：

```
var cancelSource = new CancellationTokenSource (5000);
try { await Foo (cancelSource.Token); }
```

```
catch (OperationCanceledException ex) { Console.WriteLine ("Cancelled"); }
```

CancellationToken结构提供了一个Register方法，它可用于注册一个回调代理，然后在取消操作发生时触发：它会返回一个对象，用于撤销注册。

在出现未处理的OperationCanceledException异常（IsCanceled返回true，而IsFaulted返回false）时，编译器的异步函数生成的任务会自动进入“已取消”状态。使用Task.Run创建的任务也具有相同特点，其中在创建时需要在构造函数中传入（同一个）CancellationToken。在异步场景中，出错任务和取消任务之间的区别并不明显，在等待时这两种情况都会抛出一个OperationCanceledException异常；但是在高级并行编程场景中，它们的区别较大（特别是在一些条件延续中）。第23.4.3节“取消任务”将介绍这方面的内容。

14.6.2 进度报告

有时候，异步操作需要在运行时报告进度情况。有一种简单的解决方法是给异步方法传入一个Action代理，然后进度发生变化时就会触发这个方法：

```
async Task Foo (Action<int> onProgressPercentChanged)
{
  return Task.Run (() =>
  {
    for (int i = 0; i < 1000; i++)
    {
    if (i % 10 == 0) onProgressPercentChanged (i / 10);
    // 执行计算密集型操作...
    }
  });
}
```

下面是调用它的方式：

```
Action<int> progress = i => Console.WriteLine (i + " %");
await Foo (progress);
```

这段代码运行在控制台应用程序上，但是它不适合运行在富客户端场景，因为它可以从工作者线程报告进度，这可能会给使用者线程带来线程安全问题。（实际上，并发操作向外部“泄露”信息的副作用是允许的，但是在UI线程上调用的方法则不行。）

IProgress<T>和Progress<T>

CLR提供了一对用于解决这个问题的类型：IProgress<T>接口和实现这个接口的类Progress<T>。实际上，它们的作用是“包装”一个代理，这样UI应用程序就可以通过同步上下文安全地报告进度。

这个接口只定义了一个方法：

```
public interface IProgress<in T>
{
  void Report (T value);
}
```

IProgress<T>的用法很简单：我们的方法几乎不需要修改：

```
Task Foo (IProgress<int> onProgressPercentChanged)
```

```
{
  return Task.Run (() =>
  {
    for (int i = 0; i < 1000; i++)
    {
      if (i % 10 == 0) onProgressPercentChanged.Report (i / 10);
      // 执行一些计算密集型操作...
    }
  });
}
```

Progress<T>类有一个构造方法，它可以接受Action<T>类型包装的代理，

```
var progress = new Progress<int> (i => Console.WriteLine (i + " %"));
await Foo (progress);
```

（Progress<T>还有一个ProgressChanged事件，我们可以订阅这个事件，而不要（或者同时）给构造函数传入一个操作代理。）在实例化Progress<int>时，这个类会捕捉同步上下文（如果有）。然后当Foo调用Report时，它会通过上下文调用代理对象。

将替换为包含一系列属性的自定义类型，就可以在异步方法中实现更复杂的进度报告。

提示： 如果熟悉Reactive框架，那么开发者会发现，IProgress<T>和异步函数返回的任务可以共同实现与IObserver<T>类似的功能。它们的区别在于，除了IProgress<T>生成的值及其类型不同，这个任务还会产生一个最终返回值。

由IProgress<T>生成的值一般是"废弃值"（例如，完成比或已下载字节），而由IObserver<T>的MoveNext生成的值通常由结果组成，这正是调用它的初衷。

WinRT的异步方法也提供了进度报告功能，但是它采用的协议受到COM（相对）较羸弱的类型系统的影响。报告进度的异步WinRT方法不接受一个IProgress<T>对象，而是返回以下接口之一，用于代替IAsyncAction和IAsyncOperation<TResult>：

```
IAsyncActionWithProgress<TProgress>
IAsyncOperationWithProgress<TResult, TProgress>
```

有趣的是，这两个方法都基于IAsyncInfo（而非IAsyncAction和IAsyncOperation<TResult>）。

好消息是AsTask扩展方法也有重载方法，可以接受以IProgress<T>代替上面的接口，所以.NET开发者可以忽略COM接口，转而使用下面的方法：

```
var progress = new Progress<int> (i => Console.WriteLine (i + " %"));
CancellationToken cancelToken = ...
var task = someWinRTobject.FooAsync().AsTask (cancelToken, progress);
```

14.6.3 基于任务的异步模式（TAP）

Framework 4.5提供了大量返回任务的异步方法，它们都可用于代替await（主要与I/O相关）。很多方法（至少有一部分）采用了一种基于任务的异步模式（Task-based Asynchronous Pattern, TAP），这是到目前为止最合理的形式。一个TAP方法必须：

- 返回一个"热"（正在运行的）Task或Task<TResult>

- 拥有“Async”后缀（除了一些特殊情况，如任务组合器）
- 如果支持取消和/或进度报告，重载后可接受取消令牌和/或IProgress<T>
- 快速返回调用者（具有一小段初始同步阶段）
- 在I/O密集操作中不占用线程

正如前面所介绍，TAP方法很容易通过C#异步函数实现。

14.6.4 任务组合器

使用统一协议调用异步函数（它们都一致返回任务）的一个优点是，可以使用和编写任务组合器（task combinator）——一些适用于组合各种用途的任务的函数。

CLR包含两个任务组合器：Task.WhenAny和Task.WhenAll。在下面的介绍过程中，我们假设定义了以下方法：

```
async Task<int> Delay1() { await Task.Delay (1000); return 1; }
async Task<int> Delay2() { await Task.Delay (2000); return 2; }
async Task<int> Delay3() { await Task.Delay (3000); return 3; }
```

1. WhenAny

Task.WhenAny返回这样一个任务：当任务组中任意一个任务完成，它也就完成。下面的任务会在1秒钟内完成：

```
Task<int> winningTask = await Task.WhenAny (Delay1(), Delay2(), Delay3());
Console.WriteLine ("Done");
Console.WriteLine (winningTask.Result); // 1
```

因为Task.WhenAny本身会返回一个任务，所以我们要等待它，然后它会返回先完成的任务。这个例子完全不会阻塞——包括访问Result属性的最后一行语句（因为winningTask已经完成）。但是，最好还是要等待任务（winningTask）：

```
Console.WriteLine (await winningTask); // 1
```

因为这时任何异常都会重新抛出，而不需要包装在一个AggregateException异常中。事实上，我们可以在一步操作中同时执行两个await：

```
int answer = await await Task.WhenAny (Delay1(), Delay2(), Delay3());
```

如果后面没有一个未完成任务出现错误，那么除非后面等待了这个任务（或者查询了它的Exception属性），否则该异常将不会被捕捉到。

WhenAny适合用于应用操作超时时间或取消操作（原本这些操作并不支持超时或取消）：

```
Task<string> task = SomeAsyncFunc();
Task winner = await (Task.WhenAny (someOperation, Task.Delay(5000)));
if (winner != task) throw new TimeoutException();
string result = await task; // 解开结果/重新抛出异常
```

注意，因为这个例子使用不同类型的任务去调用WhenAny，所以完成的任务会报告为一个普通Task（而非Task<string>）。

2. WhenAll

Task.WhenAll返回这样一个任务：当传入的所有任务都完成时，它才完成。下面的任务会在3秒钟之后完成（同时演示了分叉/联合模式）：

```
await Task.WhenAll (Delay1(), Delay2(), Delay3());
```

不使用WhenAll，而依次等待task1、task2和task3，也可以得到相似的结果：

```
Task task1 = Delay1(), task2 = Delay2(), task3 = Delay3();
await task1; await task2; await task3;
```

除了三次等待的效率低于一次等待之外，这两种方式的区别是，如果task1出错，那么我们就无法等待task2/task3，而且它们中间发生的所有异常都无法处理。事实上，这正是CLR 4.5不约束未处理任务异常行为的原因所在：尽管上面整个代码块有一个异常处理块，但是如果task2或task3的异常会在垃圾回收之后的某个时间使应用程序崩溃，那么这是很不合理的。

相反，Task.WhenAll只有在所有任务完成之后才会完成——即使中间出现了错误。而如果出现了多个错误，那么它们的异常会组合到任务的AggregateException之中（这是AggregateException真正发挥作用的时候——即开发者会关心所有的异常）。然而，等待组合的任务只能捕捉到第一个异常，所以如果要查看所有异常，则必须这样做：

```
Task task1 = Task.Run (() => { throw null; } );
Task task2 = Task.Run (() => { throw null; } );
Task all = Task.WhenAll (task1, task2);
try { await all; }
catch
{
  Console.WriteLine (all.Exception.InnerExceptions.Count); // 2
}
```

使用类型为Task<TResult>的任务调用WhenAll，会返回一个Task<TResult[]>，这是所有任务的结果组合。如果执行等待操作时，那么这个结果会变成TResult[]：

```
Task<int> task1 = Task.Run (() => 1);
Task<int> task2 = Task.Run (() => 2);
int[] results = await Task.WhenAll (task1, task2); // { 1, 2 }
```

下面是一个实际例子：并行下载多个URI，然后计算它们的总下载大小：

```
async Task<int> GetTotalSize (string[] uris)
{
  IEnumerable<Task<byte[]>> downloadTasks = uris.Select (uri =>
    new WebClient().DownloadDataTaskAsync (uri));

  byte[][] contents = await Task.WhenAll (downloadTasks);
  return contents.Sum (c => c.Length);
}
```

但是，这段代码的效率有一些不足，即我们只能在每一个任务都完成之后才能处理字节数组。如果在下载之后马上将字节数组压缩为实际长度，那么效率会有所提高。这正是异步lambda发挥作用的地方，因为我们需要在LINQ的Select查询操作符中插入一个await表达式：

```
async Task<int> GetTotalSize (string[] uris)
```

```
{
  IEnumerable<Task<int>> downloadTasks = uris.Select (async uri =>
  (await new WebClient().DownloadDataTaskAsync (uri)).Length);
  int[] contentLengths = await Task.WhenAll (downloadTasks);
  return contentLengths.Sum();
}
```

3. 自定义组合器

编写自定义的任务组合器是很实用的。最简单的组合器可以接受一个任务（如下所示），它允许在特定超时时间里等待任意任务：

```
async static Task<TResult> WithTimeout<TResult> (this Task<TResult> task,
                                                    TimeSpan timeout)
{
  Task winner = await (Task.WhenAny (task, Task.Delay (timeout)));
  if (winner != task) throw new TimeoutException();
  return await task; // 解开结果/重新抛出异常
}
```

下面的代码通过一个CancellationToken“抛弃”一个任务：

```
static Task<TResult> WithCancellation<TResult> (this Task<TResult> task,
                                  CancellationToken cancelToken)
{
  var tcs = new TaskCompletionSource<TResult>();
  var reg = cancelToken.Register (() => tcs.TrySetCanceled ());
  task.ContinueWith (ant =>
  {
    reg.Dispose();
    if (ant.IsCanceled)
      tcs.TrySetCanceled();
    else if (ant.IsFaulted)
      tcs.TrySetException (ant.Exception.InnerException);
    else
      tcs.TrySetResult (ant.Result);
  });
  return tcs.Task;
}
```

任务组合器有时候可能很复杂，需要使用第22章介绍的各种信号结构。但是这实际上是好事，因为它可以将与并发相关的复杂性从业务逻辑代码分离，然后封装在允许单独测试的可重用方法中。

下面的组合器作用与WhenAll类似，唯一不同的是如果任意任务出现错误，那么最终任务也会马上出错：

```
async Task<TResult[]> WhenAllOrError<TResult>
  (params Task<TResult>[] tasks)
{
  var killJoy = new TaskCompletionSource<TResult[]>();
  foreach (var task in tasks)
    task.ContinueWith (ant =>
    {
    if (ant.IsCanceled)
      killJoy.TrySetCanceled();
    else if (ant.IsFaulted)
```

并发与异步

```
        killJoy.TrySetException (ant.Exception.InnerException);
      });
    return await await Task.WhenAny (killJoy.Task, Task.WhenAll (tasks));
  }
```

这里先创建了一个TaskCompletionSource，它的唯一作用是终止出错的任务。因此，这里不会调用它的SetResult方法，只会调用它的TrySetCanceled和TrySetException方法。这个例子更适合使用ContinueWith，而不是GetAwaiter().OnCompleted，因为我们不需要访问任务的结果，也不需要在此时弹回UI线程。

14.7 旧模式

Framework在实现异步时采用了其他一些模式，它们位于任务和异步函数之前。这些模式现在已经很少使用，因为基于任务的异步处理已经成为Framework 4.5的主要模式。

14.7.1 异步编程模型（APM）

最老的模式是APM（Asynchronous Programming Model），它使用一对以“Begin”和“End”开头的方法，以及一个接口IAsyncResult。为了演示它的用法，我们将使用System.IO的Stream类，然后调用它的Read方法。首先，下面它的异步版本是：

```
public int Read (byte[] buffer, int offset, int size);
```

可以想象，基于任务的异步版本是：

```
public Task<int> ReadAsync (byte[] buffer, int offset, int size);
```

下面就是APM版本：

```
public IAsyncResult BeginRead (byte[] buffer, int offset, int size,
                               AsyncCallback callback, object state);
public int EndRead (IAsyncResult asyncResult);
```

调用Begin*方法，初始化操作，返回一个IAsyncResult对象，它作为异步操作的令牌。当操作完成或者出错时，就会触发AsyncCallback代理：

```
public delegate void AsyncCallback (IAsyncResult ar);
```

无论谁处理这个代理，它都会调用End*方法，负责提供操作的返回值，以及在操作出错时重新抛出异常。

APM不仅使用不方便，而且也很难正确实现。处理APM方法的最简单方法是调用Task.Factory.FromAsync适配器方法，它会将一个APM方法对转换为一个Task。在内部，它使用一个TaskCompletionSource创建任务，它会在APM操作完成或出错时发送出去。

FromAsync方法需要接收以下参数：

- 指定一个Begin*XXX*方法的代理
- 指定一个End*XXX*方法的代理
- 传递给这些方法的额外参数

FromAsync经过重载，可以接收几乎能够匹配.NET框架中所有异步方法签名的代理类型和参数。例如，假设流是Stream，缓冲区是byte[]，那么可以这样做：

```
Task<int> readChunk = Task<int>.Factory.FromAsync (
  stream.BeginRead, stream.EndRead, buffer, 0, 1000, null);
```

异步代理

CLR仍然支持异步代理，这个特性可以使用APM风格的BeginInvoke/EndInvoke方法以异步方式调用任何代理：

```
Func<string> foo = () => { Thread.Sleep(1000); return "foo"; };
foo.BeginInvoke (asyncResult =>
  Console.WriteLine (foo.EndInvoke (asyncResult)), null);
```

异步代理有较大的过载，而且有非常麻烦的冗余任务需要处理：

```
Func<string> foo = () => { Thread.Sleep(1000); return "foo"; };
Task.Run (foo).ContinueWith (ant => Console.WriteLine (ant.Result));
```

14.7.2 基于事件的异步模式（EAP）

基于事件的异步模式（Event-based Asynchronous Pattern, EAP）在Framework 2.0时引入，它是代替APM的更简单方法，特别是在UI场景中。然而，它只能通过有限的类型实现，其中最主要的类是System.Net的WebClient。EAP只是一个模式；它并没有任何辅助类型。本质上，这个模式是：一个类提供了一组内部管理并发性的成员。具体与下面的代码类似：

```
// WebClient类的成员：

public byte[] DownloadData (Uri address);    // 异步版本
public void DownloadDataAsync (Uri address);
public void DownloadDataAsync (Uri address, object userToken);
public event DownloadDataCompletedEventHandler DownloadDataCompleted;

public void CancelAsync (object userState);  // 取消操作
public bool IsBusy { get; }                  // 表示仍然在运行
```

*Async方法可以以异步方式初始化一个操作。当操作完成时，*Completed事件就会触发（自动提交给捕捉的同步上下文）。这个事件会传回一个事件参数对象，其中包含：

- 一个表示操作是否取消的标记（通过使用者调用CancelAsync）
- 一个表示异常（如果有）抛出的Error对象
- 在调用Async方法时可能提供的userToken对象

EAP类型还可能触发一个进度报告事件，它会在进度发生变化的任意时刻触发（它也通过同步上下文提交）：

```
public event DownloadProgressChangedEventHandler DownloadProgressChanged;
```

实现EAP需要编写大量的模板代码，因此这个模式的代码相当复杂。

14.7.3 BackgroundWorker

位于System.ComponentModel的BackgroundWorker是EAP的通用实现。它允许富客户端应用启动一个工作者线程，然后执行完成和报告百分比进度，而不需要显式捕捉同步上下文。例如：

```
var worker = new BackgroundWorker { WorkerSupportsCancellation = true };
worker.DoWork += (sender, args) =>
{                                              // 这个方法运行在工作者线程上
  if (args.Cancel) return;
  Thread.Sleep(1000);
  args.Result = 123;
};
worker.RunWorkerCompleted += (sender, args) =>
{                                              // 运行在UI线程上
  // 这里可以安全地更新UI控件...
  if (args.Cancelled)
    Console.WriteLine ("Cancelled");
  else if (args.Error != null)
    Console.WriteLine ("Error: " + args.Error.Message);
  else
    Console.WriteLine ("Result is: " + args.Result);
};
worker.RunWorkerAsync(); // 捕捉同步上下文并启动操作
```

RunWorkerAsync启动操作，然后触发一个池化工作者线程的DoWork事件。它还会捕捉同步上下文，而且当操作完成或出错时，RunWorkerCompleted事件就会通过同步上下文触发（像延续一样）。

BackgroundWorker可以创建过程级并发性，其中DoWork事件完全运行在工作者线程上。如果需要在该事件处理器上更新UI控件（而非提交完成百分比进度），则必须使用Dispatcher.BeginInvoke或类似的方法。

关于BackgroundWorker更详细的介绍，请参见www.albahari.com/threading。

第15章 流与I/O

本文介绍.NET的输入和输出基础类型，主要包括以下几方面内容：

- .NET流体系结构及其支持各种I/O类型的统一读写编程接口
- 处理磁盘文件和目录
- 隔离存储区及其分离程序与用户数据的作用

本章主要介绍System.IO命名空间中的类型，即底层I/O功能的基础。.NET Framework也支持一些更高级的I/O功能，形式包括SQL连接和命令、LINQ to SQL和LINQ to XML、WCF、Web Services和Remoting。

15.1 流体系结构

.NET流体系结构主要包括以下概念：后备存储流、装饰器流和流适配器，如图15-1所示。

后备存储是支持输入和输出的终端，例如文件或网络连接。准确地说，它可以是下面的一种或两种：

- 支持顺序读取字节的源。
- 支持顺序写入字节的目标。

但是，除非对程序员公开，否则后备存储是无用的。`Stream`正是实现这个功能的标准.NET类；它支持标准的读、写和寻址方法。与数组不同，流不是直接将所有数据保存到内存中，而是按序列方式处理数据——一次一个字节或一个可管理大小的块。因此，无论后备存储的大小如何，流都只占用很少的内存。

流分成两类：

后备存储流

　　它们是与特定类型后备存储硬连接的，例如`FileStream`或`NetworkStream`。

装饰器流

　　它们使用另一种流，以某种方式转换数据，例如`DeflateStream`或`CryptoStream`。

装饰器流具有以下体系结构优势：

- 它们能够释放用于实现自我压缩和加密的后备存储流。
- 在装饰后，流不受接口变化的影响。
- 装饰器支持实时连接。
- 装饰器支持相互连接（例如，压缩器后紧跟一个加密器）。

后备存储流和装饰器流都只支持字节。虽然这种方式既灵活又高效，但是应用程序通常采用更高级的方式，例如文本或XML。通过在一个类中创建专门支持特定格式的类型化方法，并在这个类中封装一个流，适配器弥补了这个缺陷。例如，文本读取器具有一个`ReadLine`方法；而XML编写器则有一个`WriteAttributes`方法。

> **提示：** 适配器会封装一个流，这与装饰器类似。然而，与装饰器不同的是，适配器本身不是一个流；它一般会完全隐藏面向字节的方法。

总之，后备存储流负责处理原始数据；装饰器流支持透明的二进制转换，例如加密；适配器则支持一些处理更高级类型的类型化方法，例如字符串和XML。图15-1说明了它们之间的关系。为了构成一个关系链，我们只需要将一个对象传递给另一个对象的构造函数。

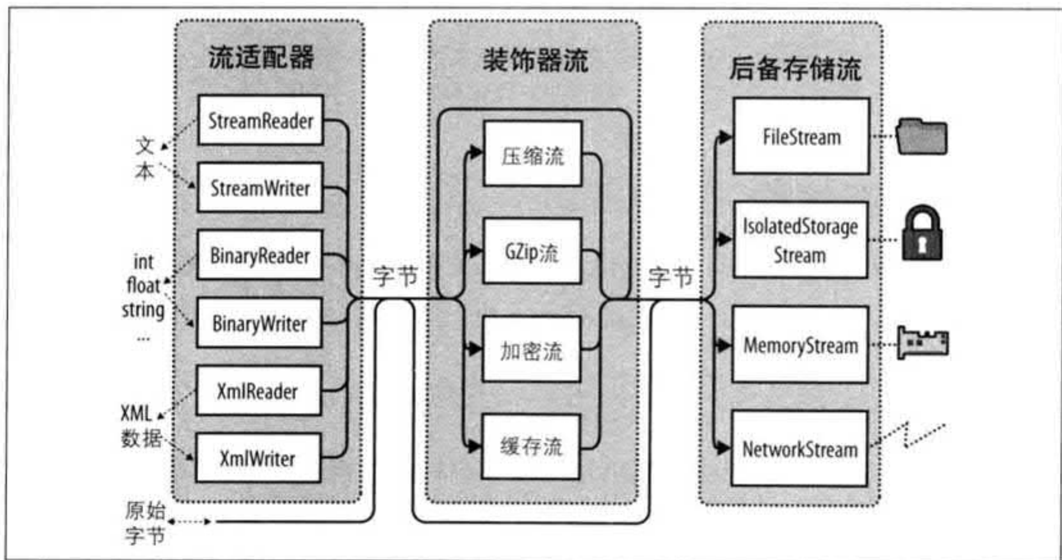


图15-1：流体系结构

15.2 使用流

抽象的`Stream`类是所有流的基类。它定义了三种基础操作的方法和属性：读取、写入和查找；以及一些管理任务，例如关闭、清除和配置超时（见表15-1）。

表15-1：Stream类成员

分类	成员
读取	`public abstract bool CanRead { get; }`
	`public abstract int Read (byte[] buffer, int offset, int count)`
	`public virtual int ReadByte();`
写入	`public abstract bool CanWrite { get; }`
	`public abstract void Write (byte[] buffer, int offset, int count);`
	`public virtual void WriteByte (byte value);`
查找	`public abstract bool CanSeek { get; }`
	`public abstract long Position { get; set; }`
	`public abstract void SetLength (long value);`
	`public abstract long Length { get; }`
	`public abstract long Seek (long offset, SeekOrigin origin);`
关闭/清除	`public virtual void Close();`
	`public void Dispose();`
	`public abstract void Flush();`
超时	`public virtual bool CanTimeout { get; }`
	`public virtual int ReadTimeout { get; set; }`
	`public virtual int WriteTimeout { get; set; }`
其他	`public static readonly Stream Null; // 流为空`
	`public static Stream Synchronized (Stream stream);`

在下面的示例中，我们使用一个文件流来执行读取、写入和查找操作：

```
using System;
using System.IO;

class Program
{
  static void Main()
  {
    // 在当前目录中创建一个文件test.txt：
    using (Stream s = new FileStream ("test.txt", FileMode.Create))
    {
      Console.WriteLine (s.CanRead);                   // True
      Console.WriteLine (s.CanWrite);                  // True
      Console.WriteLine (s.CanSeek);                   // True

      s.WriteByte (101);
      s.WriteByte (102);
      byte[] block = { 1, 2, 3, 4, 5 };
      s.Write (block, 0, block.Length);                // 写入5个字节的块

      Console.WriteLine (s.Length);                    // 7
      Console.WriteLine (s.Position);                  // 7
```

```
      s.Position = 0;                              // 将位置移回到开头

      Console.WriteLine (s.ReadByte());            // 101
      Console.WriteLine (s.ReadByte());            // 102

      // 将数据从流读回数组块：
      Console.WriteLine (s.Read (block, 0, block.Length));    // 5

      // 假设最后一个Read返回5，那么我们将位于文件结尾，所以Read现在返回0：
      Console.WriteLine (s.Read (block, 0, block.Length));    // 0
    }
  }
}
```

要实现异步读或写，只需要调用ReadAsync/WriteAsync，替代Read/Write，然后等待表达式。（我们还必须在调用方法前添加async关键字，具体内容已在第14章介绍过。）

```
async static void AsyncDemo()
{
  using (Stream s = new FileStream ("test.txt", FileMode.Create))
  {
    byte[] block = { 1, 2, 3, 4, 5 };
    await s.WriteAsync (block, 0, block.Length);     // 异步写

    s.Position = 0; // 移回开头
    // 将流字节读回到block数组：
  Console.WriteLine (await s.ReadAsync (block, 0, block.Length));  // 5
  }
}
```

使用异步方法，不需要捆绑线程就可以轻松编写适应慢速流（可能是网络流）的响应式和可扩展应用。

提示： 为了保持简洁，将继续在本章的大多数例子中使用异步方法；然而，建议在大多数涉及网络I/O的场景中使用异步读/写操作。

15.2.1 读取与写入

一个流可能支持只读、只写、读写。如果CanWrite返回false，那么流就是只读的；如果CanRead返回false，那么流就是只写的。

Read可以将流中的一个数据块读取到数组中。它返回接收到的一些字节，字节数一定小于或等于count参数。如果它小于count，那么表示已经到达流的结尾，或者流是以小块方式提供数据的（通常是网络流）。无论是哪一种情况，数组的剩余字节都是不可写的，它们之前的值都是保留的。

警告： 使用Read时，只有当该方法返回0时，才可以肯定已经到达流的末尾。所以，如果流有1,000字节，那么下面的代码无法将它全部读取到内存中：

```
// 假设s是一个流：
byte[] data = new byte [1000];
s.Read (data, 0, data.Length);
```

Read方法可以读取第1～1,000个字节的任意位置，但是不读取流中的其余字节。

下面是读取1,000字节流的正确方法：

```
byte[] data = new byte [1000];
// 除非流的字节数小于1000，否则读取字节直到1000个：

int bytesRead = 0;
int chunkSize = 1;
while (bytesRead < data.Length && chunkSize > 0)
  bytesRead +=chunkSize = s.Read (data, bytesRead, data.Length - bytesRead);
```

提示： BinaryReader类型具有实现相同结果的简单方法：

```
byte[] data = new BinaryReader (s).ReadBytes (1000);
```

如果流的长度小于1,000字节，那么返回的字节数组就会保存实际长度的流数据。如果流支持查找，那么将1000替换成(int)s.Length就能够读取完整的内容。

我们将在本章后面的“流适配器”小节中对BinaryReader类型进行深入介绍。

ReadByte方法简单一些：它每次只读取一个字节，在流结束时返回-1。ReadByte实际上返回的是一个int，而不是byte，因为后者不能为-1。

Write和WriteByte方法都支持将数据发送到流中。当它们无法发送指定的字节时，就会抛出一个异常。

警告： 在Read和Write方法中，参数offset指的是缓冲数组中开始读或写的索引位置，而不是流中的位置。

15.2.2 查找

如果CanSeek返回true，那么表示流是可查找的。在一个可查找的流中（例如文件流），我们可以通过调用SetLength查询或修改它的Length，也可以随时修改正在读写的Position。Position属性是与流的开始位置相关的；然而，Seek方法则支持移动到流的当前位置或结束位置。

提示： 修改FileStream的Position属性一般需要几毫秒时间。如果要在循环中执行几百万次位置修改，那么Framework 4.0中新的MemoryMappedFile类可能比FileStream更适合（详细信息参见本章后面的“内存映射文件”）。

如果流不支持查找（例如加密流），那么确定其长度的唯一方法是遍历整个流。而且，如果需要重新读取之前的位置，必须先关闭这个流，然后再重新从头开始读取。

15.2.3 关闭和清除

流在使用完毕之后必须清理，以释放底层资源，例如文件和套接字句柄。一个保证关闭的简单方法是在块中初始化流。通常，流采用以下标准的清理语法：

- Dispose和Close的功能相同
- 重复清除或关闭流不会产生错误

关闭一个装饰流会同时关闭装饰器及其后备存储流。在装饰器系列中，关闭最外层的装饰器（系列的头部）会关闭整个系列。

有一些流（例如文件流）会将数据缓冲到后备存储并从中取回数据，减少回程，从而提升性能。这意味着写入到流中的数据不会直接存储到后备存储器；而是等到缓冲区填满时再写入。Flush方法可以强制将所有内部缓冲的数据写入。当流关闭时，Flush会自动被调用，所以下面的代码是不必要的：

```
s.Flush(); s.Close();
```

15.2.4 超时

如果CanTimeout返回true，那么流支持读写超时设定。网络流支持超时设定；文件流和内存流则不支持。对于支持超时设定的流，ReadTimeout和WriteTimeout属性可用来确定以毫秒为单位的预期超时时间，其中0表示不设定超时。Read和Write方法会在超时发生时抛出一个异常。

15.2.5 线程安全

通常，流并不是线程安全的，这意味着两个线程在并发读或写同一个流时可能会发生错误。Stream类提供了一个简单的解决方法，即使用静态Synchronized方法。这个方法可以接受任何类型的流，并返回一个线程安全的包装器。这个包装器会为每个读取、写入或查找操作获得排他锁，从而保证每次只有一个线程执行此类操作。在实践中，允许多个线程同时访问同一个流的数据，其他类型的活动（例如并发读取）则需要额外的锁，才能保证每一个线程都访问流中要求的部分。我们将在第22章全面介绍线程安全。

15.2.6 后备存储流

图15-2显示的是.NET Framework中主要的后备存储流。此外，通过Stream的静态Null域，我们也能够获得一个“空流”。

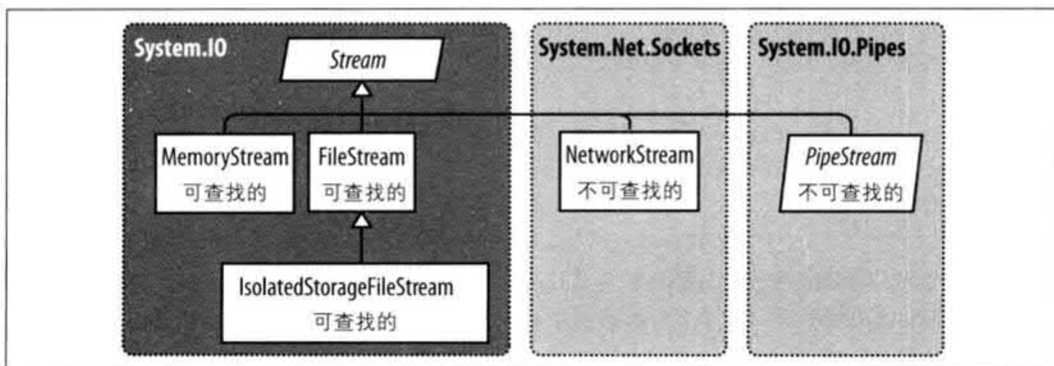


图15-2：后备存储流

在下面的小节我们将介绍FileStream和MemoryStream；在本章最后一节中，我们将介绍IsolatedStorageStream。在第16章中，我们将介绍NetworkStream。

15.2.7 FileStream

在本节前面，我们演示了使用FileStream读取和写入字节型数据的基本方法。现在，我们将学习这个类的具体特性。

提示： FileStream不适用于Metro应用。相反，要转而使用Windows.Storage的Windows Runtime类型（参见第642页的“Windows Runtime的文件I/O”）。

---

1. 创建一个FileStream

实例化FileStream的最简单方法是使用File类的以下静态外观方法之一：

```
FileStream fs1 = File.OpenRead  ("readme.bin");             // 只读
FileStream fs2 = File.OpenWrite (@"c:\temp\writeme.tmp");   // 只写
FileStream fs3 = File.Create    (@"c:\temp\writeme.tmp");   // 读/写
```

如果文件已经存在，那么OpenWrite和Create的行为是不同的。Create会截去全部已有的内容；OpenWrite则会原封不动地保留流中从位置0开始的已有内容。如果写入的字节小于文件已有字节，那么OpenWrite所产生的流会同时保存新旧内容。

我们还可以直接实例化一个FileStream。它的构造函数支持所有特性，允许指定文件名或底层文件句柄、文件创建和访问模式、共享选项、缓冲和安全性。下面的代码会直接打开一个已有文件进行读/写操作，而不需要重写这个文件：

```
var fs = new FileStream ("readwrite.tmp", FileMode.Open); // 读/写
```

我们后面会了解FileMode的更多用法。

File类的快捷方法

下面的静态方法能够一次性将整个文件读取到内存中：

- File.ReadAllText（返回一个字符串）
- File.ReadAllLines（返回一个字符串数组）
- File.ReadAllBytes（返回一个字节数组）

下面的静态方法能够一次性写入一个完整的文件：

- File.WriteAllText
- File.WriteAllLines
- File.WriteAllBytes
- File.AppendAllText（适用于给日志文件附加内容）

从Framework 4.0开始，增加了一个静态方法File.ReadLines。这个方法与ReadAllLines类似，唯一不同的是它返回一个延后判断的IEnumerable<string>。这个方法效率更高，因为它不会一次性将整个文件加载到内存中。LINQ是处理这些结果的理想方法：下面的代码会统计出长度大于80个字符的行数：

```
int longLines = File.ReadLines ("filePath")
                    .Count (l => l.Length > 80);
```

2. 指定文件名

文件名可以是绝对路径（例如*c:\temp\test.txt*），也可以是当前目录的相对路径（例如*test.txt*或*temp\test.txt*）。我们可以通过静态的`Environment.CurrentDirectory`属性来访问或修改当前目录。

警告： 当程序启动时，当前目录不一定是程序执行文件所在的路径。因此，一定不要使用当前目录来定位与可执行文件一起打包的额外运行时文件。

`AppDomain.CurrentDomain.BaseDirectory`会返回应用程序根目录，正常情况下它就是程序可执行文件所在的文件夹。使用`Path.Combine`可以指定相对于这个目录的文件名：

```
string baseFolder = AppDomain.CurrentDomain.BaseDirectory;
string logoPath = Path.Combine (baseFolder, "logo.jpg");
Console.WriteLine (File.Exists (logoPath));
```

我们还可以通过UNC路径读写一个网络文件，例如*\\JoesPC\PicShare\pic.jpg*或*\\10.1.1.2\PicShare\pic.jpg*。

3. 指定FileMode

`FileStream`的所有构造函数接受文件名需要一个`FileMode`枚举参数。图15-3说明了如何选择`FileMode`，而这些选择的结果与调用File类的静态方法类似。

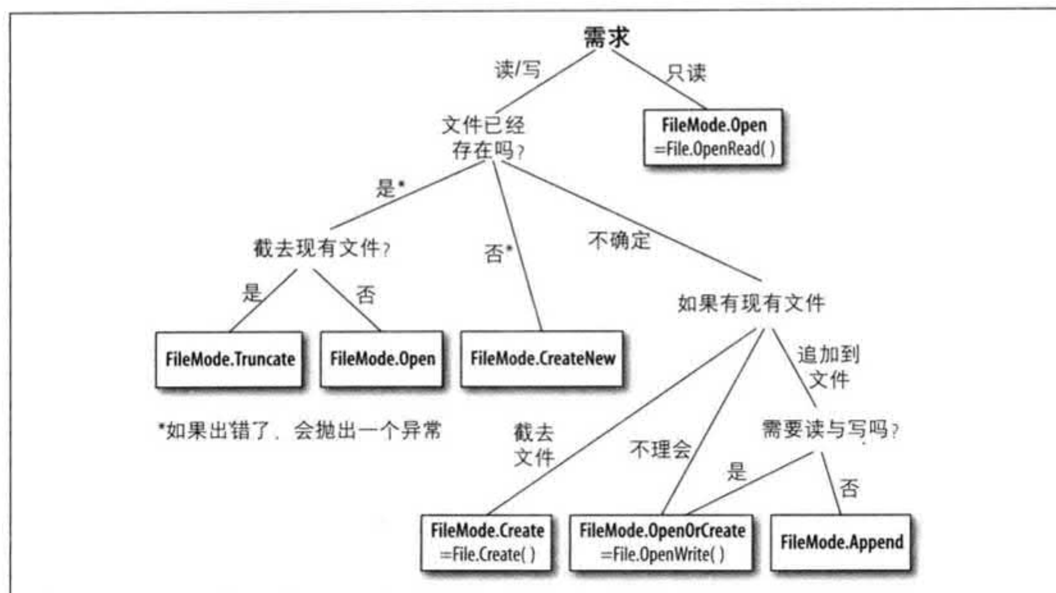


图15-3：选择FileMode

警告： 用`File.Create`和`FileMode.Create`处理隐藏文件会抛出一个异常。必须先删除隐藏文件再重新创建：

```
if (File.Exists ("hidden.txt")) File.Delete ("hidden.txt");
```

只使用文件名和FileMode创建一个FileStream会得到（只有一种异常）一个可读写的流。如果传入一个FileAccess参数，就可以要求降低读写模式：

```
[Flags]
public enum FileAccess { Read = 1, Write = 2, ReadWrite = 3 }
```

下面的代码返回一个只读流，相当于调用File.OpenRead：

```
using (var fs = new FileStream ("x.bin", FileMode.Open, FileAccess.Read))
...
```

FileMode.Append是最奇怪的一个方法：使用这个模式会得到只写流。相反，要附加读写支持，我们使用FileMode.Open或FileMode.OpenOrCreate，然后再查找流的结尾：

```
using (var fs = new FileStream ("myFile.bin", FileMode.Open))
{
  fs.Seek (0, SeekOrigin.End);
  ...
```

4. FileStream高级特性

下面是创建FileStream时可以选择的其他参数：

- 一个FileShare枚举值，描述了在完成文件处理之前，可以给同一个文件的其他进程授予的访问权限（None、Read [default]、ReadWrite或者Write）。
- 以字节为单位的内部缓冲区大小（当前的默认值是4KB）。
- 一个标记，表示是否由操作系统管理异步I/O。
- 一个FileSecurity对象，描述给新文件分配什么用户和角色权限。
- 一个FileOptions标记枚举值，包括请求操作系统加密（Encrypted）、在临时文件关闭时自动删除（DeleteOnClose）和优化提示（RandomAccess和SequentialScan）。此外，还有一个WriteThrough标记要求操作系统禁用写后缓存，适用于事务文件或日志。

使用FileShare.ReadWrite打开一个文件允许其他进程或用户同时读写同一个文件。为了避免混乱，我们可以使用以下方法在读或写之前锁定文件的特定部分：

```
// 定义于FileStream类：
public virtual void Lock (long position, long length);
public virtual void Unlock (long position, long length);
```

如果所请求的文件段的部分或全部已经被锁定，那么Lock会抛出一个异常。这是系统在基于文件的数据库中采用的方法，例如Access和FoxPro。

15.2.8 MemoryStream

MemoryStream使用一个数组作为后备存储。这在一定程度是与使用流的目的相违背的，因为整个后备存储都必须一次性驻留在内存中。然而，MemoryStream仍然有一定的用途；一个示例是随机访问一个不可查找的流。如果原始流具有可管理大小，那么通过以下方式可以将它复制到一个MemoryStream中：

```
static MemoryStream ToMemoryStream (this Stream input, bool closeInput)
```

```
{
  try
  {                                                  // 读和写入4KB的数据块
    byte[] block = new byte [0×1000];
    MemoryStream ms = new MemoryStream();
    while (true)
    {
      int bytesRead = input.Read (block, 0, block.Length);
      if (bytesRead == 0) return ms;
      ms.Write (block, 0, bytesRead);
    }
  }
  finally { if (closeInput) input.Close (); }
}
```

使用closeInput参数的原因是为了避免出现方法编写者和使用者均认为对象会关闭流的误会。

调用ToArray可以将一个MemoryStream转换为一个字节数组。GetBuffer方法也可以实现相同操作，而且效率更高，它将返回一个底层存储数组的直接引用。但是，这个数组通常会比流的实际长度长一些。

提示：MemoryStream的关闭和清除不是必需的。如果关闭了一个MemoryStream，我们就无法再读或写这个流，但是我们仍然可以调用ToArray来获得底层数据。消除实际上不会对内存流执行任何操作。

在本章后面“压缩”小节和第21章的“密码学概述”小节中，我们将进一步介绍MemoryStream实例。

15.2.9 PipeStream

PipeStream是在Framework 3.5引入的。它支持一种简单的方法，其中一个进程可以通过Windows管道协议与另一个进程进行通信。

管道的类型有两种：

匿名管道

支持在同一个computer.id的父子进程之间单向通信。

命名管道

支持同一台计算机或Windows网络中不同计算机的任意进程之间进行通信。

管道很适合用于在一台计算机上进行进程间通信（IPC）：它不依赖于任何网络传输，性能更好，也不会有防火墙问题。

提示：管道是基于流实现的，所以一个进程会等待接收字节，而另一个进程则负责发送字节。另一种进程通信方法可以通过共享内存块实现，我们将在“内存映射文件”一节中介绍它的实现方法。

PipeStream是一个抽象类，它有4个实现子类。其中两个支持匿名管道和两个支持命名管道：

匿名管道

AnonymousPipeServerStream和AnonymousPipeClientStream

命名管道

NamedPipeServerStream和NamedPipeClientStream

命名管道使用更简单，所以我们先介绍它的使用方法。

提示：管道是一个底层概念，它支持发送和接收字节（或消息，即字节组）。WCF和Remoting API支持使用IPC通道进行通信的更高级消息框架。

1. 命名管道

通过命名管道，各方将使用一个同名管道进行通信。这个协议定义了两个不同的角色：客户端和服务器。客户端与服务器之间的通信采用以下方式：

- 服务器初始化一个NamedPipeServerStream，然后调用WaitForConnection。
- 客户端初始化一个NamedPipeClientStream，然后调用Connect（使用一个可选的超时时间）。

然后，双方通过读写流进行通信。

下面的示例演示了服务器发送一个字节（100），然后等待接收一个字节：

```
using (var s = new NamedPipeServerStream ("pipedream"))
{
  s.WaitForConnection();
  s.WriteByte (100);
  Console.WriteLine (s.ReadByte());
}
```

下面是对应的客户端代码：

```
using (var s = new NamedPipeClientStream ("pipedream"))
{
  s.Connect();
  Console.WriteLine (s.ReadByte());
  s.WriteByte (200);                        // 返回值为200
}
```

命名管道流默认是双向通信的，所以任何一方都可以读或写它们的流。这意味着客户端和服务器都必须同意使用一种协议来协调它们的操作，所以双方是不能同时发送或接收消息的。

通信双方还需要统一每次传输的数据长度。我们的示例并没有强调这个方面，因为我们在每个方向只传输了一个字节。为了支持传输更长的消息，管道提供了一种消息传输模式。如果启用这个模式，调用Read的一方可以通过检查IsMessageComplete属性来确定消息是否完成传输。为了演示这一点，我们将编写一个帮助方法，它会从一个启用消息的PipeStream读取完整的消息，换言之，一直读取到IsMessageComplete变成true：

```
static byte[] ReadMessage (PipeStream s)
{
  MemoryStream ms = new MemoryStream();
  byte[] buffer = new byte [0×1000];              // 读入4KB的数据块

  do { ms.Write (buffer, 0, s.Read (buffer, 0, buffer.Length)); }
  while (!s.IsMessageComplete);

  return ms.ToArray();
}
```

（为了实现这种异步性，要将“s.Read”替换为“await s.ReadAsync”。）

警告： 只需要等待Read返回0，我们就可以确定一个PipeStream是否完成消息的读取。这是因为，与其他大多数流不同，管道流和网络流并没有确定的结尾。相反，它们会在消息传输期间临时中断。

现在，我们可以激活消息传输模式。在服务器端，创建流时指定PipeTransmissionMode.Message，就可以激活消息传输：

```
using (var s = new NamedPipeServerStream ("pipedream", PipeDirection.InOut,
                                          1, PipeTransmissionMode.Message))
{
  s.WaitForConnection();

  byte[] msg = Encoding.UTF8.GetBytes ("Hello");
  s.Write (msg, 0, msg.Length);

  Console.WriteLine (Encoding.UTF8.GetString (ReadMessage (s)));
}
```

在客户端，调用Connect之后设置ReadMode就可以激活消息传输：

```
using (var s = new NamedPipeClientStream ("pipedream"))
{
  s.Connect();
  s.ReadMode = PipeTransmissionMode.Message;

  Console.WriteLine (Encoding.UTF8.GetString (ReadMessage (s)));

  byte[] msg = Encoding.UTF8.GetBytes ("Hello right back!");
  s.Write (msg, 0, msg.Length);
}
```

2. 匿名管道

匿名管道支持在父子进程之间进行单向通信。匿名管道不使用系统级名称，而是通过一个私有句柄进行调整。

与命名管道一样，匿名管道也区分客户端和服务器角色。然而，通信系统有一些不同，它采用以下方法：

1. 服务器初始化一个AnonymousPipeServerStream，提交一个In或Out的PipeDirection。
2. 服务器调用GetClientHandleAsString获取管道的标识，然后传递回客户端（一般作为启动子进程的一个参数）。
3. 子进程初始化一个AnonymousPipeClientStream，指定相反的PipeDirection。
4. 服务器调用DisposeLocalCopyOfClientHandle，释放第2步产生的本地句柄。
5. 父子进程通过读/写流来进行通信。

因为匿名管道是单向的，所以服务器必须为双向通信创建两个管道。在下面的代码中，服务器向子进程发送一个字节，然后再从该进程接收一个字节：

```
string clientExe = @"d:\PipeDemo\ClientDemo.exe";

HandleInheritability inherit = HandleInheritability.Inheritable;
```

```
using (var tx = new AnonymousPipeServerStream (PipeDirection.Out, inherit))
using (var rx = new AnonymousPipeServerStream (PipeDirection.In, inherit))
{
  string txID = tx.GetClientHandleAsString();
  string rxID = rx.GetClientHandleAsString();

  var startInfo = new ProcessStartInfo (clientExe, txID + " " + rxID);
  startInfo.UseShellExecute = false;          // 子进程需要的
  Process p = Process.Start (startInfo);

  tx.DisposeLocalCopyOfClientHandle();        // 释放未托管的处理资源
  rx.DisposeLocalCopyOfClientHandle();

  tx.WriteByte (100);
  Console.WriteLine ("Server received: " + rx.ReadByte());

  p.WaitForExit();
}
```

下面的客户端代码将会编译到*d:\PipeDemo\ClientDemo.exe*：

```
string rxID = args[0];         // 注意我们对调了接收和传输角色
string txID = args[1];

using (var rx = new AnonymousPipeClientStream (PipeDirection.In, rxID))
using (var tx = new AnonymousPipeClientStream (PipeDirection.Out, txID))
{
  Console.WriteLine ("Client received: " + rx.ReadByte());
  tx.WriteByte (200);
}
```

与命名管道一样，客户端和服务器必须协调它们的发送和接收，并且统一每一次传输的数据长度。但是，匿名管道不支持消息模式，所以必须实现自己的消息长度认同协议。一种方法是在每次传输的前4个字节中发送一个整数值，定义后续消息的长度。BitConverter类具有一些用于转换整数和4字节数组的方法。

15.2.10 BufferedStream

BufferedStream可以装饰或包装另一个具有缓冲功能的流，它是.NET Framework的诸多核心装饰流类型之一，图15-4列出了所有类型。

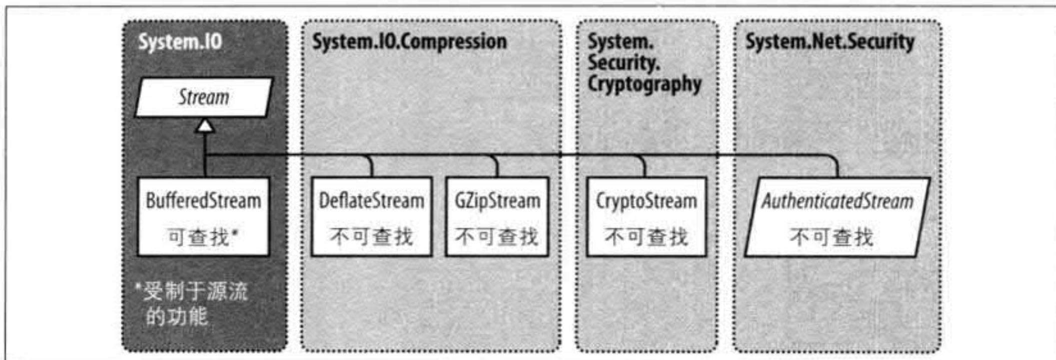


图15-4：装饰器流

缓冲能够减少后备存储的方法，从而提高性能。在下面的代码中，我们将一个FileStream包装在一个20KB的BufferedStream中：

```
// 将100K数据写入到一个文件中：
File.WriteAllBytes ("myFile.bin", new byte [100000]);

using (FileStream fs = File.OpenRead ("myFile.bin"))
using (BufferedStream bs = new BufferedStream (fs, 20000))    // 20K缓冲区
{
  bs.ReadByte();
  Console.WriteLine (fs.Position);                             // 20000
}
```

在这个示例中，通过向前读缓冲，底层流在读取1个字节之后前进20,000字节。不需要再次访问FileStream，我们就能够再调用19,999次ReadByte。

就像这个示例一样，组合使用BufferedStream和FileStream的好处并不明显，因为FileStream已经有内置的缓冲了。它的唯一用途可能就是扩大一个已有FileStream的缓冲区。

关闭一个BufferedStream会自动关闭底层的后备存储流。

15.3 流适配器

Stream只支持字节处理；要读写一些数据类型，例如字符串、整数或XML元素，我们必须插入适配器。下面是Framework支持的适配器：

文本适配器（处理字符串和字符数据）

  TextReader、TextWriter

  StreamReader、StreamWriter

  StringReader、StringWriter

二进制适配器（处理基本数据类型，例如int、bool、string和float）

  BinaryReader、BinaryWriter

XML适配器（已在第11章介绍）

  XmlReader、XmlWriter

这些类型的关系如图15-5所示。

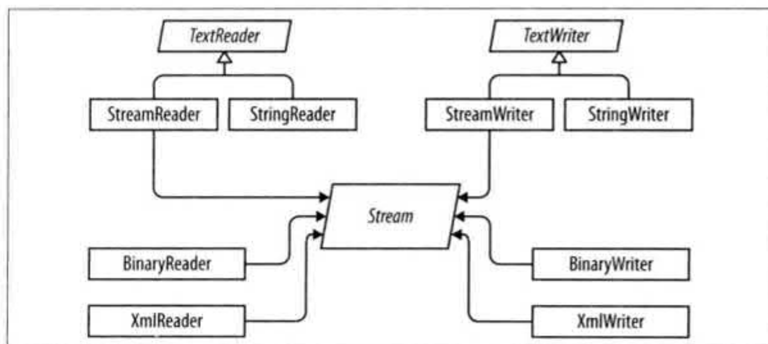


图15-5：阅读器和编写器

15.3.1 文本适配器

TextReader和TextWriter都是专门处理字符和字符串的适配器的抽象基类。它们在框架中都是两个通用的实现：

StreamReader/StreamWriter
    使用Stream存储它的原始数据，将流的字节转换成字符或字符串。

StringReader/StringWriter
    使用内存字符串实现TextReader/TextWriter。

表15-2按分类列出了TextReader的成员。不需要将位置前移，Peek就可以返回流中的下一个字符。

如果到达流的末尾，那么Peek与不带参数的Read都会返回-1；否则，它们会返回一个能够强制转换为char的整数。接收一个char[]缓冲区参数的Read重载函数功能与ReadBlock方法相似。ReadLine会一直读取到出现CR（13号字符）或LF（10号字符）或CR+LF对为止。然后，它会返回一个丢弃CR/LF字符的字符串。

表15-2：TextReader成员

分类	成员
读取一个字符	public virtual int Peek(); // 将结果强制转换为char
	public virtual int Read(); // 将结果强制转换为char
读取多个字符	public virtual int Read (char[] buffer, int index, int count);
	public virtual int ReadBlock (char[] buffer, int index, int count);
	public virtual string ReadLine();
	public virtual string ReadToEnd();
关闭	public virtual void Close();
	public void Dispose(); // 与Close相同
其他	public static readonly TextReader Null;
	public static TextReader Synchronized (TextReader reader);

提示： Windows的新换行字符是模仿机械打字机的：回车符（13号字符）后面加上一个换行符（10号字符）。C#字符串表示是"\r\n"（可以认为是“ReturN”）。如果顺序调换，结果可能是两行，也可能一行也没有！

TextWriter有与之类似的写方法，如表15-3所示。Write和WriteLine方法还重载了接受所有基本数据类型及object类型的方法。这些方法会直接调用ToString方法处理所传入的数据（也可以选择在调用该方法或创建TextWriter时通过指定的IFormatProvider进行处理）。

表15-3：TextWriter成员

分类	方法
写一个字符	public virtual void Write (char value);
写多个字符	public virtual void Write (string value);

表15-3：TextWriter成员（续）

分类	方法
	`public virtual void Write (char[] buffer, int index, int count);`
	`public virtual void Write (string format, params object[] arg);`
	`public virtual void WriteLine (string value);`
关闭和清除	`public virtual void Close();`
	`public void Dispose(); // 与Close相同`
	`public virtual void Flush();`
格式化和编码	`public virtual IFormatProvider FormatProvider { get; }`
	`public virtual string NewLine { get; set; }`
	`public abstract Encoding Encoding { get; }`
其他	`public static readonly TextWriter Null;`
	`public static TextWriter Synchronized (TextWriter writer);`

WriteLine会给指定文本附加CR+LF。我们可以使用NewLine属性修改这些字符，这对于支持UNIX文件格式的互操作性很有用。

提示：和Stream一样，TextReader和TextWriter为他们的读/写方法提供了基于任务的异步版本。

1. StreamReader和StreamWriter

在下面的示例中，先用一个StreamWriter将两行文本写入到一个文件中，然后再用一个StreamReader读取文件的内容：

```
using (FileStream fs = File.Create ("test.txt"))
using (TextWriter writer = new StreamWriter (fs))
{
  writer.WriteLine ("Line1");
  writer.WriteLine ("Line2");
}

using (FileStream fs = File.OpenRead ("test.txt"))
using (TextReader reader = new StreamReader (fs))
{
  Console.WriteLine (reader.ReadLine());     // Line1
  Console.WriteLine (reader.ReadLine());     // Line2
}
```

因为文本适配器通常与文件有关，所以File类也有一些静态方法支持快捷处理，例如CreateText、AppendText和OpenText：

```
using (TextWriter writer = File.CreateText ("test.txt"))
{
  writer.WriteLine ("Line1");
  writer.WriteLine ("Line2");
}
```

```
using (TextWriter writer = File.AppendText ("test.txt"))
  writer.WriteLine ("Line3");

using (TextReader reader = File.OpenText ("test.txt"))
  while (reader.Peek() > _1)
  Console.WriteLine (reader.ReadLine());      // Line1
                                              // Line2
                                              // Line3
```

这也演示了通过reader.Peek()测试文件结尾的判断方法。另一个方法是不停地读取，直至reader.ReadLine返回null。

我们还可以以整数方式读写其他的类型，但是因为TextWriter在类型中调用ToString，所以必须解析字符串才能读取数据：

```
using (TextWriter w = File.CreateText ("data.txt"))
{
  w.WriteLine (123);                 // 写入"123"
  w.WriteLine (true);                // 写入单词"true"
}
using (TextReader r = File.OpenText ("data.txt"))
{
  int myInt = int.Parse (r.ReadLine());      // myInt == 123
  bool yes = bool.Parse (r.ReadLine());      // yes == true
}
```

2. 字符编码

TextReader和TextWriter本身是与流或后备存储无关的抽象类。然而，类型StreamReader和StreamWriter都与底层的字节流相关，所以它们必须进行字符和字节之间的转换。它们是通过System.Text命名空间的Encoding类进行这些操作的，创建StreamReader或StreamWriter需要选择一种编码方式。如果不进行选择，那么就使用默认的UTF-8编码。

警告：如果明确指定一个编码方式，默认情况下StreamWriter会在流开头写入一个前缀，用于指定编码方式。这通常不是一种好做法，按照下面的方式指定编码方式会更好：

```
var encoding = new UTF8Encoding (
  encoderShouldEmitUTF8Identifier:false,
  throwOnInvalidBytes:true);
```

第2个参数让StreamWriter（或StreamReader）在遇到无法根据编码转换为有效字符串的字节时抛出一个异常，其行为与未指定编码方式的默认行为相同。

最简单的编码方式是ASCII，因为每一个字符都是用一个字节表示的。ASCII编码将Unicode字符集的前127个字符映射为一个字节，其中包括键盘上的所有字符。而包括特殊符号与非英语字符在内的其他大多数字符都无法表示，它们会被转换为□字符。默认的UTF-8编码方式也能够映射所有分配的Unicode字符，但是更复杂一些。它将前127个字符编码为一个字节，以兼容ASCII；其他字符则编码为一定数量的字节（通常是两个或三个）。假设有以下代码：

```
using (TextWriter w = File.CreateText ("but.txt"))    // 使用默认的UTF-8编码。
```

```
  w.WriteLine ("but-");

using (Stream s = File.OpenRead ("but.txt"))
  for (int b; (b = s.ReadByte()) > -1;)
    Console.WriteLine (b);
```

单词“but”后面紧跟的不是一个标准的连字符，而是长破折号（—）：U+2014。这个字符在编辑器上显示是看不出问题的！让我们来查看一下输出：

```
98      // b
117     // u
116     // t
226     // 长破折号字节 1 注意，多个字节序列各部分字节值大于等于128。
128     // 长破折号字节 2
148     // 长破折号字节 3
13      // <CR>
10      // <LF>
```

因为长破折号不属于Unicode字符集的前127个字符，所以它在UTF-8中需要多于一个的字节来表示（在这里是3个字节）。UTF-8在处理西方字母时很高效，因为最常用的字符只需要1个字节。只需要忽略127之后的字节，它就能够轻松向下兼容ASCII。缺点是在流中查找是很麻烦的，因为字符的位置与它在流中的字节位置是无关的。另一种方式是UTF-16（在Encoding类中仅仅标记为“Unicode”）。下面的示例说明了如何使用UTF-16编写相同的字符串：

```
using (Stream s = File.Create ("but.txt"))
using (TextWriter w = new StreamWriter (s, Encoding.Unicode))
  w.WriteLine ("but-");

foreach (byte b in File.ReadAllBytes ("but.txt"))
  Console.WriteLine (b);
```

输出结果是：

```
255     // 字节顺序标记1
254     // 字节顺序标记2
98      // 'b' 字节 1
0       // 'b' 字节 2
117     // 'u' 字节 1
0       // 'u' 字节 2
116     // 't' 字节 1
0       // 't' 字节 2
20      // '--' 字节 1
32      // '--' 字节 2
13      // <CR> 字节 1
0       // <CR> 字节 2
10      // <LF> 字节 1
0       // <LF> 字节 2
```

技术上，UTF-16使用2个或4个字节来表示一个字符（所分配或保护的Unicode字符接近一百万个，所以2个字节并不总是足够的）。然而，因为C#的char类型本身只有16位，所以UTF-16编码方式总是使用2个字节来表示一个.NET 的char类型。这样就能够很容易转到流中特定的字符索引。

UTF-16使用2个字节前缀来确定字节对采用“小字节序”还是“大字节序”（最低有效字节在前还是最高有效字节在前）。Windows系统采用的默认标准是小字节序。

3. StringReader和StringWriter

StringReader和StringWriter适配器并不封装流；相反，它们使用一个字符串或StringBuilder作为底层数据源。这意味着不需要进行任何的字节转换，事实上，这些类所执行的操作都可以通过字符串或StringBuilder与一个索引变量轻松实现。并且它们的优点是与StreamReader/StreamWriter使用相同的基类。例如，假设我们有一个包含XML的字符串，然后希望用一个XmlReader来解析这个字符串。XmlReader.Create方法可以接受以下参数：

- URI
- Stream
- TextReader

那么，我们如何解析XML字符串呢？因为StringReader是TextReader的子类，所以我们可以实例化和传入一个StringReader：

```
XmlReader r = XmlReader.Create (new StringReader (myString));
```

15.3.2 二进制适配器

BinaryReader和BinaryWriter能够读写基本的数据类型：bool、byte、char、decimal、float、double、short、int、long、sbyte、ushort、uint和ulong以及字符串和数组等。

与StreamReader和StreamWriter不同的是，二进制适配器能够高效地存储基本数据类型，因为它们位于内存中。所以，一个int占用4个字节；一个double占用8个字节。字符串是通过文本编码（与StreamReader和StreamWriter一样）写入的，但是带有长度前缀，从而不需要特殊分隔符就能够读取一系列字符串。

假设我们有一个简单的类型，其定义如下：

```
public class Person
{
  public string Name;
  public int Age;
  public double Height;
}
```

我们可以给Person类添加以下方法，使用二进制适配器将它的数据保存到一个流中，或者从一个流中加载数据：

```
public void SaveData (Stream s)
{
  var w = new BinaryWriter (s);
  w.Write (Name);
  w.Write (Age);
  w.Write (Height);
  w.Flush();                       // 保证清空BinaryWriter缓冲区。
                                   // 我们不会清理/关闭它，所以可以在流中写入更多的数据。
}

public void LoadData (Stream s)
{
  var r      = new BinaryReader (s);
```

```
    Name       = r.ReadString();
    Age        = r.ReadInt32();
    Height     = r.ReadDouble();
  }
```

BinaryReader也支持读入字节数组。下面的代码将读取一个可查找流中的全部内容：

```
byte[] data = new BinaryReader (s).ReadBytes ((int) s.Length);
```

这比直接从一个流中读取数据更方便一些，因为它不需要使用循环来保证已经读取所有数据。

15.3.3 关闭和清理流适配器

清理流适配器有4种方法：

1. 只关闭适配器。
2. 先关闭适配器，然后再关闭流。
3. （对于编写器）先清理适配器，然后再关闭流。
4. （对于读取器）直接关闭流。

提示：对于适配器和流，Close（关闭）和Dispose（清理）是同义词。

方法（1）和（2）在语义上是相同的，因为关闭一个适配器会自动关闭底层的流。每当嵌入一个using语句，就表示采用方法（2）：

```
using (FileStream fs = File.Create ("test.txt"))
using (TextWriter writer = new StreamWriter (fs))
  writer.WriteLine ("Line");
```

因为嵌入语句是从内向外清理的，所以适配器先关闭，然后再关闭流。而且，如果适配器构造函数中抛出一个异常，那么这个流仍然会关闭。因此，嵌入一个using语句是最佳方法！

警告：一定不要在关闭和清理编写器之前关闭一个流，这样会丢失仍在适配器中缓存的所有数据。

方法（3）和（4）之所以有效，是因为适配器属于随意清理对象这个分类。例如，当用完一个适配器时，可能不选择马上处理这个适配器，而是希望保留底层流，以备后面继续使用：

```
using (FileStream fs = new FileStream ("test.txt", FileMode.Create))
{
  StreamWriter writer = new StreamWriter (fs);
  writer.WriteLine ("Hello");
  writer.Flush();

  fs.Position = 0;
  Console.WriteLine (fs.ReadByte());
}
```

在这个示例中，我们写入一个文件，重新定位流，然后读取第一个字节，最后关闭这个流。如果我们清理掉StreamWriter，那么也就关闭了底层的FileStream，从而导致后面的读操作失败。前提条

件是我们要调用Flush来保证将StreamWriter的缓冲区数据写入到底层的流中。

提示： 流适配器及其可选的清理语法并没有实现扩展的清理模式，即在终结器中调用Dispose。这可以避免垃圾回收器找到弃用的适配器时自动清理这个适配器。

从Framework 4.5开始，StreamReader/StreamWriter有一个新的构造方法，它可以让流在清理之后仍然保持打开。因此，我们可以将前一个例子重写为：

```
using (var fs = new FileStream ("test.txt", FileMode.Create))
{
  using (var writer = new StreamWriter (fs, new UTF8Encoding (false, true),
                                        0x400, true))
  writer.WriteLine ("Hello");
  fs.Position = 0;
  Console.WriteLine (fs.ReadByte());
  Console.WriteLine (fs.Length);
}
```

15.4 压缩流

System.IO.Compression命名空间提供了两个通用压缩流：DeflateStream和GZipStream。这两个类都使用与ZIP格式类似的流行压缩算法。它们的区别是：GZipStream会在开头和结尾写入一个额外的协议——其中包括检测错误的CRC。GZipStream还遵循一个其他软件可识别的标准。

这两种流都支持读写操作，但是有以下限制条件：

- 压缩时总是在写入流。
- 解压缩时总是在读取流。

DeflateStream和GZipStream都是装饰器：它们负责压缩或解压缩构造方法传入的另一个流。下面的例子使用FileStream作为后台存储流，执行一系列字节的压缩和解压缩操作：

```
using (Stream s = File.Create ("compressed.bin"))
using (Stream ds = new DeflateStream (s, CompressionMode.Compress))
  for (byte i = 0; i < 100; i++)
    ds.WriteByte (i);

using (Stream s = File.OpenRead ("compressed.bin"))
using (Stream ds = new DeflateStream (s, CompressionMode.Decompress))
  for (byte i = 0; i < 100; i++)
    Console.WriteLine (ds.ReadByte()); // 写入0至99
```

即使使用两个算法中较小的算法，压缩文件大小也是241字节：比原始文件大两倍！非重复性二进制文件数据的压缩效果很差（缺少设计规范性的加密数据的压缩比是最差的），这种压缩适用于大多数文本文件。下一个例子将对从一个句子中随机选取的1,000个单词的文本流执行压缩和解压缩操作。这个例子也演示了串联后台存储流、装饰流和适配器（如本章开头图15-1所示）的用法，并且使用了异步方法：

```
string[] words = "The quick brown fox jumps over the lazy dog".Split();
Random rand = new Random();
using (Stream s = File.Create ("compressed.bin"))
```

```
using (Stream ds = new DeflateStream (s, CompressionMode.Compress))
using (TextWriter w = new StreamWriter (ds))
  for (int i = 0; i < 1000; i++)
    await w.WriteAsync (words [rand.Next (words.Length)] + " ");

Console.WriteLine (new FileInfo ("compressed.bin").Length);      // 1073

using (Stream s = File.OpenRead ("compressed.bin"))
using (Stream ds = new DeflateStream (s, CompressionMode.Decompress))
using (TextReader r = new StreamReader (ds))
  Console.Write (await r.ReadToEndAsync());  // 输出结果如下：

lazy lazy the fox the quick The brown fox jumps over fox over fox The
brown brown brown over brown quick fox brown dog dog lazy fox dog brown
over fox jumps lazy lazy quick The jumps fox jumps The over jumps dog...
```

在这个例子中，DeflateStream的压缩结果为1,073字节——大约每一个单词1字节。

内存中压缩

有时候，我们需要在内存中执行压缩。下面演示了如何使用MemoryStream执行这个操作：

```
byte[] data = new byte[1000];          // 预期空数组可以得到较好的压缩效果！

var ms = new MemoryStream();
using (Stream ds = new DeflateStream (ms, CompressionMode.Compress))
  ds.Write (data, 0, data.Length);

byte[] compressed = ms.ToArray();
Console.WriteLine (compressed.Length);         // 113

// 解压缩回数据数组：
ms = new MemoryStream (compressed);
using (Stream ds = new DeflateStream (ms, CompressionMode.Decompress))
  for (int i = 0; i < 1000; i += ds.Read (data, i, 1000 - i));
```

DeflateStream类之前的using语句是教科书式的关闭流方法，它会清除进程中所有未写入的缓冲数据。它也会关闭所封装的MemoryStream——这意味着必须在后面调用ToArray提取它的数据。

下面是另一种不需要关闭MemoryStream的方法，它使用异步读写方法：

```
byte[] data = new byte[1000];

MemoryStream ms = new MemoryStream();
using (Stream ds = new DeflateStream (ms, CompressionMode.Compress, true))
  await ds.WriteAsync (data, 0, data.Length);

Console.WriteLine (ms.Length);             // 113
ms.Position = 0;
using (Stream ds = new DeflateStream (ms, CompressionMode.Decompress))
  for (int i = 0; i < 1000; i += await ds.ReadAsync (data, i, 1000 - i));
```

在DeflateStream构造方法传入的额外标记，表示在清除底层流时不采用普通的协议。换言之，MemoryStream保持打开，从而允许将位置重置为0，然后重新读取流。

15.5 操作Zip文件

Framework 4.5引入了一个新特性：支持流行的Zip文件压缩格式，实现方法是System.IO.Compression中（位于*System.IO.Compression.dll*中）新增加的ZipArchive和ZipFile类。与DeflateStream和GZipStream相比，这种格式的优点是可以处理多个文件，并且兼容Windows资源管理器及其他压缩工具创建的Zip文件。

ZipArchive可以操作流，而ZipFile则负责操作更常见的文件。（ZipFile是ZipArchive的静态帮助类。）

ZipFile的CreateFromDirectory方法可以将指定目录的所有文件添加到一个Zip文件中：

```
ZipFile.CreateFromDirectory (@"d:\MyFolder", @"d:\compressed.zip");
```

而ExtractToDirectory则执行相反操作，可以将一个Zip文件解压缩到一个目录中：

```
ZipFile.ExtractToDirectory (@"d:\compressed.zip", @"d:\MyFolder");
```

在压缩时，可以指定是否优化文件大小或压缩速度，以及是否在存档文件中包含源目录名称。在我们的例子中，启用后者将会在存档中创建一个子目录*My-Folder*，其中保存压缩后的文件。

ZipFile包含一个Open方法，它可以读/写各个文件项目。这个方法会返回一个ZipArchive对象（也可以通过使用一个Stream对象创建ZipArchive实例而获得）。当调用Open时，必须指定一个文件名，并且指定存档文件操作方式：Read、Create或Update。然后，使用Entries属性遍历现有的项目，或者使用GetEntry查询某个文件：

```
using (ZipArchive zip = ZipFile.Open (@"d:\zz.zip", ZipArchiveMode.Read))
  foreach (ZipArchiveEntry entry in zip.Entries)
    Console.WriteLine (entry.FullName + " " + entry.Length);
```

ZipArchiveEntry还有Delete方法、ExtractToFile方法（实际是ZipFileExtensions类的扩展方法）和Open方法（返回一个可读/可写的Stream）。调用ZipArchive的CreateEntry或者CreateEntryFromFile扩展方法，可以创建新项目。下面的语句将创建存档文件*d:\zz.zip*，其中包含*foo.dll*，它位于*bin\X86*存档之中的目录结构中：

```
byte[] data = File.ReadAllBytes (@"d:\foo.dll");
using (ZipArchive zip = ZipFile.Open (@"d:\zz.zip", ZipArchiveMode.Update))
  zip.CreateEntry (@"bin\X64\foo.dll").Open().Write (data, 0, data.Length);
```

使用MemoryStream创建ZipArchive，也可以在内存中实现相同效果。

15.6 文件与目录操作

System.IO命名空间有一些执行“实用的”文件与目录操作的类型，例如复制和移动、创建目录以及设置文件属性和权限。对于大多数特性，我们可以选择两种类型：一种采用静态方法，另一种采用实例方法：

静态类

  File和Directory

实例方法类（使用文件或目录名创建）

```
FileInfo和DirectoryInfo
```

此外，还有一个静态类Path。它不操作文件或目录；相反，它具有一些文件名或目录路径的字符串处理方法。Path也能够帮助处理临时文件。

所有这些类都不适用于Metro应用（参见第15.7节的“Windows Runtime的文件输入/输出”）。

15.6.1 File类

File是一个静态类，它的方法都接受文件名参数。这个文件名可以是相对当前目录的路径，也可以是一个目录的完整路径。下面是它的一些方法（所有的public和static方法）：

```
bool Exists (string path);          // 文件存在时返回true

void Delete (string path);
void Copy          (string sourceFileName, string destFileName);
void Move          (string sourceFileName, string destFileName);
void Replace       (string sourceFileName, string destinationFileName,
                                           string destinationBackupFileName);

FileAttributes GetAttributes  (string path);
void SetAttributes            (string path, FileAttributes fileAttributes);

void Decrypt (string path);
void Encrypt (string path);

DateTime GetCreationTime      (string path); // UTC版本也支持
DateTime GetLastAccessTime    (string path);
DateTime GetLastWriteTime     (string path);

void SetCreationTime          (string path, DateTime creationTime);
void SetLastAccessTime        (string path, DateTime lastAccessTime);
void SetLastWriteTime         (string path, DateTime lastWriteTime);

FileSecurity GetAccessControl (string path);
FileSecurity GetAccessControl (string path,AccessControlSections includeSections);
void SetAccessControl (string path, FileSecurity fileSecurity);
```

如果目标文件已存在，那么Move会抛出一个异常；但是Replace不会。这两个方法允许将文件重命名或移动到另一个目录。

如果文件被标记为只读，那么Delete会抛出一个UnauthorizedAccessException；调用GetAttributes可以预先判断其属性。下面是GetAttributes返回的FileAttribute枚举类型的所有成员：

```
Archive, Compressed, Device, Directory, Encrypted,
Hidden, Normal, NotContentIndexed, Offline, ReadOnly,
ReparsePoint, SparseFile, System, Temporary
```

这个枚举类型的成员是可以组合的。下面的代码将演示如何在不影响其他属性的前提下切换一个文件属性：

```
string filePath = @"c:\temp\test.txt";

FileAttributes fa = File.GetAttributes (filePath);
if ((fa & FileAttributes.ReadOnly) > 0)
```

```
{
    fa ^= FileAttributes.ReadOnly;
    File.SetAttributes (filePath, fa);
}

// 例如，现在我们可以删除该文件：
File.Delete (filePath);
```

提示：FileInfo提供了一个更简单的修改文件只读标记的方法：

```
new FileInfo (@"c:\temp\test.txt").IsReadOnly = false;
```

---

1. 压缩与加密属性

Compressed和Encrypted文件属性与Windows资源浏览器中文件或目录的属性对话框的压缩和加密复选框相对应。这种压缩和加密是透明的，实际上是由操作系统完成所有的操作，它支持读写普通数据。

我们不能使用SetAttributes来修改文件的Compressed或Encrypted属性，它会出错但不会报错！后一种情况的解决方法很简单：调用File类的Encrypt()和Decrypt()方法。经过压缩后，它会变得更复杂；有一种解决方法是使用System.Management的Windows管理规范（Windows Management Instrumentation，WMI）API。下面的方法会压缩一个目录，成功后返回0（失败则返回一个WMI错误码）。

```
static uint CompressFolder (string folder, bool recursive)
{
  string path = "Win32_Directory.Name='" + folder + "'";
  using (ManagementObject dir = new ManagementObject (path))
  using (ManagementBaseObject p = dir.GetMethodParameters ("CompressEx"))
  {
    p ["Recursive"] = recursive;
    using (ManagementBaseObject result = dir.InvokeMethod ("CompressEx",p, null))
      return (uint) result.Properties ["ReturnValue"].Value;
  }
}
```

执行解压缩，可以将CompressEx替换成UncompressEx。

透明加密需要使用由用户登录密码生成的密钥。系统对于认证用户所执行的密码修改操作是很可靠的，但是如果密码被管理员重置，那么加密文件的数据就不可恢复。

提示：透明加密和压缩需要特殊的文件系统支持。NTFS（硬盘中使用最广泛的格式）支持这些特性；CDFS（在CD-ROM中）和FAT（在可移动内存卡中）则不支持。

Win32 互操作可用来确定一个卷是否支持压缩和加密：

```
using System;
using System.IO;
using System.Text;
using System.Runtime.InteropServices;

class SupportsCompressionEncryption
{
  const int SupportsCompression = 0x10;
```

```
  const int SupportsEncryption = 0x20000;

  [DllImport ("Kernel32.dll", SetLastError = true)]
  extern static bool GetVolumeInformation (string vol, StringBuilder name,
    int nameSize, out uint serialNum, out uint maxNameLen, out uint flags,
    StringBuilder fileSysName, int fileSysNameSize);

  static void Main()
  {
    uint serialNum, maxNameLen, flags;
    bool ok = GetVolumeInformation (@"C:\", null, 0, out serialNum,
                                    out maxNameLen, out flags, null, 0);
    if (!ok)
      throw new Win32Exception();

    bool canCompress = (flags & SupportsCompression) > 0;
    bool canEncrypt = (flags & SupportsEncryption) > 0;
  }
}
```

2. 文件安全性

GetAccessControl和SetAccessControl方法支持通过FileSecurity对象（位于命名空间System.Security.AccessControl）查询和修改操作系统授予用户和角色的权限。在创建一个新文件时，我们可以给FileStream的构造函数传入一个FileSecurity，以指定它的权限。

在这个示例中，我们列出了一个文件的现有权限，然后将运行权限授予"Users"组：

```
using System;
using System.IO;
using System.Security.AccessControl;
using System.Security.Principal;

...

FileSecurity sec = File.GetAccessControl (@"d:\test.txt");
AuthorizationRuleCollection rules = sec.GetAccessRules (true, true,typeof (NTAccount));
foreach (FileSystemAccessRule rule in rules)
{
  Console.WriteLine (rule.AccessControlType);         // Allow或Deny
  Console.WriteLine (rule.FileSystemRights);          // 例如FullControl
  Console.WriteLine (rule.IdentityReference.Value);   // 例如MyDomain/Joe
}

var sid = new SecurityIdentifier (WellKnownSidType.BuiltinUsersSid, null);
string usersAccount = sid.Translate (typeof (NTAccount)).ToString();

FileSystemAccessRule newRule = new FileSystemAccessRule
("Users", FileSystemRights.ExecuteFile, AccessControlType.Allow);
sec.AddAccessRule (newRule);
File.SetAccessControl (@"d:\test.txt", sec);
```

在后面的"特殊文件夹"将会举出另一个例子。

15.6.2 Directory类

静态的Directory类具有一组与File类相似的方法，用于检查目录是否存在（Exists）、移动目录（Move）、删除目录（Delete）、获取/设置创建时间或最后访问时间，以及获取/设置安全权限。此外，Directory类还包含以下静态方法：

```
string GetCurrentDirectory ();
void SetCurrentDirectory   (string path);

DirectoryInfo CreateDirectory       (string path);
DirectoryInfo GetParent             (string path);
string GetDirectoryRoot             (string path);

string[] GetLogicalDrives();

// 下面的方法都返回完整路径：

string[] GetFiles                   (string path);
string[] GetDirectories             (string path);
string[] GetFileSystemEntries       (string path);

IEnumerable<string> EnumerateFiles               (string path);
IEnumerable<string> EnumerateDirectories         (string path);
IEnumerable<string> EnumerateFileSystemEntries   (string path);
```

提示： 最后三个方法是Framework 4.0新增加的。它们可能比Get*方法效率更高，因为它们是延迟赋值的——在枚举序列时从文件系统取回数据。它们特别用于LINQ查询。

Enumerate*和Get*方法经过重载后可以接受searchPattern（字符串）和searchOption（枚举型）参数。如果将SearchOption.Search指定为AllSubDirectories，那么会执行子目录递归搜索。*FileSystemEntries方法组合了*Files和*Directories的结果：

下面的代码说明了如何在目录不存在时创建一个目录：

```
if (!Directory.Exists (@"d:\test"))
  Directory.CreateDirectory (@"d:\test");
```

15.6.3 FileInfo和DirectoryInfo

使用File和Directory的静态方法，我们可以方便地执行一个文件或目录操作。如果需要一次性调用多个方法，FileInfo和DirectoryInfo类支持一种简化这种调用的对象模型。

FileInfo以实例方法的形式支持大部分的File静态方法以及一些额外的属性，例如Extension、Length、IsReadOnly和Directory（返回一个DirectoryInfo对象）。例如：

```
FileInfo fi = new FileInfo (@"c:\temp\FileInfo.txt");
Console.WriteLine (fi.Exists);                // false

using (TextWriter w = fi.CreateText())
  w.Write ("Some text");

Console.WriteLine (fi.Exists);                // false
```

流与I/O

```
fi.Refresh();
Console.WriteLine (fi.Exists);               // true

Console.WriteLine (fi.Name);                         // FileInfo.txt
Console.WriteLine (fi.FullName);             // c:\temp\FileInfo.txt
Console.WriteLine (fi.DirectoryName);                // c:\temp
Console.WriteLine (fi.Directory.Name);       // temp
Console.WriteLine (fi.Extension);            // .txt
Console.WriteLine (fi.Length);               // 9

fi.Encrypt();
fi.Attributes ^= FileAttributes.Hidden; // (添加隐藏标识)
fi.IsReadOnly = true;

Console.WriteLine (fi.Attributes);   // ReadOnly、Archive、Hidden、Encrypted
Console.WriteLine (fi.CreationTime);

fi.MoveTo (@"c:\temp\FileInfoX.txt");

DirectoryInfo di = fi.Directory;
Console.WriteLine (di.Name);                 // temp
Console.WriteLine (di.FullName);             // c:\temp
Console.WriteLine (di.Parent.FullName);      // c:\
di.CreateSubdirectory ("SubFolder");
```

下面的代码说明如何使用DirectoryInfo列举文件及子目录：

```
DirectoryInfo di = new DirectoryInfo (@"e:\photos");

foreach (FileInfo fi in di.GetFiles ("*.jpg"))
  Console.WriteLine (fi.Name);

foreach (DirectoryInfo subDir in di.GetDirectories())
  Console.WriteLine (subDir.FullName);
```

15.6.4 Path

静态的Path类定义了一些处理路径和文件名的方法和字段。假设有这样一些设置代码：

```
string dir = @"c:\mydir";
string file = "myfile.txt";
string path = @"c:\mydir\myfile.txt";

Directory.SetCurrentDirectory (@"k:\demo");
```

我们可以用以下表达式来演示Path的方法和字段：

表达式	结果
Directory.GetCurrentDirectory()	k:\demo\
Path.IsPathRooted (file)	False
Path.IsPathRooted (path)	True
Path.GetPathRoot (path)	c:\
Path.GetDirectoryName (path)	c:\mydir

`Path.GetFileName (path)`	`myfile.txt`
`Path.GetFullPath (file)`	`k:\demo\myfile.txt`
`Path.Combine (dir, file)`	`c:\mydir\myfile.txt`
文件扩展名：	
`Path.HasExtension (file)`	`True`
`Path.GetExtension (file)`	`.txt`
`Path.GetFileNameWithoutExtension (file)`	`myfile`
`Path.ChangeExtension (file, ".log")`	`myfile.log`
操作符和字符：	
`Path.AltDirectorySeparatorChar`	`/`
`Path.PathSeparator`	`;`
`Path.VolumeSeparatorChar`	`:`
`Path.GetInvalidPathChars()`	0~31号字符和"<>|
`Path.GetInvalidFileNameChars()`	0~31号字符和 "<>|:*?\/
临时文件：	
`Path.GetTempPath()`	*<local user folder>*\Temp
`Path.GetRandomFileName()`	*d2dwuzjf.dnp*
`Path.GetTempFileName()`	*<local user folder>*\Temp*tmp14B.tmp*

Combine是非常有用的，它可用来组合目录和文件名或者两个目录，而不需要先检查名称后面是否有反斜杠。

GetFullPath可以将一个相对于当前目录的路径转换为一个绝对路径。它接受例如..\..\file.txt这样的值。

GetRandomFileName会返回一个完全唯一的8.3格式文件名，但不会真正创建文件。GetTempFileName会使用一个自增计数器生成一个临时文件名，这个计数器每隔65,000次重复一遍。然后，它会用这个名称在本地临时目录创建一个0字节的文件。

> 提示：当完成之后，我们必须删除GetTempFileName生成的文件；否则，它最终会抛出一个异常（在第65,000次调用GetTempFileName后）。如果出现这个问题，我们可以组合（Combine）GetTempPath和GetRandomFileName。但是，要小心用光用户的硬盘！

15.6.5 特殊文件夹

Path和Directory还缺少一个功能，那就是查找一些特殊文件夹，例如*My Documents*、*Program Files*、*Application Data*等。System.Environment类的GetFolderPath方法提供了这个查找功能：

```
string myDocPath = Environment.GetFolderPath
  (Environment.SpecialFolder.MyDocuments);
```

Environment.SpecialFolder是一个枚举类型，它的值包括Windows中的所有特殊目录：

AdminTools	CommonVideos	Personal
ApplicationData	Cookies	PrinterShortcuts
CDBurning	Desktop	ProgramFiles
CommonAdminTools	DesktopDirectory	ProgramFilesX86
CommonApplicationData	Favorites	Programs
CommonDesktopDirectory	Fonts	Recent
CommonDocuments	History	Resources
CommonMusic	nternetCache	SendTo
CommonOemLinks	LocalApplicationData	StartMenu
CommonPictures	LocalizedResources	Startup
CommonProgramFiles	MyComputer	System
CommonProgramFilesX86	MyDocuments	SystemX86
CommonPrograms	MyMusic	Templates
CommonStartMenu	MyPictures	UserProfile
CommonStartup	MyVideos	Windows
CommonTemplates	NetworkShortcuts	

提示：上面几乎列出了所有目录，唯一例外的.NET Framework目录可以用下面的方法获得：

```
System.Runtime.InteropServices.
RuntimeEnvironment.GetRuntimeDirectory()
```

这里有一个特殊值ApplicationData：这里可以存储一些设置，它们会随用户网络位置变化而迁移（如果网络域启动了漫游配置），以及保存非漫游数据（已登录用户的数据）的LocalApplicationData和由计算机上所有用户共享的CommonApplicationData。在这些文件夹中写入应用数据，最好使用Windows注册表。在这些文件夹中存储数据的标准协议是在应用名称下创建一个子目录：

```
string localAppDataPath = Path.Combine (
  Environment.GetFolderPath (Environment.SpecialFolder.ApplicationData),
    "MyCoolApplication");

if (!Directory.Exists (localAppDataPath))
  Directory.CreateDirectory (localAppDataPath);
```

提示：运行在最严格沙箱的程序（如Silverlight应用）不能访问这些文件夹。相反，它们要使用独立存储（Isolated Storage，参见本章最后一节），而Metro应用则使用WinRT库（参见第642页的"Windows Runtime的文件I/O"）。

使用CommonApplicationData有一个可怕的问题：如果用户使用管理员身份启动程序，那么程序会在CommonApplicationData下创建文件夹和文件，而用户将来可能用受限的Windows身份登录，从而没有足够权限修改这些文件。（如果在权限受限的帐号之间切换，也存在相似的问题。）在安装过程中创建所需要的文件夹（并给所有人分配权限），就可以解决这个问题。

此外，在下创建一个文件夹CommonApplicationData之后，如果马上（未写入任何文件）运行下面的代码，就可以保证“用户”群组中所有人都可以访问这个文件夹的内容：

```
public void AssignUsersFullControlToFolder (string path)
{
  try
  {
    var sec = Directory.GetAccessControl (path);
    if (UsersHaveFullControl (sec)) return;

    var rule = new FileSystemAccessRule (
      GetUsersAccount().ToString(),
      FileSystemRights.FullControl,
      InheritanceFlags.ContainerInherit | InheritanceFlags.ObjectInherit,
      PropagationFlags.None,
      AccessControlType.Allow);

      sec.AddAccessRule (rule);
      Directory.SetAccessControl (path, sec);
  }
  catch (UnauthorizedAccessException)
  {
    // 文件夹已经由另一个用户创建
  }
}

bool UsersHaveFullControl (FileSystemSecurity sec)
{
  var usersAccount = GetUsersAccount();
  var rules = sec.GetAccessRules (true, true, typeof (NTAccount))
                                  .OfType<FileSystemAccessRule>();
  return rules.Any (r =>
    r.FileSystemRights == FileSystemRights.FullControl &&
    r.AccessControlType == AccessControlType.Allow &&
    r.InheritanceFlags == (InheritanceFlags.ContainerInherit |
                           InheritanceFlags.ObjectInherit) &&
    r.IdentityReference == usersAccount);
}

NTAccount GetUsersAccount()
{
var sid = new SecurityIdentifier (WellKnownSidType.BuiltinUsersSid, null);
return (NTAccount)sid.Translate (typeof (NTAccount));
}
```

另一个写入配置文件与日志文件的地方是应用的基目录，它可以通过AppDomain.CurrentDomain.BaseDirectory获得。然而，不推荐使用这种方法，因为操作系统很可能拒绝应用在第一次安装之后（无管理员权限的情况下）在这个文件夹中写入内容。

15.6.6 查询卷信息

DriveInfo类可用来查询计算机的驱动器信息：

```
DriveInfo c = new DriveInfo ("C");          // 查询C:盘

long totalSize = c.TotalSize;               // 以字节为单位的大小
```

```
long freeBytes = c.TotalFreeSpace;                    // 忽略磁盘配额
long freeToMe = c.AvailableFreeSpace;                 // 计入磁盘配额

foreach (DriveInfo d in DriveInfo.GetDrives())        // 所有定义的驱动器
{
  Console.WriteLine (d.Name);                         // C:\
  Console.WriteLine (d.DriveType);                    // 固定磁盘
  Console.WriteLine (d.RootDirectory);                // C:\

  if (d.IsReady)     // 如果磁盘不可用，下面两个属性会抛出异常
  {
    Console.WriteLine (d.VolumeLabel);                // The Sea Drive
    Console.WriteLine (d.DriveFormat);                // NTFS
  }
}
```

静态的GetDrives方法会返回所有映射的驱动器，包括CD-ROM、内存卡和网络连接。DriveType是一个枚举类型，它包括以下值：

```
Unknown、NoRootDirectory、Removable、Fixed、Network、CDRom、Ram
```

15.6.7 捕获Filesystem事件

FileSystemWatcher类可用来监控一个目录（或者子目录）的活动。当有文件或子目录被创建、修改、重命名、删除以及属性变化时，FileSystemWatcher都会触发相应的事件。无论是用户还是进程执行这些操作，这些事件都会触发。

下面是一个示例：

```
static void Main() { Watch (@"c:\temp", "*.txt", true); }

static void Watch (string path, string filter, bool includeSubDirs)
{
  using (var watcher = new FileSystemWatcher (path, filter))
  {
    watcher.Created += FileCreatedChangedDeleted;
    watcher.Changed += FileCreatedChangedDeleted;
    watcher.Deleted += FileCreatedChangedDeleted;
    watcher.Renamed += FileRenamed;
    watcher.Error += FileError;

    watcher.IncludeSubdirectories = includeSubDirs;
    watcher.EnableRaisingEvents = true;

    Console.WriteLine ("Listening for events - press <enter> to end");
    Console.ReadLine();
  }
  // 清除FileSystemWatcher会停止触发事件。
}

static void FileCreatedChangedDeleted (object o, FileSystemEventArgs e)
{
  Console.WriteLine ("File {0} has been {1}", e.FullPath, e.ChangeType);
}

static void FileRenamed (object o, RenamedEventArgs e)
```

```
{
Console.WriteLine ("Renamed: {0}->{1}", e.OldFullPath, e.FullPath);
}

static void FileError (object o, ErrorEventArgs e)
{
  Console.WriteLine ("Error: " + e.GetException().Message);
}
```

警告： 因为FileSystemWatcher在一个独立线程上接收事件，所以事件处理代码中必须使用异常处理语句，防止错误使应用程序崩溃。见“异常处理”一节。

Error事件不会通知文件系统错误；相反，它表示的是FileSystemWatcher的事件缓冲区溢出，因为它已经被Changed、Created、Deleted或Renamed占用。我们可以通过InternalBufferSize属性修改缓冲区大小。

IncludeSubdirectories会递归执行。所以，如果IncludeSubdirectories为true，然后在C:\中创建一个FileSystemWatcher，那么在硬盘任何位置创建一个文件或目录都会触发它的事件。

警告： FileSystemWatcher有一个问题，它会在文件完全生成或更新之前打开并读取新创建或更新的文件。如果同时使用一些创建文件的软件，那么我们需要考虑采取一些策略来防止问题发生，例如创建一个扩展名未监控的文件，然后再将它重命名为完整格式。

15.7 Windows Runtime中的文件输入/输出

Metro应用都不能使用FileStream和Directory/File类。相反，Windows.Storage命名空间包含一些具有相同用途的WinRT类型，其中两个主要类是StorageFolder和StorageFile。

15.7.1 操作目录

StorageFolder类表示一个目录。调用StorageFolder的静态方法GetFolderFromPathAsync，指定文件夹的完整路径，就可以获得一个StorageFolder对象。然而，由于WinRT只允许访问特定位置的文件，所以更简单的方法是通过KnownFolders类获得一个StorageFolder对象，它包含多个静态属性，分别对应每一个（可能）允许访问的位置：

```
public static StorageFolder DocumentsLibrary { get; }
public static StorageFolder PicturesLibrary { get; }
public static StorageFolder MusicLibrary { get; }
public static StorageFolder VideosLibrary { get; }
```

提示： 声明程序包清单文件，可以进一步限制文件访问权限。具体地，Metro应用只能访问那些扩展名与所声明文件类型相匹配的文件。

此外，Package.Current.InstalledLocation会返回当前应用所在的StorageFolder（只有只读权限）。

KnownFolders包含一些访问可移除设备和主文件夹的属性。

StorageFolder包含一些常用属性（Name、Path、DateCreated、DateModified、Attributes等），一些删除/重命名文件夹的方法（DeleteAsync/RenameAsync），以及一些列举文件及子文件夹的方法（GetFilesAsync和GetFoldersAsync）。

从方法名称可以看出，这些方法都是异步方法，它们都会返回一个对象，使用AsTask扩展方法可以将它们转换为一个任务，或者直接等待。下面的代码获取一个目录，然后列出文档文件夹中的所有文件：

```
StorageFolder docsFolder = KnownFolders.DocumentsLibrary;
IReadOnlyList<StorageFile> files = await docsFolder.GetFilesAsync();
foreach (IStorageFile file in files)
  Debug.WriteLine (file.Name);
```

使用CreateFileQueryWithOptions方法，可以添加筛选条件，只查询带有特定扩展名的文件：

```
StorageFolder docsFolder = KnownFolders.DocumentsLibrary;
var queryOptions = new QueryOptions (CommonFileQuery.DefaultQuery,
                                     new[] { ".txt" });
var txtFiles = await docsFolder.CreateFileQueryWithOptions (queryOptions)
                               .GetFilesAsync();
foreach (StorageFile file in txtFiles)
  Debug.WriteLine (file.Name);
```

QueryOptions类还包含一些可以进一步控制查询的属性。例如，FolderDepth属性请求执行递归列举目录内容：

```
queryOptions.FolderDepth = FolderDepth.Deep;
```

15.7.2 操作文件

StorageFile是操作文件的基础类。使用静态方法StorageFile.GetFileFromPathAsync，可以使用完整路径获得一个文件实例；调用StorageFolder或IStorageFolder对象的GetFileAsync方法，则可以使用相对路径获得一个文件实例：

```
StorageFolder docsFolder = KnownFolders.DocumentsLibrary;
StorageFile file = await docsFolder.GetFileAsync ("foo.txt");
```

如果文件不存在，那么这时会抛出一个FileNotFoundException异常。

StorageFile拥有Name、Path等属性，以及各种操作文件的方法，如Move、Rename、Copy和Delete（都是Async）。CopyAsync方法会返回一个与新文件相对应的StorageFile。此外，还有一个CopyAndReplaceAsync，它接受一个目标StorageFile对象，而非目标名称和文件夹。

StorageFile还包含一些通过.NET流（OpenStreamForReadAsync和OpenStreamForWriteAsync）打开文件执行读/写操作的方法。例如，下面的语句将在文档文件夹中创建和写入一个文件*test.txt*：

```
StorageFolder docsFolder = KnownFolders.DocumentsLibrary;

StorageFile file = await docsFolder.CreateFileAsync
  ("test.txt", CreationCollisionOption.ReplaceExisting);

using (Stream stream = await file.OpenStreamForWriteAsync())
using (StreamWriter writer = new StreamWriter (stream))
  await writer.WriteLineAsync ("This is a test");
```

警告： 如果指定CreationCollisionOption.ReplaceExisting，并且文件已存在，那么它会自动给文件名附加一个数字，以保证文件名的唯一性。

下面的语句将读回文件：

```
StorageFolder docsFolder = KnownFolders.DocumentsLibrary;
StorageFile file = await docsFolder.GetFileAsync ("test.txt");

using (var stream = await file.OpenStreamForReadAsync ())
using (StreamReader reader = new StreamReader (stream))
  Debug.WriteLine (await reader.ReadToEndAsync());
```

15.7.3 Metro应用的独立存储

Metro应用还能够访问一些与其他应用隔离的私有文件夹，它们可用于存储应用的专属数据：

```
Windows.Storage.ApplicationData.Current.LocalFolder
Windows.Storage.ApplicationData.Current.RoamingFolder
Windows.Storage.ApplicationData.Current.TemporaryFolder
```

这里每一个静态属性都会返回一个StorageFolder对象，它们可以按照前面介绍的方式读/写和列举文件。

15.8 内存映射文件

内存映射文件是Framework 4.0新增加的。它们有两个主要特性：

- 文件数据的高效随机访问
- 在同一台计算机的不同进程之间共享内存

内存映射文件的类型位于System.IO.MemoryMappedFiles命名空间。在内部，它们是封装了支持内存映射文件的Win32 API。

15.8.1 内存映射文件与随机文件I/O

虽然常规的FileStream也支持随机文件I/O（通过设置流的Position属性实现），但是它在连续I/O方面进行了优化。一般原则大致是：

- FileStream的连续I/O速度要比内存映射文件快10倍。
- 内存映射文件的随机I/O速度要比FileStream快10倍。

修改FileStream的Position属性可能需要耗费几毫秒时间，并在循环中会进一步累加。FileStream不适用于多线程访问，因为它在读或写时位置会发生改变。

要创建一个内存映射文件，我们要：

1. 获取一个普通的FileStream。
2. 使用文件流实例化MemoryMappedFile。

3. 在内存映射文件对象上调用CreateViewAccessor。

最后一步可以得到一个MemoryMappedViewAccessor对象，它具有一些随机读写简单类型、结构和数组的方法（见“使用视图访问器”）。

下面的代码创建了一个百万字节的文件，然后使用内存映射文件API读取内容，并在位置500,000处写入一个字节：

```
File.WriteAllBytes ("long.bin", new byte [1000000]);

using (MemoryMappedFile mmf = MemoryMappedFile.CreateFromFile ("long.bin"))
using (MemoryMappedViewAccessor accessor = mmf.CreateViewAccessor())
{
  accessor.Write (500000, (byte) 77);
  Console.WriteLine (accessor.ReadByte (500000)); // 77
}
```

调用CreateFromFile时还可以指定一个映射名称和容量。指定一个非空映射名称允许将内存块共享给其他进程（见下一节）；指定一个容量会自动将文件大小扩大为该值。下面的代码创建一个1,000字节的文件：

```
using (var mmf = MemoryMappedFile.CreateFromFile
                 ("long.bin", FileMode.Create, null, 1000))
...
```

15.8.2 内存映射文件和共享内存

内存映射文件也可以作为在同一台计算机上不同进程间共享内存的一种手段。一个进程可以调用MemoryMappedFile.CreateNew创建一个共享内存块，而另一个进程则可以用相同的名称调用MemoryMappedFile.OpenExisting来共享同一个内存块。虽然它仍然是一个内存映射文件，但是已经完全脱离磁盘而进入内存中。

下面的代码创建了一个500字节的共享内存映射文件，然后在位置0写入整数12345：

```
using (MemoryMappedFile mmFile = MemoryMappedFile.CreateNew ("Demo", 500))
using (MemoryMappedViewAccessor accessor = mmFile.CreateViewAccessor())
{
  accessor.Write (0, 12345);
  Console.ReadLine();        // 保证共享内存块在用户按回车键之前都是可用的。
}
```

而下面的代码会打开同一个内存映射文件并读取整数：

```
// 这可以在一个独立的EXE中运行：
using (MemoryMappedFile mmFile = MemoryMappedFile.OpenExisting ("Demo"))
using (MemoryMappedViewAccessor accessor = mmFile.CreateViewAccessor())
  Console.WriteLine (accessor.ReadInt32 (0)); // 12345
```

15.8.3 使用视图访问器

在MemoryMappedFile中调用CreateViewAccessor可以得到一个视图访问器，它可以用来执行随机位置的读/写。

Read*/Write*方法可以接受数字类型、bool、char以及包含值类型元素或域的数组和结构体。引用类型（及包含引用类型的数组或结构体）是禁止使用的，因为它们无法映射到一个未托管的内存中。所以，如果想要写入一个字符串，我们必须将它编码到一个字节数组中：

```
byte[] data = Encoding.UTF8.GetBytes ("This is a test");
accessor.Write (0, data.Length);
accessor.WriteArray (4, data, 0, data.Length);
```

注意，我们先指定了长度。这意味着我们知道后面需要读取多少个字节：

```
byte[] data = new byte [accessor.ReadInt32 (0)];
accessor.ReadArray (4, data, 0, data.Length);
Console.WriteLine (Encoding.UTF8.GetString (data));          // 测试
```

下面是一个读/写结构体的示例：

```
struct Data { public int X, Y; }
...
var data = new Data { X = 123, Y = 456 };
accessor.Write (0, ref data);
accessor.Read (0, out data);
Console.WriteLine (data.X + " " + data.Y); // 123 456
```

我们还可以通过指针直接访问底层的未托管内存。下面继续前一个示例的代码：

```
unsafe
{
  byte* pointer = null;
  accessor.SafeMemoryMappedViewHandle.AcquirePointer (ref pointer);
  int* intPointer = (int*) pointer;
  Console.WriteLine (*intPointer);              // 123
}
```

指针在处理大结构时的优势是：它们可以直接处理原始数据，而不需要使用Read/Write在托管内存和未托管内存之间进行数据复制。我们将在第25章详细介绍这方面的内容。

15.9 隔离存储区

每一个.NET程序都可以访问该程序独有的本地存储区域，即独立存储（isolated storage）。如果程序无法访问标准文件系统，因此也无法在ApplicationData、LocalApplicationData、CommonApplicationData、MyDocuments等文件夹写入内容（参见15.6.5节的“特殊文件夹”），那么很适合使用独立存储。使用受限“互联网”权限的Silverlight应用和ClickOnce应用就属于这种情况。

独立存储有以下缺点：

- API很难用。
- 只能通过IsolatedStorageStream实现读/写，无法先获得一个文件或目录路径，然后执行普通的文件I/O。
- 主机存储（相当于CommonApplicationData）不允许操作系统权限受限的用户删除或重写其他用户所创建的文件（但是他们可以修改）。这实际上是一个bug。

在安全性方面，隔离存储区的作用更多的是阻止其他的应用程序进入，而不是阻止其中的应用程序出去！隔离存储区的数据受到严格保护，不会受到其他运行在最严格权限集（例如“互联网”域）之下的.NET应用程序的入侵。在其他情况中，如果另一个应用程序确实需要访问隔离存储区，也不会有很严格的安全措施阻止它。相对于CommonApplicationData而言，使用隔离存储区的优点是应用程序无法干扰其他应用程序，错误或意外行为不可能导致这些干扰。

在沙箱中运行的应用程序可以通过权限设置获得有限的隔离存储区配额。默认情况下，互联网和Silverlight应用程序在Framework 4.0中的配额是1MB。

提示： 从Framework 4.0开始，一个托管的UI应用程序（例如Silverlight）可以调用IsolatedStorageFile对象的IncreaseQuotaTo方法，要求获取增加隔离存储区配额的用户权限。

我们可以通过Quota属性查询当前的配额。

15.9.1 隔离类型

隔离存储区可以按程序和用户进行划分。因此，有三种基本划分方式：

按本地用户划分

一个用户、程序或计算机分配一个隔离存储区。

按漫游用户划分

一个用户或程序分配一个隔离存储区。

按主机划分

一个程序或计算机分配一个隔离存储区（由程序的所有用户共享）。

漫游用户区间的数据会跟随用户在网络中移动，但是需要合适的操作系统和域支持。如果得不到支持，那么它与按本地用户划分的方式类似。

到目前为止，我们只讨论了按程序划分的隔离存储区。隔离存储区会根据选择的模式，将程序看作是以下两种形式之一：

- 程序集
- 运行在特定应用程序上下文中的程序集

后者称为域隔离，它的使用频率高于程序集隔离。域隔离是按两个方面划分的：当前运行的程序集和启动它的原始可执行文件或Web应用程序。程序集隔离是按当前运行的程序集划分的，所以调用同一个程序集的不同应用程序会共享同一个存储区。

警告： 程序集和应用程序是根据它们的强名称标识的。如果没有强名称，那么转而使用该程序集的完整文件路径或URI。因此，如果移动或重命名一个弱命名的程序集，那么它的隔离存储区也会重置。

这样，总共就有6种隔离存储划分类型。表15-4对各种隔离方式进行比较。

表15-4：隔离存储区

类型	计算机	应用程序	程序集	用户	获取存储的方法
域用户（默认）	✓	✓	✓	✓	GetUserStoreForDomain
域漫游		✓	✓	✓	
域主机	✓	✓	✓		GetMachineStoreForDomain
程序集用户	✓		✓	✓	GetUserStoreForAssembly
程序集漫游			✓	✓	
程序集主机	✓		✓		GetMachineStoreForAssembly

这里不存在只按域进行隔离的方式。然而，如果想要在一个应用程序的所有程序集之间共享一个隔离存储区，那么只需要在其中一个程序集中提供一个公共方法，用它实例化和返回一个IsolatedStorageFileStream对象。如果指定一个IsolatedStorageFile对象，那么任何程序集都可以访问任意隔离存储区，因为隔离限制只影响创建，不影响后续使用。

类似地，只按主机进行隔离的方式也不存在。如果想要在多个应用程序中共享一个隔离存储区，那么变通方法是编写一个可供所有应用程序引用的通用程序集，然后在这个通用程序集上提供一个方法，用它创建和返回按程序隔离的IsolatedStorageFileStream。这个通用程序集必须采用强命名。

15.9.2 读写隔离存储区

隔离存储区使用一些与普通文件流很相似的流。要获得一个隔离存储流，我们首先需要通过调用IsolatedStorageFile的一个静态方法指定隔离类型，如表15-4所示。然后，使用这个类型、文件名和FileMode创建一个IsolatedStorageFileStream：

```
// IsolatedStorage类位于System.IO.IsolatedStorage

using (IsolatedStorageFile f = IsolatedStorageFile.GetMachineStoreForDomain())
using (var s = new IsolatedStorageFileStream ("hi.txt",FileMode.Create,f))
using (var writer = new StreamWriter (s))
  writer.WriteLine ("Hello, World");

// 读回文字：

using (IsolatedStorageFile f = IsolatedStorageFile.GetMachineStoreForDomain())
using (var s = new IsolatedStorageFileStream ("hi.txt", FileMode.Open, f))
using (var reader = new StreamReader (s))
  Console.WriteLine (reader.ReadToEnd());     // Hello, world
```

提示：IsolatedStorageFile的命名并不好，因为它实际上不是一个文件，而是文件的容器（实际上是一个目录）。

获取IsolatedStorageFile的一个更好（但复杂）的方法是调用IsolatedStorageFile.GetStore，传入正确组合的IsolatedStorageScope标记（如图15-6所示）：

```
var flags = IsolatedStorageScope.Machine
          | IsolatedStorageScope.Application
          | IsolatedStorageScope.Assembly;
```

```
using (IsolatedStorageFile f = IsolatedStorageFile.GetStore (flags,
   typeof (StrongName), typeof (StrongName)))
{
   ...
```

这种方法的优点是可以告诉GetStore使用何种依据来标记程序，而不是由它自行选择。最常见的情况是，使用程序中程序集的强名称（就像本例所采用的方法），因为强名称是唯一的，而且可以在不同版本中保持一致。

警告：由CLR自动选择依据的危险是它还需要考虑Authenticode签名（参见第18章）。这通常是不可取的，因为这意味着如果Authenticode发生变化，就会引起身份信息发生变化。特别地，如果一开始不使用Authenticode，而后来才决定使用，那么在独立存储的角度上看，CLR会认为应用是不同的，这就意味着用户会丢失不同版本的数据。

IsolatedStorageScope是一个枚举标记，必须正确组合它的成员值，才能够得到一个有效的存储。图15-6列出了所有有效的组合方式。注意，我们可以用它们访问漫游存储（它们与本地存储很像，但是能够通过Windows漫游配置实现“漫游”）。

	程序集	程序集和域
本地用户	程序集 | 用户	程序集 | 域 | 用户
漫游用户	程序集 | 用户 | 漫游	程序集 | 域 | 用户 | 漫游
机器	程序集 | 机器	程序集 | 域 | 机器

图15-6：有效的IsolatedStorageScope组合

下面的代码说明了如何向一个按程序集和漫游用户隔离的存储区写数据：

```
var flags = IsolatedStorageScope.Assembly
          | IsolatedStorageScope.User
          | IsolatedStorageScope.Roaming;

using (IsolatedStorageFile f = IsolatedStorageFile.GetStore (flags, null, null))
using  (var s = new IsolatedStorageFileStream ("a.txt", FileMode.Create, f))

using (var writer = new StreamWriter (s))
  writer.WriteLine ("Hello, World");
```

15.9.3 存储位置

以下是.NET写隔离存储文件的位置：

范围	位置
本地用户	[LocalApplicationData]\IsolatedStorage
漫游用户	[ApplicationData]\IsolatedStorage
主机	[CommonApplicationData]\IsolatedStorage

调用Environment.GetFolderPath方法就可以获得中括号内每一个文件夹的位置。下面是在Windows Vista中以上文件夹的默认位置：

范围	位置
本地用户	\Users\<user>\AppData\Local\IsolatedStorage
漫游用户	\Users\<user>\AppData\Roaming\IsolatedStorage
主机	\ProgramData\IsolatedStorage

在Windows XP上的结果是：

范围	位置
本地用户	*\Documents and Settings\<user>\Local Settings\Application Data\IsolatedStorage*
漫游用户	*\Documents and Settings\<user>\Application Data\IsolatedStorage*
主机	*\Documents and Settings\All Users\Application Data\IsolatedStorage*

这里只列出了基础文件夹；数据文件本身位于内部的子目录中，名称是对程序集名称进行散列得到的。这正是使用（也是不使用）隔离存储区的原因。一方面，它使隔离成为可能：一个具有严格权限的应用程序在与其他应用程序交互时，可能会被拒绝获取目录列表——尽管与其他程序一样拥有相同的文件系统权限。另一方面，它使外部应用程序无法执行管理任务。有时候，在记事本中编辑一个XML配置文件是很方便的（也很必要），这样应用程序才能够正确启动。隔离存储区则不支持这种方式。

15.9.4 列举隔离存储区

IsolatedStorageFile对象也具有一些列举存储区文件的方法：

```
using (IsolatedStorageFile f = IsolatedStorageFile.GetUserStoreForDomain())
{
  using (var s = new IsolatedStorageFileStream ("f1.x",FileMode.Create,f))
    s.WriteByte (123);

  using (var s = new IsolatedStorageFileStream ("f2.x",FileMode.Create,f))
    s.WriteByte (123);

foreach (string s in f.GetFileNames ("*.*"))
  Console.Write (s + " ");                    // f1.x f2.x
}
```

我们还可以创建和删除一些子目录和文件：

```
using (IsolatedStorageFile f = IsolatedStorageFile.GetUserStoreForDomain())
{
  f.CreateDirectory ("subfolder");

foreach (string s in f.GetDirectoryNames ("*.*"))
  Console.WriteLine (s);                      // subfolder

using (var s = new IsolatedStorageFileStream (@"subfolder\sub1.txt",FileMode.Create, f))
  s.WriteByte (100);
```

```
f.DeleteFile (@"subfolder\sub1.txt");
f.DeleteDirectory ("subfolder");
}
```

由于拥有了足够的权限，我们还可以列举当前用户创建的所有隔离存储区及主机存储区。这个功能可能违反程序保密性，但是不会违反用户保密性。下面是一个示例：

```
System.Collections.IEnumerator rator =
  IsolatedStorageFile.GetEnumerator (IsolatedStorageScope.User);

while (rator.MoveNext())
{
  var isf = (IsolatedStorageFile) rator.Current;

Console.WriteLine (isf.AssemblyIdentity);    // 强名称或者URI
Console.WriteLine (isf.CurrentSize);
Console.WriteLine (isf.Scope);               // 用户 + ...
}
```

GetEnumerator方法一般是不接受参数的（这会使它包含的类不支持foreach）。GetEnumerator可以接受以下三个值：

IsolatedStorageScope.User
: 列举属于当前用户的所有本地存储。

IsolatedStorageScope.User | IsolatedStorageScope.Roaming
: 列举属于当前用户的所有漫游存储。

IsolatedStorageScope.Machine
: 列举计算机上的所有主机存储。

一旦有了IsolatedStorageFile对象，我们就可以调用GetFiles和GetDirectories列出它的内容。

第16章 网络

Framework在System.Net.*命名空间中包含各种支持标准网络协议通信的类，例如HTTP、TCP/IP和FTP。下面是其中一些主要组件的小结：

- WebClient外观类：支持通过HTTP或FTP执行简单的下载/上传操作。
- WebRequest和WebResponse类：支持更多的客户端HTTP或FTP操作。
- HttpListener类：可用来编写HTTP服务器。
- SmtpClient类：支持通过SMTP创建和发送电子邮件。
- Dns类：支持域名和地址之间的转换。
- TcpClient、UdpClient、TcpListener和Socket类：支持传输层和网络层的直接访问。

Framework支持主要的Internet协议，但是它的功能不仅限于Internet连接，诸如TCP/IP等协议也可以广泛应用于局域网上。

本章介绍的大多数类型都位于System.Net和System.Net.Sockets命名空间中；然而，有一些例子也用到了System.IO的类型。

16.1 网络体系结构

图16-1说明了.NET网络连接类型以及它们所在的通信层。大多数类型都位于传输层或应用层。传输层定义了发送和接收字节的基础协议（TCP和UDP）；应用层则定义支持特定应用程序的上层协议，例如获取Web页（HTTP）、传输文件（FTP）、发送邮件（SMTP）和域名与IP地址转换（DNS）。

通常，在应用层编程是最方便的。然而，有一些原因要求我们必须直接在传输层上进行操作，例如当需要使用一种Framework不支持的应用程序协议（例如POP3）来接收邮件时。此外，当需要为某个特殊应用程序（例如对等客户端）发明一种自定义协议时，也是如此。

HTTP属于应用层协议，它专门用于扩展通用的通信。它的基本运行方式是“请给我这个URL的网页”，可以很好地理解为“返回使用这些参数调用这个方法的结果值。”HTTP具有丰富的特性，它们在多层次业务应用程序和面向服务的体系结构中是非常有用的，例如验证和加密协议、消息组块、可扩展头信息和Cookies，并且多个服务器应用程序可以共享一个端口和IP地址。因此，HTTP在

Framework中得到很好的支持，包括本章介绍的直接支持以及通过WCF、Web Services和ASP.NET等技术实现的更高级支持。

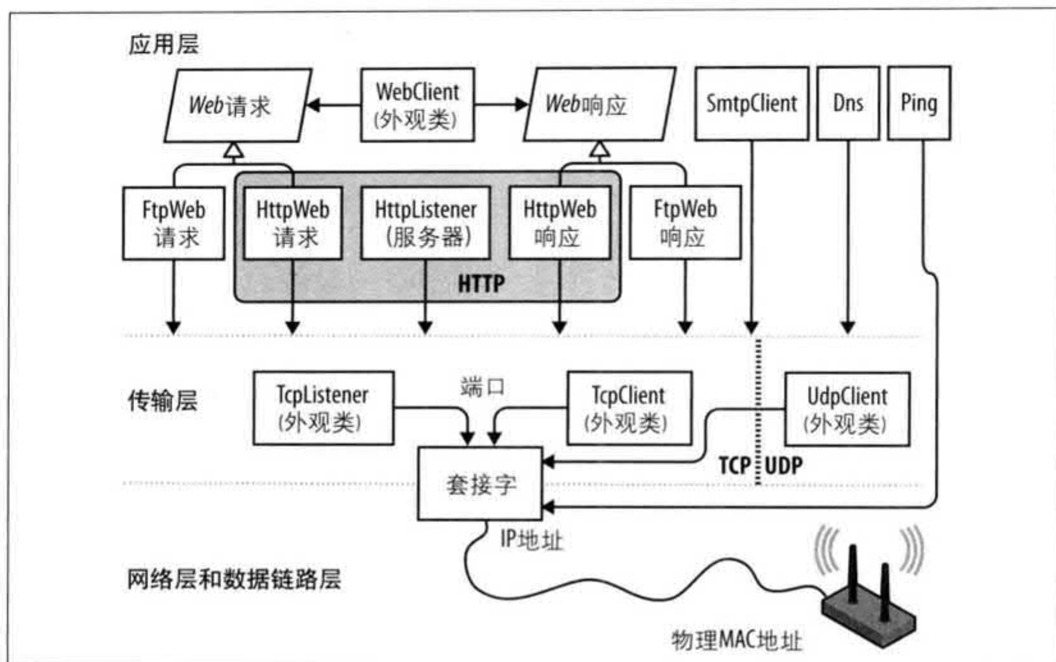


图16-1：网络体系结构

Framework提供FTP客户端支持，这是最常用的Internet文件发送和接收协议。服务器端支持是通过IIS或UNIX服务器软件等形式实现的。

正如前面内容所介绍的，网络是一个充满专业词汇缩略词的领域。表16-1是一个便捷的网络TLA（Three-Letter and more Acronym，字母缩写词）大全。

表16-1：网络TLA大全

缩写	全称	说明
DNS	Domain Name Service（域名服务）（例如，199.54.213.2）	域名（例如，*ebay.com*）和IP地址转换
FTP	File Transfer Protocol（文件传输协议）	Internet文件发送和接收的协议
HTTP	Hypertext TransferProtocol（超文本传输协议）	查询网页和运行Web服务
IIS	Internet InformationServices（Internet信息服务）	微软的Web服务器软件
IP	Internet Protocol（Internet协议）	TCP与UDP之下的网络层协议
LAN	Local Area Network（局域网）	大多数LAN使用TCP/IP等Internet协议
POP	Post Office Protocol（邮局协议）	查询Internet邮件
SMTP	Simple MailTransfer Protocol（简单邮件传输协议）	发送Internet邮件

表16-1：网络TLA大全（续）

缩写	全称	说明
TCP	Transmission and Control Protocol（传输和控制协议）	传输层Internet协议，大多更高级服务的基础
UDP	Universal Datagram Protocol（通用数据报协议）	低开销服务使用传输层Internet协议，例如VoIP
UNC	Universal Naming Convention（通用命名转换）	*computer\sharename\filename*
URI	Uniform Resource Identifier（统一资源标识符）	使用普遍的资源命名系统（例如，*http://www.amazon.com*或*mailto:joe@bloggs.org*）
URL	Uniform Resource Locator（统一资源定位符）	技术意义（逐渐停止使用）：URI子集；流行意义：URI简称

16.2 地址与端口

要实现通信，计算机或设备都需要一个地址。Internet使用了两套地址系统：

IPv4

这是目前的主流地址系统；IPv4地址有32位。当用字符串表示时，IPv4地址可以写为用点号分隔的4个十进制数（例如，101.102.103.104）。地址可能是全世界唯一的，也可能在一个特定子网中是唯一的（例如，企业网络）。

IPv6

这是更新的128位地址系统。这些地址用字符串表示为用冒号分隔的十六进制数（例如，[3EA0:FFFF:198A:E4A3:4FF2:54f-A:41BC:8D31]）。.NET Framework中要求地址加上方括号。

System.Net命名空间的IPAddress类是采用其中一种协议的地址。它有一个构造函数可以接收字节数组，以及一个静态的Parse方法接收正确格式的字符串：

```
IPAddress a1 = new IPAddress (new byte[] { 101, 102, 103, 104 });
IPAddress a2 = IPAddress.Parse ("101.102.103.104");
Console.WriteLine (a1.Equals (a2));                     // True
Console.WriteLine (a1.AddressFamily);                   // InterNetwork
IPAddress a3 = IPAddress.Parse
  ("[3EA0:FFFF:198A:E4A3:4FF2:54fA:41BC:8D31]");
Console.WriteLine (a3.AddressFamily);                   // InterNetworkV6
```

TCP和UDP协议将每一个IP地址划分为65,535个端口，从而允许一台计算机在一个地址上运行多个应用程序，每一个应用程序使用一个端口。许多应用程序都分配有标准端口，例如，HTTP使用端口80；SMTP使用端口25。

提示： 从49152到65535的TCP和UDP端口是官方保留的，它们只用于测试和小规模部署。

IP地址和端口组合在.NET Framework中是使用IPEndPoint类表示的：

```
IPAddress a = IPAddress.Parse ("101.102.103.104");
IPEndPoint ep = new IPEndPoint (a, 222);                    // 端口222
Console.WriteLine (ep.ToString());                           // 101.102.103.104:222
```

提示：防火墙可以阻挡端口。在许多企业环境中，事实上只有少数端口是开放的，通常情况下，只开放端口80（不加密HTTP）和端口443（安全HTTP）。

16.3 URI

URI是一个具有特殊格式的字符串，它描述了一个Internet或LAN的资源，例如网页、文件或电子邮件地址。例如，*http://www.ietf.org*、*ftp://myisp/doc.txt*和*mailto:joe@bloggs.com*。正确的格式是由Internet工程任务组（Internet Engineering Task Force，*http://www.ietf.org/*）定义的。

URI一般分成三个元素：协议（scheme）、权限（authority）和路径（path）。System命名空间的Uri类正是采用这种划分方式，为每一种元素提供对应的属性，如图16-2所示。

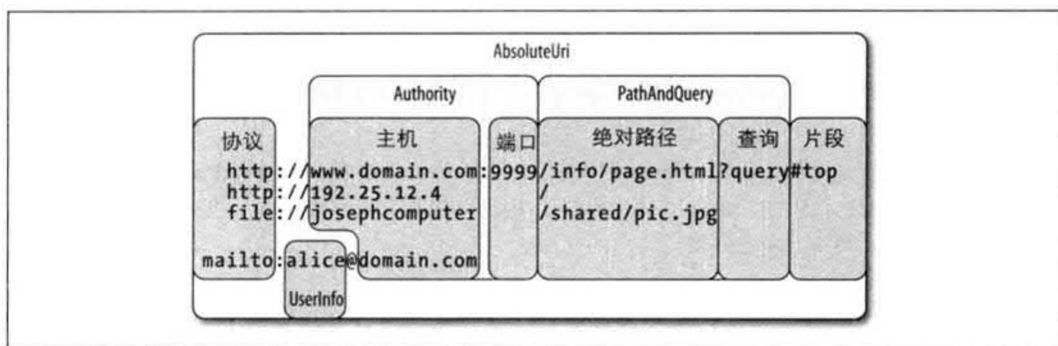


图16-2：URI属性

提示：Uri类适合用来验证URI字符串的格式或将URI分割成相应的组成部分。另外，可以将URI作为一个简单的字符串进行处理，大多数网络连接方法都有接收Uri对象或字符串的重载方法。

在构造函数中传入以下字符串之一，就可以创建一个Uri对象：

- URI字符串，例如*http://www.ebay.com*或*file://janespc/sharedpics/dolphin.jpg*。
- 硬盘中一个文件的绝对路径，例如*c:\myfiles\data.xls*。
- LAN中一个文件的UNC路径，例如\\janespc\sharedpics\dolphin.jpg。

文件和UNC路径会自动转换为URI：添加协议“file:”，反斜杠会转换为斜杠。Uri的构造函数在创建Uri之前也会对传入的字符串执行一些基本的清理操作，包括将协议和主机名转换为小写、删除默认端口号和空端口号。如果传入一个不带协议的URI字符串，例如“*www.test.com*”，那么会抛出一个UriFormatException异常。

Uri有一个IsLoopback属性，它表示Uri是否引用本地主机（IP地址为127.0.0.1）；以及一个IsFile

属性，它表示Uri是否引用一个本地或UNC（IsUnc）路径。如果IsFile返回true，LocalPath属性会返回一个符合本地操作系统习惯的AbsolutePath（带反斜杠），然后可以用它来调用File.Open。

Uri的实例有一些只读属性。要修改一个Uri，我们需要实例化一个UriBuilder对象，这是一个可写属性，它可以通过Uri属性转换为Uri。

Uri也具有一些比较和截取路径的方法：

```
Uri info = new Uri ("http://www.domain.com:80/info/");
Uri page = new Uri ("http://www.domain.com/info/page.html");

Console.WriteLine (info.Host);      // www.domain.com
Console.WriteLine (info.Port);      // 80
Console.WriteLine (page.Port);      // 80（Uri知道默认的HTTP端口）

Console.WriteLine (info.IsBaseOf (page));     // True
Uri relative = info.MakeRelativeUri (page);
Console.WriteLine (relative.IsAbsoluteUri);   // False
Console.WriteLine (relative.ToString());      // page.html
```

相对于Uri，如这个例子中的*page.html*，在调用除IsAbsoluteUri和ToString()以外的属性和方法时，都会抛出一个异常。可以按照以下方式实例化一个相对Uri：

```
Uri u = new Uri ("page.html", UriKind.Relative);
```

警告：URI后面的斜杠是很重要的，服务器会根据它来决定是否处理路径组成部分。

例如，假设URI为*http://www.albahari.com/nutshell/*，那么HTTP Web服务器会查找网站Web文件夹的*nutshell*子目录，然后返回默认文档（通常是*index.html*）。

如果没有结尾的斜杠，Web服务器则会直接在网站根文件夹查找一个名为*nutshell*的文件（没有扩展名），这通常不是我们想要的。如果文件不存在，大多数Web服务器将认为用户输入错误，然后返回301 Permanent Redirect错误，表示客户端应该尝试在结尾加上斜杠。默认情况下，.NET HTTP客户端采用与Web浏览器相同的方式响应301错误，使用建议的URI再尝试一次。这意味着，如果忽略末尾本应该添加的斜杠，那么请求仍然有效，但是会经过一个不必要的额外回程。

Uri类也具有一些静态的便利方法，如EscapeUriString()方法将ASCII值大于127的所有字符转换成十六进制，从而将一个字符串转换成一个有效的URL；CheckHostName()和CheckSchemeName()方法接受一个字符串，然后检查它的指定属性的语法是否正确（虽然它们不会确定主机或URI是否存在）。

16.4 客户端类

WebRequest和WebResponse是管理HTTP和FTP客户端活动及“file:”协议的通用基类。它们封装了这些协议共用的“请求/响应”模型：客户端发起请求，然后等待服务器的响应。

WebClient是一个便利的门面（façade）类，它负责调用WebRequest和WebResponse，可以节省很多编码。WebClient支持字符串、字节数组、文件或流；而WebRequest和WebResponse只支持流。但是，WebClient也不是万能的，因为它也不支持某些特性（如cookie）。

HttpClient是另一个基于WebRequest和WebResponse的类（更准确说是基于HttpWebRequest和

HttpWebResponse），并且是Framework 4.5新引入的类。WebClient主要作为请求/响应类之上薄薄的一层，而HttpClient则增加了更多的功能，能够处理基于HTTP的Web API、基于REST的服务和自定义验证模式。

WebClient和HttpClient都支持以字符串或字节数组方式处理简单的文件下载/上传操作。它们都拥有一些异步方法，但是只有WebClient支持进度报告。

提示： WinRT应用程序不能使用WebClient，它必须使用WebRequest/WebResponse或HttpClient（用于HTTP连接）。

16.4.1 WebClient

下面是WebClient的使用步骤：

1. 实例化一个WebClient对象。
2. 设置Proxy属性值。
3. 在需要验证时设置Credentials属性值。
4. 使用相应的URI调用Download*XXX*或Upload*XXX*方法。

它的下载方法有：

```
public void DownloadFile     (string address, string fileName);
public string DownloadString (string address);
public byte[] DownloadData   (string address);
public Stream OpenRead       (string address);
```

每一个方法都重载方法，它们接受使用Uri对象代替字符串地址的参数。上传方法也类似；它们返回包含服务器响应的值：

```
public byte[] UploadFile  (string address, string fileName);
public byte[] UploadFile  (string address, string method, string fileName);
public string UploadString(string address, string data);
public string UploadString(string address, string method, string data);
public byte[] UploadData  (string address, byte[] data);
public byte[] UploadData  (string address, string method, byte[] data);
public byte[] UploadValues(string address, NameValueCollection data);
public byte[] UploadValues(string address, string method,
                                     NameValueCollection data);
public Stream OpenWrite   (string address);
public Stream OpenWrite   (string address, string method);
```

UploadValues方法可用于以POST方法参数提交一个HTTP表单的值。WebClient还包含一个BaseAddress属性，可用于为所有地址添加一个字符串前缀，如*http://www.mysite.com/data/*。

下面的例子演示如何下载本书的代码示例页面，并且保存到当前文件夹中，然后在默认Web浏览器中显示：

```
WebClient wc = new WebClient();
wc.Proxy = null;
wc.DownloadFile ("http://www.albahari.com/nutshell/code.aspx", "code.htm");
```

```
System.Diagnostics.Process.Start ("code.htm");
```

提示：WebClient被动实现了IDisposable——因为它继承了Component（这使它能够显示在Visual Studio的Designer组件托盘）。然而，它的Dispose方法在运行时并没有执行太多实际操作，所以不需要清理WebClient的实例。

从Framework 4.5开始，WebClient提供了长任务方法的异步版本（第14章），它们会返回可以等待的任务：

```
await wc.DownloadFileTaskAsync ("http://oreilly.com", "webpage.htm");
```

（这些方法使用“TaskAsync”后缀，不同于使用“Async”后缀的EAP旧异步方法。）但是，新方法不支持取消操作和进度报告的标准“TAP”模式。相反，在处理延续时，必须调用WebClient对象的CancelAsync方法；而处理进度报告时，则需要处理DownloadProgressChanged/UploadProgressChanged事件。下面的例子下载一个网页并显示进度报告，如果下载过程超过5秒钟，则取消下载：

```
var wc = new WebClient();

wc.DownloadProgressChanged += (sender, args) =>
  Console.WriteLine (args.ProgressPercentage + "% complete");

Task.Delay (5000).ContinueWith (ant => wc.CancelAsync());

await wc.DownloadFileTaskAsync ("http://oreilly.com", "webpage.htm");
```

提示：当请求取消时，程序就会抛出一个WebException异常，它的Status属性是WebExceptionStatus.RequestCanceled。（由于历史原因，这里不会抛出OperationCanceledException异常。）

捕捉与进度相关的事件，将它们提交到激活的同步上下文，所以它们的处理器不需要使用Dispatcher.BeginInvoke，就可以更新UI控件。

警告：如果需要使用取消操作或进度报告，那么要避免使用同一个WebClient对象依次执行多个操作，因为这样会形成竞争条件。

16.4.2 WebRequest和WebResponse

WebRequest和WebResponse比WebClient复杂，但是更加灵活。下面是开始使用的步骤：

1. 使用一个URI调用WebRequest.Create，创建一个Web请求实例。
2. 设置Proxy属性。
3. 如果需要验证身份，则设置Credentials属性。

如果要上传数据，则：

4. 调用请求对象的GetRequestStream，然后在流中写入数据。如果需要处理响应，则转到第5步。

如果要下载数据，则：

5. 调用请求对象的GetResponse，创建一个Web响应实例。

6. 调用响应对象的GetResponseStream，然后（可以使用StreamReader）从流中读取数据。

下面的例子演示了如何下载和显示一个代码示例网页（重写了前一个例子）：

```
WebRequest req = WebRequest.Create
               ("http://www.albahari.com/nutshell/code.html");
req.Proxy = null;
using (WebResponse res = req.GetResponse())
using (Stream rs = res.GetResponseStream())
using (FileStream fs = File.Create ("code.html"))
  rs.CopyTo (fs);
```

下面是用异步方式实现相同的例子：

```
WebRequest req = WebRequest.Create
               ("http://www.albahari.com/nutshell/code.html");
req.Proxy = null;
using (WebResponse res = await req.GetResponseAsync())
using (Stream rs = res.GetResponseStream())
using (FileStream fs = File.Create ("code.html"))
  await rs.CopyToAsync (fs);
```

警告： Web响应对象包含一个ContentLength属性，它表示服务器报告的响应流长度（以字节为单位）。这个值位于响应头，可能不存在或者不正确。特别地，如果HTTP服务器选择使用“块”模式分割大响应，那么ContentLength值通常为－1。动态生成的网页也有相同的结果。

静态方法Create会创建一个WebRequest类型的子类实例，如HttpWebRequest或FtpWebRequest。它选择的子类取决于URI的前缀，具体如表16-2所示。

表16-2：URI前缀与Web请求类型

前缀	Web请求类型
http:或https:	HttpWebRequest
ftp:	FtpWebRequest
file:	FileWebRequest

提示： 将Web请求对象转换为具体的类型（HttpWebRequest或FtpWebRequest），就可以访问特定协议的特性。

此外，调用WebRequest.RegisterPrefix，可以注册自定义的前缀。这需要使用一个前缀及一个工厂对象，它有一个Create方法，可以实例化一个Web请求对象。

“https:”协议是指通过安全套接层（Secure Sockets Layer，SSL）实现的安全（加密）HTTP。WebClient和WebRequest都会在遇到这种前缀时激活SSL（参见本章后面第16.5节“HTTP访问”中第16.5.6节“SSL”）。“file:”协议会将请求转发到一个FileStream对象。其目的是确定一个与读

取URI一致的协议，它可能是一个网页、FTP站点或文件路径。

WebRequest包含一个Timeout属性，其单位为毫秒。如果出现超时，那么程序就会抛出一个WebException异常，其中包含一个Status属性：WebExceptionStatus.Timeout。HTTP的默认超时时间为100秒，而FTP的超时时间为无限。

WebRequest对象不能回收并用于处理多个请求——每一个实例只适用于一个作业。

16.4.3 HttpClient

HttpClient是Framework 4.5新引入的类，它在HttpWebRequest和HttpWebResponse之上提供了另一层封装。它的设计是为了支持越来越多的Web API和REST服务，在处理比获取网页等更复杂的协议时实现比WebClient更佳的体验。具体地：

- 一个HttpClient就可以支持并发请求。如果要使用WebClient处理并发请求，则需要为每一个并发线程创建一个新实例，这时需要自定义请求头、cookies和验证模式，因此会比较麻烦。
- HttpClient可用于编写和插入自定义消息处理器。这样可以创建单元测试桩函数，以及创建自定义管道（用于记录日志、压缩、加密等）。调用WebClient的单元测试代码则很难编写。
- HttpClient包含丰富且可扩展的请求头与内容类型系统。

提示： HttpClient不能完全代替WebClient，因为它不支持进度报告。WebClient也有一个优点，它支持FTP、*file://*和自定义URI模式，它也适用于所有Framework版本。

使用HttpClient的最简单方法是创建一个实例，然后使用URI调用其中一个Get*方法：

```
string html = await new HttpClient().GetStringAsync ("http://linqpad.net");
```

（另外还有GetByteArrayAsync和GetStreamAsync。）HttpClient的所有I/O密集型方法都是异步的（它们没有同步实现版本）。

与WebClient不同，想要获得最佳性能的HttpClient，必须重用相同的实例（否则诸如DNS解析等操作会出现不必要的重复执行）。HttpClient允许并发操作，所以下面的语句是合法的，它们可以同时下载两个网页：

```
var client = new HttpClient();
var task1 = client.GetStringAsync ("http://www.linqpad.net");
var task2 = client.GetStringAsync ("http://www.albahari.com");
Console.WriteLine (await task1);
Console.WriteLine (await task2);
```

HttpClient包含一个Timeout属性和一个BaseAddress属性，它会为每一个请求添加一个URI前缀。HttpClient在一定程度上就是一层实现：通常使用的大部分属性都定义在另一个类中，即HttpClientHandler。要访问这个类，可以先创建一个实例，然后将这个实例传递给HttpClient的构造方法：

```
var handler = new HttpClientHandler { UseProxy = false };
var client = new HttpClient (handler);
...
```

这个例子在处理器中禁用了代理支持。此外，还有其他一些属性可用于控制cookies、自动重定向、验证等（后续章节及第16.5节“HTTP访问”将介绍这些属性）。

1. GetAsync与响应消息

GetStringAsync、GetByteArrayAsync和GetStreamAsync方法是更常用的GetAsync方法的快捷方法。GetAsync方法会返回一个响应消息：

```
var client = new HttpClient();
// GetAsync方法也接受一个CancellationToken.
HttpResponseMessage response = await client.GetAsync ("http://...");
response.EnsureSuccessStatusCode();
string html = await response.Content.ReadAsStringAsync();
```

HttpResponseMessage包含一些访问请求头（参见第16.5节“HTTP访问”）和HTTP StatusCode的属性。与WebClient不同，除非显式调用EnsureSuccessStatusCode，否则返回不成功状态（如404，资源未找到）不会抛出异常。然而，通信或DNS错误会抛出异常（参见第16.4.6节“异常处理”）

HttpResponseMessage包含一个CopyToAsync方法，它可以将数据写到另一个流中，适用于将输入写到一个文件中：

```
using (var fileStream = File.Create ("linqpad.html"))
await response.Content.CopyToAsync (fileStream);
```

GetAsync是与HTTP的4种动作相关的4个方法之一（其他方法是PostAsync、PutAsync和DeleteAsync）。后面第16.5.3节“上传表单数据”将演示PostAsync。

2. SendAsync与请求消息

前面介绍的4个方法都是SendAsync的快捷方法，这是访问所有数据的唯一低层方法。要使用这个类，首先需要创建一个HttpRequestMessage：

```
var client = new HttpClient();
var request = new HttpRequestMessage (HttpMethod.Get, "http://...");
HttpResponseMessage response = await client.SendAsync (request);
response.EnsureSuccessStatusCode();
...
```

创建一个HttpRequestMessage对象，意味着可以自定义请求的属性，如请求头（第16.5.1节“请求头”）和内容本身，它们可用于上传数据。

3. 上传数据与HttpContent

在创建一个HttpRequestMessage对象之后，设置它的Content属性，就可以上传内容。这个属性的类型是抽象类HttpContent。Framework包含以下具体的子类，分别对应不同类型的内容（也可以编写自定义子类）：

- ByteArrayContent
- StreamContent
- FormUrlEncodedContent（参见第16.5.3节“上传表单数据”）
- StreamContent

例如：

```
var client = new HttpClient (new HttpClientHandler { UseProxy = false });
var request = new HttpRequestMessage (
  HttpMethod.Post, "http://www.albahari.com/EchoPost.aspx");
request.Content = new StringContent ("This is a test");
HttpResponseMessage response = await client.SendAsync (request);
response.EnsureSuccessStatusCode();
Console.WriteLine (await response.Content.ReadAsStringAsync());
```

4. HttpMessageHandler

前面介绍过，大多数自定义请求的属性都不是在HttpClient中定义，而是在HttpClientHandler中定义。后者实际上是抽象类HttpMessageHandler的子类，其定义如下：

```
public abstract class HttpMessageHandler : IDisposable
{
  protected internal abstract Task<HttpResponseMessage> SendAsync
    (HttpRequestMessage request, CancellationToken cancellationToken);

  public void Dispose();
  protected virtual void Dispose (bool disposing);
}
```

SendAsync方法是在HttpClient的SendAsync方法中调用。

HttpMessageHandler非常容易继承，同时也提供了HttpClient的扩展点。

5. 单元测试与桩处理器

创建HttpMessageHandler的子类，就可以创建一个帮助进行单元测试的桩处理器：

```
class MockHandler : HttpMessageHandler
{
  Func <HttpRequestMessage, HttpResponseMessage> _responseGenerator;

  public MockHandler
    (Func <HttpRequestMessage, HttpResponseMessage> responseGenerator)
  {
    _responseGenerator = responseGenerator;
  }

  protected override Task <HttpResponseMessage> SendAsync
    (HttpRequestMessage request, CancellationToken cancellationToken)
  {
    cancellationToken.ThrowIfCancellationRequested();
    var response = _responseGenerator (request);
    response.RequestMessage = request;
    return Task.FromResult (response);
  }
}
```

它的构造方法可以接受一个函数，它规定了如何使用桩处理器生成请求的响应。这是最通用的方法，因为同一个处理器可以测试多个请求。

通过使用Task.FromResult，就可以将程序替换为同步实现。在响应生成器上返回一个Task

<HttpResponseMessage>，就可以保持异步性，但是这对于预期运行时间较短的桩处理器而言意义不大。下面的例子说明了如何使用所创建的桩处理器：

```
var mocker = new MockHandler (request =>
  new HttpResponseMessage (HttpStatusCode.OK)
{
  Content = new StringContent ("You asked for " + request.RequestUri)
});

var client = new HttpClient (mocker);
var response = await client.GetAsync ("http://www.linqpad.net");
string result = await response.Content.ReadAsStringAsync();
Assert.AreEqual ("You asked for http://www.linqpad.net/", result);
```

（Assert.AreEqual方法通常也会出现在一些单元测试框架中，如NUnit。）

6. 使用实现处理器链

创建DelegatingHandler的子类，就可以创建一个调用其他消息处理器的消息处理器（形成处理器链）。这种方法适用于实现自定义身份验证、压缩和加密协议。下面的程序演示了一个简单的日志处理器：

```
class LoggingHandler : DelegatingHandler
{
  public LoggingHandler (HttpMessageHandler nextHandler)
  {
     InnerHandler = nextHandler;
  }

  protected async override Task <HttpResponseMessage> SendAsync
    (HttpRequestMessage request, CancellationToken cancellationToken)
  {
    Console.WriteLine ("Requesting: " + request.RequestUri);
    var response = await base.SendAsync (request, cancellationToken);
    Console.WriteLine ("Got response: " + response.StatusCode);
    return response;
  }
}
```

注意，这里在重写SendAsync时保持了异步性。在重载返回任务的方法时添加async修饰符是绝对合法的——如这个例子所示。

相对于将信息直接写到控制台，更好的方法是给构造方法传入某种日志记录对象。最好是接受一对Action<T>代理，然后让它们记录请求和响应对象的日志信息。

16.4.4 代理

代理服务器（proxy server）是一个中间服务器，负责转发HTTP和FTP请求。有时候，一些单位会建立一个代理服务器，作为员工访问互联网的唯一方式——主要是为了简化安全性。代理本身拥有地址，并且可能需要执行身份验证，所以只有特定的局域网用户可以访问互联网。

创建一个WebClient或WebRequest对象，就可以使用WebProxy对象通过代理服务器转发请求：

```
// 使用代理的IP地址和端口创建一个WebProxy。
```

```
// 如果代理需要用户名/密码，则可以选择设置Credentials。

WebProxy p = new WebProxy ("192.178.10.49", 808);
p.Credentials = new NetworkCredential ("username", "password");
// 或者：
p.Credentials = new NetworkCredential ("username", "password", "domain");

WebClient wc = new WebClient();
wc.Proxy = p;
  ...

// WebRequest对象也有相同的过程：
WebRequest req = WebRequest.Create ("...");
req.Proxy = p;
```

如果要使用HttpClient访问代理，那么首先要创建一个HttpClientHandler，设置它的Proxy属性，然后将它传递给HttpClient的构造方法：

```
WebProxy p = new WebProxy ("192.178.10.49", 808);
p.Credentials = new NetworkCredential ("username", "password", "domain");

var handler = new HttpClientHandler { Proxy = p };
var client = new HttpClient (handler);
...
```

警告： 如果已知不存在代理，那么可以在WebClient和WebRequest对象上将Proxy属性设置为null。否则，Framework可能会尝试自动检查代理设置，这会给请求增加30秒延迟。如果Web请求执行速度过慢，那么很可能就是这个原因造成的！

HttpClientHandler还有一个UseProxy属性，将它设置为false，就可以将Proxy属性置空，从而禁止自动检测。

如果在创建NetworkCredential时提供一个域，那么就会使用基于Windows的身份验证协议。如果想要使用当前已验证的Windows用户，则可以在代理的Credentials属性上设置静态的CredentialCache.DefaultNetworkCredentials值。

作为重复设置Proxy的替代方法，我们可以按照以下方式设置全局默认值：

```
WebRequest.DefaultWebProxy = myWebProxy;
```

或者：

```
WebRequest.DefaultWebProxy = null;
```

这里设置的值适用于整个应用程序域的生命周期（除非其他代码修改了这个值）。

16.4.5 身份验证

创建一个NetworkCredential对象，将它设置到WebClient或WebRequest的Credentials属性上，就可以向HTTP或FTP站点提供用户名和密码：

```
WebClient wc = new WebClient();
wc.Proxy = null;
wc.BaseAddress = "ftp://ftp.albahari.com";
```

```
// 验证身份，然后上传或下载文件到FTP服务器。
// 同样的方法也适用于HTTP和HTTPS。

string username = "nutshell";
string password = "oreilly";
wc.Credentials = new NetworkCredential (username, password);

wc.DownloadFile ("guestbook.txt", "guestbook.txt");

string data = "Hello from " + Environment.UserName + "!\r\n";
File.AppendAllText ("guestbook.txt", data);

wc.UploadFile ("guestbook.txt", "guestbook.txt");
```

HttpClient通过HttpClientHandler提供相同的属性：

```
var handler = new HttpClientHandler();
handler.Credentials = new NetworkCredential (username, password);
var client = new HttpClient (handler);
...
```

这种方法适用于基于对话框的身份验证协议，如Basic和Digest，并且可以通过AuthenticationManager类实现扩展。此外，（如果在创建NetworkCredential对象时加入一个域名）它还支持Windows NTLM和Kerberos。

提示： 使用基于表单的身份验证时，不需要设置Credentials。我们将另外讨论基于表单的身份验证（参见第16.5.5节“表单验证”）。

身份验证最终由一个WebRequest子类型处理（这里是FtpWebRequest），它会自动协商一个兼容协议。在HTTP中，还有另一种选择：例如在来自Microsoft Exchange服务器Web邮件页面的初始响应中，那么它可能还包含以下请求头信息：

```
HTTP/1.1 401 Unauthorized
Content-Length: 83
Content-Type: text/html
Server: Microsoft-IIS/6.0
WWW-Authenticate: Negotiate
WWW-Authenticate: NTLM
WWW-Authenticate: Basic realm="exchange.somedomain.com"
X-Powered-By: ASP.NET
Date: Sat, 05 Aug 2006 12:37:23 GMT
```

401代码信号表示需要授权；“WWW-Authenticate”请求头表示身份验证协议已识别。然而，如果在WebClient或WebRequest对象中配置正确的用户名和密码，那么这条消息就会隐藏，因为Framework会自动选择兼容的身份验证协议，然后重新提交添加请求头的原始请求。例如：

```
Authorization: Negotiate TlRMTVNTUAAABAAAt5II2gjACDArAAACAwACACgAAAAQ
ATmKAAAAD0lVDRdPUksHUq9VUA==
```

这种机制具有透明性，但是它会在每一个请求上产生额外回程时间。将PreAuthenticate属性设置为true，就可以避免同一个URI的后续请求出现额外回程时间。这个属性定义在WebRequest类中，并且只适用于HttpWebRequest。WebClient则不支持这个特性。

1. CredentialCache

使用`CredentialCache`对象，可以强制使用特定的身份验证协议。身份缓存包含一个或多个`NetworkCredential`对象，其中每一个对象都对应于一种特定协议和URI前缀。例如，在登录Exchange Server时，我们可能希望避免使用Basic协议，因为这种方式采用明文传输密码：

```
CredentialCache cache = new CredentialCache();
Uri prefix = new Uri ("http://exchange.somedomain.com");
cache.Add (prefix, "Digest", new NetworkCredential ("joe", "passwd"));
cache.Add (prefix, "Negotiate", new NetworkCredential ("joe", "passwd"));

WebClient wc = new WebClient();
wc.Credentials = cache;
...
```

验证协议由字符串表示。有效的协议字符串包括：

```
Basic, Digest, NTLM, Kerberos, Negotiate
```

在这个例子中，`WebClient`将选择Negotiate，因为服务器没有在它的验证头信息中表示它支持Digest。Negotiate是一种Windows协议，它可以是Kerberos或NTLM，这取决于服务器的功能。

静态属性`CredentialCache.DefaultNetworkCredentials`可用于将当前通过验证的Windows用户添加到身份缓存中，而不需要再指定密码：

```
cache.Add (prefix, "Negotiate", CredentialCache.DefaultNetworkCredentials);
```

2. 使用HTTPClient根据头信息实现验证

如果使用`HttpClient`，则可以使用另一种方法实现验证，即直接设置验证头信息：

```
var client = new HttpClient();
client.DefaultRequestHeaders.Authorization =
  new AuthenticationHeaderValue ("Basic",
    Convert.ToBase64String (Encoding.UTF8.GetBytes ("username:password")));
...
```

这个方法同样适用于自定义身份验证系统，如OAuth。后面将深入讨论头信息。

16.4.6 异常处理

`WebRequest`、`WebResponse`、`WebClient`及其流都会在遇到网络或协议错误时抛出一个`WebException`异常。`HttpClient`也有相同行为，但是它将`WebException`封装在一个`HttpRequestException`中。使用`WebException`的`Status`属性，就可以确定具体的错误类型；它会返回一个枚举值`WebExceptionStatus`，其中包含以下成员：

CacheEntryNotFound	PipelineFailure	SecureChannelFailure
ConnectFailure	ProtocolError	SendFailure
ConnectionClosed	ProxyNameResolutionFailure	ServerProtocolViolation
KeepAliveFailure	ReceiveFailure	Success
MessageLengthLimitExceeded	RequestCanceled	Timeout

网络

NameResolutionFailure	RequestProhibitedByCachePolicy	TrustFailure
Pending	RequestProhibitedByProxy	UnknownError

无效域名会导致发生NameResolutionFailure；网络中断会导致发生ConnectFailure；而请求时间超过WebRequest.Timeout毫秒数则会导致发生Timeout。

“页面不存在”、“页面已永久删除”和“未登录”等错误属于HTTP或FTP协议的特有错误，所以它们都属于ProtocolError状态。在HttpClient中，除非在响应对象中调用EnsureSuccessStatusCode，否则这些错误不会显示。在执行这些操作之前，查询StatusCode属性，就可以获得特定的状态代码：

```
var client = new HttpClient();
var response = await client.GetAsync ("http://linqpad.net/foo");
HttpStatusCode responseStatus = response.StatusCode;
```

在使用WebClient和WebRequest/WebResponse时，必须先捕捉WebException异常，然后再：

1. 将WebException的Response属性强制转换为HttpWebResponse或FtpWebResponse。
2. 检查响应对象的Status属性（HttpStatusCode或FtpStatusCode枚举值）和/或StatusDescription属性（字符串）。

例如：

```
WebClient wc = new WebClient();
try
{
  wc.Proxy = null;
  string s = wc.DownloadString ("http://www.albahari.com/notthere");
}
catch (WebException ex)
{
  if (ex.Status == WebExceptionStatus.NameResolutionFailure)
    Console.WriteLine ("Bad domain name");
  else if (ex.Status == WebExceptionStatus.ProtocolError)
  {
    HttpWebResponse response = (HttpWebResponse) ex.Response;
    Console.WriteLine (response.StatusDescription);  // "Not Found"
    if (response.StatusCode == HttpStatusCode.NotFound)
      Console.WriteLine ("Not there!");                 // "Not there!"
  }
  else throw;
}
```

提示： 如果想要获得3个数字的状态码，如401或404，则可以直接将HttpStatusCode或FtpStatusCode枚举值转换为整数。

默认情况下，由于WebClient和WebRequest会自动产生重定向响应，所以一定不会出现重定向错误。将AllowAutoRedirect设置为false，就可以在WebRequest对象中关闭这个行为。

重定向错误包括301（页面已永久删除）、302（发现/重定向）和307（临时重定向）。

如果使用WebClient或WebRequest类的方式不正确，那么就会抛出一个异常，这个异常一般是

InvalidOperationException或ProtocolViolationException，而不是WebException。

16.5 HTTP访问

这一节将介绍WebClient、HttpWebRequest/HttpWebResponse和HttpClient的HTTP请求和响应特性。

16.5.1 请求头

WebClient、WebRequest和HttpClient都可以添加自定义HTTP请求头，以及在响应中列举请求头信息。请求头只是一些键/值对，其中包含相应的元数据，如消息内容类型或服务器软件。下面的例子演示了如何在请求中添加自定义头信息，然后在HttpClient的响应消息中列举所有请求头信息：

```
WebClient wc = new WebClient();
wc.Proxy = null;
wc.Headers.Add ("CustomHeader", "JustPlaying/1.0");
wc.DownloadString ("http://www.oreilly.com");

foreach (string name in wc.ResponseHeaders.Keys)
  Console.WriteLine (name + "=" + wc.ResponseHeaders [name]);

Age=51
X-Cache=HIT from oregano.bp
X-Cache-Lookup=HIT from oregano.bp:3128
Connection=keep-alive
Accept-Ranges=bytes
Content-Length=95433
Content-Type=text/html
...
```

相反，HttpClient包含了一些强类型集合，其中包含与标准HTTP头信息相对应的属性。DefaultRequestHeaders属性包含适用于每一个请求的头信息：

```
var client = new HttpClient (handler);

client.DefaultRequestHeaders.UserAgent.Add (
  new ProductInfoHeaderValue ("VisualStudio", "2012"));

client.DefaultRequestHeaders.Add ("CustomHeader", "VisualStudio/2012");
```

而HttpRequestMessage类的Headers属性则包含请求特有的头信息。

16.5.2 查询字符串

查询字符串只是通过问号（?）附加到URI后面的字符串，它可用于向服务器发送简单的数据。使用下面的语法，就可以在查询字符串中指定多个键/值对：

```
?key1=value1&key2=value2&key3=value3...
```

WebClient包含一个字典风格的属性，它可以简化查询字符串的操作。下面的语句会在谷歌中搜索单词“WebClient”，然后以法语显示结果页面：

```
WebClient wc = new WebClient();
wc.Proxy = null;
```

网络

```
wc.QueryString.Add ("q", "WebClient");    // 搜索"WebClient"
wc.QueryString.Add ("hl", "fr");          // 以法语显示页面
wc.DownloadFile ("http://www.google.com/search", "results.html");
System.Diagnostics.Process.Start ("results.html");
```

如果要使用WebRequest或HttpClient实现相同效果，那么必须手工赋给请求URI正确格式的字符串：

```
string requestURI = "http://www.google.com/search?q=WebClient&hl=fr";
```

如果查询中包含符号或空格，那么必须使用Uri的EscapeDataString方法才能创建合法的URI：

```
string search = Uri.EscapeDataString ("(WebClient OR HttpClient)");
string language = Uri.EscapeDataString ("fr");
string requestURI = "http://www.google.com/search?q=" + search +
                    "&hl=" + language;
```

最终的URI为：

```
http://www.google.com/search?q=(WebClient%20OR%20HttpClient)&hl=fr
```

（EscapeDataString与EscapeUriString类似，唯一不同的是前者进行了特殊字符的编码，如&和=，否则它们会破坏查询字符串。）

提示： Microsoft的Web Protection库（*http://wpl.codeplex.com*）提供了另一种编码/解码方法，它能够防止跨站脚本攻击。

16.5.3 上传表单数据

WebClient的UploadValues方法可以以HTML表单的方式提交数据：

```
WebClient wc = new WebClient();
wc.Proxy = null;

var data = new System.Collections.Specialized.NameValueCollection();
data.Add ("Name", "Joe Albahari");
data.Add ("Company", "O'Reilly");

byte[] result = wc.UploadValues ("http://www.albahari.com/EchoPost.aspx",
                                 "POST", data);

Console.WriteLine (Encoding.UTF8.GetString (result));
```

NameValueCollection中的键（如searchtextbox和searchMode）与HTML表单的输入框相对应。

使用WebRequest上传表单数据的操作更为复杂。（如果需要使用cookies等特性，则必须采用这种方法。）下面是具体的操作过程：

1. 将请求的ContentType设置为"application/x-www-form-urlencoded"，将它的方法设置为"POST"。
2. 创建一个包含上传数据的字符串，并且将其编码为：

   ```
   name1=value1&name2=value2&name3=value3...
   ```

3. 使用Encoding.UTF8.GetBytes将字符串转换为字节数组。

4. 将Web请求的ContentLength属性设置为字节数组的长度。

5. 调用Web请求的GetRequestStream，然后写入数据数组。

6. 调用GetResponse，读取服务器的响应。

使用WebRequest实现前一个例子，结果是：

```
var req = WebRequest.Create ("http://www.albahari.com/EchoPost.aspx");
req.Proxy = null;
req.Method = "POST";
req.ContentType = "application/x-www-form-urlencoded";

string reqString = "Name=Joe+Albahari&Company=O'Reilly";
byte[] reqData = Encoding.UTF8.GetBytes (reqString);
req.ContentLength = reqData.Length;

using (Stream reqStream = req.GetRequestStream())
  reqStream.Write (reqData, 0, reqData.Length);

using (WebResponse res = req.GetResponse())
using (Stream resSteam = res.GetResponseStream())
using (StreamReader sr = new StreamReader (resSteam))
  Console.WriteLine (sr.ReadToEnd());
```

如果使用HttpClient，则要创建和生成FormUrlEncodedContent对象，然后再将它传递到PostAsync方法，或者设置到请求的Content属性上。

```
string uri = "http://www.albahari.com/EchoPost.aspx";
var client = new HttpClient();
var dict = new Dictionary<string,string>
{
  { "Name", "Joe Albahari" },
  { "Company", "O'Reilly" }
};
var values = new FormUrlEncodedContent (dict);
var response = await client.PostAsync (uri, values);
response.EnsureSuccessStatusCode();
Console.WriteLine (await response.Content.ReadAsStringAsync());
```


16.5.4 Cookies

Cookie是一种名称/值字符串对，它是HTTP服务器通过响应头发送到客户端的。Web浏览器客户端通常会记住cookie，然后在终止之前，后续请求都会将它们重复发送给服务器（相同地址）。Cookie使服务器知道它是否正在连接之前连接过的相同客户端，从而不需要在URI重复添加复杂的查询字符串。

默认情况下，HttpWebRequest会忽略从服务器接收的任意cookie。为了接收cookie，必须创建一个CookieContainer对象，然后将它分配到WebRequest。然后，就可以列举响应中接收到的cookie：

```
var cc = new CookieContainer();

var request = (HttpWebRequest) WebRequest.Create ("http://www.google.com");
request.Proxy = null;
request.CookieContainer = cc;
using (var response = (HttpWebResponse) request.GetResponse())
{
```

```
  foreach (Cookie c in response.Cookies)
  {
    Console.WriteLine (" Name: " + c.Name);
    Console.WriteLine (" Value: " + c.Value);
    Console.WriteLine (" Path: " + c.Path);
    Console.WriteLine (" Domain: " + c.Domain);
  }
  // 读取响应流...
}
```

```
Name: PREF
Value: ID=6b10df1da493a9c4:TM=1179025486:LM=1179025486:S=EJCZri0aWEHlk4tt
Path: /
Domain: .google.com
```

如果使用HttpClient实现相同效果，则需要创建一个HttpClientHandler实例：

```
var cc = new CookieContainer();
var handler = new HttpClientHandler();
handler.CookieContainer = cc;
var client = new HttpClient (handler);
...
```

WebClient门面类不支持cookie。

如果需要在将来的请求上接收到的cookie，则只需要给每一个新的WebRequest对象设置相同的CookieContainer对象，或者在HttpClient中使用相同对象发送请求。CookieContainer是可序列化类，所以它可以写到磁盘中——参见第17章。此外，可以先使用一个新的CookieContainer，然后再按照以下方式手动添加cookie：

```
Cookie c = new Cookie ("PREF",
                       "ID=6b10df1da493a9c4:TM=1179...",
                       "/",
                       ".google.com");
freshCookieContainer.Add (c);
```

第3个和第4个参数表示发起者的路径和域名。客户端上的CookieContainer可以保存来自不同位置的cookie；WebRequest只发送和域名与服务器相匹配的cookie。

16.5.5 表单验证

上一节介绍了如何使用NetworkCredentials对象实现其中一些类型验证，如Basic或NTLM（在Web浏览器中弹出一个对话框）。然而，大多数需要验证的网站都会使用某种基于表单的方法。在各种图形美化的HTML表单中，用户在文本框中输入用户名和密码，单击按钮提交数据，然后在成功验证之后接收到cookie。这个cookie属于所访问网页的私有数据。通过使用WebRequest或HttpClient，就可以实现前两节介绍的所有特性。

实现表单验证的典型网站都包含以下HTML代码：

```
<form action="http://www.somesite.com/login" method="post">
  <input type="text" id="user" name="username">
  <input type="password" id="pass" name="password">
  <button type="submit" id="login-btn">Log In</button>
</form>
```

下面的例子演示了如何使用WebRequest/WebResponse登录网站：

```
string loginUri = "http://www.somesite.com/login";
string username = "username";   // （用户名）
string password = "password";   // （密码）
string reqString = "username=" + username + "&password=" + password;
byte[] requestData = Encoding.UTF8.GetBytes (reqString);

CookieContainer cc = new CookieContainer();
var request = (HttpWebRequest)WebRequest.Create (loginUri);
request.Proxy = null;
request.CookieContainer = cc;
request.Method = "POST";

request.ContentType = "application/x-www-form-urlencoded";
  request.ContentLength = requestData.Length;

using (Stream s = request.GetRequestStream())
s.Write (requestData, 0, requestData.Length);

using (var response = (HttpWebResponse) request.GetResponse())
  foreach (Cookie c in response.Cookies)
    Console.WriteLine (c.Name + " = " + c.Value);

// 我们现在已经成功登录。只要给后续的WebRequest对象添加cc，就可以保持已验证用户的状态
```

使用HttpClient实现的方法如下：

```
string loginUri = "http://www.somesite.com/login";
string username = "username";
string password = "password";

CookieContainer cc = new CookieContainer();
var handler = new HttpClientHandler { CookieContainer = cc };

var request = new HttpRequestMessage (HttpMethod.Post, loginUri);
request.Content = new FormUrlEncodedContent (new Dictionary<string,string>
{
  { "username", username },
  { "password", password }
});

var client = new HttpClient (handler);
var response = await client.SendAsync (request);
response.EnsureSuccessStatusCode();
...
```

16.5.6 SSL

如果指定前缀，那么WebClient、HttpClient和WebRequest都会自动使用SSL。唯一复杂的问题与设计不佳的X.509证书有关。如果服务器的网站证书由于某种原因失效（例如，它是测试证书），那么通信时会抛出异常。为了解决这个问题，需要在静态类ServicePointManager上附加一个自定义证书验证器：

```
using System.Net;
using System.Net.Security;
using System.Security.Cryptography.X509Certificates;
```

网络

```
...
static void ConfigureSSL()
{
  ServicePointManager.ServerCertificateValidationCallback = CertChecker;
}
```

ServerCertificateValidationCallback是一个代理。如果它返回true，那么证书就是可以接受的：

```
static bool CertChecker (object sender, X509Certificate certificate,
                         X509Chain chain, SslPolicyErrors errors)
{
  // 如果证书满足要求，则返回true
  ...
}
```

16.6 编写HTTP服务器

我们可以使用HttpListener类编写自定义HTTP服务器。下面是一个监听端口51111的简单服务器，它会等待一个客户端请求，然后返回一行回复。

警告： HttpListener不支持Windows XP之前的操作系统。

```
static void Main()
{
  new System.Threading.Thread (Listen).Start();      // 以并行方式运行服务器
  Thread.Sleep (500);                                // 等待半秒钟

  WebClient wc = new WebClient();                    // 发起一个客户端请求
  Console.WriteLine (wc.DownloadString
    ("http://localhost:51111/MyApp/Request.txt"));
}

static void Listen()
{
  HttpListener listener = new HttpListener();
  listener.Prefixes.Add ("http://localhost:51111/MyApp/");// 监听端口51111
  listener.Start();

  // 等待一个客户端请求
  HttpListenerContext context = listener.GetContext();

  // 返回对请求的回复
  string msg = "You asked for: " + context.Request.RawUrl;
  context.Response.ContentLength64 = Encoding.UTF8.GetByteCount (msg);
  context.Response.StatusCode = (int) HttpStatusCode.OK;

  using (Stream s = context.Response.OutputStream)
  using (StreamWriter writer = new StreamWriter (s))
    writer.Write (msg);

  listener.Stop();
}
```

输出：You asked for: /MyApp/Request.txt

HttpListener内部并不使用.NET Socket对象；相反，它是调用Windows HTTP Server API。这个API是在Windows XP及以上平台支持的，支持计算机中多个应用程序监听相同的IP地址和端口，前提是每一个应用程序都注册不同的地址前缀。在我们的例子中，我们注册了前缀*http://localhost/myapp*，所以其他应用程序可以自由监听同一个IP和端口的其他前缀，如*http://localhost/anotherapp*。这是很有价值的，因为在企业防火墙中开放新端口可能需要经过一些规章流程。

调用GetContext时，HttpListener会等待下一个客户端请求，这个方法会返回一个对象，其中包含Request和Response属性。每一个属性都与WebRequest和WebResponse对象类似，但是它们属于服务器。例如，我们可以读写请求和响应对象的头信息和Cookie，使用的方法与客户端非常类似。

我们可以基于预期客户端类型选择如何支持HTTP协议特性。至少，每一个请求都需要设置内容长度和状态码。

下面是一个非常简单的网页服务器，它最多支持50个并发请求：

```
using System;
using System.IO;
using System.Net;
using System.Text;
using System.Threading;

class WebServer
{
  HttpListener _listener;
  string _baseFolder;       // 你的网页文件夹

  public WebServer (string uriPrefix, string baseFolder)
  {
    System.Threading.ThreadPool.SetMaxThreads (50, 1000);
    System.Threading.ThreadPool.SetMinThreads (50, 50);
    _listener = new HttpListener();
    _listener.Prefixes.Add (uriPrefix);
    _baseFolder = baseFolder;
  }

  public void Start()       // 在独立的线程上运行，方法与我们前面使用的一样
  {
    _listener.Start();
    while (true)
      try
      {
        HttpListenerContext request = _listener.GetContext();
        ThreadPool.QueueUserWorkItem (ProcessRequest, request);
      }
      catch (HttpListenerException) { break; }            // Listener stopped.
      catch (InvalidOperationException) { break; }        // Listener stopped.
  }

  public void Stop() { _listener.Stop(); }

  void ProcessRequest (object listenerContext)
  {
    try
    {
      var context = (HttpListenerContext) listenerContext;
```

网络

```
        string filename = Path.GetFileName (context.Request.RawUrl);
        string path = Path.Combine (_baseFolder, filename);
        byte[] msg;
        if (!File.Exists (path))
        {
          context.Response.StatusCode = (int) HttpStatusCode.NotFound;
          msg = Encoding.UTF8.GetBytes ("Sorry, that page does not exist");
        }
        else
        {
          context.Response.StatusCode = (int) HttpStatusCode.OK;
          msg = File.ReadAllBytes (path);
        }
        context.Response.ContentLength64 = msg.Length;
        using (Stream s = context.Response.OutputStream)
          s.Write (msg, 0, msg.Length);
      }
      catch (Exception ex) { Console.WriteLine ("Request error: " + ex); }
    }
  }
```

下面是启动程序的Main方法：

```
static void Main()
{
  // 监控默认端口（80），发布e:\mydocs\webroot中的文件：
  var server = new WebServer ("http://localhost/", @"e:\mydocs\webroot");

  // 在并行线程上启动服务器：
  new System.Threading.Thread (server.Start).Start();

  Console.WriteLine ("Server running... press Enter to stop");
  Console.ReadLine();
  server.Stop();
}
```

可以使用任何一个Web浏览器作为客户端进行测试；这里的URI是*http://localhost/*加上网站名称。

警告： 如果其他软件占用了同一个端口，那么HttpListener不会启动（除非这个软件也使用Windows HTTP Server API）。可能监听端口80的应用程序包括Web服务器或点对点程序，如Skype。

使用异步函数可以实现服务器的可扩展性和提高运行效率。然而，从UI线程开始都可能会影响可扩展性，因为在每一个请求中，执行过程都会在每一次等待之后返回UI线程。这种过载是毫无意义的，因为我们并没有共享的状态，所以在UI中，最好去掉UI线程，或者像这样：

```
Task.Run (Start);
```

或者在调用GetContextAsync之后调用ConfigureAwait(false)。

注意，即使这些方法已经是异步的，但是我们仍然在Task.Run中调用ProcessRequestAsync。这样可以允许调用者马上能够处理另一个请求，而不需要同步等待方法执行结束（一直到第一个await）。

16.7 使用FTP

对于简单的FTP上传和下载操作，我们可以使用WebClient按照前面的方式实现：

```
WebClient wc = new WebClient();
wc.Proxy = null;
wc.Credentials = new NetworkCredential ("nutshell", "oreilly");
wc.BaseAddress = "ftp://ftp.albahari.com";
wc.UploadString ("tempfile.txt", "hello!");
Console.WriteLine (wc.DownloadString ("tempfile.txt")); // hello!
```

然而，FTP不仅仅有文件上传和下载操作。这个协议还包含一系列的命令或方法，并且被定义为属于WebRequestMethods.Ftp的字符串常量：

AppendFile	ListDirectory	Rename
DeleteFile	ListDirectoryDetails	UploadFile
DownloadFile	MakeDirectory	UploadFileWithUniqueName
GetDateTimestamp	PrintWorkingDirectory	
GetFileSize	RemoveDirectory	

要运行这些命令，我们要将字符串常量赋值给Web请求的Method属性，然后调用GetResponse()。下面是获取目录列表的例子：

```
var req = (FtpWebRequest) WebRequest.Create ("ftp://ftp.albahari.com");
req.Proxy = null;
req.Credentials = new NetworkCredential ("nutshell", "oreilly");
req.Method = WebRequestMethods.Ftp.ListDirectory;

using (WebResponse resp = req.GetResponse())
using (StreamReader reader = new StreamReader (resp.GetResponseStream()))
  Console.WriteLine (reader.ReadToEnd());

结果为：
.
..
guestbook.txt
tempfile.txt
test.doc
```

为了获取目录列表，我们需要读取响应流才能得到结果。然而，其他大多数命令是不需要这个步骤的。例如，要获取GetFileSize命令的结果，我们只需要查询响应的ContentLength属性：

```
var req = (FtpWebRequest) WebRequest.Create (
                          "ftp://ftp.albahari.com/tempfile.txt");
req.Proxy = null;
req.Credentials = new NetworkCredential ("nutshell", "oreilly");

req.Method = WebRequestMethods.Ftp.GetFileSize;

using (WebResponse resp = req.GetResponse())
  Console.WriteLine (resp.ContentLength);     // 6
```

GetDateTimestamp命令的使用方法与此类似，只是需要查询响应的LastModified属性。这要求将响

应强制转换为FtpWebResponse：

```
...
req.Method = WebRequestMethods.Ftp.GetDateTimestamp;

using (var resp = (FtpWebResponse) req.GetResponse() )
  Console.WriteLine (resp.LastModified);
```

要使用Rename命令，我们必须给请求的RenameTo属性指定新的文件名（不带目录前缀）。例如，将*incoming*目录的文件*tempfile.txt*重命名为*deleteme.txt*：

```
var req = (FtpWebRequest) WebRequest.Create (
"ftp://ftp.albahari.com/tempfile.txt");
req.Proxy = null;
req.Credentials = new NetworkCredential ("nutshell", "oreilly");

req.Method = WebRequestMethods.Ftp.Rename;
req.RenameTo = "deleteme.txt";

req.GetResponse().Close();   // 执行重命名操作
```

下面是删除文件的方法：

```
var req = (FtpWebRequest) WebRequest.Create ("ftp://ftp.albahari.com/deleteme.txt");
req.Proxy = null;
req.Credentials = new NetworkCredential ("nutshell", "oreilly");

req.Method = WebRequestMethods.Ftp.DeleteFile;

req.GetResponse().Close();            // 执行删除操作
```

提示：在这些例子中，我们一般使用异常处理块来捕捉网络和协议错误。下面是一个典型的catch块：

```
catch (WebException ex)
{
  if (ex.Status == WebExceptionStatus.ProtocolError)
  {
    // 获取错误的详细信息：
    var response = (FtpWebResponse) ex.Response;
    FtpStatusCode errorCode = response.StatusCode;
    string errorMessage = response.StatusDescription;
    ...
  }
  ...
}
```

16.8 使用DNS

静态的Dns类封装了DNS（Domain Name Service，域名服务），它可以执行原始IP地址（如：66.135.192.87）和人性化的域名（如：*ebay.com*）之间的转换操作。

GetHostAddresses方法可以将域名转换为IP地址（或地址）：

```
foreach (IPAddress a in Dns.GetHostAddresses ("albahari.com"))
  Console.WriteLine (a.ToString());  // 208.43.7.176
```

GetHostEntry方法则执行相反操作，将地址转换为域名：

```
IPHostEntry entry = Dns.GetHostEntry ("208.43.7.176");
Console.WriteLine (entry.HostName);                    // si-eios.com
```

GetHostEntry方法还接受一个IPAddress对象，所以我们可以用一个字节数组来表示IP地址：

```
IPAddress address = new IPAddress (new byte[] { 208, 43, 7, 176 });
IPHostEntry entry = Dns.GetHostEntry (address);
Console.WriteLine (entry.HostName);                    // si-eios.com
```

在使用WebRequest或TcpClient等类时，域名会自动解析为IP地址。然而，如果想要在应用程序的生命周期内向同一个地址发送多个网络请求，有时候需要先使用DNS将域名显式地转换为IP地址，然后再直接使用得到的IP地址进行通信，从而提高运行性能。这样就能够避免重复解析同一个域名，有助于（使用TcpClient、UdpClient或Socket）处理传输层协议。

DNS类还提供了基于任务的可等待异步方法：

```
foreach (IPAddress a in await Dns.GetHostAddressesAsync ("albahari.com"))
Console.WriteLine (a.ToString());
```

16.9 通过SmtpClient发送邮件

System.Net.Mail命名空间的SmtpClient类可用来通过普遍使用的简单邮件传输协议（Simple Mail Transfer Protocol，SMTP）发送邮件消息。要发送一条简单的文本消息，我们需要实例化SmtpClient，将它的Host属性设置为SMTP服务器地址，然后调用Send：

```
SmtpClient client = new SmtpClient();
client.Host = "mail.myisp.net";
client.Send ("from@adomain.com", "to@adomain.com", "subject", "body");
```

为了防止垃圾邮件，Internet中大多数SMTP服务器都只接受来自ISP订阅者的连接，所以我们需要使用适合当前连接的SMTP地址才能成功发送邮件。

MailMessage对象支持更多的选项，包括添加附件：

```
SmtpClient client = new SmtpClient();
client.Host = "mail.myisp.net";
MailMessage mm = new MailMessage();

mm.Sender = new MailAddress ("kay@domain.com", "Kay");
mm.From = new MailAddress ("kay@domain.com", "Kay");
mm.To.Add (new MailAddress ("bob@domain.com", "Bob"));
mm.CC.Add (new MailAddress ("dan@domain.com", "Dan"));
mm.Subject = "Hello!";
mm.Body = "Hi there. Here's the photo!";
mm.IsBodyHtml = false;
mm.Priority = MailPriority.High;

Attachment a = new Attachment ("photo.jpg",System.Net.Mime.MediaTypeNames.Image.Jpeg);
mm.Attachments.Add (a);
client.Send (mm);
```

SmtpClient可以为需要执行身份验证的服务器指定Credentials，如果支持EnableSsl，也可以将

TCP Port修改为非默认值。通过修改DeliveryMethod属性，我们可以使用SmtpClient代替IIS发送邮件消息，或者直接将消息写到指定目录下的一个*.eml*文件中：

```
SmtpClient client = new SmtpClient();
client.DeliveryMethod = SmtpDeliveryMethod.SpecifiedPickupDirectory;
client.PickupDirectoryLocation = @"c:\mail";
```

16.10 使用TCP

TCP和UDP是大多数Internet（与局域网）服务所依赖的传输层协议的基础。HTTP、FTP和SMTP使用TCP；DNS使用UDP。TCP是面向连接的，具有可靠性机制；UDP是无连接的，负载更小，并且支持广播。*BitTorrent*和Voice over IP都使用UDP。

传输层比其他上层协议具有更高灵活性，性能可能也更高，但是它要求用户自己处理一些具体任务，如身份验证和加密。

（在WinRT中也可以实现TCP和UDP通信：参见16.12节的“在Windows Runtime中建立TCP连接”。）

对于TCP，我们可以选择使用简单易用的TcpClient和TcpListener外观类，或者使用功能丰富的Socket类。事实上，它们可以混合使用，因为我们可以通过TcpClient的Client属性获得底层的Socket对象。Socket类包含更多的配置选项，它支持网络层（IP）的直接访问，也支持一些非Internet的协议，如Novell的SPX/IPX。

和其他协议一样，TCP也区分客户端和服务器：客户端发起请求，而服务器则等待请求。下面是一个TCP客户端请求的基本结构：

```
using (TcpClient client = new TcpClient())
{
  client.Connect ("address", port);
  using (NetworkStream n = client.GetStream())
  {
    // 读写网络流……
  }
}
```

在创建后，TcpClient会立即在指定IP或域名地址和端口的服务器之间建立连接。在连接建立之前，构造函数会保持阻断。然后，NetworkStream会提供一种双向通信手段，同时支持从服务器发送和接收字节数据。

下面是一个简单的TCP服务器：

```
TcpListener listener = new TcpListener (<ip address>, port);
listener.Start();

while (keepProcessingRequests)
  using (TcpClient c = listener.AcceptTcpClient())
  using (NetworkStream n = c.GetStream())
  {
    // 读写网络流
  }

listener.Stop();
```

TcpListener会请求监听的本地IP地址（例如，具有两个网卡的计算机可能有两个地址）。我们可以使用IPAddress.Any要求它监听所有（或者只监听）本地IP地址。在接收到客户端请求之前，AcceptTcpClient会保持阻断，之后可以像客户端一样调用GetStream。

当在传输层操作时，我们需要确定一个协议来规定谁发起通话、何时通话以及通话时间多长，与无线电话很类似。如果双方同时发起通话或接听，那么通信就会中断！

现在让我们发明一种协议，其中客户端先发话“Hello”，然后服务器响应“Hello right back!”下面是实现的代码：

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Threading;

class TcpDemo
{
  static void Main()
  {
    new Thread (Server).Start();          // 并发地运行服务器方法
    Thread.Sleep (500);                   // 指定启动服务器的时间
    Client();
  }

  static void Client()
  {
    using (TcpClient client = new TcpClient ("localhost", 51111))
    using (NetworkStream n = client.GetStream())
    {
      BinaryWriter w = new BinaryWriter (n);
      w.Write ("Hello");
      w.Flush();
      Console.WriteLine (new BinaryReader (n).ReadString());
    }
  }

  static void Server()      // 处理一个客户端请求，然后退出
  {
    TcpListener listener = new TcpListener (IPAddress.Any, 51111);
    listener.Start();
    using (TcpClient c = listener.AcceptTcpClient())
    using (NetworkStream n = c.GetStream())
    {
      string msg = new BinaryReader (n).ReadString();
      BinaryWriter w = new BinaryWriter (n);
      w.Write (msg + " right back!");
      w.Flush();              // 必须调用Flush，因为我们不会清理编写器
    }
    listener.Stop();
  }
}
```

```
Hello
Hello right back!
```

在这个例子中，我们使用同一台主机的本地回路来运行客户端和服务器。我们随意选择了一个未分配的端口（大于49152），然后使用BinaryWriter和BinaryReader进行文本消息编码。不关闭或清理读取器和编写器，从而保持底层的NetworkStream一直打开，直到会话完成后才关闭。

BinaryWriter和BinaryReader似乎并不是读写字符串的最佳选择。然而，相对于StreamReader和StreamWriter，它们有一个重要优势：它们会给字符串加一个表示长度的数字前缀，使BinaryReader总是能够知道需要读取多少个字符。如果调用StreamReader.ReadToEnd，可能会被无限期阻断，因为NetworkStream并没有结尾！只要打开连接，网络流就无法确定客户端何时停止发送数据。

提示：实际上，StreamReader与NetworkStream是完全不同的，即使只想调用ReadLine。这是因为StreamReader是一个向前读取的缓冲区，因此它读取的字符会多于当前可用数，然后无限期阻断或者阻断到套接字超时。诸如FileStream等其他流则不存在与StreamReader不一致的问题，因为它们有一个确定的结尾，其中Read会立即返回0。

TCP并发性

TcpClient和TcpListener提供了基于任务的异步方法，可用于实现可扩展的并发性。使用这些方法，只需要将阻塞方法替换为他们对应的*Async版本方法，然后等待任务返回。

下面的例子编写了一个异步TCP服务器，它接受一个长度为5000字节的请求，倒转这些字节，然后将它们送回客户端：

```
async void RunServerAsync ()
{
  var listener = new TcpListener (IPAddress.Any, 51111);
  listener.Start ();
  try
  {
    while (true)
      Accept (await listener.AcceptTcpClientAsync ());
  }
  finally { listener.Stop(); }
}

async Task Accept (TcpClient client)
{
  await Task.Yield ();
  try
  {
    using (client)
    using (NetworkStream n = client.GetStream ())
    {
      byte[] data = new byte [5000];

      int bytesRead = 0; int chunkSize = 1;
      while (bytesRead < data.Length && chunkSize > 0)
        bytesRead += chunkSize =
          await n.ReadAsync (data, bytesRead, data.Length - bytesRead);

      Array.Reverse (data); // Reverse the byte sequence
      await n.WriteAsync (data, 0, data.Length);
    }
  }
  catch (Exception ex) { Console.WriteLine (ex.Message); }
}
```

这个程序的可扩展性在于去掉了请求过程中的线程阻塞。所以，如果有一千个客户同时通过一个慢速连接到达（例如，每一个请求需要耗费几秒钟时间），那么在这段时间里，这个程序可能不需要创建1000个线程（与同步解决方法不同）。相反，它只需要在`await`表达式前后的一小段代码执行时间里占用线程。

16.11 使用TCP接收POP3邮件

.NET Framework并没有提供任何POP3的应用层支持，所以要从一个POP3服务器接收邮件，必须在TCP层编写代码。幸好，这是一个简单的协议；POP3的会话方式如下：

客户端	邮件服务器	说明
客户端连接……	`+OK Hello there.`	欢迎消息
`USER joe`	`+OK Password required.`	
`PASS password`	`+OK Logged in.`	
`LIST`	`+OK`	列出服务器中每个消息的ID和文件大小
	`1 1876`	
	`2 5412`	
	`3 845`	
	`.`	
`RETR 1`	`+OK 1876 octets`	接收指定ID的消息
	`Content of message #1...`	
	`.`	
`DELE 1`	`+OK Deleted.`	从服务器删除一条消息
`QUIT`	`+OK Bye-bye.`	

除了多行的LIST和RETR命令是用一个点号分隔各行命令之外，每条命令都以换行符（CR + LF）结束。因为我们无法使用`StreamReader`来读取`NetworkStream`，所以我们编写一个帮助方法，以无缓冲的方式读取一行文本：

```
static string ReadLine (Stream s)
{
  List<byte> lineBuffer = new List<byte>();
  while (true)
  {
    int b = s.ReadByte();
    if (b == 10 || b < 0) break;
    if (b != 13) lineBuffer.Add ((byte)b);
  }
  return Encoding.UTF8.GetString (lineBuffer.ToArray());
}
```

我们还需要创建一个帮助方法来发送命令。因为我们总是会接收到以“+OK”开头的响应，所以可以同时读取和验证响应内容：

```
static void SendCommand (Stream stream, string line)
{
  byte[] data = Encoding.UTF8.GetBytes (line + "\r\n");
  stream.Write (data, 0, data.Length);
  string response = ReadLine (stream);
  if (!response.StartsWith ("+OK"))
    throw new Exception ("POP Error: " + response);
}
```

在编写这些方法之后，接收邮件就很简单了。我们会在端口110（默认的POP3端口）建立一个TCP连接，然后开始与服务器进行通信。

在这个例子中，在将消息从服务器删除之前，我们都会将每一条邮件消息写到一个随机命名的*.eml*文件中：

```
using (TcpClient client = new TcpClient ("mail.isp.com", 110))
using (NetworkStream n = client.GetStream())
{
  ReadLine (n);                             // 读取欢迎消息
  SendCommand (n, "USER username");
  SendCommand (n, "PASS password");
  SendCommand (n, "LIST");                  // 取回消息ID
  List<int> messageIDs = new List<int>();
  while (true)
  {
    string line = ReadLine (n);             // 例如，"1 1876"
    if (line == ".") break;
    messageIDs.Add (int.Parse (line.Split (' ')[0] ));      // 消息ID
  }

  foreach (int id in messageIDs)            // 取回每一条消息
  {
    SendCommand (n, "RETR " + id);
    string randomFile = Guid.NewGuid().ToString() + ".eml";
    using (StreamWriter writer = File.CreateText (randomFile))
      while (true)
      {
        string line = ReadLine (n);         // 读取下一行消息
        if (line == ".") break;             // 一个点表示消息结束
        if (line == "..") line = ".";       // “去掉”两个点
        writer.WriteLine (line);            // 将内容写到输出文件
      }
    SendCommand (n, "DELE " + id);          // 从服务器删除消息
  }
  SendCommand (n, "QUIT");
}
```

16.12 在Windows Runtime中建立TCP连接

Windows Runtime通过`Windows.Networking.Sockets`命名空间实现TCP功能。与.NET实现一样，其中主要有两个类，分别充当服务器和客户端角色。在WinRT中，它们分别是`StreamSocketListener`和`StreamSocket`。

下面的方法在端口51111建立一个服务器，等待客户端连接。然后，读取一个包含长度前缀的字符串消息：

```
async void Server()
{
```

```
    var listener = new StreamSocketListener();
    listener.ConnectionReceived += async (sender, args) =>
    {
      using (StreamSocket socket = args.Socket)
      {
        var reader = new DataReader (socket.InputStream);
        await reader.LoadAsync (4);
        uint length = reader.ReadUInt32();
        await reader.LoadAsync (length);
        Debug.WriteLine (reader.ReadString (length));
      }
      listener.Dispose();    // 接收一条消息后关闭监听器。
    };
    await listener.BindServiceNameAsync ("51111");
  }
```

这个例子使用一个WinRT类型DataReader（位于Windows.Networking命名空间）读取输入流，而不是将它转换为一个.NET Stream对象和使用BinaryReader。除了支持异步性，DataReader与BinaryReader完全不同。LoadAsync方法会以异步方式将若干字节读取到内部缓冲区中，然后调用ReadUInt32或ReadString方法读取更多内容。这意味着，如果想要连续读取1000个整数，那么要先调用LoadAsync，传入值4000，然后再循环调用1000次ReadInt32。这样可以避免在循环中调用异步操作的过载（因为每一个异步操作都会造成一定的过载）。

提示：DataReader/DataWriter包含一个ByteOrder属性，可用于控制是采用小端字节还是大端字节进行数字编码。其中，大端字节是默认方式。

等待AcceptAsync所获得的StreamSocket对象拥有单独的输入和输出流。所以，如果要写回一条消息，则需要使用套接字的OutputStream。

下面的客户端代码可以演示OutputStream和DataWriter的用法：

```
async void Client()
{
  using (var socket = new StreamSocket())
  {
    await socket.ConnectAsync (new HostName ("localhost"), "51111",
                               SocketProtectionLevel.PlainSocket);
    var writer = new DataWriter (socket.OutputStream);
    string message = "Hello!";
    uint length = (uint) Encoding.UTF8.GetByteCount (message);
    writer.WriteUInt32 (length);
    writer.WriteString (message);
    await writer.StoreAsync();
  }
}
```

首先直接创建一个StreamSocket，然后使用主机名和端口调用ConnectAsync。（在HostName的构造方法中传入一个DNS名称或IP地址字符串。）指定SocketProtectionLevel.Ssl，就可以请求SSL加密（如果服务器有配置）。

同样，这里要使用WinRT的DataWriter，而不是.NET的BinaryWriter，同时写入字符串长度（以字节表示，而非字符），后面紧跟UTF-8编码的字符串。最后，调用StoreAsync，将缓冲区写回后台系统，然后关闭套接字。

第17章

序列化

本章介绍序列化与反序列化，通过它对象可以表示成一个纯文本或者二进制形式。除非另行声明，否则本章中的所有类型都存在于下面的命名空间中：

```
System.Runtime.Serialization
System.Xml.Serialization
```

17.1 序列化概念

序列化是把内存中的一个对象或者对象图（一组互相引用的对象）转换点一个字节流或者一组可以保存或传输的XML节点。反序列化正好相反，它把一个数据流重新构造成一个内存中的对象或对象图。

序列化和反序列化通常用于：

- 通过网络或应用程序边界传输对象。
- 在文件或数据库中保存对象的表示。

另外，序列化与反序列化也用于深度克隆对象。数据契约和XML序列化引擎也可以被用作通用目的工具来加载和保存已知结构的XML文件。

.NET Framework从两个角度来支持序列化与反序列化：第一，从想进行序列化和反序列化对象的客户端角度；第二，从想控制其如何被序列化的类型角度。

17.1.1 序列化引擎

在.NET Framework中有4种序列化机制：

- 数据契约序列化器
- 二进制序列化器
- （基于属性）的XML序列化器（`XmlSerializer`）
- `IXmlSerializable`接口

其中前三种“引擎”可以完成大部分或所有序列化操作。而最后的IXmlSerializable接口是一个可以通过使用XmlReader和XmlWriter进行序列化的起桥梁作用的钩子（hook）。IXmlSerializable可以联合数据契约序列化器或者XmlSerializer来处理更复杂的XML序列化任务。

表17-1比较了这些引擎。

表17-1：各种引擎之间的比较

特性	数据契约序列化器	二进制序列化器	XmlSerializer	IXmlSerializable
自动化级别	***	*****	****	–
类型耦合	可选	紧密	松散	松散
版本容差	*****	***	*****	*****
保留对象引用	可选	是	否	可选
是否可以序列化非public字段	是	是	否	是
对互操作消息的适用性	*****	**	****	****
读写XML文件的灵活性	**	-	****	*****
压缩输出	**	****	**	**
性能	***	****	*到***	***

IXmlSerializable的分数假设已经使用XmlReader和XmlWriter最优化地（手）写代码。XML序列化引擎要求回收相同的XmlSerializer对象以达到更佳的性能。

1. 为何有三种引擎

出现这三种引擎在一定程度上是由于历史原因。Framework在序列化上基于两个完全不同的目的：

- 真实的序列化包含类型及其引用的.NET对象图
- XML和SOAP消息之间的互操作标准

第一种由Remoting的需求而产生；而第二种是由于Web服务。写一个序列化引擎来同时完成这两项任务非常困难，所以Microsoft编写了两个引擎：二进制序列化器和XML序列化器。

后来在.NET Framework 3.0中出现WCF时，其部分目标在于统一Remoting和Web服务。这就要求一个新的序列化引擎，所以就出现了数据契约序列化器。数据契约序列化器统一了旧有的两个和消息有关的引擎的特性。但是在这个上下文之外，这两个旧的序列化引擎还是很重要的。

2. 数据契约序列化器

数据契约序列化器在这三种序列化引擎中是最新的也是最有用的引擎，并被WCF使用。它在下面两种情形下尤其强大：

- 通过符合标准的消息协议来交换信息
- 需要好的版本容差能力，并且能够保留对象引用

数据契约序列化器支持一种数据契约模型：它能帮助把类型的底层细节与被序列化过的数据的结构解

耦。这为我们提供了优秀的版本容差性，也就意味着我们可以反序列化从早期或者后来版本序列化过来的数据类型。甚至可以反序列化已经被重命名或者被移到不同程序集中的类型。

数据契约序列化器可以处理大多数的对象图，尽管它需要比二进制序列化器更多的辅助。如果能够灵活地构造XML，它也可被用作通用目的的读写XML文件的工具。但是如果需要存储数据属性或者要处理随机出现的XML元素，就不能使用数据契约序列化器了。

3. 二进制序列化器

二进制序列化器比较容易使用、非常的自动化，并且在.NET Framework中自始至终都被很好地支持。Remoting在同一进程中的两个应用域之间通信时使用二进制序列化器（见第24章）。

二进制序列化器被高度地自动化了：只需要一个属性就可以使一个复杂类型可完全序列化。当所有类型都要求被高保真序列化时，二进制序列化器要比数据契约序列化器快。但是它把类型的内部结构与被序列化数据的格式紧密耦合，导致了比较差的版本容差性（在Framework 2.0之前，即使添加一个字段也会成为破坏版本的变化）。二进制引擎也不是真正地为生成XML而设计的，尽管它为基于SOAP的消息提供了一个有限的可以和简单类型互操作的格式化器。

4. XmlSerializer

XML序列化引擎只能产生XML，它没有其他能够保持和恢复复杂对象图的引擎那么强大（它不能够恢复共享的对象引用）。但是对于处理比较随意的XML结构，它是三者之中最灵活的。例如，可以选择把对象属性序列化为元素还是XML属性，还可以选择集合的外部元素的处理方式。XML引擎也提供了较好的版本容差性。

`XmlSerializer`被ASMX Web服务使用。

5. IXmlSerializable

实现`IXmlSerializable`意味着通过使用一个`XmlReader`和`XmlWriter`来完成序列化。`IXmlSerializable`接口被`XmlSerializer`和数据契约序列化器所识别，所以它可以有选择地被用来处理更复杂的类型。它也可以直接被WCF和ASMX Web服务使用。我们在第11章已经详细介绍过`XmlReader`和`XmlWriter`。

17.1.2 格式化器

数据契约和二进制序列化器的输出可以通过一个可置换的格式化器来确定。格式化器的作用与这两种序列化引擎一样，尽管它们使用完全不同的类来完成任务。

一个格式化器会确定最终的格式以适应特定媒介或者序列化上下文。大多数情况下，可以在XML格式化器和二进制格式化器之间选择。一个XML格式化器被设计为在一个XML读取器/写入器、文本文件/流或者SOAP消息包的上下文中工作。一个二进制格式化器被设计为在一个任意的字节流上下文中工作，特别是一个文件/流或者一个特有的消息包。二进制输出通常比XML小。

提示： 术语“二进制”在格式化器的上下文中和“二进制”序列化引擎没有关系。这两种引擎都可以使用XML格式化器和二进制格式化器。

理论上，序列化引擎与它们的格式化器是分离的。但实际上，每种引擎都被设计为适应其中一种类型的格式化器。数据契约序列化器适用于XML格式化器以满足XML消息的可互操作性需求。这对XML

格式化器有效，但同时意味着其二进制格式化器通常达不到预期的要求。相对的，二进制引擎提供了较好的二进制格式化器，但是其XML格式化器非常受限，仅仅提供了简单的SOAP互操作性。

17.1.3 显式与隐式序列化

序列化和反序列化可以通过两种方式来启动。

第一种是显式地，通过请求被序列化或反序列化的对象。当显式地序列化或反序列化时，就同时选择了序列化引擎和格式化器。

第二种方式相反，隐式的序列化被Framework启动，这发生在：

- 一个序列化器要递归循环一个子对象
- 使用一个依赖于序列化的特性，例如WCF、Remoting或Web服务

WCF总是使用数据契约序列化器，尽管它可以和其他引擎的属性和接口进行互操作。

Remoting总是使用二进制序列化引擎。

Web服务总是使用`XmlSerializer`。

17.2 数据契约的序列化

下面是使用数据契约序列化器的基本步骤：

1. 决定是使用`DataContractSerializer`还是`NetDataContractSerializer`。
2. 使用`[DataContract]`和`[DataMember]`属性修饰要序列化的对象和成员。
3. 实例化序列化器后调用`WriteObject`或`ReadObject`。

如果选择`DataContractSerializer`，同时需要注册已知类型（也能够被序列化的子类型），并且要决定是否保留对象引用。

可能也需要采取特殊措施来保证集合能被正确地序列化。

提示：与数据契约序列化器相关的类型被定义在`System.Runtime.Serialization`命名空间中，并包含在同名的程序集中。

17.2.1 DataContractSerializer与NetDataContractSerializer

有两个数据契约序列化器：

`DataContractSerializer`

.NET类型与数据契约类型松耦合。

`NetDataContractSerializer`

.NET类型与数据契约类型紧耦合。

`DataContractSerializer`可以产生可互操作的符合标准的XML，例如：

```
<Person xmlns="...">
  ...
</Person>
```

但是它需要预先显式地注册可序列化的子类型，这样它就能把一个数据契约名称，例如"Person"映射到一个.NET类型。NetDataContractSerializer不需要这种辅助，因为它把要序列化对象的类型和程序集的全名输出，而不同于二进制序列化引擎：

```
<Person z:Type="SerialTest.Person" z:Assembly=
  "SerialTest, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null">
  ...
</Person>
```

但是这种输出是私有的。为了反序列化，它也要依赖于特定命名空间和程序集下的特定的.NET类型。

如果把一个对象图说成是一个"黑箱"，可以选择这两种序列化器中的任意一种，取决于哪种优势更明显。如果通过WCF通信或者读/写一个XML文件，可能倾向于使用DataContractSerializer。

这两种序列化器之间的另外一个不同点是NetDataContractSerializer总是会保留引用相等性，而DataContractSerializer只会根据需要做这些。

接下来的小节会更详细地介绍以上主题。

17.2.2 使用序列化器

选择序列化器后，下一步就是添加相应的属性到要序列化的类型和成员上。至少应该：

- 添加[DataContract]属性到每个类型上。
- 添加[DataMember]属性到每个包含的成员上。

示例：

```
namespace SerialTest
{
  [DataContract] public class Person
  {
    [DataMember] public string Name;
    [DataMember] public int Age;
  }
}
```

这些属性足以使一个类型被数据契约引擎隐式地序列化。

可以显式地通过实例化一个DataContractSerializer或NetDataContractSerializer，然后调用WriteObject或ReadObject方法来序列化和反序列化一个对象实例：

```
Person p = new Person { Name = "Stacey", Age = 30 };

var ds = new DataContractSerializer (typeof (Person));

using (Stream s = File.Create ("person.xml"))
  ds.WriteObject (s, p);                          // 序列化

Person p2;
```

```
using (Stream s = File.OpenRead ("person.xml"))
  p2 = (Person) ds.ReadObject (s);                    // 反序列化

Console.WriteLine (p2.Name + " " + p2.Age);           // Stacey 30
```

DataContractSerializer的构造方法需要一个根对象类型（显式序列化的对象类型），相反的，NetDataContractSerializer就不需要：

```
var ns = new NetDataContractSerializer();

// NetDataContractSerializer在其他方面与DataContractSerializer的用法相同
...
```

两种序列化器都默认使用XML格式化器。使用XmlWritter，可以为了可读性让输出包含缩进：

```
Person p = new Person { Name = "Stacey", Age = 30 };
var ds = new DataContractSerializer (typeof (Person));

XmlWriterSettings settings = new XmlWriterSettings() { Indent = true };
using (XmlWriter w = XmlWriter.Create ("person.xml", settings))
  ds.WriteObject (w, p);

System.Diagnostics.Process.Start ("person.xml");
```

结果如下：

```
<Person xmlns="http://schemas.datacontract.org/2004/07/SerialTest"
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <Age>30</Age>
  <Name>Stacey</Name>
</Person>
```

XML元素名称<Person>对应于数据契约名称，也就是默认的.NET类型名称。可以在数据契约上显式地指定这个名称：

```
[DataContract (Name="Candidate")]
public class Person { ... }
```

XML命名空间对应于数据契约的命名空间，默认是*http://schemas.datacontract.org/2004/07/*加上.NET类型的命名空间。可以用类似的方式重写它：

```
[DataContract (Namespace="http://oreilly.com/nutshell")]
public class Person { ... }
```

提示：指定名称和命名空间可以把契约标识与.NET类型名称解耦。它能够保证当重构和改变类型的名称或命名空间时，序列化不会受到影响。

也可以重写数据成员的名称：

```
[DataContract (Name="Candidate", Namespace="http://oreilly.com/nutshell")]
public class Person
{
  [DataMember (Name="FirstName")] public string Name;
  [DataMember (Name="ClaimedAge")] public int Age;
}
```

输出如下：

```
<?xml version="1.0" encoding="utf-8"?>
<Candidate xmlns="http://oreilly.com/nutshell"
           xmlns:i="http://www.w3.org/2001/XMLSchema-instance" >
  <ClaimedAge>30</ClaimedAge>
  <FirstName>Stacey</FirstName>
</Candidate>
```

[DataMember]可以支持public和private字段和属性。字段和属性的数据类型可以是下列类型的任何一种：

- 任何基本类型
- DateTime、TimeSpan、Guid、Uri或者Enum值
- 上述类型的Nullable版本
- Byte[]（在XML中序列化为base 64）
- 任何用DataContract修饰的已知类型
- 任何IEnumerable 类型
- 任何被[Serializable]修饰，或者实现了ISerializable的类型
- 实现了IXmlSerializable的任何类型

指定二进制格式

可以同时使用二进制格式化器和DataContractSerializer或者NetDataContractSerializer。过程是一样的：

```
Person p = new Person { Name = "Stacey", Age = 30 };
var ds = new DataContractSerializer (typeof (Person));

var s = new MemoryStream();
using (XmlDictionaryWriter w = XmlDictionaryWriter.CreateBinaryWriter (s))
  ds.WriteObject (w, p);

var s2 = new MemoryStream (s.ToArray());
Person p2;
using (XmlDictionaryReader r = XmlDictionaryReader.CreateBinaryReader (s2,
                               XmlDictionaryReaderQuotas.Max))
p2 = (Person) ds.ReadObject (r);
```

二进制格式化器输出会比XML格式化器稍微小一些，当类型中包含大的数组时就会明显地看到小得多。

17.2.3 序列化子类

在使用NetDataContractSerializer时，不需要特别地处理子类的序列化，除非子类需要[DataContract]属性。序列化器会写实际类型的全匹配名称，它会序列化为：

```
<Person ... z:Type="SerialTest.Person" z:Assembly=
  "SerialTest, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null">
```

但是一个DataContractSerializer必须要了解它可能序列化或反序列化的所有子类型。为了说明

这一点，假设Person类有以下的子类：

```
[DataContract] public class Person
{
  [DataMember] public string Name;
  [DataMember] public int Age;
}
[DataContract] public class Student : Person { }
[DataContract] public class Teacher : Person { }
```

然后我们定义下面的方法用来克隆一个Person：

```
static Person DeepClone (Person p)
{
  var ds = new DataContractSerializer (typeof (Person));
  MemoryStream stream = new MemoryStream();
  ds.WriteObject (stream, p);
  stream.Position = 0;
  return (Person) ds.ReadObject (stream);
}
```

然后调用这个方法：

```
Person person = new Person { Name = "Stacey", Age = 30 };
Student student = new Student { Name = "Stacey", Age = 30 };
Teacher teacher = new Teacher { Name = "Stacey", Age = 30 };

Person p2 = DeepClone (person);              // OK
Student s2 = (Student) DeepClone (student); // SerializationException
Teacher t2 = (Teacher) DeepClone (teacher); // SerializationException
```

如果传入一个Person调用DeepClone，就可正常工作；但是如果传入一个Student或者一个Teacher，由于序列化器没有办法知道Student或Teacher应该被解析成哪个.NET类型（或程序集），就会抛出一个异常。

解决办法是指定所有允许或“已知”的子类型。可以在构造DataContractSerializer时像下面这样做：

```
var ds = new DataContractSerializer (typeof (Person),
  new Type[] { typeof (Student), typeof (Teacher) } );
```

或者在类型上添加KnownType属性：

```
[DataContract, KnownType (typeof (Student)), KnownType (typeof (Teacher))]
public class Person
...
```

下面就是按上面的方式序列化Student后的结果：

```
<Person xmlns="..."
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
        i:type="Student" >
...
<Person>
```

因为我们指定Person为根类型，根元素名称依然为“Person”。实际的子类型单独地通过type属性被描述。

提示： 当序列化子类型时，不管使用哪种序列化器，NetDataContractSerializer会导致性能上的损失。就好像是当遇到子类型时，它就必须停下来思考一下。

当在一个应用程序服务器上处理大量并发请求时才会考虑序列化性能。

17.2.4 对象引用

其他对象的引用也会被序列化。考虑下面的类：

```
[DataContract] public class Person
{
  [DataMember] public string Name;
  [DataMember] public int Age;
  [DataMember] public Address HomeAddress;
}

[DataContract] public class Address
{
  [DataMember] public string Street, Postcode;
}
```

下面是使用DataContractSerializer把它序列化为XML的结果：

```
<Person...>
  <Age>...</Age>
  <HomeAddress>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </HomeAddress>
  <Name>...</Name>
</Person>
```

提示： 我们在前面定义的DeepClone方法也会克隆HomeAddress，把它和简单的成员复制区分开。

当使用DataContractSerializer时，如果要继承Address类或者继承根类型，也会应用同样的规则。例如我们定义了一个USAddress：

```
[DataContract]
public class USAddress : Address { }
```

把它的一个实例赋给Person：

```
Person p = new Person { Name = "John", Age = 30 };
p.HomeAddress = new USAddress { Street = "Fawcett St", Postcode = "02138" };
```

p不能被序列化。解决办法是或者应用[KnownType]属性到Address，像下面这样：

```
[DataContract, KnownType (typeof (USAddress))]
public class Address
{
  [DataMember] public string Street, Postcode;
}
```

或者在构造时指定DataContractSerializer有关USAddress的信息：

```
var ds = new DataContractSerializer (typeof (Person),
  new Type[] { typeof (USAddress) } );
```

我们不需要指定有关Address的信息，因为它是数据成员HomeAddress的声明类型。

保留对象引用

NetDataContractSerializer总是会保留引用相等性。而DataContractSerializer不会，除非指定它保留。

这意味着如果相同的对象在两个不同的地方被引用，DataContractSerializer通常会把它写两次。如果我们更改先前的示例，让Person也保存一个WorkAddress：

```
[DataContract] public class Person
{
  ...
  [DataMember] public Address HomeAddress, WorkAddress;
}
```

然后序列化一个如下的实例：

```
Person p = new Person { Name = "Stacey", Age = 30 };
p.HomeAddress = new Address { Street = "Odo St", Postcode = "6020" };
p.WorkAddress = p.HomeAddress;
```

我们会在XML中看到两次相同的地址描述：

```
...
<HomeAddress>
  <Postcode>6020</Postcode>
  <Street>Odo St</Street>
</HomeAddress>
...
<WorkAddress>
  <Postcode>6020</Postcode>
  <Street>Odo St</Street>
</WorkAddress>
```

而反序列化后，WorkAddress和HomeAddress会成为不同的对象。这种系统的优势在于它可以保持XML简单并且符合标准。缺点在于它会引入很大的XML，失去引用完整性，并且无法处理循环引用。

可以在构造DataContractSerializer时指定参数preserveObjectReferences为true来要求引用完整性。

```
var ds = new DataContractSerializer (typeof (Person),null, 1000, false, true, null);
```

当preserveObjectReferences为true时第三个参数是强制性的，它指示序列化器要追踪的最大的对象引用数。如果超出了这个数量，序列化器就会抛出一个异常（这可以防止通过恶意的构造流来进行拒绝服务攻击）。

下面是一个Person拥有相同的WorkAddress和HomeAddress地址的XML的例子：

```
<Person xmlns="http://schemas.datacontract.org/2004/07/SerialTest"
```

```
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/"
        z:Id="1">
  <Age>30</Age>
  <HomeAddress z:Id="2">
    <Postcode z:Id="3">6020</Postcode>
    <Street z:Id="4">Odo St</Street>
  </HomeAddress>
  <Name z:Id="5">Stacey</Name>
  <WorkAddress z:Ref="2" i:nil="true" />
</Person>
```

这样做的代价就是减少了可互操作性（注意Id和Ref属性特有的命名空间）。

17.2.5 版本容差

可以添加或删除数据成员而不用破坏向前或向后的版本兼容性。默认，数据契约序列化器做如下工作：

- 跳过类型中没有[DataMember]的数据
- 如果在序列化流中缺少了任何[DataMember]，它不受影响

与其跳过无法识别的数据，不如引导序列化器把无法识别的数据成员存储到一个黑盒中，然后决定它们是否被重新序列化。这巧妙地绕过类型后期版本序列化过的数据。为了激活这个特性，要实现IExtensibleDataObject，这个接口相当于黑盒提供者。它要求实现一个属性来获取或设置这个黑盒：

```
[DataContract] public class Person : IExtensibleDataObject{
  [DataMember] public string Name;
  [DataMember] public int Age;

ExtensionDataObject IExtensibleDataObject.ExtensionData { get; set; }
}
```

必要的成员

如果某个成员对于一个类型是非常重要的，可以通过指定[IsRequired]要求它必须出现：

```
[DataMember(IsRequired = true)] public int ID;
```

如果成员没有出现，在序列化时会抛出一个异常。

17.2.6 成员顺序

数据契约序列化器对数据成员的数据要求极其苛刻。反序列化器实际上会跳过任何被认为在序列外的成员。

在序列化成员时按下面的顺序：

1. 从基类到子类
2. 根据Order从低到高（对于[Order]属性被设置的数据成员）
3. 字母表顺序（使用传统的字符串比较法）

所以在前面的示例中，Age在Name前面，在下面的示例中，Name在Age前面：

```
[DataContract] public class Person
{
  [DataMember (Order=0)] public string Name;
  [DataMember (Order=1)] public int Age;
}
```

如果Person有一个基类，那么基类的数据成员先被序列化。

要指定顺序的主要原因是为了遵循特定的XML Schema。XML元素的顺序等同于数据成员顺序。

如果需要和其他成员互操作，最容易的方法不是指定成员Order，而是纯粹依赖于字母排序法。这样对于添加和删除成员在序列化和反序列化时就不会出现任何差异。而唯一紊乱的情况是在基类和子类之间移动一个成员。

17.2.7 Null和空值

有两种方式来处理值为null或空的数据成员：

1. 显式地写null或空值
2. 从序列化输出中忽略这些数据成员

在XML中，一个显式的null值例如：

```
<Person xmlns="..."
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <Name i:nil="true" />
</Person>
```

写null或空值会浪费空间，特别是有大量属性或字段经常为空的类型。更重要的是，可能需要遵循一个期望使用可选元素（例如，minOccurs="0"）而不是nil值的XML Schema。

可以指示序列化器不要为null或空的值产生数据成员，像下面这样：

```
[DataContract] public class Person
{
  [DataMember (EmitDefaultValue=false)] public string Name;
  [DataMember (EmitDefaultValue=false)] public int Age;
}
```

如果Name值为null，它就会被忽略；如果Age值为0（int型的默认值），也会被忽略。

提示： 在产生（rehydrating）对象时，数据契约反序列化器会绕过类型的构造方法和字段初始化器。因此可以忽略数据成员，而不会破坏那些通过构造方法或字段初始化器来赋无默认值的字段。为了说明这一点，我们假设Person的Age的默认值为30，像下面这样：

```
[DataMember (EmitDefaultValue=false)]
public int Age = 30;
```

现在假设我们实例化了一个Person，显式地设置其Age从30到0，然后进行序列化。输出不会包含Age，因为int类型的默认值是0。这意味着，在反序列化时，Age会被忽略，这个字段会保留其默认值0，绕过了字段初始化器和构造方法。

序列化

17.3 数据契约与集合

数据契约序列化器可以保持和恢复任何可遍历集合。例如，假设我们定义了Person包含List<Address>类型的Addresses字段：

```
[DataContract] public class Person
{
  ...
  [DataMember] public List<Address> Addresses;
}
[DataContract] public class Address
{
  [DataMember] public string Street, Postcode;
}
```

下面是序列化一个包含两个Address的Person的结果：

```
<Person ...>
  ...
  <Addresses>
    <Address>
      <Postcode>6020</Postcode>
      <Street>Odo St</Street>
    </Address>
    <Address>
      <Postcode>6152</Postcode>
      <Street>Comer St</Street>
    </Address>
  </Addresses>
  ...
</Person>
```

注意序列化器没有对它序列化的集合的类型信息进行任何编码。Address字段的类型如果为Address[]，输出结果完全一样。这允许改变集合字段或属性的类型，而在序列化和反序列化时不会产生错误。

有时候，需要一个比指定的集合类型更具体的类型，一个典型的示例就是接口：

```
[DataMember] public IList<Address> Addresses;
```

序列化器可以正确地序列化，但是在反序列化时就会出问题：反序列化器无法知道到底实例化哪种具体类型，所以，它选择了最简单的数组。序列化会坚持使用这种策略，即使指定了一个具体类型来实例化这个字段：

```
[DataMember] public IList<Address> Addresses = new List<Address>();
```

注意反序列化器要绕过字段初始化器。一个变通方案就是为数据成员提供一个私有字段，并添加一个公开的属性来访问它：

```
[DataMember(Name = "Addresses")] List<Address> _addresses;

public IList<Address> Addresses { get { return _addresses; } }
```

在重要的应用程序中，可以通过这种方式来使用属性。唯一不同的是我们把私有字段标记为

[DataMember]，而不是一个公共属性。

17.3.1 子类集合元素

序列化器处理子类集合元素的过程是透明的。当子类要在其他地方使用时，必须声明有效的子类型：

```
[DataContract, KnownType (typeof (USAddress))]
public class Address
{
  [DataMember] public string Street, Postcode;
}

public class USAddress : Address { }
```

添加USAddress到Person的Address，然后生成的XML是：

```
...
  <Addresses>
    <Address i:type="USAddress">
      <Postcode>02138</Postcode>
      <Street>Fawcett St</Street>
    </Address>
  </Addresses>
```

17.3.2 自定义集合与元素名称

如果继承一个集合类，可以用CollectionDataContractAttribute自定义每个元素的XML名称：

```
[CollectionDataContract (ItemName="Residence")]
public class AddressList : Collection<Address> { }

[DataContract] public class Person
{
  ...
  [DataMember] public AddressList Addresses;
}
```

生成的结果如下：

```
...
  <Addresses>
    <Residence>
      <Postcode>6020</Postcode
      <Street>Odo St</Street>
    </Residence>
...
```

CollectionDataContract也可以指定Namespace和Name。Name是在集合被序列化为根对象时使用的名称，而不是用在集合被序列化为另外一个对象的属性（如上例）时的名称。

也可以使用CollectionDataContract来控制字典的序列化:

```
[CollectionDataContract(ItemName = "Entry",
                        KeyName  = "Kind",
                       ValueName  = "Number")]
```

序列化

```
public class PhoneNumberList : Dictionary<string, string> { }

[DataContract] public class Person
{
   ...
    [DataMember] public PhoneNumberList PhoneNumbers;
}
```

输出的格式如下：

```
...
  <PhoneNumbers>
    <Entry>
      <Kind>Home</Kind>
      <Number>08 1234 5678</Number>
    </Entry>
    <Entry>
      <Kind>Mobile</Kind>
      <Number>040 8765 4321</Number>
   </Entry>
 </PhoneNumbers>
```

17.4 扩展数据契约

这一部分介绍如何通过[Serializable]和IXmlSerializable来扩展数据契约的能力。

17.4.1 序列化与反序列化钩子

如果要在序列化之前或之后执行一个自定义方法，可以通过在方法上标记以下属性：

[OnSerializing]
    指示在序列化之前调用这个方法。

[OnSerialized]
    指示在序列化之后调用这个方法。

反序列化也支持类似的属性：

[OnDeserializing]
    指示在反序列化之前调用这个方法。

[OnDeserialized]
    指示在反序列化之后调用这个方法。

自定义方法只能定义一个StreamingContext类型的参数。这个参数是为了与二进制引擎保持一致而被要求的，它不被数据契约序列化器使用。

[OnSerializing]和[OnDeserialized]在处理超出数据契约引擎能力之外的成员时有用，例如一个超额的集合或者没有实现标准接口的集合。下面是基本的解决方法：

```
[DataContract] public class Person
{
  public SerializationUnfriendlyType Addresses;
  [DataMember (Name="Addresses")]
```

```
    SerializationFriendlyType _serializationFriendlyAddresses;

    [OnSerializing]
    void PrepareForSerialization (StreamingContext sc)
    {
      // Copy Addresses > _serializationFriendlyAddresses
      // ...
    }

    [OnDeserialized]
    void CompleteDeserialization (StreamingContext sc)
    {
      // Copy _serializationFriendlyAddresses > Addresses
      // ...
    }
  }
```

[OnSerializing]标记的方法也可以被用作有条件的序列化字段：

```
public DateTime DateOfBirth;

[DataMember] public bool Confidential;

[DataMember (Name="DateOfBirth", EmitDefaultValue=false)]
DateTime? _tempDateOfBirth;

[OnSerializing]
void PrepareForSerialization (StreamingContext sc)
{
  if (Confidential)
    _tempDateOfBirth = DateOfBirth;
  else
    _tempDateOfBirth = null;
}
```

注意数据契约反序列化器会绕过字段初始化器和构造方法。标记了[OnDeserializing]的方法在反序列化过程中起着伪构造方法的作用，并且它对初始化被排除在序列外的字段很有用：

```
[DataContract] public class Test
{
  bool _editable = true;

  public Test() { _editable = true; }

[OnDeserializing]
  void Init (StreamingContext sc)
  {
    _editable = true;
  }
}
```

如果不是Init方法，_editable在反序列化Test实例后会是false，尽管字段初始化器和构造方法试图把它变为true。

使用这4个属性修饰的方法可能是私有的。如果子类需要参与其中，那么它们可以使用相同的属性定义自己的方法，然后它们一样可以执行。

17.4.2 与[Serializable]互操作

数据契约序列化器也可以序列化标记了二进制序列化引擎中的属性或接口的类型。这种功能是非常重要的，因为这是为了支持已经被写入Framework 3.0 以下版本（包括.NET Framework）中的二进制引擎。

提示： 下面两项可以标记一个可被二进制引擎序列化的类型：

- [Serializable]属性
- 实现ISerializable

二进制互操作性对于序列化已有类型并且需要同时支持这两种引擎的情况比较有用。它也提供了扩展数据契约序列化器的另一种方式，因为二进制引擎的ISerializable要比数据契约属性更灵活。但是，数据契约序列化器不能通过ISerializable格式化添加的数据。

一个类型想在两种引擎上都达到最佳效果是不能同时使用两种引擎里的属性的。否则某些类型会出问题，例如string和DateTime，由于历史原因，它们不能与二进制引擎属性分离。数据契约序列化器通过筛选这些基本类型并对它们单独进行特殊处理。数据契约序列化器会对其他所有标记了二进制引擎的类型应用类似的规则。这意味着它还会使用NonSerialized属性和调用ISerializable方法。不会把它当作二进制引擎本身，这保证了输出与使用了数据契约属性的格式一样。

警告： 被设计为使用二进制序列化引擎的类型期望对象引用被保留。可以通过DataContract-Serializer或者NetDataContractSerializer打开这个选项。

注册已知类型的规则也可以应用到通过二进制接口序列化的对象或子对象上。

下面的示例说明了一个标记了[Serializable]数据成员的类：

```
[DataContract] public class Person
{
    ...
    [DataMember] public Address MailingAddress;
}
[Serializable] public class Address
{
   public string Postcode, Street;
}
```

这是序列化后的结果：

```
<Person ...>
  ...
  <MailingAddress>
    <Postcode>6020</Postcode>
    <Street>Odo St</Street>
  </MailingAddress>
  ...
```

如果Address实现了ISerializable，结果被低效地格式为：

```
<MailingAddress>
  <Street xmlns:d3p1="http://www.w3.org/2001/XMLSchema"
    i:type="d3p1:string" xmlns="">str</Street>
  <Postcode xmlns:d3p1="http://www.w3.org/2001/XMLSchema"
    i:type="d3p1:string" xmlns="">pcode</Postcode>
</MailingAddress>
```

17.4.3 与IXmlSerializable互操作

数据契约序列化器的一个限制是它几乎不能控制XML的结构。在一个WCF应用程序中，这实际上是有好处的，因为它使得基础结构更容易符合标准消息协议。

如果需要控制XML的结构，可以实现IXmlSerializable接口，然后使用XmlReader和XmlWriter来手动地读和写XML，数据契约序列化器仅允许在那些需要这一控制的类型上执行这些操作。我们会在本章的最后部分更详细地介绍IXmlSerializable接口。

17.5 二进制序列化器

二进制序列化引擎被Remoting隐式地使用。它可以用来完成把对象保存到磁盘或从磁盘上还原对象之类的任务。二进制序列化被高度地自动化了，并可以用最少的操作来处理复杂的对象图。

有两种方式让一个类型支持二进制序列化。第一种是基于属性；第二种是实现ISerializable接口。添加属性相对比较简单，而实现ISerializable更灵活。实现ISerializable主要是为了：

- 动态地控制什么要被序列化。
- 让可序列化类型能够被其他部分更友好地继承。

入门

一个类型可以使用单个属性指定为可序列化的：

```
[Serializable] public sealed class Person
{
  public string Name;
  public int Age;
}
```

[Serializable]属性使序列化器包含类型中所有的字段。这既包含私有字段，也包含公共字段（但不包含属性）。每一个字段本身都可序列化，否则就会抛出一个异常。基本.NET类型，例如string和int支持序列化（许多其他.NET类型也是）。

提示： [Serializable]属性不能被继承，所以子类不会自动成为可序列化的，除非也在子类上标记上这个属性。

对于自动属性，二进制序列化引擎会序列化底层的被编译出的字段。但是，当增加属性时，重新编译这个类型会改变这个字段的名称，这就会破坏已序列化数据的兼容性。处理方法就是在[Serializable]的类型里避免使用自动属性或者实现ISerializable接口。

为了序列化一个Person的实例，可以实例化一个格式化器，然后调用Serialize方法。在二进制引擎中有两个可用的格式化器：

BinaryFormatter
: 两者之中效率稍高，在更少的时间里产生更小的输出。它的命名空间是System.Runtime.Serialization.Formatters.Binary，程序集为*mscorlib*。

SoapFormatter
: 它支持在使用Remoting时基本的SOAP样式的消息。它的命名空间是System.Runtime.Serialization.Formatters.Soap，程序集为*System.Runtime.Serialization.Formatters.Soap.dll*。

警告：SoapFormatter没有BinaryFormatter实用。SoapFormatter不支持泛型或者筛选对版本容差有必要的额外数据。

否则这两个格式化器在使用上就完全一样了。下面用一个Binaryformatter来格式化一个Person：

```
Person p = new Person() { Name = "George", Age = 25 };

IFormatter formatter = new BinaryFormatter();

using (FileStream s = File.Create ("serialized.bin"))
  formatter.Serialize (s, p);
```

所有对于重新构造Person对象有必要的数据都被写到了*serialized.bin*文件中。Deserialize方法用来还原对象：

```
using (FileStream s = File.OpenRead ("serialized.bin"))
{
  Person p2 = (Person) formatter.Deserialize (s);
  Console.WriteLine (p2.Name + " " + p.Age); // George 25
}
```

警告：反序列化器在重新创建对象时会绕过所有的构造方法。在这个过程中实际调用了FormatterServices.GetUninitializedObject方法来完成这个工作。可以自己调用这个方法来实现可能会非常复杂的设计模式。

序列化过的数据包含类型和程序集的全部信息，所以如果试图把序列化的结果转换到一个不同程序集中的Person类型，结果会产生一个错误。在反序列化过程中，序列化器会完全恢复对象引用到序列化的状态。集合同样如此，它会对集合像其他类型一样处理（所有在System.Collections.* 下的类型都被标记为了可序列化）。

提示：二进制引擎可以处理大且复杂的对象图而不需要特别辅助（不用保证所有参与的成员都可序列化）。唯一要注意的是，序列化器的性能会随着对象图的引用数量的增加而降低。这样在一个要处理大量并发请求的Remoting服务器上就会成为一个问题。

17.6 二进制序列化属性

17.6.1 [NonSerialized]

不同于数据契约对要序列化的字段使用选择性加入方针，二进制引擎使用选择性排除方针。对于不想序列化的字段，例如用来进行临时计算的字段或用作文件排序或窗口句柄的字段，必须显式地使用[NonSerialized]属性来标记它们：

```
[Serializable] public sealed class Person
{
  public string Name;
  public DateTime DateOfBirth;

  // Age可以计算得到，所以不必进行序列化
  [NonSerialized] public int Age;
}
```

这就指示序列化器忽略Age成员。

警告： 不序列化的成员在反序列化后总是为空或null，即使在构造方法或字段初始化器中设置了它们。

17.6.2 [OnDeserializing]和[OnDeserialized]

反序列化绕过所有的构造方法和字段初始化器。如果每个字段都参与了序列化，则正确地执行，但是如果某些字段通过[NonSerialized]被排除在序列化之外时就会有问题。我们可以通过添加一个名为Valid的bool型字段来说明这一点：

```
public sealed class Person
{
  public string Name;
  public DateTime DateOfBirth;

  [NonSerialized] public int Age;
  [NonSerialized] public bool Valid = true;

  public Person() { Valid = true; }
}
```

反序列化后的Person的Valid会是false，尽管有构造方法和字段初始化器。

解决方案和数据契约序列化器是一样的：使用[OnDeserializing]属性来定义一个特别的反序列化构造方法。一个标记了这个属性的方法会在反序列化之前被调用：

```
[OnDeserializing]
void OnDeserializing (StreamingContext context)
{
  Valid = true;
}
```

我们也可以定义一个标记了[OnDeserialized]的方法用来在反序列化后计算Age字段：

```
[OnDeserialized]
```

```
void OnDeserialized (StreamingContext context)
{
  TimeSpan ts = DateTime.Now - DateOfBirth;
  Age = ts.Days / 365;                // 粗略的整数年
}
```

17.6.3 [OnSerializing]和[OnSerialized]

二进制引擎也支持[OnSerializing]和[OnSerialized]属性，这两个属性用来标记在序列化之前或之后要被调用的方法上。为了说明它们的作用，我们定义一个包含泛型List的玩家的Team类：

```
[Serializable] public sealed class Team
{
  public string Name;
  public List<Person> Players = new List<Person>();
}
```

这个类可以使用二进制格式化器来正确地格式化和反格式化，但是SOAP格式化器就不行。这是因为SOAP格式化器拒绝格式化泛型类型。一个简单的解决方案就是在序列化之前把Players转化为一个数组，然后在反序列化时再把它转化为泛型的List。为了完成这项工作，我们添加另外一个字段来存储这个数据，把原来Players字段标记为[NonSerialized]，然后按下面的代码来转换：

```
[Serializable] public sealed class Team
{
  public string Name;
  Person[] _playersToSerialize;

  [NonSerialized] public List<Person> Players = new List<Person>();

  [OnSerializing]
  void OnSerializing (StreamingContext context)
  {
    _playersToSerialize = Players.ToArray();
  }

  [OnSerialized]
  void OnSerialized (StreamingContext context)
  {
    _playersToSerialize = null; // 指定它没有数据成员
  }

  [OnDeserialized]
  void OnDeserialized (StreamingContext context)
  {
    Players = new List<Person> (_playersToSerialize);
  }
}
```

17.6.4 [OptionalField]属性与版本

默认，添加一个字段会破坏已经序列化的数据的兼容性，除非新的字段上附加了[OptionalField]属性。

为了说明这一点，假设我们从一个仅包含一个字段的Person类开始，我们称为版本1：

```
[Serializable] public sealed class Person        // Version 1
```

```
{
    public string Name;
}
```

后来，我们意识到需要第二个字段，所以按下面的代码创建了版本2：

```
[Serializable] public sealed class Person       // Version 2
{
    public string Name;
    public DateTime DateOfBirth;
}
```

如果两台电脑通过Remoting来交换Person对象，反序列化会出问题，除非它们都在完全相同的时间把Person更新到版本2。而 [OptionalField]属性就可以解决这个问题：

```
[Serializable] public sealed class Person // Version 2 Robust
{
  public string Name;
  [OptionalField (VersionAdded = 2)] public DateTime DateOfBirth;
}
```

这相当于指示反序列化器当在数据流中没有DateOfBirth时，只需要把漏掉的字段处理为没有被序列化。这意味着最后会得到一个空的DateTime（然后可以在标记了[OnDeserializing]的方法中给它赋一个不同的值）。

整数型的VersionAdded参数在每次添加字段时要增加。不过它只作为文档性的参数，不会对序列化有影响。

警告： 版本健壮性十分重要，避免重命名和删除字段，同时避免追溯性地添加[NonSerialized] 属性，永远不要改变字段的类型。

到目前为止我们关注了向后兼容性问题：反序列化器无法在序列化流中找到期望的字段。但是对于双向通信，反序列化器在遇到它不知道如何处理的额外字段时就会出现向前兼容性问题。二进制格式化器会自动地丢弃这些字段，而SOAP格式化器会抛出一个异常。所以如果在双向通信时，要求版本健壮性，必须使用二进制格式化器，否则需要通过实现ISerializable来手动地控制序列化。

17.7 使用ISerializable进行二进制序列化

实现ISerializable可以让一个类型完全控制其二进制序列化和反序列化。

ISerializable接口的定义如下：

```
public interface ISerializable
{
  void GetObjectData (SerializationInfo info, StreamingContext context);
}
```

GetObjectData在序列化时被触发；它的任务就是把想序列化的所有字段存放到SerializationInfo（一个名称/值的字典） 对象里。下面是我们如何实现GetObjectData方法来分别序列化Name和DateOfBirth两个字段。

```
public virtual void GetObjectData (SerializationInfo info,StreamingContext context)
{
  info.AddValue ("Name", Name);
  info.AddValue ("DateOfBirth", DateOfBirth);
}
```

在这个示例中，我们选择了根据字段名称来相应地命名，但这不是必须的。可以使用任何名称，只要在反序列化时使用相同的名称就可以了。值本身可以是任何可序列化的类型；Framework会根据需要递归序列化。把所有的空值存到字典里也是合法的。

提示： 把GetObjectData方法设置为virtual可以让子类扩展序列化而不用重新实现这个接口。

SerializationInfo也包含相应的属性以用来控制实例应该反序列化的类型和程序集。StreamingContext参数是它包含的结构，一个枚举值指示这个序列化的实例保存的位置（磁盘、Remoting等，尽管这个值不总是有）。

除了实现ISerializable，一个控制其序列化的类型也需要提供一个反序列化构造方法，这个方法包含和GetObjectData方法一样的两个参数。构造方法可以被声明为任何访问级别，运行时总能够找到它。特别是，可以声明它为protected级别，这样子类就可以调用它了。

在下面的示例中，我们在Team类上实现ISerializable，当它要处理List时，我们序列化这个数据为一个数组而不是一个泛型的列表，这样就提供了SOAP格式化器的兼容性：

```
[Serializable] public class Team : ISerializable
{
  public string Name;
  public List<Person> Players;

  public virtual void GetObjectData (SerializationInfo si,StreamingContext sc)
  {
    si.AddValue ("Name", Name);
    si.AddValue ("PlayerData", Players.ToArray());
  }

  public Team() {}

  protected Team (SerializationInfo si, StreamingContext sc)
  {
    Name = si.GetString ("Name");

    // 序列化Players到一个与序列匹配数组：
    Person[] a = (Person[]) si.GetValue ("PlayerData", typeof (Person[]));

    // 用数组a创建新的List：
    Players = new List<Person> (a);
  }
}
```

为了一些常用的类型，SerializationInfo类包含一些“Get”方法，例如GetString，这样是为了使序列化写起来更容易。如果指定没有数据的名称时就会有一个异常抛出。这通常会发生在代码序列化和反序列化时存在版本不匹配的情况。例如要添加一个额外的字段，但是未考虑序列化旧有的实例。为了解决这个问题，我们可以：

- 在处理后续版本中新增的数据成员的代码块中添加异常处理。

• 实现自己的版本号系统，例如：

```
public string MyNewField;

public virtual void GetObjectData (SerializationInfo si,StreamingContext sc)
{
  si.AddValue ("_version", 2);
  si.AddValue ("MyNewField", MyNewField);
  ...
}

protected Team (SerializationInfo si, StreamingContext sc)
{
  int version = si.GetInt32 ("_version");
  if (version >= 2) MyNewField = si.GetString ("MyNewField");
  ...
}
```

继承可序列化类

在前面的示例中，我们密封了依赖于序列化属性的类。为了说明原因，考虑下面的类结构：

```
[Serializable] public class Person
{
  public string Name;
  public int Age;
}

[Serializable] public sealed class Student : Person
{
  public string Course;
}
```

在这个示例中，Person和Student都可以序列化，两个类都使用默认的运行时序列化行为，因为它们都没有实现ISerializable。

现在假设开发Person的人员由于某些原因决定实现ISerializable，并提供了反序列化构造方法来控制Person的序列化。新版本的Person如下所示：

```
[Serializable] public class Person : ISerializable
{
  public string Name;
  public int Age;

  public virtual void GetObjectData (SerializationInfo si,StreamingContext sc)
  {
    si.AddValue ("Name", Name);
    si.AddValue ("Age", Age);
  }

  protected Person (SerializationInfo si, StreamingContext sc)
  {
    Name = si.GetString ("Name");
    Age = si.GetInt32 ("Age");
  }

  public Person() {}
}
```

尽管Person的实例可以正常工作，但是这个变化破坏了Student实例的序列化。序列化一个Student实例看上去会成功，但是由于Person上的ISerializable.GetObjectData的实现不知道Student继承类型的成员，所以Student上的Course字段不会被保存到序列化流中。并且反序列化Student实例会抛出一个异常，因为运行时要在Student上寻找一个反序列化构造方法。

这个问题的解决方案是从一开始就对公开和不密封的类实现ISerializable。对于internal的类，这一点不是很重要，因为可以很容易地根据需要来修改子类）。

如果我们开始按上面的示例来写Person类，那么Student类如下所示：

```
[Serializable]
public class Student : Person
{
  public string Course;

  public override void GetObjectData (SerializationInfo si,StreamingContext sc)
  {
    base.GetObjectData (si, sc);
    si.AddValue ("Course", Course);
  }

  protected Student (SerializationInfo si, StreamingContext sc): base (si, sc)
  {
    Course = si.GetString ("Course");
  }

  public Student() {}
}
```

17.8 XML序列化

Framework提供了专门的XML序列化引擎，即在System.Xml.Serialization命名空间下的XmlSerializer。它适合把.NET类型序列化为XML文件，它也被ASMX Web服务隐式地使用。

和二进制类似，可以使用以下两种方式：

- 在类型上使用定义在System.Xml.Serialization上的属性。
- 实现IXmlSerializable。

然而不同于二进制引擎，实现接口（例如IXmlSerializable）就会完全避开引擎，要完全使用XmlReader和XmlWriter来实现序列化。

17.8.1 基于属性的序列化入门

为了使用XmlSerializer，要实例化它，并调用Serialize和Deserialize方法传入Stream和对象实例。为了说明这一点，假设我们定义了下面的类：

```
public class Person
{
  public string Name;
  public int Age;
}
```

下面的代码保存一个Person到一个XML文件，然后恢复它：

```
Person p = new Person();
p.Name = "Stacey"; p.Age = 30;
XmlSerializer xs = new XmlSerializer (typeof (Person));

using (Stream s = File.Create ("person.xml"))
  xs.Serialize (s, p);

Person p2;
using (Stream s = File.OpenRead ("person.xml"))
  p2 = (Person) xs.Deserialize (s);

Console.WriteLine (p2.Name + " " + p2.Age); // Stacey 30
```

Serialize和Deserialize方法可以与Stream、XmlWriter/XmlReader或者TextWriter/TextReader一起工作。下面是输出的结果XML：

```
<?xml version="1.0"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Name>Stacey</Name>
  <Age>30</Age>
</Person>
```

XmlSerializer可以序列化没有标记任何属性的类型，例如Person类型。默认，它会序列化类型上的所有公共字段和属性。可以使用[XmlIgnore]属性来排除不想被序列化的成员：

```
public class Person
{
    ...
    [XmlIgnore] public DateTime DateOfBirth;
}
```

不同于其他两个引擎，XmlSerializer不能识别[OnDeserializing]属性，在反序列化时依赖于一个无参数的构造方法，如果没有无参的构造方法，就会抛出一个异常。在我们的示例中，Person有一个隐式的无参的构造方法。这就意味这个字段初始化器会在反序列化之前执行。

```
public class Person
{
  public bool Valid = true;  // 在反序列化之前执行
}
```

尽管XmlSerializer可以序列化任何类型，但是它会识别以下类型，并且会进行特殊的处理：

- 基本类型，DateTime、TimeSpan、Guid以及这些类型的可空类型版本
- Byte[]（它会被转化为base 64编码）
- 一个XmlAttribute或者XmlElement（它们的内容会被注入到流中）
- 任何实现了IXmlSerializable的类型
- 任何集合类型

XML反序列化器允许版本容差：如果缺少元素或属性，或者有多余的数据出现，它都可以正常工作。

1. 属性、名称、命名空间

字段和属性默认都被序列化为XML元素。可以要求序列化器像下面这样使用属性：

```
[XmlAttribute] public int Age;
```

可以像下面这样来控制元素或属性的名称：

```
public class Person
{
  [XmlElement ("FirstName")] public string Name;
  [XmlAttribute ("RoughAge")] public int Age;
}
```

这是序列化后的结果：

```
<Person RoughAge="30" ...>
  <FirstName>Stacey</FirstName>
</Person>
```

默认的XML命名空间为空（不同于数据契约序列化器使用类型的命名空间）。为了指定一个XML命名空间，[XmlElement]和[XmlAttribute]都接受一个Namespace的参数。也可以对类型本身使用[XmlRoot]来给它分配名称和命名空间：

```
[XmlRoot ("Candidate", Namespace = "http://mynamespace/test/")]
public class Person { ... }
```

这样就会命名person元素为Candidate，同时给这个元素及其子元素赋予一个命名空间。

2. XML元素顺序

XmlSerializer会按照成员在类中定义的顺序写元素。可以通过在XmlElement属性上指定Order值来改变这个顺序：

```
public class Person
{
  [XmlElement(Order = 2)] public string Name;
  [XmlElement(Order = 1)] public int Age;
}
```

一旦使用了Order，所有要序列化的成员都得使用。

而反序列化器并不关心元素的顺序，不管元素以任何顺序出现，类型总能够被恰当地反序列化。

17.8.2 子类和子对象

1. 继承根类型

假如根类型有两个子类，像下面这样：

```
public class Person { public string Name; }

public class Student : Person { }
public class Teacher : Person { }
```

定义可重用的方法来序列化根类型：

```
public void SerializePerson (Person p, string path)
{
  XmlSerializer xs = new XmlSerializer (typeof (Person));
  using (Stream s = File.Create (path))
    xs.Serialize (s, p);
}
```

为了让这个方法也能够适用于Student和Teacher，必须让XmlSerializer知道这两个子类。有两种方式实现这个功能。第一种方式就是用[XmlInclude]属性来注册每个子类：

```
[XmlInclude(typeof(Student))]
[XmlInclude(typeof(Teacher))]
public class Person { public string Name; }
```

第二种方式是在构造XmlSerializer时指定每个子类：

```
XmlSerializer xs = new XmlSerializer(typeof(Person),
                       new Type[] { typeof(Student), typeof(Teacher) });
```

在这两种情况下，序列化器通过type属性来记录子类的类型（就像数据契约格式化器一样）：

```
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:type="Student">
  <Name>Stacey</Name>
</Person>
```

然后反序列化器就可以从这个属性知道要去实例化一个Student而不是一个Person。

提示：可以在子类上使用[XmlType]控制输出XML中type属性的名字：

```
[XmlType("Candidate")]
public class Student : Person { }
```

生成的结果如下：

```
<Person xmlns:xsi="..."
        xsi:type="Candidate">
```

2. 序列化子对象

XmlSerializer会自动地递归对象引用，例如Person中的HomeAddress字段：

```
public class Person
{
   public string Name;
   public Address HomeAddress = new Address();
}

public class Address { public string Street, PostCode; }
```

对于下面的值：

```
Person p = new Person(); p.Name = "Stacey";
```

```
p.HomeAddress.Street = "Odo St";
p.HomeAddress.PostCode = "6020";
```

序列化器会生成下面的XML：

```
<Person ... >
  <Name>Stacey</Name>
  <HomeAddress>
    <Street>Odo St</Street>
    <PostCode>6020</PostCode>
  </HomeAddress>
</Person>
```

警告： 如果有两个属性或字段引用了相同的对象，那么这个对象会被序列化两次。如果想保留引用相等性，必须使用其他的序列化引擎。

3. 继承子对象

如果需要序列化引用了Address的子类的Person对象，像下面这样：

```
public class Address { public string Street, PostCode; }
public class USAddress : Address { }
public class AUAddress : Address { }

public class Person
{
  public string Name;
  public Address HomeAddress = new USAddress();
}
```

有两种完全不同的方式来处理，取决于怎样构建XML。如果想要元素的名称总是和字段或属性的名称匹配，同时要在XML中使用type属性记录子属性或字段的类型，像下面这样：

```
<Person ...>
  ...
  <HomeAddress xsi:type="USAddress">
    ...
  </HomeAddress>
</Person>
```

可以使用[XmlInclude]来注册Address的每个子类：

```
[XmlInclude (typeof (AUAddress))]
[XmlInclude (typeof (USAddress))]
public class Address
{
  public string Street, PostCode;
}
```

如果想让元素的名称能够反映子类的名称，期望达到下面的效果：

```
<Person ...>
  ...
  <USAddress>
```

```
    ...
  </USAddress>
</Person>
```

应该在字段或属性上设置多个[XmlElement]属性：

```
public class Person
{
  public string Name;

  [XmlElement ("Address", typeof (Address))]
  [XmlElement ("AUAddress", typeof (AUAddress))]
  [XmlElement ("USAddress", typeof (USAddress))]
  public Address HomeAddress = new USAddress();
}
```

每个[XmlElement]属性映射一个元素名称到一个类型。如果要使用这种方式，就不再需要在Address类型上设置[XmlInclude]属性（尽管它们的存在不会破坏序列化）。

提示：如果在[XmlElement]属性上省略了名称（只指定了类型），那么会使用类型的默认名称（受[XmlType]影响，而不是[XmlRoot]）。

17.8.3 序列化集合

XmlSerializer识别和序列化具体的集合类型，而不需要其他干涉：

```
public class Person
{
  public string Name;
  public List<Address> Addresses = new List<Address>();
}

public class Address { public string Street, PostCode; }
```

序列化后的结果如下：

```
<Person ... >
  <Name>...</Name>
  <Addresses>
    <Address>
      <Street>...</Street>
      <Postcode>...</Postcode>
    </Address>
   <Address>
     <Street>...</Street>
     <Postcode>...</Postcode>
   </Address>
    ...
  </Addresses>
</Person>
```

[XmlArray]属性可以重命名外部元素即集合属性对应的元素，例如Addresses。

[XmlArrayItem]属性允许你重命名内部元素即集合中每个项对应的元素，例如Address元素。

序列化

例如下面的类：

```
public class Person
{
  public string Name;

  [XmlArray ("PreviousAddresses")]
  [XmlArrayItem ("Location")]
  public List<Address> Addresses = new List<Address>();
}
```

序列化后结果如下：

```
<Person ... >
  <Name>...</Name>
  <PreviousAddresses>
    <Location>
      <Street>...</Street>
      <Postcode>...</Postcode>
    </Location>
    <Location>
      <Street>...</Street>
      <Postcode>...</Postcode>
    </Location>
    ...
  </PreviousAddresses>
</Person>
```

[XmlArray]和[XmlArrayItem]属性也允许指定XML的命名空间。

而为了序列化时不产生外部元素，例如：

```
<Person ... >
  <Name>...</Name>
  <Address>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </Address>
  <Address>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </Address>
</Person>
```

可以添加[XmlElement]属性到集合类型的字段或属性上：

```
public class Person
{
  ...
  [XmlElement ("Address")]
  public List<Address> Addresses = new List<Address>();
}
```

处理子类集合元素

处理子类集合元素的规则也自然地遵循其他子类规则。使用type属性来编码一个子类元素，例如：

```
<Person ... >
  <Name>...</Name>
  <Addresses>
    <Address xsi:type="AUAddress">
    ...
```

像前面一样在基类Addess上添加[XmlInclude]属性。不管是否要阻止外部元素的序列化，它都可以工作。

如果想让子类元素也根据其类型来命名，例如：

```
<Person ... >
  <Name>...</Name>
  <! start of optional outer element >
  <AUAddress>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </AUAddress>
  <USAddress>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </USAddress>
  <! end of optional outer element >
</Person>
```

就必须在集合类型的字段或属性上设置多个[XmlArrayItem]或者[XmlElement]属性。

如果想包含外部集合元素，就要设置多个[XmlArrayItem]属性：

```
[XmlArrayItem ("Address", typeof (Address))]
[XmlArrayItem ("AUAddress", typeof (AUAddress))]
[XmlArrayItem ("USAddress", typeof (USAddress))]
public List<Address> Addresses = new List<Address>();
```

如果想排除外部集合元素，就要设置多个[XmlElement]属性：

```
[XmlElement ("Address", typeof (Address))]
[XmlElement ("AUAddress", typeof (AUAddress))]
[XmlElement ("USAddress", typeof (USAddress))]
public List<Address> Addresses = new List<Address>();
```

序列化

17.8.4 IXmlSerializable接口

尽管基于属性的XML序列化是灵活的，但是它也有限制。例如，不能添加序列化钩子，也不能序列化非公有的成员。如果XML以不同方式出现相同的元素或属性，那么它就非常难以使用。

对于最后一个问题，通过在XmlSerializer构造方法传入一个XmlAttributeOverrides对象可以突破这个限制。但是如果要通过一种势在必行且更容易的方式解决时，这就需要IXmlSerializable：

```
public interface IXmlSerializable
{
  XmlSchema GetSchema();
  void ReadXml (XmlReader reader);
  void WriteXml (XmlWriter writer);
}
```

实现这个接口可以完全控制XML的读写。

提示： 一个实现IXmlSerializable的集合类在序列化集合时会绕过XmlSerializer的规则。如果不想序列化一些额外字段或属性时，这会比较有用。

实现IXmlSerializable的规则如下：

- ReadXml应该读取最外层起始元素，然后读取内容，最后才是最外层结束元素。
- WriteXml应该只写入内容。

例如：

```
using System;
using System.Xml;
using System.Xml.Schema;
using System.Xml.Serialization;

public class Address : IXmlSerializable
{
  public string Street, PostCode;

  public XmlSchema GetSchema() { return null; }

  public void ReadXml(XmlReader reader)
  {
    reader.ReadStartElement();
    Street = reader.ReadElementContentAsString("Street", "");
    PostCode = reader.ReadElementContentAsString("PostCode", "");
    reader.ReadEndElement();
  }

  public void WriteXml(XmlWriter writer)
  {
    writer.WriteElementString("Street", Street);
   writer.WriteElementString("PostCode", PostCode);
  }
}
```

通过XmlSerializer序列化和反序列化一个Address时会自动地调用WriteXml和ReadXml方法。如果Person定义成下面：

```
public class Person
{
  public string Name;
  public Address HomeAddress;
}
```

在序列化HomeAddress字段时，IXmlSerializable会有选择性地被调用。

在第11章的第一节我们介绍过XmlReader和XmlWriter类，并且在“使用XmlReader/XmlWriter的模式”部分我们也提供了使用IXmlSerializable的示例。

第18章 程序集

程序集是.NET中的基本部署单元，也是所有类的容器。程序集包含已编译的类和它们的IL代码、运行时资源，以及用于控制版本、安全性和引用其他程序集的信息。程序集也为类解析和安全许可定义了边界。一般来说，一个程序集包含单个PE（Windows Portable Executable，可移植的执行体）文件，如果是应用程序，则带有*.exe*扩展名；如果是可重用的库，则扩展名为*.dll*。

本章中的大部分类来自以下命名空间：

```
System.Reflection
System.Resources
System.Globalization
```

18.1 程序集包含的内容

程序集包含4项内容：

一个程序集清单
向.NET运行时提供信息，例如程序集的名称、版本、请求的权限以及引用的其他程序集。

一个应用程序清单
向操作系统提供信息，例如程序集应该被如何部署和是否需要管理提升。

一些已编译的类
程序集中定义的类的IL代码和元数据。

资源
嵌入程序集中的其他数据，例如图像和可本地化的文本。

所有这些内容中，只有程序集清单是必需的，尽管程序集几乎总是包含已编译的类。

程序集不管是可执行文件还是库，结构是类似的。主要的不同点是，可执行文件定义一个入口点。

18.1.1 程序集清单

程序集清单有两个目的：

- 向托管宿主环境描述程序集。
- 到程序集中模块、类和资源的目录。

因此，程序集是自描述的。消费者可以发现程序集的数据、类和函数等所有内容，无需额外的文件。

提示： 程序集清单不是显式地添加到程序集的，而是作为编译的一部分自动嵌入到程序集中的。

下面总结了程序集清单中存储的主要数据：

- 程序集的简单名称
- 版本号（AssemblyVersion）
- 程序集的公共密钥和已签名的散列（如果是强命名的）
- 一系列引用的程序集，包括它们的版本和公共密钥
- 组成程序集的一系列模块
- 程序集中定义的一系列类和包含每个类的模块
- 一组可选的由程序集要求或拒绝的安全权限（AssemblyPermission）
- 附属程序集针对的文化（AssemblyCulture）

清单也可以存储以下信息数据：

- 完整的标题和描述（AssemblyTitle和AssemblyDescription）
- 公司和版权信息（AssemblyCompany和AssemblyCopyright）
- 显示版本（AssemblyInformationalVersion）
- 自定义数据的其他属性

这些数据有些来自提供给编译器的参数，例如一系列引用的程序集或签署程序集的公共密钥。其他的数据来自程序集属性（括号中的内容）。

提示： 可以利用.NET工具*ildasm.exe*查看程序集清单的内容。第18章将介绍如何使用反射以编程方式实现相同的操作。

指定程序集属性

可以利用程序集属性指定绝大部分清单内容。例如：

```
[assembly: AssemblyCopyright ("\x00a9 Corp Ltd. All rights reserved.")]
[assembly: AssemblyVersion ("2.3.2.1")]
```

这些声明通常都定义在项目的一个文件中。Visual Studio为此对每个新C#项目都在*Properties*文件夹中自动创建一个名为*AssemblyInfo.cs*的文件，预定义了一组默认的程序集属性，为进一步的自定义提供起点。

18.1.2 应用程序清单

应用程序清单是一个XML文件，它向操作系统提供关于程序集的信息。如果存在的话，应用程序清

单在.NET托管宿主环境加载程序集之前被读取和处理，因而可以影响操作系统如何启动应用程序的进程。

应用程序清单在XML命名空间urn:schemas-microsoft-com:asm.v1中具有一个根元素assembly：

```
<?xml version="1.0" encoding="utf-8"?>
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1">
  <!-- contents of manifest -->
</assembly>
```

当程序集在Windows Vista中运行时，以下清单指示操作系统请求管理提升：

```
<?xml version="1.0" encoding="utf-8"?>
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1">
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges>
        <requestedExecutionLevel level="requireAdministrator" />
      </requestedPrivileges>
    </security>
  </trustInfo>
</assembly>
```

第21章将介绍请求管理提升的后果。

Metro应用有更详细的配置清单，具体见Package.appxmanifest文件。它包含程序功能声明，它决定了操作系统所分配的权限。编辑这个文件的最简单方法是使用Visual Studio，双击配置清单文件就可以显示编辑界面。

部署应用程序清单

可以用两种方式部署.NET应用程序清单：

- 作为程序集所在文件夹中的一个特殊命名的文件
- 嵌入程序集中

作为一个单独的文件，其名称必须匹配程序集的名称，后缀为*.manifest*。如果程序集名为*MyApp.exe*，那么它的应用程序清单就应该叫做*MyApp.exe.manifest*。

要将应用程序清单文件嵌入程序集中，首先构建程序集，然后像下面这样调用.NET mt工具：

```
mt -manifest MyApp.exe.manifest -outputresource:MyApp.exe;#1
```

提示： .NET工具*ildasm.exe*对嵌入式应用程序清单的存在视而不见。但是如果在Solution Explorer中双击程序集，Visual Studio会指出嵌入式应用程序清单是否存在。

18.1.3 模块

程序集的内容实际上存储在一个或多个称为模块的中间容器中。一个模块对应于一个包含程序集内容的文件。采用额外的容器层的原因是，为了在构建包含多种编程语言中编译的代码的程序集时，允许程序集跨多个文件，这是一个很有用的特性。

图19-1展示了一个带有单个模块的程序集。图19-2展示了一个多文件程序集。在多文件程序集中，主模块总是包含清单；其他的模块可以包含IL和资源。清单描述组成程序集的所有其他模块的相对位置。

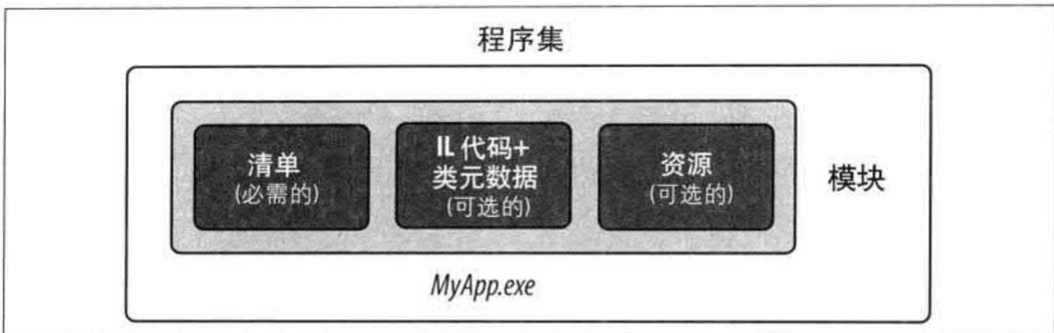


图18-1：单文件程序集

多文件程序集必须从命令行编译，Visual Studio中不支持。为了编译程序集，需要利用/t开关调用csc编译器来创建每个模块，然后再用程序集链接器工具*al.exe*将它们链接起来。

尽管很少有需要多文件程序集的情况，即使在处理单模块程序集时，但是时常需要了解模块这一额外的容器层。主要应用场景跟反射有关（见第19章中的“反射程序集”和“发出程序集和类”）。

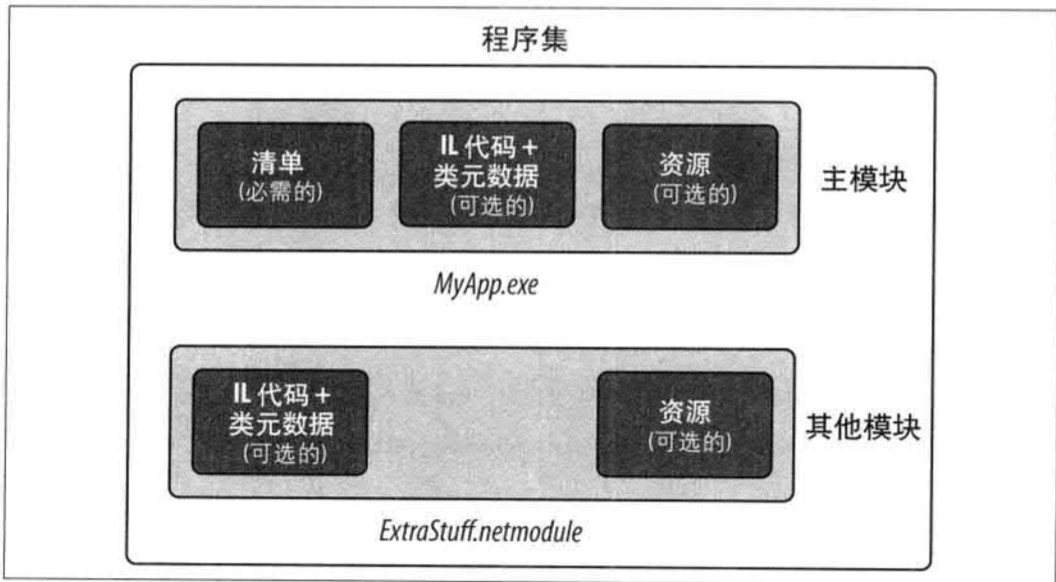


图18-2：多文件程序集

18.1.4 程序集类

System.Reflection中的Assembly类是在运行时访问程序集元数据的入口。有很多方式可以获得程序集对象，最简单的方式是通过Type的Assembly属性：

```
Assembly a = typeof (Program).Assembly;
```

也可以通过调用Assembly的静态方法来获得Assembly对象：

GetExecutingAssembly
: 返回定义当前正在执行的函数的程序集。

GetCallingAssembly
: 跟GetExecutingAssembly执行相同的操作，但是针对的是调用当前正在执行的函数的函数。

GetEntryAssembly
: 返回定义应用程序初始入口方法的程序集。

一旦有了Assembly对象，就可以使用它的属性和方法来查询程序集的元数据和反射它的类。表18-1概述了这些函数。

表18-1：程序集成员

函数	目的	见章节
FullName、GetName	返回完全限定的名称或者AssemblyName对象	“程序集名称”
CodeBase、Location	程序集文件的位置	“解析和加载程序集”
Load、LoadFrom、LoadFile	手动将程序集加载到当前应用程序域中	“解析和加载程序集”
GlobalAssemblyCache	指出程序集是否定义在GAC中	“全局程序集高速缓存”
GetSatelliteAssembly	找到给定文化的卫星程序集	“资源和卫星程序集”
GetType、GetTypes	返回定义在程序集中的一个或所有类	“反射和激活类”
EntryPoint	返回应用程序的入口方法，例如MethodInfo	“反射和调用成员”
GetModules、ManifestModule	返回程序集的所有模块或主模块	“反射程序集”
GetCustomAttributes	返回程序集的属性	“处理属性”

18.2 强名称和程序集签名

强命名的程序集具有唯一的、不可更改的身份。通过向清单添加以下两类元数据来实现：

- 属于程序集创作者的唯一编号
- 程序集的已签名散列，证实程序集产生的唯一编号持有者

这需要一个公共/私有密钥对。公共密钥提供唯一的身份识别号，私有密钥帮助签名。

提示： 强名称签名不同于*Authenticode*签名。本章稍后将介绍Authenticode。

公共密钥对于保证程序集引用的唯一性有价值：强命名的程序集将公共密钥合并到它的身份中。签名对于安全性有价值，它防止恶意人员篡改程序集。没有私有密钥，无法发布程序集的修改版本时不出现签名中断（导致加载时错误）。当然，有些人会利用另外的密钥对来重新签名程序集，但是这会给

程序集另外一个身份。任何引用初始程序集的应用程序都应该避免这种密钥，因为公共密钥标记被写入引用中。

警告：向弱命名的程序集添加一个强名称会更改它的身份。因此，有必要一开始就给生产型程序集（Production Assembly）命名一个强名称。

强命名的程序集也可以注册在GAC中。

18.2.1 如何强命名程序集

要给程序集命名一个强名称，首先利用实用工具*sn.exe*生成一个公共/私有密钥对：

```
sn.exe -k MyKeyPair.snk
```

这会生成一个新的密钥对，并存储在名为*MyApp.snk*的文件中。如果后来丢失了这个文件，就永远不能以同一个身份重新编译此程序集了。

然后利用`/keyfile`开关进行编译：

```
csc.exe /keyfile:MyKeyPair.snk Program.cs
```

Visual Studio在Project Properties窗口中完成这两步操作。

警告：强命名的程序集不能引用弱命名的程序集。这是要强命名所有生产型程序集的另一个重要原因。

同一个密钥对可以对多个程序集进行签名，如果其简单名称不同，那么它们仍然具有不同的身份。至于一个组织中选择使用多少密钥对文件，取决于很多因素。每个程序集具有一个独立的密钥对是有利的，在以后转移某个特定应用程序（以及它引用的程序集）的所有权时，可以做到最小暴露。但是使得创建可以识别所有程序集的安全策略更难了，也使得验证动态加载的程序集更为困难了。

提示：在C# 2.0之前，编译器不支持`/keyfile`开关，应该利用`AssemblyKeyFile`属性指定一个密钥文件。这带来一个安全风险，因为到密钥文件的路径会保持嵌入在程序集的元数据中。例如，利用*ildasm*可以非常容易地看到，被用来签名CLR 1.1中的*mscorlib*的密钥文件路径如下所示：

```
F:\qfe\Tools\devdiv\EcmaPublicKey.snk
```

显然，需要访问Microsoft的.NET框架构建机器上的这个文件夹，才能利用该信息的优势！

18.2.2 延迟签名

在有数百个开发人员的组织中，你可能想要限制对程序集进行签名的密钥对的访问，原因有两个：

- 如果密钥对泄漏，你的程序集就不再是不可篡改的了。
- 测试程序集如果已签名和泄漏，就会被恶意地宣称为真正的程序集。

但是，扣留开发人员的密钥对意味着，他们无法用自己正确的身份去编译和测试程序集。延迟签名是一个用于解决这个问题的系统。

延迟签名的程序集用正确的公共密钥进行标记，但是没有用私有密钥签名。延迟签名的程序集相当于被篡改的程序集，通常会被CLR拒绝。但是，开发人员告诉CLR忽略对这一台机器上延迟签名程序集的验证，允许未签名的程序集运行。到最终部署时，私有密钥持有者再用真正的密钥对重新签名程序集。

要延迟签名，需要一个只包含公共密钥的文件。可以通过利用-p开关调用sn从密钥对抽取此文件：

```
sn -k KeyPair.snk
sn -p KeyPair.snk PublicKeyOnly.pk
```

`KeyPair.snk`保存得很安全，*PublicKeyOnly.pk*是免费发布的。

提示：也可以利用-e开关从现有已签名程序集中得到*PublicKeyOnly.pk*：

```
sn -e YourLibrary.dll PublicKeyOnly.pk
```

然后通过利用`/delaysign+`开关调用`csc`，可以用*PublicKeyOnly.pk*进行延迟签名：

```
csc /delaysign+ /keyfile: PublicKeyOnly.pk /target:library YourLibrary.cs
```

选中“Delay sign”复选框，Visual Studio也能完成这件事情。

下一步是告诉.NET运行时，在运行延迟签名程序集的开发计算机上跳过程序集身份验证。这可以按每个程序集或者每个公共密钥来做到，方法是利用`Vr`开关调用`sn`工具：

```
sn -Vr YourLibrary.dll
```

警告：Visual Studio不自动执行这一步。必须从命令行手动禁用程序集验证，否则，程序集将不会执行。

最后一步是在部署之前完全签名程序集。这就是利用实际的签名（只可以用对私有密钥的访问权产生）替换空签名。为此，利用-R开关调用sn：

```
sn -R YourLibrary.dll KeyPair.snk
```

然后可以在开发机器上恢复程序集验证，如下所示：

```
sn -Vu YourLibrary.dll
```

不需要重新编译任何引用延迟签名程序集的应用程序，因为只改变了程序集的签名，没有改变它的身份。

18.3 程序集名称

程序集的身份包含四种来自其清单的元数据：

- 它的简单名称
- 它的版本（如果未指定，就是 0.0.0.0）
- 它的文化（如果不是卫星程序集，就是 neutral）
- 它的公共密钥标记（如果不是强命名的，就是 null）

简单名称不是来自任何属性，而是来自它最初被编译成的文件的名称（不带扩展名）。所以，*System.Xml.dll*程序集的简单名称是“System.Xml”。文件重命名不会改变程序集的简单名称。

版本号来自AssemblyVersion属性。它是一个由4个部分组成的字符串，如下所示：

```
major.minor.build.revision
```

可以像下面这样指定版本号：

```
[assembly: AssemblyVersion ("2.5.6.7")]
```

文化来自AssemblyCulture属性，并且适用于卫星程序集，后面“资源和卫星程序集”一节中有介绍。

公共密钥标记来自在编译时通过/keyfile开关提供的密钥对，正如前面我们在“强名称和程序集签名”一节中介绍的。

18.3.1 完全限定名称

完全限定程序集名称是一个包含4个身份识别组件的字符串，格式如下：

simple-name, Version=**version**, Culture=**culture**, PublicKeyToken=*public-key*

例如，*System.Xml.dll*的完全限定名称是：

```
"System.Xml, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"
```

如果程序集没有AssemblyVersion属性，则版本显示为“0.0.0.0”。如果未签名，则其公共密钥标记显示为“null”。

Assembly对象的FullName属性返回它的完全限定名称。编译器在清单中记录程序集引用时总是使用完全限定名称。

提示： 完全限定程序集名称不包含它在磁盘上的目录路径。定位在另一个目录中的程序集完全是另一码事，在“解析和加载程序集”一节单独介绍。

18.3.2 AssemblyName类

AssemblyName类的完全限定程序集名称的每一个组件都具有一个类型化属性。AssemblyName有两个目的：

- 解析或构建完全限定程序集名称。
- 存储一些额外的数据，以帮助解析（寻找）程序集。

可以通过以下三种方式获得AssemblyName：

- 实例化一个AssemblyName，提供完全限定名称。
- 在一个现有Assembly上调用GetName。
- 调用AssemblyName.GetAssemblyName，提供到磁盘上程序集文件的路径。

可以不用任何参数实例化一个AssemblyName，然后设置它的每个属性以构建完全限定名称。以这种方式构造的AssemblyName是易变的。

下面是它的一些重要属性和方法：

```
string       FullName    { get; }         // 完全限定名称
string       Name        { get; set; }    // 简单名称
Version      Version     { get; set; }    // 程序集版本
CultureInfo  CultureInfo { get; set; }    // 针对卫星程序集
string       CodeBase    { get; set; }    // 位置

byte[]       GetPublicKey();              // 160字节
void         SetPublicKey (byte[] key);
byte[]       GetPublicKeyToken();         // 8字节版本
void         SetPublicKeyToken (byte[] publicKeyToken);
```

Version本身是一个强类型化的表示，具有Major、Minor、Build和版本号属性。GetPublicKey返回完全加密的公共密钥；GetPublicKeyToken返回建立身份时使用的最后8个字节。

要使用AssemblyName来获得程序集的简单名称：

```
Console.WriteLine (typeof (string).Assembly.GetName().Name); // mscorlib
```

要得到程序集版本：

```
string v = myAssembly.GetName().Version.ToString();
```

我们将在本章的“解析和加载程序集”一节中介绍CodeBase属性。

18.3.3 程序集信息版本和文件版本

由于版本是程序集名称的一个有机部分，所以改变AssemblyVersion属性就会改变程序集的身份。这将影响与引用程序集的兼容性，在不间断的更新中会出现意想不到的情况。要解决这个问题，有以下两个独立的程序集级别的属性用于表示与版本相关的信息，两者都被CLR忽略：

AssemblyInformationalVersion

: 显示给最终用户的版本。这在“Windows File Properties”对话框中作为“Product Version”出现。可以包含任何字符串，例如说“5.1 Beta 2”。通常，应用程序中的所有程序集会被分配相同的信息版本号。

AssemblyFileVersion

: 用于引用此程序集的构建号。这在“Windows File Properties”对话框中作为“File Version”出现。跟AssemblyVersion一样，它必须包含一个字符串，最多由4个用句点分隔的数字组成。

18.4 Authenticode签名

*Authenticode*是一个代码签名系统，其目的是证明发行商的身份。Authenticode和强名称签名是独立的，可以用任何一个或同时用两个系统对程序集进行签名。

尽管强名称签名可以证明程序集A、B和C来自相同的一方（假设私有密钥没有泄漏），但是它不知道到底是哪一方。为了知道这一方是Joe Albahari还是Microsoft Corporation，就需要用到Authenticode。

Authenticode在从Internet下载程序时有用，因为它可以证明程序来自由Certificate Authority命名的一方，并且在此期间没有被修改。事实上，Authenticode的主要好处之一是，它防止在第一次运行下载的应用程序时出现图18-3所示的安全警告。这使得Authenticode特别适用于安装程序或单独的可执行文件。（程序集应用Authenticode的一个比较普遍的原因是，它是Windows Logo程序的一个要求。）

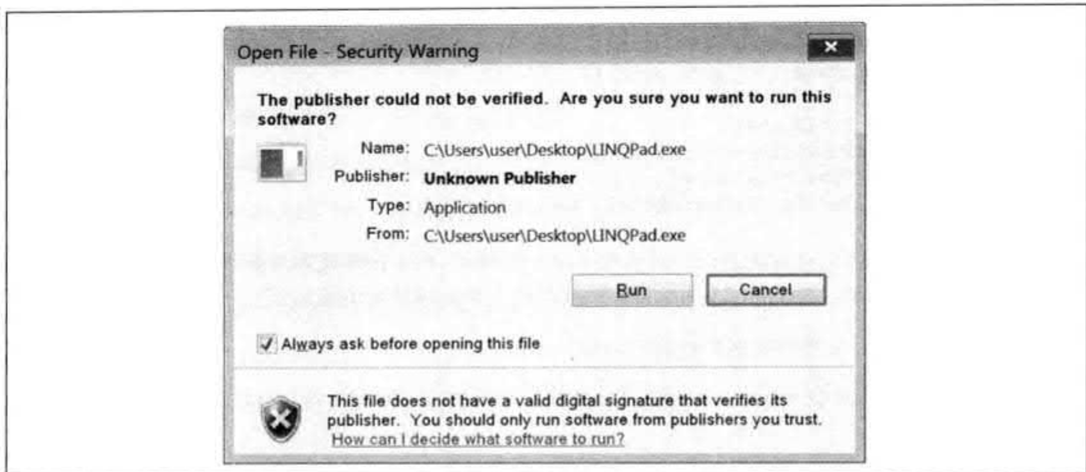


图18-3：未签名文件警告

Authenticode不仅适用于.NET程序集，也适用于未托管可执行文件和二进制文件，例如ActiveX控件或.msi部署文件。但是Authenticode不能保证程序免于恶意软件的攻击，尽管它可以减少这种情况的发生。个人或实体都愿意将其名称（由护照或公司文档证明）附在可执行文件或库后面。

提示： CLR不将Authenticode签名看作程序集身份的一部分。但是它根据需要可以读取和验证Authenticode签名，我们将在后面进行介绍。

利用Authenticode进行签名需要以个人身份或公司身份（公司章程等）作为证据联系Certificate Authority（CA）。一旦CA检查了你的文档，它就会给你发一个X.509代码签名证书，一般有效期为1~5年。这让你能够利用*signtool*实用工具签署程序集。你也可以利用*makecert*实用工具自己制造证书，但是这只对测试有用，因为此证书在其他计算机上无效。

Authenticode依赖于公共密钥基础设施：本质上，它证明程序是利用一个凭证由CA验证的证书签署的。CA本身是受信任的，因为所有CA都被加载到了操作系统（要查看它们，进入Windows控制面板并选择“Internet选项”→“内容”选项卡→“证书”按钮→“受信任的根证书颁发机构”选项卡）。如果泄漏了，CA可以撤销发行商的证书，所以验证Authenticode签名需要定期向CA询问最新的证书撤销列表。

因为Authenticode使用加密签名，所以如果后来有人篡改了文件，那么Authenticode签名就成了无效的。第21章中将讨论加密、散列和签名。

18.4.1 如何用Authenticode签名

1. 获得和安装证书

第一步是从CA获得代码签名证书（见下栏内容）。然后，可以将证书作为密码保护的文件对待，或者将证书加载到计算机的证书存储库中。后一种做法的优点是，签名时可以无需指定密码，避免了在自动构建的脚本或批文件中具有一个可见的密码。

> **从何处获得代码签名证书**
>
> 很多的代码签名CA都已被作为根证书颁发机构预先加载到Windows中了。这些机构包括Comodo、Go Daddy、GlobalSign、thawte和VeriSign。
>
> 还有一个叫做Ksoftware的代理销售商，它目前每年提供99美元的Comodo代码签名。
>
> Ksoftware、Comodo、Go Daddy和GlobalSign发布的Authenticode证书被宣称不太严格，因为它们也会签署诸如Java小程序之类的非Microsoft程序。除此之外，所有供应商的产品在功能上都差不多。
>
> 注意，用于SSL的证书一般都不能用于Authenticode签名（除非使用相同的X.509基础设施）。这在一定程度上是因为，用于SSL的证书是证明域的所有权，Authenticode是证明身份。

要将证书加载到计算机的证书存储库中，可进入Windows控制面板并选择"Internet选项"→"内容"选项卡→"证书"按钮→"导入"。导入完成后，就单击证书上的"查看"按钮，进入"详细信息"选项卡，并复制后面签名时识别证书需要用到的SHA-1散列。

提示： 如果还想对程序集进行强名称签名（强烈推荐），那么必须在Authenticode签名之前进行强名称签名。这是因为，CLR知道Authenticode签名，却不知道强名称签名。所以，如果对一个程序集先进行Authenticode签名，后进行强名称签名，那么Authenticode签名会将添加的CLR强名称看作未经授权的修改，并认为程序集已被篡改。

2. 利用signtool.exe签名

可以利用Visual Studio随附的*signtool*实用工具对程序进行Authenticode签名。如果利用`signwizard`标志调用它，那么它会显示一个UI；否则，可以像下面这样以命令行方式使用它：

```
signtool sign /sha1 (thumbprint) filename
```

如计算机证书存储库中显示的一样，thumbprint指的是证书的指纹。（如果证书是在一个文件中，那么利用`/f`指定文件名，利用`/p`指定密码。）

例如：

```
signtool sign /sha1 ff813c473dc93aaca4bac681df472b037fa220b3 LINQPad.exe
```

也可以利用`/d`和`/du`指定描述和产品URL：

```
... /d LINQPad /du http://www.linqpad.net
```

大多数情况下，还想要指定时间戳服务器。

3. 时间戳

证书过期之后，就不再能够对程序签名了。但是在过期之前签名的程序仍然有效——如果在签名时利用/t开关指定了时间戳服务器的话。为此，CA将提供一个URI，下面是Comodo（或Ksoftware）的URI：

```
... /t http://timestamp.comodoca.com/authenticode
```

4. 验证一个程序已经被签名

查看文件的Authenticode签名最容易的方式是，在Windows资源管理器中（“数字签名”选项卡）查看文件的属性。*Signtool*实用工具也为此提供了一个选项。

18.4.2 Authenticode验证

操作系统和CLR都可能验证Authenticode签名。

Windows在运行标记为“被阻塞”（实际上，这意味着程序从Internet下载之后第一次运行）的程序之前验证Authenticode签名。然后，Authenticode信息的状态显示在图19-3所示的对话框中。

当我们要求程序集证据时，CLR会读取和验证Authenticode签名。下面是具体的做法：

```
Publisher p = someAssembly.Evidence.GetHostEvidence<Publisher>();
```

（System.Security.Policy中的）Publisher类具有一个Certificate属性。如果这返回一个非空值，那么它就已经被Authenticode签名。然后，可以查询该对象，了解证书的详细信息。

警告： 在Framework 4.0之前，CRL会在程序集加载时读取和验证Authenticode签名，而不是等待直到调用GetHostEvidence。这潜在地对性能造成影响，因为Authenticode验证可能会回到CA去更新证书撤销列表——如果存在Internet连接问题的话，这最多会花30秒时间（直到失败）。因此，可能的话，最好避免对.NET 3.5或更早的程序集进行Authenticode签名（但是，签名*.msi*设置文件没有问题）。

不管Framework是什么版本，如果程序具有非法的或无法证实的Authenticode签名，那么CLR将仅仅通过GetHostEvidence使此信息可用：它不会向用户显示警告或阻止程序集运行。

正如我们前面所说，Authenticode签名对程序集的身份或名称没有影响。

18.5 全局程序集高速缓存

作为安装.NET Framework的一部分，在计算机上创建一个中心仓库，用于存储.NET程序集，这就是所谓的全局程序集高速缓存（Global Assembly Cache，GAC）。GAC包含.NET Framework本身的一个集中副本，并且它也可以用来集中自定义的程序集。

选择是否将程序集加载到GAC时，考虑的主要因素跟版本控制相关。对于GAC中的程序集，版本控制集中在机器级别，由计算机管理员控制。对于GAC外的程序集，版本控制以应用程序为基础进行处理，所以每个应用程序关注自己的依赖性和更新问题（通常通过维护它引用的每个程序集的副本来实现）。

在少数机器集中式版本控制真正有益的情况下，GAC很有用。例如，考虑一组相互依赖的插件，每

个插件都引用一些共享的程序集。我们假设每个插件都在自己的目录中，因此就可能存在共享一个程序集的多个副本（也许其中一些比另一些早）。另外，我们还假设，为了效率和类型的兼容性，对每个共享程序集，宿主应用程序只加载一次。对于宿主应用程序来说，程序集解决方案的任务很困难，需要仔细规划并了解程序集加载上下文的细微之处。这里给出的简单解决方案是将共享程序集放入GAC中。这确保了CLR总是做出直观且一致的程序集解决方案选择。

但是在比较典型的情况中，最好避免使用GAC，因为它会增加以下困难：

- XCOPY或ClickOnce部署不再可能；需要一个管理设置文件来安装应用程序。
- 更新GAC中的程序集也需要管理特权。
- GAC的使用让开发和测试复杂化，因为相对于本地副本，*fusion*（CLR的程序集解决方案机制）总是更偏爱GAC程序集。
- 版本控制和并行执行需要一些规划，一个错误就可能中断其他应用程序。

优点是，对于非常大的程序集，GAC可以缩短启动时间，因为CLR只需要在安装时验证一次GAC中程序集的签名，而不是每次加载程序集时都要验证。按百分比来说，如果用*ngen.exe*工具为程序集生成了本机映像（选择非重叠的基地址），就会有这一优势。在MSDN站点可以在线查看一篇描述这些问题的文章，题目是"The Performance Benefits of NGen"。

提示： GAC中的程序集总是完全受信任的，即使是从运行在受限的沙箱中调用程序集。第20章中将进一步讨论。

18.5.1 如何将程序集安装到GAC

要将程序集安装到GAC，第一步是给程序集命名一个强名称。然后，可以使用.NET命令行工具gacutil安装它：

```
gacutil /i MyAssembly.dll
```

如果程序集已经以相同的公共密钥和版本存在于GAC中，那么它会被更新，所以不需要先卸载旧的程序集。

要卸载程序集（注意不带文件扩展名）：

```
gacutil /u MyAssembly
```

也可以指定将安装到GAC的程序集作为Visual Studio中设置项目的一部分。

利用/l开关调用`gacutil`将列出GAC中的所有程序集。也可以利用*mscorcfg* MMC管理单元来实现（Windows→管理工具→Framework Configuration）。

程序集一旦加载到GAC，应用程序就可以引用它了，无需此程序集的本地副本。

警告： 如果存在本地副本，则被忽略并被GAC映像取代。这意味着无法引用或测试库的重新编译的版本，直到更新GAC为止。只要保留程序集的版本和身份，就会如此。

程序集

18.5.2 GAC和版本控制

更改程序集的AssemblyVersion会给它一个新的身份。为了说明这一点，我们假设编写了一个*utils*程序集，版本为1.0.0.0，进行强命名，然后安装到GAC中。然后假设添加了一些新特性，版本更改为1.0.0.1，重新编译并重新安装到GAC中。不是覆盖原来的程序集，而是GAC保留两个版本的程序集。这意味着：

- 当编译另一个使用*utils*的应用程序时，可以选择引用哪个版本。
- 任何以前编译为引用*utils* 1.0.0.0的应用程序将继续引用*utils* 1.0.0.0。

这叫做并行执行。并行执行防止共享程序集被单方更新后出现的“DLL 地狱”：为较旧版本设计的应用程序可能会非预期地中断。

但是，当要向现有程序集应用bug修复或更新时，就会出现冲突。有两种选择：

- 将修复后的程序集以相同版本号重新安装到GAC中。
- 以新版本号编译修复后的程序集并安装到GAC中。

第一种选择的问题是，无法有选择地将更新应用到某些应用程序。要么全部要么全不。第二种选择的问题是，不重新编译，应用程序就不会正常地使用较新的程序集版本。解决的办法是：可以创建一个颁发者策略，允许程序集版本重定向，缺点是增加部署复杂性。

并行执行有益于减轻共享程序集的某些问题。如果完全避免GAC，而不是允许每个应用程序维护其私有版本的*utils*，就可以消除共享程序集的所有问题。

18.6 资源和卫星程序集

应用程序通常不仅仅包含可执行代码，还包含诸如文本、图像或XML文件等内容。这些内容可以表示为程序集中的资源。资源有两个重叠的用例：

- 合并不能进入源代码的数据，例如图像
- 存储在多语言应用程序中可能需要转换的数据

程序集资源最终是一个带有名称的字节流。可以将程序集看作包含一个按字符串排列的字节数组字典。如果我们反汇编一个包含资源*banner.jpg*和*data.xml*的程序集，就会在*ildasm*中看到以下字节流：

```
.mresource public banner.jpg
{
  // 偏移：0x00000F58 长度：0x000004F6
}
.mresource public data.xml
{
  // 偏移：0x00001458 长度：0x0000027E
}
```

在本例中，*banner.jpg*和*data.xml*都作为自己的嵌入式资源直接包含在程序集中。这是最简单的工作方式。

Framework可以通过中间的*.resources*容器添加内容。一些容器包含可能需要转换成不同语言的内容。

本地化的*.resources*可打包为在运行时根据用户的操作系统语言被自动挑选的单个卫星程序集。

图18-4演示了一个程序集，它包含两个直接嵌入的资源和一个名为*welcome.resources*的*.resources*容器，我们为这个容器创建了两个本地化卫星。

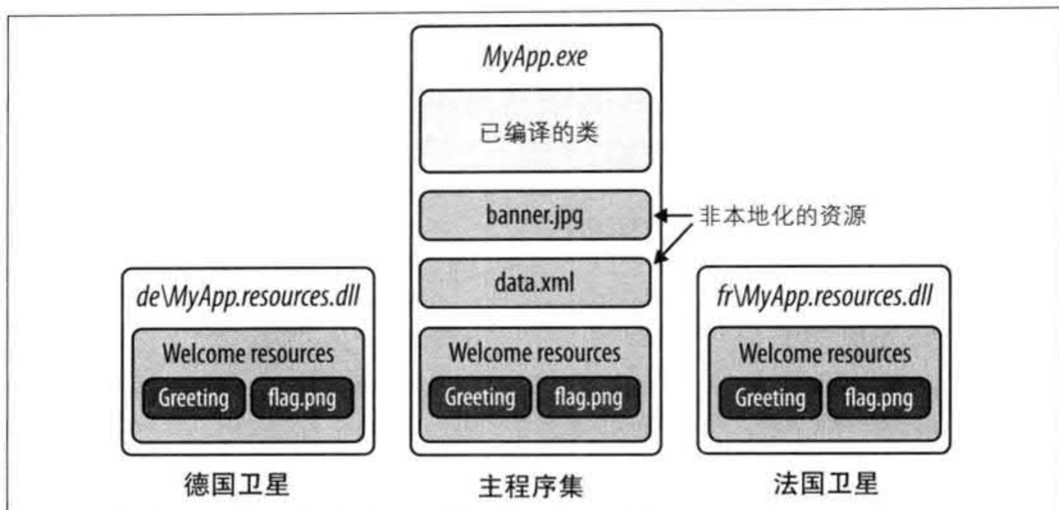


图18-4：资源

18.6.1 直接嵌入资源

提示： Metro应用也不支持在程序集中嵌入资源。相反，要先在部署包中添加所需要的额外文件，然后通过读取应用的Storage文件夹（Package.Current.InstalledLocation），就可以访问这些文件。

要在命令行直接嵌入资源，可在编译时使用/resource开关：

```
csc /resource:banner.jpg /resource:data.xml MyApp.cs
```

可以像下面这样，可选地在程序集中为资源指定一个不同的名称：

```
csc /resource:<file-name>,<resource-name>
```

要使用Visual Studio直接嵌入资源：

- 将文件添加到项目。
- 将构建操作设置为“Embedded Resource”。

Visual Studio总是将项目的默认命名空间作为资源名称的前缀，再加上所含任意子文件夹的名称。所以，如果项目的默认命名空间是Westwind.Reports，文件是*pictures*文件夹中的*banner.jpg*，那么资源名称就是*Westwind.Reports.pictures.banner.jpg*。

警告： 资源名称区分大小写，所以Visual Studio中包含资源的项目子文件夹名称也区分大小写。

要获得一个资源，可以在包含该资源的程序集上调用GetManifestResourceStream，返回一个流，然后可以将其读作任何其他名字：

```
Assembly a = Assembly.GetEntryAssembly();

using (Stream s = a.GetManifestResourceStream ("TestProject.data.xml"))
using (XmlReader r = XmlReader.Create (s))
  ...

System.Drawing.Image image;
using (Stream s = a.GetManifestResourceStream ("TestProject.banner.jpg"))
  image = System.Drawing.Image.FromStream (s);
```

返回的流是可搜索的，所以也可以做下面这件事：

```
byte[] data;
using (Stream s = a.GetManifestResourceStream ("TestProject.banner.jpg"))
  data = new BinaryReader (s).ReadBytes ((int) s.Length);
```

如果已经使用Visual Studio来嵌入资源，那么必须包含基于命名空间的前缀。为了避免错误，可以使用*type*在单独的参数中指定前缀。类的命名空间被用作前缀：

```
using (Stream s = a.GetManifestResourceStream (typeof (X), "XmlData.xml"))
```

x可以是想要获得的资源命名空间中的任何类型（通常是同一项目文件夹中的一个类型）。

警告： 在Visual Studio中将项目条目的构建操作设置为WPF应用程序中的“Resource”不同于将其设置为“Embedded Resource”。前者实际上将条目添加到一个名为*<AssemblyName>.g.resources*的*.resources*文件，可以使用URI作为密钥，通过WPF的Application类访问这个文件的内容。

更容易混淆的是，WPF进一步定义了术语“资源”。静态资源和动态资源都跟程序集资源没有关系。

GetManifestResourceNames返回程序集中所有资源的名称。

18.6.2 .resources文件

*.resources*文件包含的是潜在地可本地化的内容。*.resources*文件最终是程序集中的一个嵌入式资源，就像任何其他类型的文件一样。区别在于必须：

- 首先将内容打包到*.resources*文件中。
- 通过ResourceManager或*pack URI*而不是GetManifestResourceStream访问它的内容。

*.resources*文件的结构形式是二进制的，所以不是可读的；因此，必须依赖于Framework和Visual Studio提供的工具来处理它们。处理字符串或简单数据类的标准方法是使用*.resx*格式，该格式可以通过Visual Studio或resgen工具转换成*.resources*文件。*.resx*格式也适合于针对Windows Forms或ASP.NET应用程序的图像。

在WPF应用程序中，必须对需要由URI引用的图像或类似的内容使用Visual Studio的“Resource”构建操作。无论是否需要本地化，这一点都是适用的。

下面几小节中将讨论如何实现这些操作。

18.6.3 .resx文件

*.resx*文件是一种用于生成*.resources*文件的设计时格式。*.resx*文件使用XML通过名/值对进行构造，如下所示：

```
<root>
  <data name="Greeting">
    <value>hello</value>
  </data>
  <data name="DefaultFontSize" type="System.Int32, mscorlib">
    <value>10</value>
  </data>
</root>
```

要在Visual Studio中创建*.resx*文件，可以添加一个"Resource File"类的项目条目。其他工作都是自动完成的：

- 创建正确的头部。
- 设计器提供用于添加字符串、图像、文件和其他类型的数据。
- *.resx*文件自动转换成*.resources*格式，并在编译时嵌入到程序集中。
- 编写一个类用于以后访问数据。

提示： 资源设计器将图像添加为类型化的Image对象（*System.Drawing.dll*），而不是作为字节数组，这使得它们不适用于WPF应用程序。

---

1. 在命令行创建.resx文件

如果是在命令行工作，那么必须从一个具有有效头部的*.resx*文件开始。最容易的实现方式是以编程方式创建一个简单的*.resx*文件。System.Resources.ResXResourceWriter class（存储在*System.Windows.Forms.dll*程序集中）专门用于完成这项操作：

```
using (ResXResourceWriter w = new ResXResourceWriter ("welcome.resx")) { }
```

从这里，可以继续使用ResXResourceWriter通过调用AddResource添加资源，或者手动编译它编写的*.resx*文件。

处理图像最容易的方式是将文件当作二进制数据，并在检索时将它们转换成图像。这比将它们编码为类型化的Image对象更为普遍。可以在base 64格式的*.resx*文件中包含二进制数据，如下所示：

```
<data name="flag.png" type="System.Byte[], mscorlib">
  <value>Qk32BAAAAAAAAHYAAAAoAAAAMAMDAwACAgIAAAAD/AA....</value>
</data>
```

或者作为对另一个会被resgen读取的文件的引用：

```
<data name="flag.png"
  type="System.Resources.ResXFileRef, System.Windows.Forms">
  <value>flag.png;System.Byte[], mscorlib</value>
</data>
```

完成后，必须通过调用resgen转换*.resx*文件。下面的命令将*welcome.resx*转换成*welcome.resources*：

```
resgen welcome.resx
```

最后一步是在编译时包含*.resources*文件，如下所示：

```
csc /resources:welcome.resources MyApp.cs
```

2. 读取.resources文件

提示： 如果在Visual Studio中创建*.resx*文件，会通过检索其每个条目的属性自动生成一个相同名称的类。

ResourceManager类读取嵌入程序集中的*.resources*文件：

```
ResourceManager r = new ResourceManager ("welcome",
                                        Assembly.GetExecutingAssembly());
```

（如果资源是在Visual Studio中编译的，那么第一个参数必须带有命名空间前缀。）

然后通过强类转换调用GetString或GetObject，可以访问其中的内容：

```
string greeting = r.GetString ("Greeting");
int fontSize = (int) r.GetObject ("DefaultFontSize");
Image image = (Image) r.GetObject ("flag.png");   // (Visual Studio)
byte[] imgData = (byte[]) r.GetObject ("flag.png");  // （命令行）
```

要枚举*.resources*文件的内容：

```
ResourceManager r = new ResourceManager (...);
ResourceSet set = r.GetResourceSet (CultureInfo.CurrentUICulture,
                                    true, true);
foreach (System.Collections.DictionaryEntry entry in set)
  Console.WriteLine (entry.Key);
```

3. 在Visual Studio中创建pack URI资源

在WPF应用程序中，XAML文件能够通过URI访问资源。例如：

```
<Button>
  <Image Height="50" Source="flag.png"/>
</Button>
```

或者，如果资源在另一个程序集中：

```
<Button>
  <Image Height="50" Source="UtilsAssembly;Component/flag.png"/>
</Button>
```

（Component是一个字面关键词。）

要创建能够以这种方式被加载的资源，不能使用*.resx*文件。相反，必须将文件添加到项目，并将它们的构建操作设置为“Resource”（而不是“Embedded Resource”）。Visual Studio然后将它们编译成一个名为*<AssemblyName>.g.resources*的*.resources*文件，此文件包含已编译的XAML（*.baml*）文件。

调用Application.GetResourceStream，采用编程方式加载以URI为密钥的资源：

```
Uri u = new Uri ("flag.png", UriKind.Relative);
using (Stream s = Application.GetResourceStream (u).Stream)
```

注意我们使用了相对URI。也可以按以下格式使用绝对URI（三个逗号不是录入错误）：

```
Uri u = new Uri ("pack://application:,,,/flag.png");
```

如果指定一个Assembly对象，那么可以使用ResourceManager检索内容：

```
Assembly a = Assembly.GetExecutingAssembly();
ResourceManager r = new ResourceManager (a.GetName().Name + ".g", a);
using (Stream s = r.GetStream ("flag.png"))
  ...
```

ResourceManager也可以枚举给定程序集中*.g.resources*文件的内容。

18.6.4 卫星程序集

嵌入*.resources*文件中的数据是可本地化的。

当应用程序运行在以不同语言显示所有内容的Windows版本上时，资源本地化是相关的。为了保持一致性，应用程序也应该使用同一语言。

典型的设置如下：

- 主程序集包含针对默认或*fallback*语言的*.resources*。
- 独立的卫星程序集包含已本地化的转换成不同语言的*.resources*。

当程序集运行时，Framework检查当前操作系统的语言（从CultureInfo.CurrentUICulture）。每当使用ResourceManager请求一个资源时，Framework就寻找一个已本地化的卫星程序集。如果有一个程序集可用并且它包含请求的资源密钥，那么就使用它来取代主程序集的版本。

这意味着，可以简单地通过添加新卫星程序集而增强语言支持，无需更改主程序集。

提示： 卫星程序集不能包含可执行代码，只能包含资源。

卫星程序集部署在程序集文件夹的子目录中，如下所示：

```
programBaseFolder\MyProgram.exe
                 \MyLibrary.exe
                 \XX\MyProgram.resources.dll
                 \XX\MyLibrary.resources.dll
```

*XX*指代两个字母的语言代码（例如，“de”代表德语）或者语言和区域代码（例如，“en-GB”代表大不列颠的英语）。这种命名系统允许CLR自动寻找并加载正确的卫星程序集。

1. 构建卫星程序集

回想一下前一个*.resx*例子，其中包含：

```
<root>
  ...
```

```
    <data name="Greeting"
      <value>hello</value>
    </data>
  </root>
```

然后在运行时检索了问候语，如下所示：

```
ResourceManager r = new ResourceManager ("welcome",Assembly.GetExecutingAssembly());
Console.Write (r.GetString ("Greeting"));
```

假设我们想要该代码运行在德语版Windows上时输出Hallo，首先增加一个名为*welcome.de.resx*的*.resx*文件，它将*hello*替换为*hallo*：

```
<root>
  <data name="Greeting">
    <value>hallo<value>
  </data>
</root>
```

在Visual Studio中，这就是需要做的全部工作。当重新构建时，一个名为*MyApp.resources.dll*的卫星程序集就会自动创建在名叫de的子目录中。

如果是使用命令行，那么调用*resgen*将*.resx*文件转换成*.resources*文件：

```
resgen MyApp.de.resx
```

然后调用`al`构建卫星程序集：

```
al /culture:de /out:MyApp.resources.dll /embed:MyApp.de.resources /t:lib
```

可以指定`/template:MyApp.exe`，以导入主程序集的强名称。

2. 测试卫星程序集

要在不同语言的操作系统上模拟运行，必须使用`Thread`类更改`CurrentUICulture`：

```
System.Threading.Thread.CurrentThread.CurrentUICulture
  = new System.Globalization.CultureInfo ("de");
```

`CultureInfo.CurrentUICulture`是相同属性的只读版本。

提示： 一个实用的测试方法是在仍然可以用英语方式读取的单词中添加ℓΟ¢αℓïʐɘ，但是不使用标准的Roman Unicode字符。

---

3. Visual Studio设计器支持

Visual Studio中的设计器对本地化组件和可视元素提供广泛的支持。WPF设计器具有自己的本地化工作流；其他基于`Component`的设计器使用一个仅在设计时有效的属性，使得卫星程序集像是一个组件或者具有`Language`属性的Windows Forms控件。要为另一种语言进行自定义，只需简单地更改`Language`属性并修改组件即可。控件的所有指定为`Localizable`的属性将永久存储在该语言的*.resx*文件中。任何时候，只需更改`Language`属性，就可以在语言之间切换。

18.6.5 文化和子文化

文化分成文化和子文化。一种文化代表一种特定的语言；一种子文化代表该语言的一个地区变种。Framework遵循RFC1766标准，该标准以两个字母的代码代表文化和子文化。下面是英语文化和德语文化的代码：

```
en
de
```

下面是澳大利亚英语子文化和奥地利德语子文化的代码：

```
en-AU
de-AT
```

在.NET中用System.Globalization.CultureInfo类表示文化。可以检查应用程序的当前文化，如下所示：

```
Console.WriteLine (System.Threading.Thread.CurrentThread.CurrentCulture);
Console.WriteLine (System.Threading.Thread.CurrentThread.CurrentUICulture);
```

在一台已本地化为澳大利亚文化的计算机上运行该代码，演示了两者之间的区别：

```
EN-AU
EN-US
```

CurrentCulture反映Windows控制面板的区域设置，而CurrentUICulture反映操作系统的语言。

区域设置包括诸如时区、货币和日期格式等方面内容。CurrentCulture确定DateTime.Parse之类函数的默认行为。区域设置可以被自定义到不再装配任何特定文化时的那个点。

CurrentUICulture确定计算机与用户通信时使用的语言。澳大利亚不必为此而需要一个独立的英语版本，所以它就使用美国英语。如果我在奥地利工作几个月，我会进入控制面板将CurrentCulture更改为奥地利德语。但是，因为我不会说德语，我的CurrentUICulture还保持为美国英语。

默认情况下，ResourceManager使用当前线程的CurrentUICulture属性来确定要加载的卫星程序集。ResourceManager在加载资源时使用一种fallback机制。如果定义了子文化程序集，就使用这个子文化程序集；否则，就使用通用文化。如果通用文化不存在，就使用主程序集中的默认文化。

18.7 解析和加载程序集

一个典型的应用程序包含一个可执行的主程序集和一组引用的库程序集。例如：

```
AdventureGame.exe
Terrain.dll
UIEngine.dll
```

程序集解析是指定位所引用程序集的过程。程序集解析发生在编译时和运行时。编译时解析很简单，用户或Visual Studio提供到不在当前目录的所引用程序集的完全路径，使编译器找到引用的程序集。

运行时解析复杂一些。编译器将所引用程序集的强名称写入主清单，但是未指定到哪里去寻找它们。在最简单的例子中，将所有引用的程序集放在主可执行程序集所在的文件夹中，这没有问题，因为这是（接近）CLR寻找的第一个地方。复杂性出现于：

- 将引用的程序集部署在其他地方时。
- 动态加载程序集时。

警告： 在自定义程序集加载和解析方面，Metro应用只有很少的支持。特别是，它不支持从任意文件位置加载程序集，而且没有AssemblyResolve事件。

18.7.1 程序集和类解析规则

所有类都在程序集范围内。程序集就像类的地址。例如，我们可以把一个人叫做“Joe”（不带命名空间的类名）、“Joe Bloggs”（完全类名）或“Joe Bloggs of 100 Barker Ave, WA”（程序集限定的类名）。

在编译时，我们不需要为了唯一性而进一步超越完全类名，因为不能引用两个定义相同完全类名的程序集（至少不利用特殊技巧）。但是在运行时，内存中可以有很多相同名称的类。这发生在Visual Studio设计器中，例如，每当重新构建设计的组件时。区分此类的唯一方法是通过它们的程序集；因此，程序集组成类的运行时身份的重要部分。程序集也是类到它的代码和元数据的句柄。

CLR第一次在执行时加载程序集。这发生在引用程序集的一个类时，例如，*AdventureGame.exe*实例化一个`TerrainModel.Map`类。若没有额外的配置文件，那么CLR要回答以下问题：

- *AdventureGame.exe*编译时，包含`TerrainModel.Map`的程序集的完全限定名称是什么？
- 是否已经在相同的（解析）上下文中向内存加载具有这个完全限定名称的程序集？

如果对第二个问题的回答是肯定的，那么它就使用内存中的现有副本；否则，它继续寻找程序集。CLR首先检查GAC，然后探测路径（通常是应用程序基础目录），最后激活`AppDomain.AssemblyResolve`事件。如果都不返回匹配值，那么CLR会抛出一个异常。

18.7.2 AssemblyResolve

`AssemblyResolve`事件允许干预和手动加载CLR找不到的程序集。如果处理该事件，可以在各个位置散发引用的程序集，并加载它们。

在`AssemblyResolve`事件处理程序中，通过调用`Assembly`类中三个静态方法（`Load`、`LoadFrom`或`LoadFile`）中的一个，找到并加载程序集。这些方法返回对新加载的程序集的引用，然后再返回给调用者：

```
static void Main()
{
  AppDomain.CurrentDomain.AssemblyResolve += FindAssembly;
  ...
}

static Assembly FindAssembly (object sender, ResolveEventArgs args)
{
  string fullyQualifiedName = args.Name;
  Assembly a = Assembly.LoadFrom (...);
  return a;
}
```

ResolveEventArgs事件比较特殊，因为它具有返回类。如果有多个处理程序，那么第一个返回非空Assembly的程序优先。

18.7.3 加载程序集

Assembly类中的三个Load方法在AssemblyResolve处理程序内部和外部都很有用。在事件处理程序外部时，它们可以加载和执行编译时没有引用的程序集。可能会加载程序集的一个示例情况是在执行插件时。

警告：在调用Load、LoadFrom或LoadFile之前慎重考虑：这些方法将程序集永久地加载到当前应用程序域，即使不对产生的Assembly对象执行任何操作。加载程序集具有一些副作用：它会锁定程序集文件，还会影响后续的类解析。

卸载程序集的唯一方式是卸载整个应用程序域（另一个避免锁定程序集的方法是对检测路径的程序集执行阴影拷贝（shadow copying）——请参考MSDN文章*http://albahari.com/shadowcopy*）。

如果只想检查一个程序集，不想执行它的任何代码，那么可以加载到只反射上下文中（见第18章）。

要从完全限定名称（不带位置）加载程序集，可调用Assembly.Load。这指示CLR使用普通自动解析系统寻找程序集。CLR本身使用Load寻找所引用的程序集。

要从文件名加载程序集，可调用LoadFrom或LoadFile。

要从URI加载程序集，可调用LoadFrom。

要从字节数组加载程序集，可调用Load。

提示：通过调用AppDomain的GetAssemblies方法，可以看到哪些程序集当前被加载到内存中：

```
foreach (Assembly a in
AppDomain.CurrentDomain.GetAssemblies())
{
  Console.WriteLine (a.Location);           // 文件路径
  Console.WriteLine (a.GetName().Name);     // 简单名
}
```

从文件名加载

LoadFrom和LoadFile都可以从文件名加载程序集。它们有两点区别。首先，如果有一个相同身份的程序集从另一个位置加载到了内存中，那么LoadFrom提供前一副本：

```
Assembly a1 = Assembly.LoadFrom (@"c:\temp1\lib.dll");
Assembly a2 = Assembly.LoadFrom (@"c:\temp2\lib.dll");
Console.WriteLine (a1 == a2);                              // true
```

LoadFile提供新副本：

```
Assembly a1 = Assembly.LoadFile (@"c:\temp1\lib.dll");
Assembly a2 = Assembly.LoadFile (@"c:\temp2\lib.dll");
Console.WriteLine (a1 == a2);                              //false
```

但是，如果从同一位置加载了两次，那么两种方法都提供前一次已缓存的副本。相反，从同一字节数组两次加载一个程序集，会提供两个不同的Assembly对象。

警告： 在内存中，来自2个相同程序的类型是兼容的。这是避免加载重复程序集的主要原因，也是尽量使用LoadFrom而不使用LoadFile的原因。

LoadFrom和LoadFile的另一个区别是，LoadFrom会告诉CLR前向引用的位置，而LoadFile则不会。为了说明这一点，可以假设*\folder1*中的应用加载*\folder2*中的一个程序集*TestLib.dll*，它引用*\folder2\Another.dll*：

```
\folder1\MyApplication.exe

\folder2\TestLib.dll
\folder2\Another.dll
```

如果使用LoadFrom加载*TestLib*，那么CLR会找到和加载*Another.dll*。

如果使用LoadFile加载*TestLib*，那么CLR就无法找到*Another.dll*，然后抛出一个异常，除非事件处理了AssemblyResolve事件。

在下面的章节中将说明如何在一些实际应用中使用这些方法。

静态引用类型和LoadFrom/LoadFile

如果直接在代码中引用一个类型，那么就称为静态引用（*statically referencing*）该类型。编译器会将该类型的引用添加到正在编译的程序集中，以及包含该类型的程序集名称（但是不包含如何在运行时寻找该类型的信息）。

例如，假设程序集*foo.dll*中有一个类型Foo，而应用*bar.exe*中包含下面的代码：

```
var foo = new Foo();
```

那么，*bar.exe*应用会在*foo*程序集中静态引用Foo类型。我们也可以用下面的方法动态加载*foo*：

```
Type t = Assembly.LoadFrom (@"d:\temp\foo.dll").GetType ("Foo");
var foo = Activator.CreateInstance (t);
```

如果混合使用这两种方法，那么通常最终会在内存中出现两个程序集副本，因为CLR将每一个副本认为是不同的“解析环境”。

之前提到，在解析静态引用时，CLR会先检查GAC，然后检查检测路径（通常是应用的基目录），最后触发AssemblyResolve事件。但是，在这些操作之前，它会先检查程序集是否已经加载。然而，它只考虑以下情况的程序集：

- 已经从一个路径加载，否则就会出现在自己的路径上（检测路径）
- 已经从AssemblyResolve事件的响应中加载

因此，如果已经通过LoadFrom或LoadFrom从一个未检测的路径将它加载，那么最终会在内存中出现两个程序集副本（有不兼容的类型）。为了避免这种情况，在调用LoadFrom/LoadFile时必须非常小心，要先检查程序集是否已经存在于应用的基目录（除非确实想加载同一个程序集的多个版本）。

如果在AssemblyResolve事件响应中加载，则不存在这个问题（无论是使用LoadFrom、LoadFile或后面将会介绍的从字节数组加载），因为事件只触发检测路径之外的程序集。

提示：无论使用LoadFrom还是LoadFile，CLR都一定会先在GAC中查找所请求的程序集。使用ReflectionOnlyLoadFrom（它会将程序加载到只有反映的环境中），可以跳过GAC。即使从字节数组加载，也无法跳过GAC，但是这样可以解决锁定程序集文件的问题：

```
byte[] image = File.ReadAllBytes (assemblyPath);
Assembly a = Assembly.Load (image);
```

如果这样做，则必须处理AppDomain的AssemblyResolve事件，才能解析已加载程序集本身引用的任意程序集，并且跟踪所有已加载的程序集（参见18.9节“打包单个可执行文件”）。

Location与CodeBase

程序集的Location属性通常会返回其在文件系统的物理位置（如果有）。而CodeBase属性则以URI形式映射这个位置，但是一些特殊情况例外，例如：如果从互联网加载，那么CodeBase是互联网URI，而Location则是下载后存储的临时路径。另一个特殊情况是使用阴影拷贝的程序集，这时Location是空的，而CodeBase是未执行阴影复制之前的位置。ASP.NET和流行的NUnit测试框架使用阴影拷贝，在网站或单元测试运行时更新程序集（请参考MSDN文档*http://albahari.com/shadowcopy*）。在引用自定义程序集时，LINQPad也会执行类似的操作。

因此，如果要寻找程序集在磁盘的位置，只使用Location是不可靠的。更好的方法是同时检查两个属性。下面的方法返回一个程序集所在的文件夹（如果无法确定则返回null）：

```
public static string GetAssemblyFolder (Assembly a)
{
  try
  {
    if (!string.IsNullOrEmpty (a.Location))
      return Path.GetDirectoryName (a.Location);

    if (string.IsNullOrEmpty (a.CodeBase)) return null;

    var uri = new Uri (a.CodeBase);
    if (!uri.IsFile) return null;

    return Path.GetDirectoryName (uri.LocalPath);
  }
  catch (NotSupportedException)
  {
    return null;    // 由Reflection.Emit生成的动态程序集
  }
}
```

注意，因为CodeBase返回一个URI，所以使用Uri类获取其本地文件路径。

18.8 在基础文件夹外部署程序集

有时，可能会选择将程序集部署到应用程序基础目录之外的位置。例如：

```
..\MyProgram\Main.exe
..\MyProgram\Libs\V1.23\GameLogic.dll
..\MyProgram\Libs\V1.23\3DEngine.dll
..\MyProgram\Terrain\Map.dll
..\Common\TimingController.dll
```

要使之可行，必须帮助CLR在基础文件夹外寻找程序集。最容易的解决方案是处理AssemblyResolve事件。

在下面的例子中，我们假设其他所有的程序集都位于*c:\ExtraAssemblies*中：

```
using System;
using System.IO;
using System.Reflection;

class Loader
{
  static void Main()
  {
    AppDomain.CurrentDomain.AssemblyResolve += FindAssembly;

     // 我们必须切换到其他类，然后尝试在c:\ExtraAssemblies中使用任意类：
     Program.Go();
  }

  static Assembly FindAssembly (object sender, ResolveEventArgs args)
  {
    string simpleName = new AssemblyName (args.Name).Name;
    string path = @"c:\ExtraAssemblies\" + simpleName + ".dll";
    if (!File.Exists (path)) return null;     // 完整性检查
    return Assembly.LoadFrom (path);          // 加载
  }
}

class Program
{
  internal static void Go()
  {
    // 现在我们可以引用在c:\ExtraAssemblies中定义的类
  }
}
```

警告： 在这个例子中，不直接从Loader类引用*c:\ExtraAssemblies*中的类型（例如，作为字段）是极为重要的，因为CLR会在遇到Main()之前解析类。

在本例中，我们可以使用LoadFrom，也可以使用LoadFile。无论使用哪一个，CLR都验证传递给它的程序集是否具有它所请求的身份。这维护了强命名引用的完整性。

在第24章中，我们将描述创建新应用程序域时可以使用的另外一种方法。这涉及到设置应用程序域的PrivateBinPath以包括那些包含其他程序集的目录，扩展了标准程序集探测位置。这种方法的局限是，其他目录必须都位于应用程序基础目录之下。

18.9 打包单个可执行文件

假设编写了一个包含10个程序集的应用程序：1个主可执行文件，加上9个DLL。尽管这样的粒度有利于设计和调试，但是也需要能够将所有东西打包到单个“单击就运行”的可执行文件中，而无需用户执行一些设置或文件解压之类的常规操作。打包的方法是，将已编译的程序集DLL作为嵌入式资源包含在主可执行项目中，然后编写一个AssemblyResolve事件处理程序来根据需要加载它们的二进制图像。下面就是具体的做法：

```
using System;
using System.IO;
using System.Reflection;
using System.Collections.Generic;

public class Loader
{
  static Dictionary <string, Assembly> _libs
    = new Dictionary <string, Assembly>();

  static void Main()
  {
    AppDomain.CurrentDomain.AssemblyResolve += FindAssembly;
    Program.Go();
  }

  static Assembly FindAssembly (object sender, ResolveEventArgs args)
  {
    string shortName = new AssemblyName (args.Name).Name;
    if (_libs.ContainsKey (shortName)) return _libs [shortName];

    using (Stream s = Assembly.GetExecutingAssembly().
           GetManifestResourceStream ("Libs." + shortName + ".dll"))
    {
      byte[] data = new BinaryReader (s).ReadBytes ((int) s.Length);
      Assembly a = Assembly.Load (data);
      _libs [shortName] = a;
      return a;
    }
  }
}

public class Program
{
  public static void Go()
  {
    // 运行主程序...
  }
}
```

由于Loader类定义在主可执行文件中，所以对Assembly.GetExecutingAssembly的调用将总是返回主可执行程序集，其中包含了作为嵌入式资源的已编译的DLL。在本例中，我们给每个嵌入式资源程序集加上前缀“Libs.”。如果使用了Visual Studio IDE，那么应该将“Libs.”更改为项目的默认命名空间（进入Project Properties→Application）。还需要确保包含在主项目中的每个DLL文件的“Build Action IDE”属性被设置为“Embedded Resource”。

将所请求的程序集高速缓存在一个字典中的原因是为了确保，如果CLR再次请求同一程序集，可以返回完全相同的对象。否则，程序集的类将与那些以前加载的类不兼容（除了它们的二进制图像相同之外）。

这个例子的一个变体是在编译时压缩引用的程序集，然后使用DeflateStream将它们解压在FindAssembly中。

选择性打补丁

假设这个例子中我们想要可执行文件能够自动地更新，也许是从网络服务器或网站进行更新。直接打包可执行文件不仅复杂难实现，而且所需的文件I/O权限也不是那么容易得到（例如，如果安装在*Program Files*中）。最佳的解决方法是将任何已更新的库下载到独立存储区域中（每个存储区域作为一个单独的DLL），然后修改FindAssembly方法，以便它在从可执行文件中的资源加载某个库之前，首先检查这个库是否已经存在于它的独立存储区域。这保证了初始可执行文件不被修改，从而避免了在用户计算机上留下任何非法程序。如果程序集是强命名的（假设它们在编译时被引用），那么安全性是不能破坏的，如果有什么错误，应用程序总是可以恢复到它的初始状态，只要删除它的独立存储区域中的所有文件即可。

18.10 处理未引用的程序集

有时，显式地加载可能在编译时没有引用的程序集是很有用的。

如果所讨论的程序集是一个可执行文件，那么想要运行它，只需在当前应用程序域上调用ExecuteAssembly即可。ExecuteAssembly使用LoadFrom语义加载可执行文件，然后利用可选的命令行参数调用它的实体方法。例如：

```
string dir = AppDomain.CurrentDomain.BaseDirectory;
AppDomain.CurrentDomain.ExecuteAssembly (Path.Combine (dir, "test.exe"));
```

ExecuteAssembly同步地工作，意味着调用方法一直被阻止，直到被调用程序集退出。要异步地工作，必须在另一个线程上调用ExecuteAssembly（见第14章）。

但是在大多数情况下，想要加载的程序集是库。加载的方法是调用LoadFrom，然后使用反射来处理程序集的类。例如：

```
string ourDir = AppDomain.CurrentDomain.BaseDirectory;
string plugInDir = Path.Combine (ourDir, "plugins");
Assembly a = Assembly.LoadFrom (Path.Combine (plugInDir, "widget.dll"));
Type t = a.GetType ("Namespace.TypeName");
object widget = Activator.CreateInstance (t);    //（见第18章）
...
```

我们使用LoadFrom而不使用LoadFile，是为了确保同一文件夹中*widget.dll*引用的任何私有程序集也已经加载。然后通过名称从程序集检索到一个类，并实例化。

下一步是使用反射来动态地在widget上调用方法和属性，我们将在下一章中描述如何实现。一个较容易且较快速的方法是，将对象强制转换为两个程序集都接受的类型。这是定义在公共程序集中的一个接口：

```
public interface IPluggable
```

```
{
  void ShowAboutBox();
  ...
}
```

这允许我们实现：

```
Type t = a.GetType ("Namespace.TypeName");
IPluggable widget = (IPluggable) Activator.CreateInstance (t);
widget.ShowAboutBox();
```

可以使用类似的方式在WCF或Remoting Server中动态发布服务。下面假设我们想要公开的库以"server"结尾：

```
using System.IO;
using System.Reflection;
...
string dir = AppDomain.CurrentDomain.BaseDirectory;
foreach (string assFile in Directory.GetFiles (dir, "*Server.dll"))
{
  Assembly a = Assembly.LoadFrom (assFile);
  foreach (Type t in a.GetTypes())
    if (typeof (MyBaseServerType).IsAssignableFrom (t))
    {
      // 公开类t
    }
}
```

但是，这会使得一些人非常容易添加非法程序集，甚至可能是意外地！假设没有编译时引用，CLR就无法检查程序集的身份。如果加载的每个程序集都被分配一个已知的公共密钥，那么解决方案是显式地检查那个密钥。在下面这个例子中，我们假设所有的库都和执行程序集一样，被分配相同的密钥对：

```
byte[] ourPK = Assembly.GetExecutingAssembly().GetName().GetPublicKey();

foreach (string assFile in Directory.GetFiles (dir, "*Server.dll"))
{
  byte[] targetPK = AssemblyName.GetAssemblyName (assFile).GetPublicKey();
  if (Enumerable.SequenceEqual (ourPK, targetPK))
  {
    Assembly a = Assembly.LoadFrom (assFile);
    ...
```

注意AssemblyName是如何在加载程序集之前检查公共密钥的。为了比较字节数组，我们使用了LINQ的SequenceEqual方法（System.Linq）。

第19章

反射和元数据

如前面各章所介绍的，C#程序可以编译为一种程序集，这种程序集中可以包含元数据、编译后的代码和各种资源。在运行时检查元数据和编译代码的操作称为“反射”。

程序集中编译后的代码几乎含有初始源代码的所有内容。某些信息会丢失，如局部变量名、注释和预处理程序语句。但是，反射可以访问非常多的其他信息，甚至可以编写一个反编译器。

许多可以在.NET中获得的和通过C#公开的服务（如动态绑定、串行化、数据绑定和Remoting）都是以元数据的存在为依据的。自己开发的程序也可以利用这些元数据，甚至可以通过使用自定义属性的新信息扩充它们。System.Reflection命名空间含有反射API。通过System.Reflection.Emit命名空间中的类，在运行时也可以在IL（中间语言）中动态创建新的元数据和可执行指令。

本章的示例假定导入了System、System.Reflection以及System.Reflection.Emit命名空间。

提示： 本章使用的术语“动态”是指使用反射执行某些任务时，这些任务的类型安全仅在运行时才被强制设定。这与通过C#的dynamic关键字进行动态绑定的原则类似，尽管它们的机制和功能各不相同。

比较这两种方法，动态绑定更易于使用并且可以更有效地利用动态语言互操作性的DLR。相对来说使用反射的限制更多一点，只能与CLR一同使用，但是这在使用CLR可以完成的任务方面有更多的灵活性。例如，通过反射可以获得类型和成员列表、实例化某个名称来自字符串的对象，而且还可以在运行时构建程序集。

19.1 反射和激活类型

本节将介绍如何获取类型、检查元数据和使用反射动态地实例化对象。

19.1.1 获取类型

System.Type的实例代表了类型的元数据。因为Type的应用领域非常广泛，所以它存在于System命名空间中，而非System.Reflection命名空间中。

通过调用对象上的GetType或者使用C#的typeof运算符，可以获得System.Type实例：

```
Type t1 = DateTime.Now.GetType();          // 在运行时获得的类型
Type t2 = typeof (DateTime);               // 在编译时获得的类型
```

可以使用typeof获得数组类型和以下泛型类型：

```
Type t3 = typeof (DateTime[]);             // 1-d数组类型
Type t4 = typeof (DateTime[,]);            // 2-d数组类型
Type t5 = typeof (Dictionary<int,int>);    // 封闭式泛型类型
Type t6 = typeof (Dictionary<,>);          // 未绑定的泛型类型
```

还可以通过名称获取类型。如果引用了该类型的程序集，例如Assembly.GetType（本章稍后将介绍该程序集）：

```
Type t = Assembly.GetExecutingAssembly().GetType ("Demos.TestProgram");
```

如果没有程序集对象，可以通过其程序集限定名称获取类型（该类型的全称会带有程序集完整的限定名称）。当调用Assembly.Load(string)时，会隐式地加载该程序集：

```
Type t = Type.GetType ("System.Int32, mscorlib, Version=2.0.0.0, " +
                       "Culture=neutral, PublicKeyToken=b77a5c561934e089");
```

一旦拥有了System.Type对象，就可以使用它的属性访问类型的名称、程序集、基础类型、可见性等。例如：

```
Type stringType = typeof (String);
string name = stringType.Name;               // 字符串
Type baseType = stringType.BaseType;         // typeof(Object)
Assembly assem = stringType.Assembly;        // mscorlib.dll
bool isPublic = stringType.IsPublic;         // true
```

一个System.Type实例就是打开类型（及其定义的程序集）的全部元数据的一个入口。

提示：System.Type是个抽象的概念，因此实际上typeof运算符获得的是Type子类。对于*mscorlib*来说，CLR使用的这些子类都是内部的，称为RuntimeType。

---

1. TypeInfo和Metro应用

Metro应用模板隐藏了大多数类型成员，转而将它们封装在TypeInfo类中。调用GetTypeInfo，就可以得到这个类。所以，将前一个例子运行在Metro应用中，可以这样做：

```
Type stringType = typeof(string);
string name = stringType.Name;
Type baseType = stringType.GetTypeInfo().BaseType;
Assembly assem = stringType.GetTypeInfo().Assembly;
bool isPublic = stringType.GetTypeInfo().IsPublic;
```

警告：本章的许多代码清单都需要修改才能在Metro应用中正常运行。所以，如果有某个例子由于缺少成员而无法编译，则要在Type表达式上添加.GetTypeInfo()。

完整的.NET框架也包含TypeInfo，所以能在Metro中正常运行的代码也可以在标准库.NET应用中运

行，但是只适用于Framework 4.5（旧版本不支持）。TypeInfo还包含其他一些反射成员的属性和方法。

Metro应用只实现了有限的反射机制。特别是它们无法访问非公共成员类型，也无法使用Reflection.Emit。

2. 获取数组类型

如前所述，可以将typeof和GetType与数组类型一起使用。还可以通过调用元素类型上的MakeArrayType获取数组类型：

```
Type tSimpleArray = typeof (int).MakeArrayType();
Console.WriteLine (tSimpleArray == typeof (int[]));  // True
```

可以向MakeArray传递整型参数，以创建多维矩形数组：

```
Type tCube = typeof (int).MakeArrayType (3);          //立方体形
Console.WriteLine (tCube == typeof (int[,,]));        // True
```

GetElementType返回数组的元素类型：

```
Type e = typeof (int[]).GetElementType();             // e == typeof (int)
```

GetArrayRank返回矩形数组的维数：

```
int rank = typeof (int[,,]).GetArrayRank();           // 3
```

3. 获取嵌套类型

要重新获得嵌套类型，可调用包含类型的GetNestedTypes。例如：

```
foreach (Type t in typeof (System.Environment).GetNestedTypes())
  Console.WriteLine (t.FullName);
```

输出：System.Environment+SpecialFolder

在使用嵌套类型时需要特别注意的是CLR会认为嵌套类型拥有特定“嵌套”可访问等级。例如：

```
Type t = typeof (System.Environment.SpecialFolder);
Console.WriteLine (t.IsPublic);                         // False
Console.WriteLine (t.IsNestedPublic);                   // True
```

19.1.2 类型名称

类型具有Namespace、Name和FullName特性。在大多数情况中，FullName是前两者的组合：

```
Type t = typeof (System.Text.StringBuilder);

Console.WriteLine (t.Namespace);          // System.Text
Console.WriteLine (t.Name);               // StringBuilder
Console.WriteLine (t.FullName);           // System.Text.StringBuilder
```

这个法则有两种异常情况：嵌套类型和封闭式泛型类型。

提示： Type还具有AssemblyQualifiedName特性，使用它可以返回带有逗号和其程序集完整名称的FullName值。同样可以将该字符串传递给Type.GetType，然后会在默认的加载环境中单独获取类型。

1. 嵌套类型名称

对于嵌套类型来说，包含类型仅在FullName中出现：

```
Type t = typeof (System.Environment.SpecialFolder);
Console.WriteLine (t.Namespace);      // System
Console.WriteLine (t.Name);           // SpecialFolder
Console.WriteLine (t.FullName);       // System.Environment+SpecialFolder
```

+表示将包含类型与嵌套的命名空间区分开。

2. 泛型类型名称

泛型类型名称带有'后缀，还带有类型参数的编号。如果泛型类型被绑定，那么该法则同时应用于Name和FullName：

```
Type t = typeof (Dictionary<,>);      // 未绑定
Console.WriteLine (t.Name);           // Dictionary'2
Console.WriteLine (t.FullName);       // System.Collections.Generic.Dictionary'2
```

然而，如果该泛型类型是封闭式的，FullName（仅仅）获得基本的额外附加信息。下面枚举了每个类型参数的完整的程序集限定名称：

```
Console.WriteLine (typeof (Dictionary<int,string>).FullName);

// 输出：
System.Collections.Generic.Dictionary'2[[System.Int32, mscorlib,
Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089],
[System.String, mscorlib, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089]]
```

这样可以确保AssemblyQualifiedName（类型完整名称和程序集名称的组合）包含足够的信息，以严格区分泛型类型及其类型参数。

3. 数组和指针类型名称

数组通过在typeof表达式中使用的相同后缀表示：

```
Console.WriteLine (typeof ( int[] ).Name);          // Int32[]
Console.WriteLine (typeof ( int[,] ).Name);         // Int32[,]
Console.WriteLine (typeof ( int[,] ).FullName);     // System.Int32[,]
```

指针类型也与数组类似：

```
Console.WriteLine (typeof (byte*).Name); // Byte*
```

4. Ref和out参数类型名称

描述ref和out参数的类型带有&后缀：

```
Type t = typeof (bool).GetMethod ("TryParse").GetParameters()[1]
                                            .ParameterType;
Console.WriteLine (t.Name);  // Boolean&
```

本章稍后会对这部分进行详细介绍。

19.1.3 基本类型和接口

类型可以公开BaseType特性：

```
Type base1 = typeof (System.String).BaseType;
Type base2 = typeof (System.IO.FileStream).BaseType;

Console.WriteLine (base1.Name);       // 对象
Console.WriteLine (base2.Name);       // 数据流
```

GetInterfaces方法会返回类型实现的接口：

```
foreach (Type iType in typeof (Guid).GetInterfaces())
  Console.WriteLine (iType.Name);

IFormattable
IComparable
IComparable'1
IEquatable'1
```

反射为C#的静态is运算符提供了两种等价的动态运算符：

IsInstanceOfType
: 可以接收类型和实例。

IsAssignableFrom
: 可以接收两个类型。

下面列出了使用第一个运算符的示例：

```
object obj = Guid.NewGuid();
Type target = typeof (IFormattable);

bool isTrue = obj is IFormattable;                     // 静态C#运算符
bool alsoTrue = target.IsInstanceOfType (obj);         // 动态等价运算符
```

IsAssignableFrom具有更多用途：

```
Type target = typeof (IComparable), source = typeof (string);
Console.WriteLine (target.IsAssignableFrom (source));          // True
```

IsSubclassOf方法的使用原则与IsAssignableFrom方法的使用原则相同，但是IsSubclassOf不包含接口。

19.1.4 实例化类型

可以使用两种方法通过对象的类型动态地实例化对象：

- 调用静态Activator.CreateInstance方法。
- 调用ConstructorInfo对象上的Invoke，ConstructorInfo对象是通过调用类型（高级环境）上的GetConstructor获得的。

Activator.CreateInstance可以接收已传递到构造方法的Type和可选的参数：

```
int i = (int) Activator.CreateInstance (typeof (int));
DateTime dt = (DateTime) Activator.CreateInstance (typeof (DateTime), 2000, 1, 1);
```

使用CreateInstance可以设定许多其他选项，如用于加载类型的程序集、目标应用程序域和是否与非全局构造方法绑定。如果运行时无法找到适当的构造方法，那么会抛出MissingMethodException。

当参数值无法在重载的构造方法之间消除时，必须调用ConstructorInfo上的Invoke。例如，假设X类拥有两个构造方法：一个构造方法接收string类型的参数，而另一个构造方法接收StringBuilder类型的参数。当类型不明确时，应该将一个null参数传递给Activator.CreateInstance。在这种情况需要使用ConstructorInfo进行替换：

```
// 引用接收String类型参数的构造方法：
ConstructorInfo ci = typeof (X).GetConstructor (new[] { typeof (string) };

// 使用重载方法创建对象，传入null类型参数：
object foo = ci.Invoke (new object[] { null });
```

或者，在Metro应用中：

```
ConstructorInfo ci = typeof (X).GetTypeInfo().DeclaredConstructors
.FirstOrDefault (c =>
c.GetParameters().Length == 1 &&
c.GetParameters()[0].ParameterType == typeof (string));
```

要获取非全局构造方法，需要设定BindingFlags，请参阅本节稍后对它的详细介绍。

警告： 在构造对象时进行动态实例化会增加几微秒的时间。相对而言这是一个较长的时间，因为CLR实例化对象的速度非常快（在小型类上简单的new操作不足十纳秒）。

要根据元素类型动态实例化数组，应首先调用MakeArrayType。还可以实例化泛型类型，本章会在下一节介绍该内容。

要动态地实例化委托，应调用Delegate.CreateDelegate。下列示例演示了实例化实例委托和静态委托的过程：

```
class Program
{
  delegate int IntFunc (int x);

  static int Square (int x) { return x * x; }        // 静态方法
  int        Cube (int x)   { return x * x * x; }    // 实例方法

  static void Main()
  {
    Delegate staticD = Delegate.CreateDelegate
      (typeof (IntFunc), typeof (Program), "Square");d

    Delegate instanceD = Delegate.CreateDelegate
      (typeof (IntFunc), new Program(), "Cube");

    Console.WriteLine (staticD.DynamicInvoke (3));   // 9
    Console.WriteLine (instanceD.DynamicInvoke (3)); // 27
  }
}
```

可以通过引用调用DynamicInvoke返回的Delegate对象，如上例所示，还可以通过转换为类型化的委托来实现：

```
IntFunc f = (IntFunc) staticD;
Console.WriteLine (f(3));    // 9 (但是更加快速!)
```

可以将MethodInfo传递到CreateDelegate中，而非方法名称中。将在后面的章节中简略介绍MethodInfo，还会介绍将以动态方式创建的委托转换回静态委托类型的基本原理。

19.1.5 泛型类型

Type可以代表封闭式或未绑定的泛型类型。在编译时，封闭式泛型类型可以实例化，而未绑定的类型不能实例化：

```
Type closed = typeof (List<int>);
List<int> list = (List<int>) Activator.CreateInstance (closed);      // 可以

Type unbound = typeof (List<>);
object anError = Activator.CreateInstance (unbound);                 // 运行时错误
```

MakeGenericType方法可以将未绑定的泛型类型转换为封闭式泛型类型。只需传递需要的类型参数就可以实现：

```
Type unbound = typeof (List<>);
Type closed = unbound.MakeGenericType (typeof (int));
```

使用GetGenericTypeDefinition方法可以实现相反的操作：

```
Type unbound2 = closed.GetGenericTypeDefinition();   // unbound == unbound2
```

当Type为泛型时，IsGenericType会返回true，而当泛型类型为未绑定时，IsGenericTypeDefinition会返回true。下列代码可以检测出类型是否为可空的类型值：

```
Type nullable = typeof (bool?);
Console.WriteLine (
  nullable.IsGenericType &&
  nullable.GetGenericTypeDefinition() == typeof (Nullable<>));       // True
```

GetGenericArguments可以为封闭式泛型类型返回类型参数：

```
Console.WriteLine (closed.GetGenericArguments()[0]);         // System.Int32
Console.WriteLine (nullable.GetGenericArguments()[0]);       // System.Boolean
```

对于未绑定的泛型类型来说，GetGenericArguments会返回在泛型类型定义中指定为占位符类型的伪类型：

```
Console.WriteLine (unbound.GetGenericArguments()[0]);        // T
```

提示：在运行时，所有泛型类型不是未绑定的就是封闭式的。在typeof(Foo<>)这类表达式中泛型类型是未绑定的（相对来说这种情况比较常见）；在其他情况中，泛型类型是封闭式的。在运行时不存在开放式泛型类型：所有开放式类型都会被编译器关闭。下列类中的方法总是得到False结果：

```
class Foo<T>
{
  public void Test()
  {
    Console.Write (GetType().IsGenericTypeDefinition);
  }
}
```

19.2 反射和调用成员

使用GetMembers方法可以返回类型的成员。请看下列类：

```
class Walnut
{
  private bool cracked;
  public void Crack() { cracked = true; }
}
```

我们可以使用下列方式对它的公共成员进行反射：

```
MemberInfo[] members = typeof (Walnut).GetMembers();
foreach (MemberInfo m in members)
  Console.WriteLine (m);
```

下面列出的是运行结果：

```
Void Crack()
System.Type GetType()
System.String ToString()
Boolean Equals(System.Object)
Int32 GetHashCode()
Void .ctor()
```

通过TypeInfo反射成员

TypeInfo提供了另一个（更简单的）成员反射协议。这个API对于目标平台为Framework 4.5的应用是可选的，而Metro应用则是强制选择的，因为Metro应用没有与GetMembers方法等价的方法。

TypeInfo并没有像GetMembers这样可以返回数组的方法，而只有返回IEnumerable<T>的属性，它们一般用于运行LINQ查询。其中使用最广泛的是DeclaredMembers：

```
IEnumerable<MemberInfo> members =
  typeof(Walnut).GetTypeInfo().DeclaredMembers;
```

与GetMembers()不同，其结果包含继承的成员：

```
Void Crack()
Void .ctor()
Boolean cracked
```

此外，还有一些属性可以返回特定类型的成员（DeclaredProperties、DeclaredMethods、DeclaredEvents等），以及一些按名称返回具体成员的方法（如GetDeclaredMethod）。后者不支持重载（因为无法指定参数类型）。相反，必须通过DeclaredMethods执行LINQ查询：

```
MethodInfo method = typeof (int).GetTypeInfo().DeclaredMethods
.FirstOrDefault (m => m.Name == "ToString" &&
m.GetParameters().Length == 0);
```

如果在调用时没有使用参数，GetMembers会返回类型（及其基本类型）的所有公共成员。GetMember通过名称检索特定成员，但是因为成员可能会被重新加载，GetMember仍旧会返回一个数组：

```
MemberInfo[] m = typeof (Walnut).GetMember ("Crack");
Console.WriteLine (m[0]);                     // Void Crack()
```

MemberInfo也具有MemberTypes类型的MemberType特性。下面列出的是该特性的典型值：

```
All、Custom、Field、NestedType、TypeInfo、Constructor、Event、Method、Property
```

当调用GetMembers时，可以传递一个MemberTypes实例，以限定它返回的成员类型。还可以通过调用GetMethods、GetFields、GetProperties、GetEvents、GetConstructors或GetNestedTypes，限定返回的结果。这些方法还有专门用于特定成员的版本。

提示： 对类型的成员进行检索时应尽可能地具体，因而如果要在以后添加成员，就无需拆分代码。如果要通过名称检索方法，指定所有参数类型可以确保出现方法重载时，代码仍旧可以运行（我们在后面的内容中提供了一些示例）。

MemberInfo对象具有Name特性和以下两个Type特性：

DeclaringType
: 返回定义该成员的类型。

ReflectedType
: 根据所调用种类的GetMembers返回类型。

当根据基础类型定义的成员进行调用时，会出现两种不同情况：DeclaringType会返回基础类型；而ReflectedType会返回子类型。下列示例重点说明了这一点：

```
class Program
{
  static void Main()
  {
    // MethodInfo是MemberInfo的子类：如图19-1所示。

    MethodInfo test = typeof (Program).GetMethod ("ToString");
    MethodInfo obj = typeof (object) .GetMethod ("ToString");

    Console.WriteLine (test.DeclaringType);   // System.Object
    Console.WriteLine (obj.DeclaringType);    // System.Object

    Console.WriteLine (test.ReflectedType);   // Program
    Console.WriteLine (obj.ReflectedType);    // System.Object

    Console.WriteLine (test == obj);          // False
  }
}
```

因为test和obj对象拥有不同的ReflectedTypes，所以它们并不相同。然而，它们之间的差异只在于反射API的实现；Program类型在基础类型系统中没有独特的ToString方法。我们可以通过下列两种方式证明这两个MethodInfo对象引用了相同的方法：

```
Console.WriteLine (test.MethodHandle == obj.MethodHandle);    // True

Console.WriteLine (test.MetadataToken == obj.MetadataToken    // True
                   && test.Module == obj.Module);
```

MethodHandle对于应用域中的每个特定的方法都是唯一的；对于程序集模块中的所有类型和成员来

说，MetadataToken也是唯一的。

MemberInfo还定义了用于返回自定义属性的方法（请参阅本章稍后对该部分的介绍）。

提示： 可以通过调用MethodBase.GetCurrentMethod获得当前执行方法的MethodBase。

19.2.1 成员类型

MemberInfo本身在成员中不重要，因为它是类型的概要基础，如图19-1所示。

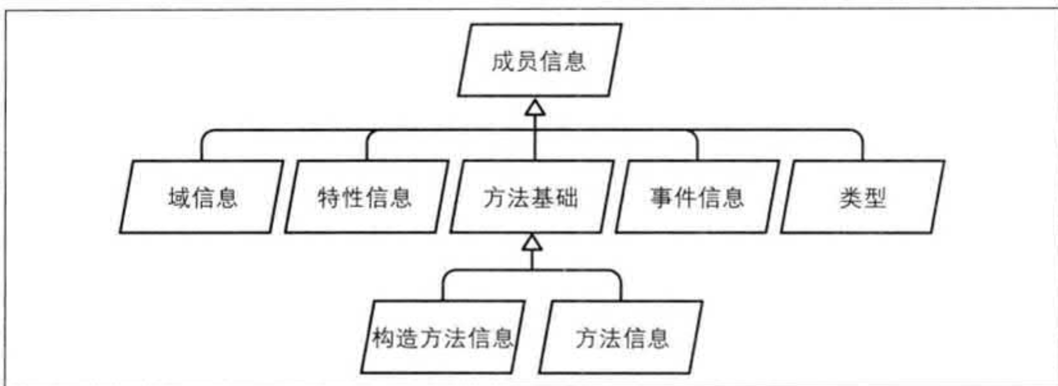


图19-1：成员类型

可以根据MemberInfo的MemberType特性，将MemberInfo投射到其子类型上。如果通过GetMethod、GetField、GetProperty、GetEvent、GetConstructor或GetNestedType（或者它们的复数版本）获取成员，就不必进行投射。表19-1概述了各种C#构造应使用的方法。

表19-1：检索成员元数据

C#构造	可以使用的方法	可以使用的名称	结果
方法	GetMethod	方法名称	MethodInfo
属性	GetProperty	属性名称	PropertyInfo
索引器	GetDefaultMembers		MemberInfo[]（如果在C#中编译会包含PropertyInfo对象）
域	GetField	域名称	FieldInfo .
枚举成员	GetField	成员名称	FieldInfo
事件	GetEvent	事件名称	GetEvent
构造方法	GetConstructor		ConstructorInfo
终止器	GetMethod	"Finalize"	MethodInfo
运算符	GetMethod	"op_+"运算符名称	MethodInfo
嵌套类型	GetNestedType	类型名称	Type

每个MemberInfo子类都具有大量特性和方法，以便公开成员元数据的可见性、修饰符、泛型类型参数、参数、返回类型和自定义属性。

下面是一个使用GetMethod的示例：

```
MethodInfo m = typeof (Walnut).GetMethod ("Crack");
Console.WriteLine (m);                                  // Void Crack()
Console.WriteLine (m.ReturnType);                       // System.Void
```

在第一次使用时所有*Info实例都由反射API缓存：

```
MethodInfo method = typeof (Walnut).GetMethod ("Crack");
MemberInfo member = typeof (Walnut).GetMember ("Crack") [0];

Console.Write (method == member);    // True
```

同样在保留对象身份时，缓存也可以提高速度较慢的API的性能。

19.2.2 C#成员与CLR成员

表19-1说明了某些C#函数构造并非与CLR构造一一对应。这是因为CLR和反射API是使用.NET语言设计的，因而可以通过Visual Basic使用反射。

有些C#构造（即索引器、枚举、运算符和终止器）在涉及CLR时就被设计出来了。尤其应注意以下几点：

- C#索引器可以转换为接收一个或多个参数的特性，而且可以标识为类型的[DefaultMember]。
- C#枚举可以通过每个成员的静态域转换为System.Enum的子类型。
- C#运算符可以转换为被特殊命名的静态方法，而且带有“op_”前缀，例如op_Addition。
- C#析构函数可以转换为覆盖Finalize的方法。

另一种复杂的情况是特性或事件实际上由两部分组成：

- 描述特性或事件的元数据（由PropertyInfo或EventInfo封装）
- 一个或两个反向方法（backing Method）

在C#程序中，反向方法被封装在特性或事件定义中。但是当将它们编译为IL时，反向方法会被表示为原始方法，而且可以像其他方法那样调用。这意味着GetMethods会返回与原始方法并列的特性和事件反向方法。下面是具体说明：

```
class Test { public int X { get { return 0; } set {} } }

void Demo()
{
  foreach (MethodInfo mi in typeof (Test).GetMethods())
    Console.Write (mi.Name + " ");
}

// 输出：
get_X set_X GetType ToString Equals GetHashCode
```

可以通过MethodInfo中的IsSpecialName特性识别这些方法。IsSpecialName会对特性、索引器、

事件访问器和运算符返回true。它只对常规的C#方法返回false，而且只有在定义了析构函数的情况下对Finalize方法返回false。

下面是C#生成的反向方法：

C#构造	成员类型	IL中的方法
特性	Property	get_XXX和set_XXX
索引器	Property	get_Item和set_Item
事件	Event	add_XXX和remove_XXX

每个反向方法都拥有其本身的关联对象MethodInfo。可以通过下列方式访问这些方法：

```
PropertyInfo pi = typeof (Console).GetProperty ("Title");
MethodInfo getter = pi.GetGetMethod();                   // get_Title
MethodInfo setter = pi.GetSetMethod();                   // set_Title
MethodInfo[] both = pi.GetAccessors();                   // Length==2
```

对于EventInfo使用GetAddMethod和GetRemoveMethod会得到类似的结果。

要以反向方式操作（从MethodInfo到它关联的PropertyInfo或EventInfo），需要执行查询。LINQ可以完成这项工作：

```
PropertyInfo p = mi.DeclaringType.GetProperties()
                   .First (x => x.GetAccessors (true).Contains (mi));
```

19.2.3 泛型类型成员

既可以为未绑定的泛型类型获取成员元数据，也可以为封闭式泛型类型获取成员元数据：

```
PropertyInfo unbound = typeof (IEnumerator<>) .GetProperty ("Current");
PropertyInfo closed = typeof (IEnumerator<int>).GetProperty ("Current");

Console.WriteLine (unbound);          // T Current
Console.WriteLine (closed);           // Int32 Current

Console.WriteLine (unbound.PropertyType.IsGenericParameter);  // True
Console.WriteLine (closed.PropertyType.IsGenericParameter);   // False
```

从未绑定的和封闭式泛型类型返回的MemberInfo对象总是独特的，即使对于签名中不带泛型类型参数的成员也是如此：

```
PropertyInfo unbound = typeof (List<>) .GetProperty ("Count");
PropertyInfo closed = typeof (List<int>).GetProperty ("Count");

Console.WriteLine (unbound);                  // Int32 Count
Console.WriteLine (closed);                   // Int32 Count

Console.WriteLine (unbound == closed);        // False

Console.WriteLine (unbound.DeclaringType.IsGenericTypeDefinition); // True
Console.WriteLine (closed.DeclaringType.IsGenericTypeDefinition); // False
```

未绑定泛型类型的成员不能被动态调用。

19.2.4 以动态方式调用成员

一旦拥有了MemberInfo对象，就可以动态地调用它或者获取/设置它的值。这种操作称为动态绑定或后期绑定，因为要在运行时选择调用成员，而不是在编译时选择调用成员。

为了说明这一点，下列代码使用了原始的静态绑定：

```
string s = "Hello";
int length = s.Length;
```

下列代码通过反射以动态方式实现了相同的效果：

```
object s = "Hello";
PropertyInfo prop = s.GetType().GetProperty ("Length");
int length = (int) prop.GetValue (s, null);          // 5
```

使用GetValue和SetValue可以获取和设置PropertyInfo或FieldInfo的值。第一个参数是一个实例，对于静态成员来说它的值可以为null。除了在调用GetValue或SetValue时需要将索引器的值作为第二个参数外，访问索引器就像访问名为Item的特性一样。

要动态调用方法（如在MethodInfo上调用Invoke），应为该方法提供一组参数。如果参数类型错误，那么在运行时就会出现异常。在进行动态调用时，会失去编译时的类型安全，但是仍旧可以拥有运行时的类型安全（就像使用dynamic关键字一样）。

19.2.5 方法的参数

假设我们想要以动态方式调用string的Substring方法。要以静态方式实现该目标，就需要使用下列代码：

```
Console.WriteLine ("stamp".Substring(2));     // "amp"
```

下面是使用反射以动态方式实现相同的效果：

```
Type type = typeof (string);
Type[] parameterTypes = { typeof (int) };
MethodInfo method = type.GetMethod ("Substring", parameterTypes);

object[] arguments = { 2 };
object returnValue = method.Invoke ("stamp", arguments);
Console.WriteLine (returnValue);                          // "amp"
```

因为重载了Substring方法，所以必须将一组参数类型传递给GetMethod，以指定我们想要获取的类型。当未指定参数类型时，GetMethod会返回AmbiguousMatchException。

MethodBase（MethodInfo和ConstructorInfo的基础类）上定义的GetParameters方法可以返回参数的元数据。可以通过下列方式继续上一个示例：

```
ParameterInfo[] paramList = method.GetParameters();
foreach (ParameterInfo x in paramList)
{
  Console.WriteLine (x.Name);                          // startIndex
  Console.WriteLine (x.ParameterType);                 // System.Int32
}
```

1. 处理ref和out参数

要传递ref和out参数，可以在获取方法前调用类型上的MakeByRefType。例如下列代码：

```
int x;
bool successfulParse = int.TryParse ("23", out x);
```

可以通过下列动态方式实现以上代码的效果：

```
object[] args = { "23", 0 };
Type[] argTypes = { typeof (string), typeof (int).MakeByRefType() };
MethodInfo tryParse = typeof (int).GetMethod ("TryParse", argTypes);
bool successfulParse = (bool) tryParse.Invoke (null, args);

Console.WriteLine (successfulParse + " " + args[1]);          // True 23
```

对于ref和out参数类型这种方法会取得相同的效果。

2. 检索和调用泛型方法

在调用GetMethod时显式地指定参数类型对于避免重载方法非常重要。然而，泛型参数类型是无法指定的。例如下列代码，System.Linq.Enumerable类重载了Where方法：

```
public static IEnumerable<TSource> Where<TSource>
(this IEnumerable<TSource> source, Func<TSource, bool> predicate);

public static IEnumerable<TSource> Where<TSource>
(this IEnumerable<TSource> source, Func<TSource, int, bool> predicate);
```

要检索出特定的重载，必须检索所有方法，然后手动查找重载情况。下列查询检索了Where以前的重载情况：

```
from m in typeof (Enumerable).GetMethods()
where m.Name == "Where" && m.IsGenericMethod
let parameters = m.GetParameters()
where parameters.Length == 2
let genArg = m.GetGenericArguments().First()
let enumerableOfT = typeof (IEnumerable<>).MakeGenericType (genArg)
let funcOfTBool = typeof (Func<,>).MakeGenericType (genArg, typeof (bool))
where parameters[0].ParameterType == enumerableOfT
      && parameters[1].ParameterType == funcOfTBool
select m
```

在这个查询中调用.Single()可以为正确的MethodInfo对象提供未绑定的类型参数。下一步是通过调用MakeGenericMethod关闭这些类型参数：

```
var closedMethod = unboundMethod.MakeGenericMethod (typeof (int));
```

在这个示例中，我们已经通过int关闭了TSource，这样就可以通过IEnumerable<int>类型的source和Func<int,bool>类型的predicate调用Enumerable.Where：

```
int[] source = { 3, 4, 5, 6, 7, 8 };
Func<int, bool> predicate = n => n % 2 == 1; // 只能使用奇数
```

现在可以使用下列代码调用封闭式泛型方法：

```
var query = (IEnumerable<int>) closedMethod.Invoke
  (null, new object[] { source, predicate });

foreach (int element in query) Console.Write (element + "|"); // 3|5|7|
```

提示： 如果要使用System.Linq.Expressions API以动态方式创建表达式（第8章已详细介绍），无需指定泛型方法。重载Expression.Call方法可以为想要调用的方法指定封闭式类型参数：

```
int[] source = { 3, 4, 5, 6, 7, 8 };
Func<int, bool> predicate = n => n % 2 == 1;

var sourceExpr = Expression.Constant (source);
var predicateExpr = Expression.Constant (predicate);

var callExpression = Expression.Call (
  typeof (Enumerable), "Where",
  new[] { typeof (int) }, // 封闭式泛型类型
  sourceExpr, predicateExpr);
```

19.2.6 使用委托提高性能

动态调用的效率较低，因为重载通常需要花费几微秒。如果要在一个循环中重复调用某个方法，可以通过动态方式调用实例化的委托（其目标为动态方法）将每次调用的重载时间降低到纳秒级。下面的示例在没有明显时间花费的情况下，以动态方式调用了1000,000次string的Trim方法：

```
delegate string StringToString (string s);

static void Main()
{
  MethodInfo trimMethod = typeof (string).GetMethod ("Trim", new Type[0]);
  var trim = (StringToString) Delegate.CreateDelegate
                    (typeof (StringToString), trimMethod);
  for (int i = 0; i < 1000000; i++)
    trim ("test");
}
```

这种方式更加快速的原因是花费时间较多的动态绑定（以粗体显示）仅发生了一次。

19.2.7 访问非全局成员

类型上的所有用于探测元数据的方法（例如GetProperty、GetField等）都拥有使用BindingFlags枚举的重载过程。该枚举起元数据筛选器的作用，使用它可以更改默认的选择标准。该枚举的最常见用法是检索非全局成员。

例如，请观察下列类：

```
class Walnut
{
  private bool cracked;
  public void Crack() { cracked = true; }

  public override string ToString() { return cracked.ToString(); }
}
```

可以通过下列方式使Walnut类变得更安全：

```
Type t = typeof (Walnut);
Walnut w = new Walnut();
w.Crack();
FieldInfo f = t.GetField ("cracked", BindingFlags.NonPublic|
                                      BindingFlags.Instance);
f.SetValue (w, false);
Console.WriteLine (w);          // False
```

使用反射访问非全局成员可以获得很好的效果，但是这样做也具有危险性，因为可能会绕过封装，在类型的内部实现中创建无法管理的独立性。

BindingFlags枚举

BindingFlags专门被设计成按位组合。为了获取任何程度上的匹配，需要以下列4种组合作为起点：

```
BindingFlags.Public | BindingFlags.Instance
BindingFlags.Public | BindingFlags.Static
BindingFlags.NonPublic | BindingFlags.Instance
BindingFlags.NonPublic | BindingFlags.Static
```

NonPublic包括internal、protected、protected internal和private。

下面的示例检索了object类型的所有静态全局成员：

```
BindingFlags publicStatic = BindingFlags.Public | BindingFlags.Static;
MemberInfo[] members = typeof (object).GetMembers (publicStatic);
```

下面的示例检索了object类型的所有非全局成员，包括静态成员和实例：

```
BindingFlags nonPublicBinding =
  BindingFlags.NonPublic | BindingFlags.Static | BindingFlags.Instance;

MemberInfo[] members = typeof (object).GetMembers (nonPublicBinding);
```

除非从基础类型继承的函数被覆盖，否则DeclaredOnly标记（flag）会排除它们。

提示：DeclaredOnly标记（flag）因其对结果集的限制（而其他绑定的标记（flag）会扩展结果集）会使人有些困惑。

19.2.8 泛型方法

泛型方法不能被直接调用；下列代码抛出一个异常：

```
class Program
{
  public static T Echo<T> (T x) { return x; }

  static void Main()
  {
    MethodInfo echo = typeof (Program).GetMethod ("Echo");
    Console.WriteLine (echo.IsGenericMethodDefinition);          // True
    echo.Invoke (null, new object[] { 123 } );                   // 异常
  }
}
```

必须调用MethodInfo的MakeGenericMethod，指定具体的泛型类型参数。执行这样的操作返回另一个MethodInfo，可以通过下列方式调用它：

```
MethodInfo echo = typeof (Program).GetMethod ("Echo");
MethodInfo intEcho = echo.MakeGenericMethod (typeof (int));
Console.WriteLine (intEcho.IsGenericMethodDefinition);             // False
Console.WriteLine (intEcho.Invoke (null, new object[] { 3 } ));    // 3
```

19.2.9 以匿名方式调用泛型接口的成员

当需要调用泛型接口的成员并且在运行前未指定类型参数时，反射非常有用。理论上，如果类型被设计得非常完美，需要使用反射的情况就会非常少；但是，类型不会总是被设计得非常完美。

例如，假设我们想要为ToString编写一个功能更加强大的版本，使其可以扩展LINQ查询的结果。可以将下列代码作为起点：

```
public static string ToStringEx <T> (IEnumerable<T> sequence)
{
  ...
}
```

这样操作还有许多限制。如果序列中含有需要封装的嵌套序列时，必须重载该方法才能实现：

```
public static string ToStringEx <T> (IEnumerable<IEnumerable<T>> sequence)
```

那么当序列含有分组或者嵌套序列的投影时，方法重载的静态解决方案会变得不切实际，因为需要使用可以调节的方式处理任意的对象图，如下面列出的：

```
public static string ToStringEx (object value)
{
  if (value == null) return "<null>";
  StringBuilder sb = new StringBuilder();

  if (value is List<>)                                                // Error
    sb.Append ("List of " + ((List<>) value).Count + " items");       // Error

  if (value is IGrouping<,>)                                          // Error
    sb.Append ("Group with key=" + ((IGrouping<,>) value).Key);       // Error

  // 如果是集合则枚举集合元素，递归地调用ToStringEx()
  // ...

  return sb.ToString();
}
```

但是，这段代码无法编译，因为无法调用未绑定的泛型类型（如List<>或IGrouping<>）的成员。在使用List<>时，可以通过使用非泛型IList接口解决该问题：

```
if (value is IList)
  sb.AppendLine ("A list with " + ((IList) value).Count + " items");
```

提示：我们这样做的原因是List<>的设计者已经预见到要实现IList类（以及IList泛型）。当自定义泛型类型时应考虑相同的原则：拥有一个可以使用户后退的非泛型的接口或基础类十分重要。

在使用IGrouping<,>时解决方案比较复杂。下面的代码定义了该接口：

```
public interface IGrouping <TKey,TElement> : IEnumerable <TElement>,IEnumerable
{
  TKey Key { get; }
}
```

没有用于访问Key特性的非泛型类型，因此必须使用反射。这个解决方案不是要调用未绑定泛型类型的成员（这是可以做到的），而是要调用封闭式泛型类型的成员，其类型参数是在运行时创建的。

提示： 下一章将通过C#的dynamic关键字以更简单的方式解决这个问题。动态绑定的一个明确的标志是出现类型必须进行巧妙处理的情况，即像我们现在做的那样。

首先确定value是否实现IGrouping<,>，如果是，那么应获取其封闭式泛型接口。LINQ查询是最容易实现这一操作的方法。然后可以检索并调用Key特性：

```
public static string ToStringEx (object value)
{
  if (value == null) return "<null>";
  if (value.GetType().IsPrimitive) return value.ToString();

  StringBuilder sb = new StringBuilder();

  if (value is IList)
    sb.Append ("List of " + ((IList)value).Count + " items: ");

  Type closedIGrouping = value.GetType().GetInterfaces()
    .Where (t => t.IsGenericType &&
                 t.GetGenericTypeDefinition() == typeof (IGrouping<,>))
    .FirstOrDefault();

  if (closedIGrouping != null)    // 调用IGrouping<,>上的Key特性
  {
    PropertyInfo pi = closedIGrouping.GetProperty ("Key");
    object key = pi.GetValue (value, null);
    sb.Append ("Group with key=" + key + ": ");
  }

  if (value is IEnumerable)
    foreach (object element in ((IEnumerable)value))
      sb.Append (ToStringEx (element) + " ");

  if (sb.Length == 0) sb.Append (value.ToString());

  return "\r\n" + sb.ToString();
}
```

这个方法非常强大：不论IGrouping<,>被显式地或隐式地实现，它都会起作用。下列代码演示了该方法：

```
Console.WriteLine (ToStringEx (new List<int> { 5, 6, 7 } ));
Console.WriteLine (ToStringEx ("xyyzzz".GroupBy (c => c) ));
```

List of 3 items: 5 6 7

Group with key=x: x
Group with key=y: y y
Group with key=z: z z z

反射和元数据

19.3 反射程序集

通过调用Assembly对象上的GetType或GetTypes，可以动态反射程序集。下列代码对当前的程序集进行了检索，其中TestProgram位于Demos命名空间中：

```
Type t = Assembly.GetExecutingAssembly().GetType ("Demos.TestProgram");
```

在Metro应用中，可以从现有类型获得程序集：

```
typeof (Foo).GetTypeInfo().Assembly.GetType ("Demos.TestProgram");
```

下一个示例列出了*e:\demo*中的*mylib.dll*程序集的所有类型：

```
Assembly a = Assembly.LoadFrom (@"e:\demo\mylib.dll");

foreach (Type t in a.GetTypes())
  Console.WriteLine (t);
```

或者，在Metro应用中：

```
Assembly a = typeof (Foo).GetTypeInfo().Assembly;

foreach (Type t in a.ExportedTypes)
  Console.WriteLine (t);
```

GetTypes仅会返回顶级类型和非嵌套类型。

19.3.1 将程序集加载到仅能应用反射的环境中

上一个示例为了列出程序集的所有类型，将程序集加载到当前的应用程序域中。这样操作可能会产生不良效果，如执行静态构造方法或干扰以后的类型分解。如果仅需要检查类型信息（而不是要实例化或调用类型），那么可以通过将程序集加载到仅能应用反射的环境中解决该问题：

```
Assembly a = Assembly.ReflectionOnlyLoadFrom (@"e:\demo\mylib.dll");
Console.WriteLine (a.ReflectionOnly); // True

foreach (Type t in a.GetTypes())
Console.WriteLine (t);
```

这段代码是编写类浏览器的开头部分。

可以使用以下三种方法将程序集加载到仅能应用反射的环境中：

- ReflectionOnlyLoad (byte[])
- ReflectionOnlyLoad (string)
- ReflectionOnlyLoadFrom (string)

提示：即使在仅能应用反射的环境中，也不能加载*mscorlib.dll*的多个版本。因此，类浏览器（如Lutz Roeder编写的.NET Reflector）是使用自定义类和映射到非托管元数据接口的数据类型编写的。

19.3.2 模块

调用多模块程序集上的GetTypes会返回所有模块中的所有类型。因此，可以忽略模块的存在并将程序集作为一种类型容器处理。但是，当模块之间具有关联性时，就应该对元数据令牌进行处理。

元数据令牌是整数，在模块范围中每个类型、成员、字符串或源的元数据令牌都是唯一的。IL使用了元数据令牌，因此如果正在分析IL，就需要解决这些元数据令牌。Module类型中定义了实现方法，这些方法是ResolveType、ResolveMember、ResolveString和ResolveSignature。本章最后一节“编写反编译器”将介绍这些内容。

通过调用GetModules可以获得程序集中所有模块的列表。通过程序集主模块的ManifestModule属性，可以直接访问程序集主模块。

19.4 使用属性

CLR允许通过属性将额外的元数据附加到类型、成员和程序集上。这是许多CLR函数（如串行化和安全性）的运行机制，从而使一些属性变为应用程序的不可见部分。

属性的关键特点是可以自己编写，然后可以像其他属性那样使用额外的信息添加代码元素。这些额外的信息可以编译到基础程序集中，而且还可以在运行时使用反射创建说明性服务（如自动单元测试）检索这些额外的信息。

19.4.1 属性基础

C#中有三种属性：

- 位映射属性
- 自定义属性
- 伪自定义属性

在这三种属性中，只有自定义属性是可扩展的。

提示：但是在C#中术语“属性”大多数情况代表自定义属性或伪自定义属性。

位映射属性（本书使用的术语）可以映射到类型元数据中的指定位。C#的大多数关键字（如public、abstract和sealed）可以编译为位映射属性。这些属性非常高效，因为它们在元数据中占用最小的空间（通常只有1位），而且CLR通过很少的间接寻址或不通过间接寻址就可以定位它们。反射API通过在类型（和其他MemberInfo子类）上指定特性公开它们，如IsPublic、IsAbstract和IsSealed。Attributes特性会返回在大多数情况中使用1位描述位映射属性的位域枚举：

```
static void Main()
{
  TypeAttributes ta = typeof (Console).Attributes;
  MethodAttributes ma = MethodInfo.GetCurrentMethod().Attributes;
  Console.WriteLine (ta + "\r\n" + ma);
}
```

下面列出的是结果：

反射和元数据

```
AutoLayout, AnsiClass, Class, Public, Abstract, Sealed, BeforeFieldInit
PrivateScope, Private, Static, HideBySig
```

比较而言，自定义属性可以编译为类型主元数据表上的二进制大对象。所有自定义属性都是由System.Attribute子类表示的，而且与位映射属性不同，它们是可以扩展的。元数据中的二进制大对象标识了属性类，而且还保存了在应用该属性时任何定位或命名参数的值。自定义属性与.NET Framework中定义的属性具有相同的结构。

第4章介绍了将自定义属性附加到C#中的类或成员上的方法。下面的代码将预定义的Obsolete属性附加到Foo类上：

```
[Obsolete] public class Foo {...}
```

该代码会命令编译器将一个ObsoleteAttribute实例合并到Foo的元数据中，然后在运行时可以通过调用Type或MemberInfo对象上的GetCustomAttributes反射该元数据。

伪自定义属性与标准的自定义属性类似。它们由System.Attribute子类表示并且通过标准方式附加：

```
[Serializable] public class Foo {...}
```

伪自定义属性与自定义属性之间的差异在于编译器或CLR在内部通过将伪自定义属性转换为位映射属性，对其进行最优化。请参阅[Serializable]（第17章）、StructLayout、In和Out示例（第25章）。反射通过指定特性（如IsSerializable）保留伪自定义属性，而且当调用GetCustomAttributes（包括SerializableAttribute）时，在许多情况中伪自定义属性还会作为System.Attribute对象返回。这意味着几乎可以忽略伪自定义属性和非伪自定义属性之间的差异（注意当使用Reflection.Emit在运行时以动态方式生成类型是一个异常情况，请参阅本章稍后的介绍）。

19.4.2 AttributeUsage属性

AttributeUsage是一种应用于Attribute类的属性。它可以指示编译器使用目标属性的方式：

```
public sealed class AttributeUsageAttribute : Attribute
{
  public AttributeUsageAttribute (AttributeTargets validOn);

  public bool AllowMultiple      { get; set; }
  public bool Inherited          { get; set; }
  public AttributeTargets ValidOn { get; }
}
```

AllowMultiple控制定义的属性是否可以对相同的目标多次应用；Inherited控制是否可以对属性应用子类型。ValidOn控制可以附加属性的目标集合（类、接口、特性、方法、参数等）。它可以接收来自AttributeTargets枚举的任何组合值，AttributeTargets枚举拥有下列成员：

全部	委托	泛型参数	参数
程序集	枚举	接口	特性
类	事件	方法	返回值
构造方法	字段	模块	结构

下面的代码演示了在.NET Framework中将AttributeUsage应用于Serializable属性的方式：

```
[AttributeUsage (AttributeTargets.Delegate |
                 AttributeTargets.Enum     |
                 AttributeTargets.Struct   |
                 AttributeTargets.Class, Inherited = false)
]
public sealed class SerializableAttribute : Attribute
{
}
```

实际上，这段代码是Serializable属性的完整定义。编写没有特性或特殊构造方法的属性类就是这样简单。

19.4.3 定义自定义属性

下面是自定义属性的定义方式：

1. 从System.Attribute或System.Attribute生成一个子类。尽管没有明确规定，但根据惯例这个类的名称应以Attribute作为后缀。
2. 以上一节描述的方式应用AttributeUsage属性。

   如果该自定义属性在其构造方法中不需要特性或参数，那么编写自定义属性的工作就完成了。
3. 编写一个或多个全局构造方法。这些构造方法的参数定义了该属性的定位参数，而且在使用该属性时这些构造方法的参数会变成强制的。
4. 为每个已命名参数声明一个全局域或特性。在使用该属性时已命名参数是可选的。

提示： 属性类型和构造方法参数必须为以下类型：

- 密封的基本类型：bool、byte、char、double、float、int、long、short或string
- Type类型
- 枚举类型
- 以上类型的一维数组

当应用了某个属性后，编译器还必须能够以静态方式评估每个特性或构造方法参数。

为实现自动单元测试，下列类定义了一个属性。它指定了应测试的方法、测试的重复次数和失败时应显示的消息：

```
[AttributeUsage (AttributeTargets.Method)]
public sealed class TestAttribute : Attribute
{
  public int Repetitions;
  public string FailureMessage;

  public TestAttribute () : this (1)     { }
  public TestAttribute (int repetitions) { Repetitions = repetitions; }
}
```

下面的Foo类带有使用Test属性以各种方式修饰的方法：

```
class Foo
```

```
{
  [Test]
  public void Method1() { ... }

  [Test(20)]
  public void Method2() { ... }

  [Test(20, FailureMessage="Debugging Time!")]
  public void Method3() { ... }
}
```

19.4.4 在运行时检索属性

在运行时有两种检索属性的标准方法：

- 调用Type或MemberInfo对象上的GetCustomAttributes。
- 调用Attribute.GetCustomAttribute或Attribute.GetCustomAttributes。

为了接收与合法属性目标（Type、Assembly、Module、MemberInfo或ParameterInfo）对应的任何反射对象，后两种方法会被重载。

提示： Framework 4.0还可以根据类型或成员调用GetCustomAttributesData()，以获得属性信息。这种方式与使用GetCustomAttributes()之间的差异是可以了解该属性的实例化方式：它可以提供使用的构造方法的重载情况，以及每个构造方法参数和已命名参数的值。当为了将属性重构为相同的设置需要发出代码或IL时，这种方式非常有用（请参阅本章后面的内容，以了解详细情况）。

下面的代码演示了在之前列举的拥有TestAttribute的Foo类中枚举每个方法的方式：

```
foreach (MethodInfo mi in typeof (Foo).GetMethods())
{
  TestAttribute att = (TestAttribute) Attribute.GetCustomAttribute
    (mi, typeof (TestAttribute));

   if (att != null)
  Console.WriteLine ("Method {0} will be tested; reps={1}; msg={2}",
                      mi.Name, att.Repetitions, att.FailureMessage);
}
```

或者，在Metro应用中：

```
foreach (MethodInfo mi in typeof (Foo).GetTypeInfo().DeclaredMethods)
...
```

下面是输出结果：

```
Method Method1 will be tested; reps=1; msg=
Method Method2 will be tested; reps=20; msg=
Method Method3 will be tested; reps=20; msg=Debugging Time!
```

为了完整地说明使用这种方式实现单元测试，下面列出了一个扩展示例，因此它实际上是调用使用Test属性修饰的方法：

```
foreach (MethodInfo mi in typeof (Foo).GetMethods())
```

```
{
  TestAttribute att = (TestAttribute) Attribute.GetCustomAttribute
    (mi, typeof (TestAttribute));

  if (att != null)
    for (int i = 0; i < att.Repetitions; i++)
      try
      {
        mi.Invoke (new Foo(), null);     // 调用无参数方法
      }
      catch (Exception ex)               // 在att.FailureMessage中封装异常
      {
        throw new Exception ("Error: " + att.FailureMessage, ex);
      }
}
```

返回属性反射，下面的示例列出了特定类型上显示的属性列表：

```
[Serializable, Obsolete]
class Test
{
  static void Main()
  {
    object[] atts = Attribute.GetCustomAttributes (typeof (Test));
    foreach (object att in atts) Console.WriteLine (att);
  }
}
```

输出结果：

```
System.ObsoleteAttribute
System.SerializableAttribute
```

19.4.5 在仅能应用反射的环境中检索属性

禁止调用加载到仅能应用反射环境中的成员上的GetCustomAttributes，因为这样做需要实例化以任意方式类型化的属性（注意不允许在仅能应用反射的环境中进行对象实例化）。为了处理这种情况，可以使用特殊类型CustomAttributeData反射这些属性。下面的示例演示了该类型的使用方式：

```
IList<CustomAttributeData> atts = CustomAttributeData.GetCustomAttributes
                                  (myReflectionOnlyType);
foreach (CustomAttributeData att in atts)
{
  Console.Write (att.GetType());                      // 属性类型

  Console.WriteLine (" " + att.Constructor);          // ConstructorInfo对象

  foreach (CustomAttributeTypedArgument arg in att.ConstructorArguments)
    Console.WriteLine (" " +arg.ArgumentType + "=" + arg.Value);

  foreach (CustomAttributeNamedArgument arg in att.NamedArguments)
    Console.WriteLine (" " + arg.MemberInfo.Name + "=" + arg.TypedValue);
}
```

在许多情况中，这些属性类型将位于不同于反射程序集的其他程序集中。限定其范围的方法是在当前的应用程序域上处理ReflectionOnlyAssemblyResolve事件：

```
ResolveEventHandler handler = (object sender, ResolveEventArgs args)
                              => Assembly.ReflectionOnlyLoad (args.Name);
AppDomain.CurrentDomain.ReflectionOnlyAssemblyResolve += handler;

// 反射属性...

AppDomain.CurrentDomain.ReflectionOnlyAssemblyResolve -= handler;
```

19.5 动态生成代码

System.Reflection.Emit命名空间含有用于在运行时创建元数据和IL的类。对于某些编程任务，以动态方式生成代码非常有用。API常规表达式就是一个例子，它会发出调整为特定常规表达式的高性能类型。在Framework中使用Reflection.Emit的其他情况包括为Remoting动态生成传输委托，以及使用最短运行时间重载动态生成执行特定XSLT转换的类型。LINQPad使用Reflection.Emit动态生成类型化的DataContext类。

19.5.1 使用DynamicMethod生成IL

DynamicMethod类是System.Reflection.Emit命名空间中的轻量级工具，它用于在运行时生成方法。与TypeBuilder不同，DynamicMethod类不需要设置包含方法的动态程序集、模块和类型。这个特点使其非常适于完成简单的任务，以及起到引入Reflection.Emit的作用。

提示： 当不再被引用时，DynamicMethod和相关的IL都会被收集起来。这意味着可以重复生成动态方法，而无需占用内存。相反，动态程序集不能从内存卸载，除非包含它们的应用程序域被卸载。

下面这段代码是对DynamicMethod的简单使用，它创建了一个将Hello world写入控制台的方法：

```
public class Test
{
  static void Main()
  {
    var dynMeth = new DynamicMethod ("Foo", null, null, typeof (Test));
    ILGenerator gen = dynMeth.GetILGenerator();
    gen.EmitWriteLine ("Hello world");
    gen.Emit (OpCodes.Ret);
    dynMeth.Invoke (null, null);                // Hello world
  }
}
```

OpCodes对于每个IL操作码都拥有一个静态只读字段。尽管ILGenerator还拥有专门用于生成标签和局部变量以及处理异常情况的方法，但是它的大部分功能是通过各种操作码实现的。这些方法通常带有Opcodes.Ret后缀，Opcodes.Ret代表“返回”。ILGenerator上的EmitWriteLine方法是发出大量低级操作码的快捷方式。我们使用下面这段代码代替对EmitWriteLine的调用，可以得到相同的结果：

```
MethodInfo writeLineStr = typeof (Console).GetMethod ("WriteLine",
                          new Type[] { typeof (string) });
gen.Emit (OpCodes.Ldstr, "Hello world");     // 加载一个字符串
gen.Emit (OpCodes.Call, writeLineStr);       // 调用一个方法
```

注意，我们将typeof(Test)传递到了DynamicMethod的构造方法中。这样做可以使动态方法访问这种类型的非全局方法，从而可以执行以下操作：

```
public class Test
{
  static void Main()
  {
    var dynMeth = new DynamicMethod ("Foo", null, null, typeof (Test));
    ILGenerator gen = dynMeth.GetILGenerator();

    MethodInfo privateMethod = typeof(Test).GetMethod ("HelloWorld",
      BindingFlags.Static | BindingFlags.NonPublic);

    gen.Emit (OpCodes.Call, privateMethod);          // 调用HelloWorld
    gen.Emit (OpCodes.Ret);

    dynMeth.Invoke (null, null);                     // Hello world
  }

  static void HelloWorld()          // 私有方法，而且可以调用它
  {
    Console.WriteLine ("Hello world");
  }
}
```

了解IL需要花费大量的时间。与了解所有的操作码相比，编译C#程序然后检查、复制并调整IL更加简单。程序集查看工具（如ildasm或Lutz Roeder的Reflector）是完成这项工作的最佳工具。

19.5.2 评估堆栈

IL的核心概念是评估堆栈。评估堆栈不同于存储局部变量和方法参数的堆栈。要调用带参数的方法，首先应将这些参数推（加载）到评估堆栈中，然后调用该方法。该方法会从评估堆栈弹出它需要的参数，在前面介绍调用Console.WriteLine的内容中对此进行过讲解。下面这段代码是一个使用整数的类似示例：

```
var dynMeth = new DynamicMethod ("Foo", null, null, typeof(void));
ILGenerator gen = dynMeth.GetILGenerator();
MethodInfo writeLineInt = typeof (Console).GetMethod ("WriteLine",
                                        new Type[] { typeof (int) });

// Ldc*操作码加载了各种类型和大小的数值。

gen.Emit (OpCodes.Ldc_I4, 123);       // 将一个4字节的整数推入堆栈中
gen.Emit (OpCodes.Call, writeLineInt);

gen.Emit (OpCodes.Ret);
dynMeth.Invoke (null, null);                     // 123
```

要同时添加两个数字，首先应将每个数字加载到评估堆栈中，然后调用Add。Add操作码会从评估堆栈弹出两个值并将结果推入评估堆栈。下面这段代码添加了2和2，然后使用先前获得的writeLine方法写结果：

```
gen.Emit (OpCodes.Ldc_I4, 2);                    // 推入一个4字节整数，其值等于2
gen.Emit (OpCodes.Ldc_I4, 2);                    // 推入一个4字节整数，其值等于2
gen.Emit (OpCodes.Add);                          // 同时添加结果
gen.Emit (OpCodes.Call, writeLineInt);
```

要计算10/2+1，既可以使用下面这段代码：

```
gen.Emit (OpCodes.Ldc_I4, 10);
gen.Emit (OpCodes.Ldc_I4, 2);
gen.Emit (OpCodes.Div);
gen.Emit (OpCodes.Ldc_I4, 1);
gen.Emit (OpCodes.Add);
gen.Emit (OpCodes.Call, writeLineInt);
```

也可以使用下面这段代码：

```
gen.Emit (OpCodes.Ldc_I4, 1);
gen.Emit (OpCodes.Ldc_I4, 10);
gen.Emit (OpCodes.Ldc_I4, 2);
gen.Emit (OpCodes.Div);
gen.Emit (OpCodes.Add);
gen.Emit (OpCodes.Call, writeLineInt);
```

19.5.3 向动态方法传递参数

可以使用Ldarg和Ldarg_*XXX*操作码，将传递到动态方法中的参数加载到堆栈中。要返回某个值，可以在结束时在堆栈中精确地保留这个值。要实现该方式，必须在调用DynamicMethod的构造方法时指定返回类型和参数类型。下面这段代码创建了一个返回两个整数和的动态方法：

```
DynamicMethod dynMeth = new DynamicMethod ("Foo",
  typeof (int),                              //返回类型= int
  new[] { typeof (int), typeof (int) },      //参数类型= int, int
  typeof (void));

ILGenerator gen = dynMeth.GetILGenerator();

gen.Emit (OpCodes.Ldarg_0);          //将第一个参数推入评估堆栈
gen.Emit (OpCodes.Ldarg_1);          //将第二个参数推入评估堆栈
gen.Emit (OpCodes.Add);              //将它们相加（结果在堆栈中）
gen.Emit (OpCodes.Ret);              //返回拥有1个值的堆栈

int result = (int) dynMeth.Invoke (null, new object[] { 3, 4 } ); // 7
```

警告： 当结束时，评估堆栈必须精确地拥有0或1个项目（由方法是否返回1个值决定）。如果违反了这个原则，CLR将无法执行方法。可以在不对堆栈进行处理的情况下，使用OpCodes.Pop从堆栈中删除1个项目。

与调用Invoke不同，将动态方法作为类型化的委托更加方便，CreateDelegate方法就可以实现。为了说明这一点，我们定义了一个名为BinaryFunction的委托：

```
delegate int BinaryFunction (int n1, int n2);
```

然后可以将上例的最后一行代码替换为下列代码：

```
BinaryFunction f = (BinaryFunction) dynMeth.CreateDelegate(typeof (BinaryFunction));
int result = f (3, 4); // 7
```

提示：委托还可以省去调用动态方法的系统开销，节省每次调用需要花费的几微秒时间。

本章稍后将介绍通过引用进行传递的方式。

19.5.4 生成局部变量

通过调用ILGenerator上的DeclareLocal声明局部变量，可以返回一个LocalBuilder对象，该对象可以与Ldloc（加载一个局部变量）或Stloc（存储一个局部变量）之类的操作码协同使用。Ldloc可以将数值推入评估堆栈；Stloc可以将数值弹出评估堆栈。例如下面这段代码：

```
int x = 6;
int y = 7;
x *= y;
Console.WriteLine (x);
```

下面这段代码能够以动态方式生成上面的代码：

```
var dynMeth = new DynamicMethod ("Test", null, null, typeof (void));
ILGenerator gen = dynMeth.GetILGenerator();

LocalBuilder localX = gen.DeclareLocal (typeof (int));            //声明x
LocalBuilder localY = gen.DeclareLocal (typeof (int));            //声明y

gen.Emit (OpCodes.Ldc_I4, 6);         //将数值6推入评估堆栈
gen.Emit (OpCodes.Stloc, localX);     //存入localX
gen.Emit (OpCodes.Ldc_I4, 7);         //将数值7推入评估堆栈
gen.Emit (OpCodes.Stloc, localY);     //存入localY

gen.Emit (OpCodes.Ldloc, localX);     //将localX推入评估堆栈
gen.Emit (OpCodes.Ldloc, localY);     //将localY推入评估堆栈
gen.Emit (OpCodes.Mul);               //多个值相乘
gen.Emit (OpCodes.Stloc, localX);     //将结果存储到localX中

gen.EmitWriteLine (localX);           //编写localX的值
gen.Emit (OpCodes.Ret);

dynMeth.Invoke (null, null);          // 42
```

提示：Lutz Roeder的Reflector还具有检查动态方法错误的强大功能。当反编译C#程序时，它通常可以显示错误之处！本章稍后介绍将动态发出的方法保存到磁盘。

19.5.5 分支

IL中没有while、do和for循环；这些循环是通过标签、相等goto和条件goto语句实现的。它们是分支操作码，如Br（无条件分支）、Brtrue（如果评估堆栈中的值为true则分支）和Blt（如果第一个值小于第二个值则分支）。

要设置分支目标，首先应调用DefineLabel（返回一个Label对象），然后在需要标记标签时调用MarkLabel，请看下列代码：

```
int x = 5;
while (x <= 10) Console.WriteLine (x++);
```

反射和元数据

可以使用以下方式发出这段代码：

```
ILGenerator gen = ...

Label startLoop = gen.DefineLabel();                    // 声明标签
Label endLoop = gen.DefineLabel();

LocalBuilder x = gen.DeclareLocal (typeof (int));      // int x
gen.Emit (OpCodes.Ldc_I4, 5);                           //
gen.Emit (OpCodes.Stloc, x);                            // x = 5
gen.MarkLabel (startLoop);
  gen.Emit (OpCodes.Ldc_I4, 10);       // 将10加载到评估堆栈中
  gen.Emit (OpCodes.Ldloc, x);         // 将x加载到评估堆栈中

  gen.Emit (OpCodes.Blt, endLoop);     // 如果（x > 10）则跳转到endLoop

  gen.EmitWriteLine (x);               // Console.WriteLine (x)

  gen.Emit (OpCodes.Ldloc, x);         // 将x加载到评估堆栈中
  gen.Emit (OpCodes.Ldc_I4, 1);        // 将1加载到评估堆栈中
  gen.Emit (OpCodes.Add);              // 将它们相加
  gen.Emit (OpCodes.Stloc, x);         // 将结果保存回x

  gen.Emit (OpCodes.Br, startLoop);    // 返回循环的开始
gen.MarkLabel (endLoop);

gen.Emit (OpCodes.Ret);
```

19.5.6 实例化对象和调用实例方法

new等价于IL中的Newobj操作码。下面这段代码使用了一个构造方法并将结构化对象加载到评估堆栈中。例如，通过下列代码构造StringBuilder：

```
var dynMeth = new DynamicMethod ("Test", null, null, typeof (void));
ILGenerator gen = dynMeth.GetILGenerator();

ConstructorInfo ci = typeof (StringBuilder).GetConstructor (new Type[0]);
gen.Emit (OpCodes.Newobj, ci);
```

一旦将对象推入评估堆栈中，就可以使用Call或Callvirt操作码调用其实例方法。为了扩展这个示例，我们通过特性访问器查询StringBuilder的MaxCapacity特性，然后写出结果：

```
gen.Emit (OpCodes.Callvirt, typeof (StringBuilder)
                              .GetProperty ("MaxCapacity").GetGetMethod());

gen.Emit (OpCodes.Call, typeof (Console).GetMethod ("WriteLine",
                                         new[] { typeof (int) } ));

gen.Emit (OpCodes.Ret);
dynMeth.Invoke (null, null);                 // 2147483647
```

为了采用C#的调用语义：

- 应使用Call调用静态方法和值类型实例方法。
- 应使用Callvirt调用引用类型实例方法（不论它们是否被声明为虚方法）。

在上一个示例中，我们在StringBuilder实例上使用Callvirt，即使MaxProperty不是虚拟的。这不会导致错误，仅会执行一个非虚拟调用。使用Callvirt调用引用类型实例方法可以避免相反条件

的危险：使用Call调用虚方法（这个危险是真实存在的。目标方法可能以后会更改其声明）。

警告： 使用Call通过虚拟调用语义来调用虚方法，然后直接调用该方法。实际上，很少需要采用这种方式，而且这样做也会影响类型的安全性。

下面的示例构造了一个传递两个参数的StringBuilder，它将，“world!”附加到StringBuilder上，然后调用StringBuilder上的ToString：

```
// 我们将调用：new StringBuilder ("Hello", 1000)

ConstructorInfo ci = typeof (StringBuilder).GetConstructor (
                     new[] { typeof (string), typeof (int) } );

gen.Emit (OpCodes.Ldstr, "Hello");            // 将字符串加载到评估堆栈中
gen.Emit (OpCodes.Ldc_I4, 1000);              // 将整数加载到评估堆栈中
gen.Emit (OpCodes.Newobj, ci);                // 构造StringBuilder

Type[] strT = { typeof (string) };
gen.Emit (OpCodes.Ldstr, ", world!");
gen.Emit (OpCodes.Call, typeof (StringBuilder).GetMethod ("Append", strT));
gen.Emit (OpCodes.Callvirt, typeof (object).GetMethod ("ToString"));
gen.Emit (OpCodes.Call, typeof (Console).GetMethod ("WriteLine", strT));
gen.Emit (OpCodes.Ret);
dynMeth.Invoke (null, null);                  // Hello, world!
```

为了增加趣味我们调用了typeof(object)上的GetMethod，然后使用Callvirt在ToString上执行虚拟方法调用。通过调用StringBuilder类型上的ToString也可以得到相同的结果：

```
gen.Emit (OpCodes.Callvirt, typeof (StringBuilder).GetMethod ("ToString",
                                                    new Type[0] ));
```

（在调用GetMethod时需要使用空类型数组，因为StringBuilder使用另一个签名重载ToString。）

提示： 如果我们已经以非虚拟方式调用了object的ToString方法：

```
gen.Emit (OpCodes.Call,
          typeof (object).GetMethod ("ToString"));
```

该结果是System.Text.StringBuilder。换句话说，涵盖了StringBuilder的ToString方法并直接调用了object的方法。

19.5.7 异常情况处理

ILGenerator为异常情况处理提供了专门的方法。请看下列C#代码：

```
try                               { throw new NotSupportedException(); }
catch (NotSupportedException ex) { Console.WriteLine (ex.Message);    }
finally                           { Console.WriteLine ("Finally");     }
```

下面是对上述代码的转换：

```
MethodInfo getMessageProp = typeof (NotSupportedException)
                            .GetProperty ("Message").GetGetMethod();
```

```
MethodInfo writeLineString = typeof (Console).GetMethod ("WriteLine",
                                        new[] { typeof (object) } );

gen.BeginExceptionBlock();
  ConstructorInfo ci = typeof (NotSupportedException).GetConstructor (
                                                      new Type[0] );
  gen.Emit (OpCodes.Newobj, ci);
  gen.Emit (OpCodes.Throw);
gen.BeginCatchBlock (typeof (NotSupportedException));
  gen.Emit (OpCodes.Callvirt, getMessageProp);
  gen.Emit (OpCodes.Call, writeLineString);
gen.BeginFinallyBlock();
  gen.EmitWriteLine ("Finally");
gen.EndExceptionBlock();
```

就像在C#中一样，可以包括多个catch块。为了再次抛出相同的异常，可以使用Rethrow操作码。

警告：ILGenerator提供了一个功能更强的辅助方法ThrowException。但是该方法含有一个小bug，这个bug使其无法与DynamicMethod一起使用，而只能与MethodBuilder一起使用（请参阅下一节）。

19.6 发出程序集和类型

尽管发出程序集和类型非常方便，但只能生成方法。如果需要发出任何其他构造（或完整的类型），就需要使用全部“最重量级的”API。这意味着以动态方式创建程序集和模块。然而，无需将程序集保存到磁盘上；程序集完全可以存在于内存中。

假设需要以动态方式创建一个类型。因为类型必须存在于程序集的模块中，所以在创建类型前必须先创建程序集和模块。下面这段代码创建了AssemblyBuilder和ModuleBuilder类型：

```
AppDomain appDomain = AppDomain.CurrentDomain;

AssemblyName aname = new AssemblyName ("MyDynamicAssembly");

AssemblyBuilder assemBuilder =
  appDomain.DefineDynamicAssembly (aname, AssemblyBuilderAccess.Run);

ModuleBuilder modBuilder = assemBuilder.DefineDynamicModule ("DynModule");
```

提示：不能将类型添加到已经存在的程序集中，因为程序集一旦被创建就不可改变。

除非在定义程序集时指定了AssemblyBuilderAccess.RunAndCollect，否则动态程序集不支持垃圾收集，它们在应用域结果之前都会驻留在内存中。可收集程序有许多不同的限制条件（具体见*http://albahari.com/dynamiccollect*）。

一旦拥有了可以包含类型的模块后，就可以使用TypeBuilder创建类型。下面的代码定义了一个名为Widget的类：

```
TypeBuilder tb = modBuilder.DefineType ("Widget", TypeAttributes.Public);
```

TypeAttributes标志枚举支持CLR类型修饰符，当使用*ildasm*反汇编类型时就可以看到这种情况。成员可见性标志也是同样的道理，这包括类型修饰符（如Abstract和Sealed）以及用于定义.NET接

口的Interface。这些修饰符还包括Serializable，使用它的效果与在C#中应用[Serializable]属性的效果相同；另外还包括Explicit，使用它的效果与应用[StructLayout(LayoutKind.Explicit)]的效果相同。本章稍后将介绍应用这些属性的方法。

提示：DefineType方法还可以接收可选的基础类型：

- 要定义结构，应设定System.ValueType的基础类型。
- 要定义委托，应设定System.MulticastDelegate的基础类型。
- 要实现接口，可使用接收接口类型数组的构造方法。
- 要定义接口，可设定TypeAttributes.Interface | TypeAttributes.Abstract。

定义委托类型需要许多额外的操作。

现在创建类型中的成员：

```
MethodBuilder methBuilder = tb.DefineMethod ("SayHello",
                                             MethodAttributes.Public,
                                             null, null);
ILGenerator gen = methBuilder.GetILGenerator();
gen.EmitWriteLine ("Hello world");
gen.Emit (OpCodes.Ret);
```

下面创建这个类型完成其定义：

```
Type t = tb.CreateType();
```

一旦创建这个类型后，可以使用初始的反射检查和执行动态绑定：

```
object o = Activator.CreateInstance (t);
t.GetMethod ("SayHello").Invoke (o, null);    // Hello world
```

19.6.1 保存已发出的程序集

Save方法可以将以动态方式生成的程序集写入指定的文件中。但是需要做到以下两点：

- 在构造AssemblyBuilder时，指定Save或RunAndSave的AssemblyBuilderAccess。
- 在构造ModuleBuilder时指定文件名（除非创建多模块程序集，否则该文件名应与程序集的文件名相同）。

还可以任意设置AssemblyName对象的属性，如Version或KeyPair（为了进行签名）。

例如：

```
AppDomain domain = AppDomain.CurrentDomain;

AssemblyName aname = new AssemblyName ("MyEmissions");
aname.Version = new Version (2, 13, 0, 1);

AssemblyBuilder assemBuilder = domain.DefineDynamicAssembly (
  aname, AssemblyBuilderAccess.RunAndSave);

ModuleBuilder modBuilder = assemBuilder.DefineDynamicModule (
```

```
    "MainModule", "MyEmissions.dll");
  // 以我们前面使用的方式创建类型……
  // ...
  assemBuilder.Save ("MyEmissions.dll");
```

这段代码将程序集写入应用程序的基础目录。要将程序集保存到不同的位置，必须在构造AssemblyBuilder时提供可选的目录：

```
AssemblyBuilder assemBuilder = domain.DefineDynamicAssembly (
  aname, AssemblyBuilderAccess.RunAndSave, @"d:\assemblies" );
```

一旦被写入文件中，动态程序集就会与其他程序集一样变为初始程序集。程序能够以静态方式引用我们刚刚创建的程序集并执行下列操作：

```
Widget w = new Widget();
w.SayHello();
```

19.6.2 Reflection.Emit对象模型

图19-2显示了.Emit类型中的基础类型，这些类型描述了CLR构造，而且它们是以System.Reflection命名空间中的对应部分为基础的。所以我们可以在创建类型时在标准构造中使用发出的构造。例如，我们以前使用下列方式调用Console.WriteLine：

```
MethodInfo writeLine = typeof(Console).GetMethod ("WriteLine",
                                       new Type[] { typeof (string) });
gen.Emit (OpCodes.Call, writeLine);
```

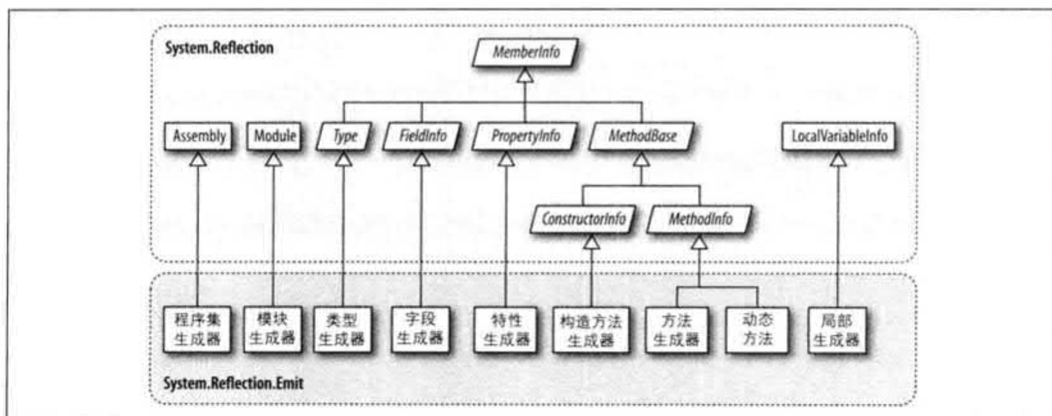


图19-2：System.Reflection.Emit

通过调用带有MethodBuilder（而非MethodInfo）的gen.Emit，我们就可以轻松地调用生成的方法。这样做非常重要，否则将无法编写在同一类型中调用另一个方法的动态方法。

注意，当构造TypeBuilder后，必须调用TypeBuilder上的CreateType。调用CreateType可以密封TypeBuilder及其所有成员，从而不能再对其成员进行添加或更改，以便获得可以实例化的Type。

在调用CreateType前，TypeBuilder及其成员处于“未创建”状态。未创建构造有许多限制。尤其

是无法调用所有返回MemberInfo对象的成员，如GetMembers、GetMethod或GetProperty，这些情况都会抛出异常。如果想要引用未创建类型的成员，必须使用原始方式：

```
TypeBuilder tb = ...

MethodBuilder method1 = tb.DefineMethod ("Method1", ...);
MethodBuilder method2 = tb.DefineMethod ("Method2", ...);

ILGenerator gen1 = method1.GetILGenerator();

// 假设用method，调用method2:

gen1.Emit (OpCodes.Call, method2);                    // 正确
gen1.Emit (OpCodes.Call, tb.GetMethod ("Method2"));   // 错误
```

调用了CreateType后，不仅可以反射和激活返回的Type，还可以反射和激活原始的TypeBuilder对象。实际上，TypeBuilder变成了Type的代理。本章稍后将介绍这个特点的重要性。

19.7 发出类型成员

本节的所有示例都假设TypeBuilder成员tb已经被实例化为以下代码：

```
AppDomain domain = AppDomain.CurrentDomain;
AssemblyName aname = new AssemblyName ("MyEmissions");

AssemblyBuilder assemBuilder = domain.DefineDynamicAssembly (
  aname, AssemblyBuilderAccess.RunAndSave);

ModuleBuilder modBuilder = assemBuilder.DefineDynamicModule (
  "MainModule", "MyEmissions.dll");

TypeBuilder tb = modBuilder.DefineType ("Widget", TypeAttributes.Public);
```

19.7.1 发出方法

在调用DefineMethod时，可以使用实例化DynamicMethod等方法时的方式指定返回类型和参数，例如以下方法：

```
public static double SquareRoot (double value)
{
  return Math.Sqrt (value);
}
```

可以这样生成：

```
MethodBuilder mb = tb.DefineMethod ("SquareRoot",
  MethodAttributes.Static | MethodAttributes.Public,
  CallingConventions.Standard,
  typeof (double),                     // 返回类型
  new[] { typeof (double) } );         // 参数类型

mb.DefineParameter (1, ParameterAttributes.None, "value");   // 分配名称

ILGenerator gen = mb.GetILGenerator();
gen.Emit (OpCodes.Ldarg_0);                                   // 加载第一个参数
gen.Emit (OpCodes.Call, typeof(Math).GetMethod ("Sqrt"));
gen.Emit (OpCodes.Ret);
```

```
Type realType = tb.CreateType();
double x = (double) tb.GetMethod ("SquareRoot").Invoke (null,new object[] { 10.0 });
Console.WriteLine (x);       // 3.16227766016838
```

调用DefineParameter是可选的，通常用于为参数分配名称。数字1引用第一个参数（0引用返回值）。如果调用DefineParameter，那么参数会被隐式地命名为__p1、__p2……以此类推。在将程序集写入磁盘时，分配名称才具有意义；这样做可以使方法拥有用户友好的性质。

提示：DefineParameter会返回ParameterBuilder对象，可以根据这个对象调用SetCustomAttribute，以便附加属性（请参阅本章稍后对这些内容的介绍）。

要发出按引用传递的参数，如下列C#方法中的参数：

```
public static void SquareRoot (ref double value)
{
  value = Math.Sqrt (value);
}
```

调用参数类型上的MakeByRefType：

```
MethodBuilder mb = tb.DefineMethod ("SquareRoot",
  MethodAttributes.Static | MethodAttributes.Public,
  CallingConventions.Standard,
  null,
  new Type[] { typeof (double).MakeByRefType() } );

  mb.DefineParameter (1, ParameterAttributes.None, "value");

  ILGenerator gen = mb.GetILGenerator();
  gen.Emit (OpCodes.Ldarg_0);
  gen.Emit (OpCodes.Ldarg_0);
  gen.Emit (OpCodes.Ldind_R8);
  gen.Emit (OpCodes.Call, typeof (Math).GetMethod ("Sqrt"));
  gen.Emit (OpCodes.Stind_R8);
  gen.Emit (OpCodes.Ret);

  Type realType = tb.CreateType();
  object[] args = { 10.0 };
  tb.GetMethod ("SquareRoot").Invoke (null, args);
  Console.WriteLine (args[0]);                          // 3.16227766016838
```

此处使用的操作码是从反编译的C#方法中复制的。注意，访问按引用传递的参数在语义上的差异：Ldind和Stind分别表示“间接加载”和“间接存储”。R8后缀表示8字节的浮点数字。

除了要使用下列代码调用DefineParameter外，发出out参数的过程是相同的：

```
mb.DefineParameter (1, ParameterAttributes.Out, "value");
```

1. 生成实例方法

要生成实例方法，应在调用DefineMethod时指定MethodAttributes.Instance：

```
MethodBuilder mb = tb.DefineMethod ("SquareRoot",
  MethodAttributes.Instance | MethodAttributes.Public
  ...
```

对于实例方法而言，参数0暗指this；余下的参数从1开始。因此，Ldarg_0会将this加载到评估堆栈中；Ldarg_1会加载第一个真正的方法参数。

2. HideBySig

如果要将一个类型子类化，需要在定义方法时指定MethodAttributes.HideBySig。HideBySig可以确保C#风格的方法隐藏的语义被应用，这种情况是指只有在子类型使用相同的签名定义方法时基础方法会被隐藏。在没有使用HideBySig的情况中，方法隐藏仅会考虑名称，因此子类型中的Foo(string)会隐藏基础类型中的Foo()，这通常是人们不希望得到的结果。

19.7.2 发出字段和属性

要创建字段，可调用TypeBuilder上的DefineField指定字段名、类型和可见性。下面这段代码创建了一个名为length的私有整型字段：

```
FieldBuilder field = tb.DefineField ("length", typeof (int),
                                     FieldAttributes.Private);
```

要创建特性或索引器需要几个额外步骤。首先，调用TypeBuilder上的DefineProperty指定特性的名称和类型：

```
PropertyBuilder prop = tb.DefineProperty (
                         "Text",                      // 特性的名称
                         PropertyAttributes.None,
                         typeof (string),             // 特定类型
                         new Type[0]                  // 索引器类型
                       );
```

（如果编写一个索引器，最后一个参数应为索引器类型的参数。）注意，我们没有指定这个特性的可见性：这个操作是单独在访问器方法上完成的。

下一步是编写get和set方法。为简单起见，方法的名称带有get_或set_前缀。这样就可以通过调用PropertyBuilder上的SetGetMethod和SetSetMethod将它们附加到特性上。

为了举出完整的示例，我们将使用下列字段和特性声明：

```
string _text;
public string Text
{
  get { return _text; }
  internal set { _text = value; }
}
```

然后以动态方式生成它：

```
FieldBuilder field = tb.DefineField ("_text", typeof (string),
                                      FieldAttributes.Private);
PropertyBuilder prop = tb.DefineProperty (
                         "Text",              // 特性的名称
                         PropertyAttributes.None,
                         typeof (string),     // 特性类型
                         new Type[0]);        // 索引器类型

MethodBuilder getter = tb.DefineMethod (
```

```
    "get_Text",                                    // 方法名称
    MethodAttributes.Public | MethodAttributes.SpecialName,
    typeof (string),                               // 返回类型
    new Type[0]);                                  // 参数类型

  ILGenerator getGen = getter.GetILGenerator();
  getGen.Emit (OpCodes.Ldarg_0);               // 将this加载到评估堆栈中
  getGen.Emit (OpCodes.Ldfld, field);          // 将字段值加载到评估堆栈中
  getGen.Emit (OpCodes.Ret);                   // 返回

  MethodBuilder setter = tb.DefineMethod (
    "set_Text",
    MethodAttributes.Assembly | MethodAttributes.SpecialName,
    null,                                      // 返回类型
    new Type[] { typeof (string) } );          // 参数类型

    ILGenerator setGen = setter.GetILGenerator();
    setGen.Emit (OpCodes.Ldarg_0);             // 将this加载到评估堆栈中
    setGen.Emit (OpCodes.Ldarg_1);             // 加载第二个参数，即value
    setGen.Emit (OpCodes.Stfld, field);        // 将value存储到字段中
    setGen.Emit (OpCodes.Ret);                 // 返回

    prop.SetGetMethod (getter);                // 将获取方法与特性链接起来
    prop.SetSetMethod (setter);                // 将设置方法与特性链接起来
```

可以使用下面的代码测试该属性：

```
  Type t = tb.CreateType();
  object o = Activator.CreateInstance (t);
  t.GetProperty ("Text").SetValue (o, "Good emissions!", new object[0]);
  string text = (string) t.GetProperty ("Text").GetValue (o, null);

  Console.WriteLine (text);             // 发出情况良好!
```

注意，在定义访问器MethodAttributes时包括了SpecialName。这会命令编译器不允许在静态引用程序集时对这些方法进行直接绑定，并确保访问器由反射工具和Visual Studio的IntelliSense进行适当的处理。

提示： 可以通过调用TypeBuilder上的DefineEvent，使用类似的方式发出事件。从而编写事件访问器方法，然后通过调用SetAddOnMethod和SetRemoveOnMethod将它们附加到EventBuilder上。

19.7.3 发出构造方法

通过调用类型生成器上的DefineConstructor，可以创建自定义的构造方法。创建自定义的构造方法不是强制的，如果没有创建自定义的构造方法就会自动获得默认的无参数构造方法。当出现子类化的情况时，默认的构造方法会调用基类构造方法，就像在C#中定义一个或多个构造方法替换这个默认的构造方法一样。

如果需要初始化字段，构造方法就是一个很好的场所。实际上，它是初始化字段的唯一场所：C#的字段初始化没有特殊的CLR支持，CLR支持仅仅是将值分配给构造方法中字段的语法捷径。因此，要进行复制可使用下面这段代码：

```
  class Widget
  {
```

```
    int _capacity = 4000;
}
```

可以使用下列代码定义构造方法：

```
FieldBuilder field = tb.DefineField ("_capacity", typeof (int),FieldAttributes.Private);
ConstructorBuilder c = tb.DefineConstructor (
  MethodAttributes.Public,
  CallingConventions.Standard,
  new Type[0]);                    // 构造方法参数

ILGenerator gen = c.GetILGenerator();

gen.Emit (OpCodes.Ldarg_0);            // 将this加载到评估堆栈中
gen.Emit (OpCodes.Ldc_I4, 4000);      // 将4000加载到评估堆栈中
gen.Emit (OpCodes.Stfld, field);      // 将其存储到字段中
gen.Emit (OpCodes.Ret);
```

调用基类构造方法

如果对另一个类型应用子类化，我们刚刚编写的构造方法会包含基类构造方法。这与C#不同，在C#中不论是使用直接方式还是使用间接方式，基类构造方法永远会被调用。例如：

```
class A { public A() { Console.Write ("A"); } }
class B : A { public B() {} }
```

实际上，编译器将第二行代码转换为：

```
class B : A { public B() : base() {} }
```

这不是生成IL的情况：如果想要执行基类构造方法（总是会出现这样的情况），就必须准确地调用基类构造方法。假设这个基类为B，下面这段代码可以实现上述操作：

```
gen.Emit (OpCodes.Ldarg_0);
ConstructorInfo baseConstr = typeof (B).GetConstructor (new Type[0]);
gen.Emit (OpCodes.Call, baseConstr);
```

调用带参数的构造方法的方式与调用方法的方式相同。

19.7.4 附加属性

可以通过调用CustomAttributeBuilder的SetCustomAttribute，将自定义属性附加到动态构造上。例如，假设需要将下列属性声明附加到字段或特性上：

```
[XmlElement ("FirstName", Namespace="http://test/", Order=3)]
```

要实现这一操作需要使用接收单个字符串的XmlElementAttribute构造方法。要使用CustomAttributeBuilder，必须检索这个构造方法以及我们想要设置的两个额外特性（Namespace和Order）：

```
Type attType = typeof (XmlElementAttribute);

ConstructorInfo attConstructor = attType.GetConstructor (
  new Type[] { typeof (string) } );
```

```
var att = new CustomAttributeBuilder (
  attConstructor,                                    // 构造方法
  new object[] { "FirstName" },                      // 构造方法参数
  new PropertyInfo[]
  {
    attType.GetProperty ("Namespace"),               // 特性
    attType.GetProperty ("Order")
  },
  new object[] { "http://test/", 3 }                 // 特性值
);

myFieldBuilder.SetCustomAttribute (att);
// 或者propBuilder.SetCustomAttribute (att);
// 或者typeBuilder.SetCustomAttribute (att); 等
```

19.8 发出泛型方法和类型

本节的所有示例都假设已经使用下列代码实例化modBuilder：

```
AppDomain domain = AppDomain.CurrentDomain;
AssemblyName aname = new AssemblyName ("MyEmissions");

AssemblyBuilder assemBuilder = domain.DefineDynamicAssembly (
  aname, AssemblyBuilderAccess.RunAndSave);

ModuleBuilder modBuilder = assemBuilder.DefineDynamicModule (
  "MainModule", "MyEmissions.dll");
```

19.8.1 定义泛型方法

要发出泛型方法：

1. 调用MethodBuilder上的DefineGenericParameters，以便获取一个GenericTypeParameterBuilder对象的数组。
2. 使用这些泛型参数调用MethodBuilder上的SetSignature。
3. 还可以使用不同的方式命名这些参数。

例如，请看下列泛型方法：

```
public static T Echo<T> (T value)
{
  return value;
}
```

可以使用以下方式发出这个泛型方法：

```
TypeBuilder tb = modBuilder.DefineType ("Widget", TypeAttributes.Public);

MethodBuilder mb = tb.DefineMethod ("Echo", MethodAttributes.Public |
                                            MethodAttributes.Static);
GenericTypeParameterBuilder[] genericParams
  = mb.DefineGenericParameters ("T");

mb.SetSignature (genericParams[0],   // 返回类型
                 null, null,
```

```
                    genericParams,        // 参数类型
                    null, null);

mb.DefineParameter (1, ParameterAttributes.None, "value");    // 可选

ILGenerator gen = mb.GetILGenerator();
gen.Emit (OpCodes.Ldarg_0);
gen.Emit (OpCodes.Ret);
```

DefineGenericParameters方法可以接收任意数量的字符串参数，这些字符串参数可对应于想要使用的泛型类型名称。在下面的示例中，我们仅需要一个基于System.Type的泛型T. GenericTypeParameterBuilder，这样在发出操作码时就可以使用它代替TypeBuilder。

使用GenericTypeParameterBuilder还可以设置基础类型的约束：

```
genericParams[0].SetBaseTypeConstraint (typeof (Foo));
```

以及接口约束：

```
genericParams[0].SetInterfaceConstraints (typeof (IComparable));
```

要复制这段代码：

```
public static T Echo<T> (T value) where T : IComparable<T>
```

可以编写：

```
genericParams[0].SetInterfaceConstraints (
  typeof (IComparable<>).MakeGenericType (genericParams[0]) );
```

对于其他约束，可调用SetGenericParameterAttributes。它可以接受GenericParameterAttributes枚举的成员，包括下列各值：

```
DefaultConstructorConstraint
NotNullableValueTypeConstraint
ReferenceTypeConstraint
Covariant
Contravariant
```

使用最后两个值得到的结果与将out和in修饰符应用于类型参数获得的结果相同。

19.8.2 定义泛型类型

可以使用类似的方式定义泛型类型。两者的差异在于定义泛型类型时需调用TypeBuilder上的DefineGenericParameters，而不是MethodBuilder。因此，要复制下列代码：

```
public class Widget<T>
{
  public T Value;
}
```

应执行以下操作：

```
TypeBuilder tb = modBuilder.DefineType ("Widget", TypeAttributes.Public);

GenericTypeParameterBuilder[] genericParams
```

```
    = tb.DefineGenericParameters ("T");
tb.DefineField ("Value", genericParams[0], FieldAttributes.Public);
```

可以使用添加方法的方式添加泛型约束。

19.9 复杂的发出目标

本节的所有示例都假设使用与上一节相同的方式对modBuilder进行实例化。

19.9.1 未创建的封闭式泛型

如果需要发出使用封闭式泛型类型的方法：

```
public class Widget
{
  public static void Test() { var list = new List<int>(); }
}
```

这个过程非常简单：

```
TypeBuilder tb = modBuilder.DefineType ("Widget", TypeAttributes.Public);

MethodBuilder mb = tb.DefineMethod ("Test", MethodAttributes.Public |
                                            MethodAttributes.Static);
ILGenerator gen = mb.GetILGenerator();

Type variableType = typeof (List<int>);

ConstructorInfo ci = variableType.GetConstructor (new Type[0]);

LocalBuilder listVar = gen.DeclareLocal (variableType);
gen.Emit (OpCodes.Newobj, ci);
gen.Emit (OpCodes.Stloc, listVar);
gen.Emit (OpCodes.Ret);
```

现在假设需要发出使用整型列表的方法，我们需要一个微件列表：

```
public class Widget
{
  public static void Test() { var list = new List<Widget>(); }
}
```

从理论方面讲，这是一个简单的修改；我们需要做的是使用下面这行代码：

```
Type variableType = typeof (List<>).MakeGenericType (tb);
```

替换下面这行代码：

```
Type variableType = typeof (List<int>);
```

但是，这样做会导致当调用GetConstructor时抛出NotSupportedException。问题在于无法调用由未创建的类型生成器封闭的泛型类型上的GetConstructor。GetField和GetMethod也存在相同的问题。

解决方案并不直观。TypeBuilder提供了下列三个静态方法：

```
public static ConstructorInfo GetConstructor (Type, ConstructorInfo);
public static FieldInfo       GetField       (Type, FieldInfo);
public static MethodInfo      GetMethod      (Type, MethodInfo);
```

尽管并不明显，但是这些方法专门用于获取由未创建的类型生成器封闭的泛型类型的成员！第一个参数是封闭式泛型类型；第二个参数是需要从未绑定的泛型类型上获取的成员。下面是修改后的示例：

```
MethodBuilder mb = tb.DefineMethod ("Test", MethodAttributes.Public |
                                            MethodAttributes.Static);
ILGenerator gen = mb.GetILGenerator();

Type variableType = typeof (List<>).MakeGenericType (tb);

ConstructorInfo unbound = typeof (List<>).GetConstructor (new Type[0]);
ConstructorInfo ci = TypeBuilder.GetConstructor (variableType, unbound);

LocalBuilder listVar = gen.DeclareLocal (variableType);
gen.Emit (OpCodes.Newobj, ci);
gen.Emit (OpCodes.Stloc, listVar);
gen.Emit (OpCodes.Ret);
```

19.9.2 循环依赖

假设需要创建互相引用的两个类型。例如：

```
class A { public B Bee; }
class B { public A Aye; }
```

可以使用下列代码动态生成这两个类型：

```
var publicAtt = FieldAttributes.Public;

TypeBuilder aBuilder = modBuilder.DefineType ("A");
TypeBuilder bBuilder = modBuilder.DefineType ("B");

FieldBuilder bee = aBuilder.DefineField ("Bee", bBuilder, publicAtt);
FieldBuilder aye = bBuilder.DefineField ("Aye", aBuilder, publicAtt);

Type realA = aBuilder.CreateType();
Type realB = bBuilder.CreateType();
```

注意，在引入这两个对象前，我们没有调用aBuilder和bBuilder上的CreateType。这个原则是：首先定义每个类型，然后调用每个类型生成器上的CreateType。

有趣的是，realA类型是合法的，但在调用bBuilder上的CreateType前它会功能失常（如果在此之前先使用aBuilder，当尝试访问Bee字段时就会出现异常）。

在创建了realB后bBuilder是如何知道去“修复”realA的呢？答案是它没有修复realA；realA可以在下次被使用时自行修复。这是可能出现的情况，因为调用了CreateType后，TypeBuilder会为真正的运行时类型变为代理。因此，（在引用了bBuilder的情况下）realA可以容易地获得升级所需的元数据。

当类型生成器需要非结构化类型的简单信息（可以预定的信息，如类型、成员和对象引用）时，这个系统就会生效。例如，在创建realA时，类型构造方法无需知道realB最终会占用多少字节的内存。这无关紧要，因为realB还没有创建！但是现在可以假设realB是一种结构，realB的最终大小是创建realA的关键信息。

如果两个结构之间的关系是非循环的，例如：

```
struct A { public B Bee; }
struct B {               }
```

可以通过先创建结构B后创建结构A，解决这个问题。请看下面的代码：

```
struct A { public B Bee; }
struct B { public A Aye; }
```

我们不会发出这段代码，因为使两种结构彼此包含对方是没有意义的（如果尝试这样做C#程序会在编译时出错）。但是下列变化都是合法和有用的：

```
public struct S<T> { ... }   // S可以为空而且这段演示代码也会运行。

class A { S<B> Bee; }
class B { S<A> Aye; }
```

在创建A时，TypeBuilder需要知道B的内存覆盖区，同理在创建B时，TypeBuilder也需要知道A的内存覆盖区。为了说明这一点，我们假设结构S是以静态方式定义的。下面是发出A类和B类的代码：

```
var pub = FieldAttributes.Public;

TypeBuilder aBuilder = modBuilder.DefineType ("A");
TypeBuilder bBuilder = modBuilder.DefineType ("B");

aBuilder.DefineField ("Bee", typeof(S<>).MakeGenericType (bBuilder), pub);
bBuilder.DefineField ("Aye", typeof(S<>).MakeGenericType (aBuilder), pub);

Type realA = aBuilder.CreateType();  //错误：无法加载类型B
Type realB = bBuilder.CreateType();
```

不论以何种顺序进行下列操作，CreateType都会抛出TypeLoadException：

- 先调用aBuilder.CreateType，然后它会输出“无法加载类型B”。
- 先调用bBuilder.CreateType，然后它会输出“无法加载类型A”。

警告： 如果以动态方式将类型化的LINQ发出到SQL DataContext中，就会出现问题。在我们的示例中，泛型EntityRef是一种结构，它与S结构相同。当数据库中的两个表通过相对的父/子关系彼此链接时，就会出现循环引用的情况。

要解决这个问题，必须允许类型生成器通过创建realA来创建realB的一部分。通过在调用CreateType前，在当前应用程序域上处理TypeResolve事件可以实现操作。因此，我们的示例将最后两行代码替换为以下代码：

```
TypeBuilder[] uncreatedTypes = { aBuilder, bBuilder };

ResolveEventHandler handler = delegate (object o, ResolveEventArgs args)
{
  var type = uncreatedTypes.FirstOrDefault (t => t.FullName == args.Name);
  return type == null ? null : type.CreateType().Assembly;
};

AppDomain.CurrentDomain.TypeResolve += handler;
```

```
Type realA = aBuilder.CreateType();
Type realB = bBuilder.CreateType();

AppDomain.CurrentDomain.TypeResolve -= handler;
```

在调用aBuilder.CreateType期间，当TypeResolve事件需要调用bBuilder上的CreateType时，它会被激发。

提示： 当定义嵌套类型而且嵌套类型与父类型互相引用时，以上述方法处理TypeResolve事件也是必要的。

19.10 解析IL

通过调用MethodBase对象上的GetMethodBody，可以获得已有方法的内容信息。这样做可以返回MethodBody对象，该对象拥有检查方法的局部变量、异常处理子句、堆栈尺寸以及原始IL的特性。与反向执行Reflection.Emit的结果不同！

在解析代码时检查方法的原始IL非常有用。简单用法是当更新程序集时，指定程序集中的哪些方法已经改变。

为了说明IL解析，我们以*ildasm*格式编写了一个反编译IL的应用程序。这段代码可以用作代码分析工具或者高级语言反编译器的开头。

提示： 在反射API时，所有C#的函数结构不是由MethodBase子类型表示，就是（在特性、事件和索引器中）将MethodBase对象附加到它们上。

19.11 编写反编译器

提示： 可以从*http://www.albahari.com/nutshell/*下载这段源代码。

下面列出了反编译器的输出样本：

```
IL_00EB: ldfld       Disassembler._pos
IL_00F0: ldloc.2
IL_00F1: add
IL_00F2: ldelema     System.Byte
IL_00F7: ldstr       "Hello world"
IL_00FC: call        System.Byte.ToString
IL_0101: ldstr       " "
IL_0106: call        System.String.Concat
```

要获得这个输出，我们必须解析组成IL的二进制令牌。第一步是调用MethodBody上的GetILAsByteArray方法，以获得作为字节数组的IL。为了使余下的操作更加容易执行，可将下列代码编写到一个类中：

```
public class Disassembler
{
  public static string Disassemble (MethodBase method)
  {
```

```
    return new Disassembler (method).Dis();
  }

  StringBuilder _output;       //我们将不断追加的结果
  Module _module;              //稍后这个结果会唾手可得
  byte[] _il;                  //原始的字节代码
  int _pos;                    //在字节代码中我们可以处理的位置

  Disassembler (MethodBase method)
  {
    _module = method.DeclaringType.Module;
    _il = method.GetMethodBody().GetILAsByteArray();
  }

  string Dis()
  {
    _output = new StringBuilder();
    while (_pos < _il.Length) DisassembleNextInstruction();
    return _output.ToString();
  }
}
```

静态的Disassemble方法是这个类的唯一全局成员。它的所有其他成员都是反编译过程的私有成员。Dis方法包含主循环，我们在这个循环中处理每条指令。

设置好这个框架后，剩下的工作是编写DisassembleNextInstruction。但是在此之前，将所有运算符加载到静态字典中可以获得一些帮助，从而可以通过它们的8或16位值进行访问。实现这项操作的最简单方法是使用反射检索所有OpCodes类中类型为OpCode的静态字段：

```
static Dictionary<short,OpCode> _opcodes = new Dictionary<short,OpCode>();

static Disassembler()
{
  Dictionary<short, OpCode> opcodes = new Dictionary<short, OpCode>();
    foreach (FieldInfo fi in typeof (OpCodes).GetFields
                            (BindingFlags.Public | BindingFlags.Static))
      if (typeof (OpCode).IsAssignableFrom (fi.FieldType))
      {
        OpCode code = (OpCode) fi.GetValue (null);    // 获取字段值
        if (code.OpCodeType != OpCodeType.Nternal)
          _opcodes.Add (code.Value, code);
      }
}
```

我们已经在静态构造方法中编写过它，因此只是将它再执行一次。

下面就可以编写DisassembleNextInstruction了。每条IL指令都由带有0、1、2、3、4或8字节的操作数的1或2字节的操作码组成（一种例外情况是内嵌转换操作码，其操作数的数量可变）。因此，应先读取操作码，然后读取操作数，最后写出结果：

```
void DisassembleNextInstruction()
{
  int opStart = _pos;

  OpCode code = ReadOpCode();
  string operand = ReadOperand (code);

  _output.AppendFormat ("IL_{0:X4}: {1,-12} {2}",
```

```
                    opStart, code.Name, operand);
  _output.AppendLine();
}
```

要读取操作码，应先读取一个字节并检测这条指令是否合法。如果这条指令不合法，可读取另一个字节并搜索2字节的指令：

```
OpCode ReadOpCode()
{
  byte byteCode = _il [_pos++];
  if (_opcodes.ContainsKey (byteCode)) return _opcodes [byteCode];

  if (_pos == _il.Length) throw new Exception ("Unexpected end of IL");

  short shortCode = (short) (byteCode * 256 + _il [_pos++]);

  if (!_opcodes.ContainsKey (shortCode))
    throw new Exception ("Cannot find opcode " + shortCode);

  return _opcodes [shortCode];
}
```

要读取操作数，必须先创建其长度，可以根据操作数的类型来实现。因为大多数操作数是4字节的，所以使用条件子句可以非常简单地筛选出例外情况。

下一步是调用FormatOperand格式化操作数：

```
string ReadOperand (OpCode c)
{
  int operandLength =
    c.OperandType == OperandType.InlineNone
      ? 0 :
    c.OperandType == OperandType.ShortInlineBrTarget ||
    c.OperandType == OperandType.ShortInlineI ||
    c.OperandType == OperandType.ShortInlineVar
      ? 1 :
    c.OperandType == OperandType.InlineVar
      ? 2 :
    c.OperandType == OperandType.InlineI8 ||
    c.OperandType == OperandType.InlineR
      ? 8 :
    c.OperandType == OperandType.InlineSwitch
      ? 4 * (BitConverter.ToInt32 (_il, _pos) + 1) :
      4;  // 其他操作数都是4字节的

  if (_pos + operandLength > _il.Length)
    throw new Exception ("Unexpected end of IL");

  string result = FormatOperand (c, operandLength);
  if (result == null)
  {                                  // 写出十六进制的操作数字节
    result = "";
    for (int i = 0; i < operandLength; i++)
      result += _il [_pos + i].ToString ("X2") + " ";
  }
  _pos += operandLength;
  return result;
}
```

如果调用FormatOperand的结果是null，意味着操作数无需进行特殊的格式化，因此仅需以十六进制的方式将其写出。通过编写永远返回null的FormatOperand方法，可以在此时测试反编译器。下面列出的是输出结果：

```
IL_00A8: ldfld       98 00 00 04
IL_00AD: ldloc.2
IL_00AE: add
IL_00AF: ldelema     64 00 00 01
IL_00B4: ldstr       26 04 00 70
IL_00B9: call        B6 00 00 0A
IL_00BE: ldstr       11 01 00 70
IL_00C3: call        91 00 00 0A
...
```

尽管操作码是正确的，但是没有使用许多操作数。要代替十六进制数，需要使用成员名称和字符串。一旦编写完FormatOperand就可以识别从这些格式获益的特殊情况。下面列出了大多数由4个字节组成的操作数和较短的分支指令：

```
string FormatOperand (OpCode c, int operandLength)
{
  if (operandLength == 0) return "";

  if (operandLength == 4)
    return Get4ByteOperand (c);
  else if (c.OperandType == OperandType.ShortInlineBrTarget)
    return GetShortRelativeTarget();
  else if (c.OperandType == OperandType.InlineSwitch)
    return GetSwitchTarget (operandLength);
  else
    return null;
}
```

需要特殊处理的4字节操作数有三种。第一种是对成员或类型的引用，对于这些操作数我们可以通过调用定义模块的ResolveMember方法，获取成员或类型名称。第二种是字符串，这些操作数存储在程序集模块的元数据中，而且通过调用ResolveString可以对它们进行检索。最后一种是分支目标，这些操作数引用了IL中的字节偏移量。通过计算当前指令（+4字节）后面的绝对地址，可以格式化这些操作：

```
string Get4ByteOperand (OpCode c)
{
  int intOp = BitConverter.ToInt32 (_il, _pos);

  switch (c.OperandType)
  {
    case OperandType.InlineTok:
    case OperandType.InlineMethod:
    case OperandType.InlineField:
    case OperandType.InlineType:
      MemberInfo mi;
      try { mi = _module.ResolveMember (intOp); }
      catch { return null; }
      if (mi == null) return null;

      if (mi.ReflectedType != null)
        return mi.ReflectedType.FullName + "." + mi.Name;
```

```
        else if (mi is Type)
          return ((Type)mi).FullName;
        else
          return mi.Name;

      case OperandType.InlineString:
        string s = _module.ResolveString (intOp);
        if (s != null) s = "'" + s + "'";
        return s;

      case OperandType.InlineBrTarget:
        return "IL_" + (_pos + intOp + 4).ToString ("X4");

      default:
        return null;
  }
```

提示：调用ResolveMember的位置是代码分析工具对方法依赖进行报告的良好入口。

对于任何4字节的操作码，上面的代码都会返回null，这会导致ReadOperand将操作数格式化为十六进制数。

应特别注意最后两种操作数是短分支目标和内嵌转换。短分支目标将目的地偏移量描述为位于当前指令尾部（即+1字节）的单个签名的字节。转换目标带有可变数量的4字节分支目的址：

```
string GetShortRelativeTarget()
{
  int absoluteTarget = _pos + (sbyte) _il [_pos] + 1;
  return "IL_" + absoluteTarget.ToString ("X4");
}

string GetSwitchTarget (int operandLength)
{
  int targetCount = BitConverter.ToInt32 (_il, _pos);
  string [] targets = new string [targetCount];
  for (int i = 0; i < targetCount; i++)
  {
    int ilTarget = BitConverter.ToInt32 (_il, _pos + (i + 1) * 4);
    targets [i] = "IL_" + (_pos + ilTarget + operandLength).ToString ("X4");
  }
  return "(" + string.Join (", ", targets) + ")";
}
```

这段代码编写了这个反编译器。通过反编译本身的方法可以对其进行测试：

```
MethodInfo mi = typeof (Disassembler).GetMethod (
  "ReadOperand", BindingFlags.Instance | BindingFlags.NonPublic);

Console.WriteLine (Disassembler.Disassemble (mi));
```

第20章
动态编程

第4章介绍了C#语言中动态绑定的实现方式。本章先简要介绍DLR，然后介绍下列动态编程方式：

- 数字类型统一
- 动态成员重载解决方案
- 实现动态对象
- 使用动态语言进行交互操作

提示： 第25章将介绍以动态方式提高COM交互操作性的方法。

本章使用的类型位于`System.Dynamic`命名空间中，但`CallSite<>`除外，它位于`System.Runtime.CompilerServices`中。

20.1 动态语言运行时

C#依靠动态语言运行时（DLR）执行动态绑定。

与其名称的意义相反，DLR并非CLR的动态版本。更确切地说，它是位于CLR之上的一个库，就像*System.Xml.dll*等其他库一样。它的主要功能是为进行统一的动态编程提供运行时服务，它既能以静态类型化语言提供服务，也能以动态类型化语言提供服务。这意味着C#、VB、IronPython和IronRuby等语言都可以使用通用协议以动态方式调用函数，并允许这些函数共享库和调用其他语言编写的代码。

DLR还使得在.NET中编写动态语言变得更加容易。并非必须发出IL，动态语言的工作重点在于表达式（第8章介绍了`System.Linq.Expressions`中相同的表达式树）。

DLR进一步确保了所有用户都可以从调用点缓存获益，调用点缓存是一种最优化方式，在这种方式中DLR可以在动态绑定过程中避免不必要地重复进行成员解决方案决定。

提示： Framework 4.0是第一个带有DLR的Framework版本。在此之前的版本，可以从CodePlex单独下载DLR。这个网站还含有其他对开发者有用的资源。

什么是调用点

当编译器遇到一个动态表达式时，它不知道谁会在运行时计算该表达式的值。例如，下面的方法：

```
public dynamic Foo (dynamic x, dynamic y)
{
  return x / y; // Dynamic expression
}
```

变量x和y可以是任意的CLR对象、COM对象或由动态语言托管的对象。因此，编译器无法使用其常用的静态发出方法调用已知类型的已知方法。但是，编译器会发出最终生成描述操作的表达式树的代码，这些代码由DLR将在运行时绑定的调用点管理。调用点基本上在调用者和被调用者之间起媒介的作用。

调用点由*System.Core.dll*中的`CallSite<>`类代表。我们可以通过反编译上一个方法观察这个情况，其结果如下所示：

```
static CallSite<Func<CallSite,object,object,object>> divideSite;

[return: Dynamic]
public object Foo ([Dynamic] object x, [Dynamic] object y)
{
  if (divideSite == null)
    divideSite =
      CallSite<Func<CallSite,object,object,object>>.Create (
        Microsoft.CSharp.RuntimeBinder.Binder.BinaryOperation (
          CSharpBinderFlags.None,
          ExpressionType.Divide,
          /* 为使代码简洁省略参数*/ ));

  return divideSite.Target (divideSite, x, y);
}
```

如前所示，调用点缓存在静态字段中，以避免每次调用时重新创建。DLR进一步缓存了绑定阶段和实际方法目标的结果（根据x和y的类型，可能有多个目标）。

这样通过调用点的`Target`（一个委托），在x和y操作数中进行传递，真正的动态调用就会实现。

注意，`Binder`类专门用于C#。每种对动态绑定提供支持的语言都会提供专门的绑定器，以帮助DLR以专门方式为该语言解释表达式。例如，如果使用整型值5和2调用`Foo`，C#绑定器会确保获得2的结果。相反，VB.NET绑定器会返回2.5的结果。

20.2 数字类型统一

第4章介绍了如何使用关键字dynamic编写跨多种数字类型的单一方法：

```
static dynamic Mean (dynamic x, dynamic y)
{
  return (x + y) / 2;
}
```

```
static void Main()
{
  int x = 3, y = 5;
  Console.WriteLine (Mean (x, y));
}
```

提示： 关键字static和dynamic可以相邻显示是C#中的一种特殊的反射！关键字internal和extern也可以应用相同的反射。

然而，这样做（并非必要）会影响静态类型的安全性。下列代码的编译过程没有出错，但是会在运行时出错：

```
string s = Mean (3, 5);      // 运行时出错！
```

通过引入泛型类型参数，然后将其强制转换为计算本身的dynamic可以解决该问题：

```
static T Mean<T> (T x, T y)
{
  dynamic result = ((dynamic) x + y) / 2;
  return (T) result;
}
```

注意，我们显式地将结果转换回了T。如果省略了这个转换，就需要依靠隐式转换，这种转换在最初可以正确运行。但是，隐式转换会在运行时调用带有8或16字节整型类型的情况中出错。为了了解出错原因，当将两个8位数字加在一起时，请考虑初始静态类型化出现的情况：

```
byte b = 3;
Console.WriteLine ((b + b).GetType().Name);  // Int32
```

我们获得了一个Int32，因为在执行算术操作前编译器将8或16位数字"提升"为了Int32。为了保持一致性，C#的绑定器命令DLR执行相同的操作，然后我们就得到了需要直接转换为到较小数字类型的Int32。当然，如果进行求和而不是求平均值，这样做会有溢出的可能。

即使应用了调用点缓存，但动态绑定仍旧会稍微降低一点性能。通过以静态方式添加覆盖几乎所有常用类型的类型化重载，可以缓解这个问题。例如，如果性能剖析显示使用doubles调用Mean使性能降低时，可以在以后的重载中添加下列代码：

```
static double Mean (double x, double y)
{
  return (x + y) / 2;
}
```

当使用在编译时已知为double类型的参数调用Mean时，编译器会支持这个重载。

20.3 动态成员重载解决方案

以静态方式使用动态的类型化参数调用方法，会使其成员的编译时的重载解决方案与运行时的重载解决方案不同。在简化某些编程任务时这样做有好处，如简化访问者的设计模式。在处理由C#静态类型化带来的限制条件时，这样做也有用。

20.3.1 简化访问者模式

实际上，使用访问者模式可以向类中添加方法，而无需更改已经存在的类。尽管访问者模式很有用，但是这种模式的静态方式比较抽象，而且不如其他大多数设计模式直观。该模式还要求被访问的类通过提供Accept方法获得友好的访问性，如果无法控制这些类，那么将无法做到这一点。

在动态绑定中，可以更容易地做到这一点，而且无需更改已经存在的类。要详细了解该情况，请看下面的类结构：

```
class Person
{
  public string FirstName { get; set; }
  public string LastName { get; set; }

  // Friends集合中可能包含Customers和Employees:
  public readonly IList<Person> Friends = new Collection<Person> ();
}

class Customer : Person { public decimal CreditLimit { get; set; } }
class Employee : Person { public decimal Salary { get; set; } }
```

假设需要编写一个方法，通过程序方式将Person的详细信息导出到XML XElement中。最明显的解决方案是在Person类中编写一个名为ToXElement()的虚方法，该方法会返回带有Person特性的XElement。然后在Customer和Employee类中重写这个方法，这样XElement也会填入CreditLimit和Salary。然而这种模式会出现问题，具体包括以下两种原因：

- 可能没有Person、Customer和Employee类，因此无法向它们添加方法（而且扩展方法无法提供各种行为）。
- Person、Customer和Employee类可能已经非常大。经常出现的反面情况是“万能对象”，在这种情况中Person类包含了海量的功能，这使得无法进行维护。避免向Person添加无需访问Person的私有成员的函数是一种解决方法。ToXElement方法也是一个很好的候选方案。

通过动态成员重载解决方案，可以在单独的类中编写ToXElement功能，而无需根据类进行复杂的转换：

```
class ToXElementPersonVisitor
{
  public XElement DynamicVisit (Person p)
  {
    return Visit ((dynamic)p);
  }

  XElement Visit (Person p)
  {
    return new XElement ("Person",
      new XAttribute ("Type", p.GetType().Name),
      new XElement ("FirstName", p.FirstName),
      new XElement ("LastName", p.LastName),
      p.Friends.Select (f => DynamicVisit (f))
    );
  }

  XElement Visit (Customer c)   // 用户的专属逻辑
  {
```

```
    XElement xe = Visit ((Person)c);         // 调用"基础"方法
    xe.Add (new XElement ("CreditLimit", c.CreditLimit));
    return xe;
  }
  XElement Visit (Employee e)                // 雇员的专属逻辑
  {
    XElement xe = Visit ((Person)e);         // 调用"基础"方法
    xe.Add (new XElement ("Salary", e.Salary));
    return xe;
  }
}
```

DynamicVisit方法执行了一次动态分派，按照运行时作出的决定调用Visit最特殊的版本。注意使用粗体显示的代码行，这行代码根据Friends集合中的每个元素调用DynamicVisit。这样做可以确保如果是用户或雇员，正确的重载方法会被调用。

下列代码演示了这个类：

```
var cust = new Customer
{
  FirstName = "Joe", LastName = "Bloggs", CreditLimit = 123
};
cust.Friends.Add (
  new Employee { FirstName = "Sue", LastName = "Brown", Salary = 50000 }
);
Console.WriteLine (new ToXElementPersonVisitor().DynamicVisit (cust));
```

下面列出的是结果：

```
<Person Type="Customer">
  <FirstName>Joe</FirstName>
  <LastName>Bloggs</LastName>
  <Person Type="Employee">
    <FirstName>Sue</FirstName>
    <LastName>Brown</LastName>
    <Salary>50000</Salary>
  </Person>
  <CreditLimit>123</CreditLimit>
</Person>
```

变化

如果要使用一个以上的访问者类，为访问者定义抽象基类是一种有用的变化：

```
abstract class PersonVisitor<T>
{
  public T DynamicVisit (Person p) { return Visit ((dynamic)p); }
  protected abstract T Visit (Person p);
  protected virtual T Visit (Customer c) { return Visit ((Person) c); }
  protected virtual T Visit (Employee e) { return Visit ((Person) e); }
}
```

这样子类无需定义它们本身的DynamicVisit方法，而是重写它们想要专门研究其行为的Visit版本。这样做还可以集中包含Person结构的方法的优点，更自然地调用基础方法：

```
class ToXElementPersonVisitor : PersonVisitor<XElement>
{
  protected override XElement Visit (Person p)
  {
    return new XElement ("Person",
     new XAttribute ("Type", p.GetType().Name),
     new XElement ("FirstName", p.FirstName),
     new XElement ("LastName", p.LastName),
     p.Friends.Select (f => DynamicVisit (f))
   );
  }

  protected override XElement Visit (Customer c)
  {
    XElement xe = base.Visit (c);
    xe.Add (new XElement ("CreditLimit", c.CreditLimit));
    return xe;
  }

  protected override XElement Visit (Employee e)
  {
    XElement xe = base.Visit (e);
    xe.Add (new XElement ("Salary", e.Salary));
    return xe;
  }
}
```

这样甚至可以对ToXElementPersonVisitor本身应用子类化。

多分派

C#和CLR一直支持虚方法调用形式的有限动态形式。这与用于虚方法调用的真正的动态编程不同，编译器在编译时必须根据基类中成员的名称和签名提供特殊的虚拟成员。这意味着：

- 编译器必须完全理解调用表达式（例如，它必须在编译时确定目标成员是字段还是特性）。
- 重载解决方案必须完全由编译器根据编译时的参数类型完成。

最后一点的结果是执行虚拟法调用的能力称为单分派。要了解其原因，请看下列方法调用（其中Walk是一个虚方法）：

```
animal.Walk (owner);
```

运行时仅根据接收器animal的类型，决定是调用狗的Walk还是调用猫的Walk方法（因此这是单分派。如果Walk的许多重载接受不同类型的拥有者，那么无需关心实际运行时owner对象的类型，重载会在编译时被选中。换句话说，只有运行时的接收器类型可以改变调用的方法。

相反，在运行前动态调用与重载解决方案不同：

```
animal.Walk ((dynamic) owner);
```

最终选择调用哪个Walk方法，由animal和owner的类型同时决定，这称为多分派，因为除了接收器之外，运行时的参数类型也用于确定调用哪个Walk方法。

20.3.2 匿名调用泛型类型的成员

C#的静态类型化严格说是一把双刃剑。一方面，它在编译时保证程序的正确性。另一方面，它偶尔会导致编码困难或无法使用代码进行表达，在这种情况中必须使用反射，动态绑定比反射更清晰、更快速。

在T未知的情况下需要使用G<T>类型的对象就是一个这样的示例。我们通过定义下列类说明这个情况：

```
public class Foo<T> { public T Value; }
```

然后编写下列方法：

```
static void Write (object obj)
{
  if (obj is Foo<>)                              // 非法
  Console.WriteLine ((Foo<>) obj).Value);        // 非法
}
```

该方法无法编译：因为无法调用未绑定的泛型类型成员。

动态绑定提供了两种方法用于处理这种情况。第一种方法是以动态方式访问Value成员，如下所示：

```
static void Write (dynamic obj)
{
  try { Console.WriteLine (obj.Value); }
  catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException) {...}
}
```

这种方式的优势是可以使用定义了Value字段或特性的任何对象。然而，仍旧存在两个问题。首先，以这种方式捕捉异常有些复杂并且效率低（而且无法预先询问DLR这样做是否可以正常运行）；其次，如果Foo是一个接口（如IFoo<T>）并且下列两个条件都成立，这个方法无法生效：

- Value被显式地实现了。
- 实现IFoo<T>的类型无法访问（稍后详细介绍）。

一个更好的解决方案是编写帮助进行重载的方法GetFooValue，然后使用动态成员重载解决方案调用它：

```
static void Write (dynamic obj)
{
  object result = GetFooValue (obj);
  if (result != null) Console.WriteLine (result);
}

static T GetFooValue<T> (Foo<T> foo) { return foo.Value; }
static object GetFooValue (object foo) { return null; }
```

注意，我们重载GetFooValue以接收一个object参数，该参数对任何类型都起反馈作用。在运行时，当调用带动态参数的GetFooValue时，C#动态绑定器将选择重载。如果该对象不是基于Foo<T>的，那么C#动态绑定器将重载对象参数，而不是抛出一个异常。

提示：还可以仅编写第一个GetFooValue，然后捕捉RuntimeBinderException。这样做的优点在于可以识别foo.Value为null的情况；缺点是会导致抛出和捕捉异常的性能开销。

第19章介绍了使用反射解决相同问题的方法，这种方法更有效（请参阅第18章的相关内容）。我们使用的示例用于设计ToString()功能更加强大的版本，它可以接收例如IEnumerable和IGrouping<,>等对象。下面的示例通过动态绑定更轻松地解决了这个问题：

```
static string GetGroupKey<TKey,TElement> (IGrouping<TKey,TElement> group)
{
  return "Group with key=" + group.Key + ": ";
}

static string GetGroupKey (object source) { return null; }

public static string ToStringEx (object value)
{
  if (value == null) return "<null>";
  if (value is string) return (string) value;
  if (value.GetType().IsPrimitive) return value.ToString();

  StringBuilder sb = new StringBuilder();

  string groupKey = GetGroupKey ((dynamic)value); //动态分派
  if (groupKey != null) sb.Append (groupKey);

  if (value is IEnumerable)
    foreach (object element in ((IEnumerable)value))
      sb.Append (ToStringEx (element) + " ");

  if (sb.Length == 0) sb.Append (value.ToString());

  return "\r\n" + sb.ToString();
}
```

在运行时：

```
Console.WriteLine (ToStringEx ("xyyzzz".GroupBy (c => c) ));

Group with key=x: x
Group with key=y: y y
Group with key=z: z z z
```

注意，我们使用了动态成员重载解决方案解决这个问题。如果使用下列代码：

```
dynamic d = value;
try { groupKey = d.Value); }
catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException) {...}
```

示例就会运行失败，因为LINQ的GroupBy运算符会返回一种实现IGrouping<,>的类型，这种类型本身是内部的，因此无法访问它：

```
internal class Grouping : IGrouping<TKey,TElement>, ...
{
  public TKey Key;
  ...
}
```

即使Key特性声明为public，包含它的类将其标识为internal，这使其无法通过IGrouping<,>接口被访问。而且像第4章介绍的那样，在以动态方式调用Value成员时，无法指定DLR绑定该接口。

20.4 实现动态对象

对象可以通过实现IDynamicMetaObjectProvider提供其绑定语义（或者通过子类化DynamicObject更容易地提供其绑定语义，DynamicObject提供了对该接口的默认实现）。第4章使用下面的示例对此做了简单的介绍：

```
static void Main()
{
  dynamic d = new Duck();
  d.Quack();                  // Quack method was called
  d.Waddle();                 // Waddle method was called
}

public class Duck : DynamicObject
{
  public override bool TryInvokeMember (
    InvokeMemberBinder binder, object[] args, out object result)
  {
    Console.WriteLine (binder.Name + " method was called");
    result = null;
    return true;
  }
}
```

20.4.1 DynamicObject

在上一个示例中，我们重写了TryInvokeMember，允许用户调用动态对象上的方法，如Quack或Waddle。DynamicObject提供了其他虚方法，这些虚方法同样可以让用户使用其他编程结构。下面列出了C#中典型的相应结构：

方法	编程结构
TryInvokeMember	方法
TryGetMember、TrySetMember	特性或字段
TryGetIndex、TrySetIndex	索引器
TryUnaryOperation	一元运算符，如!
TryBinaryOperation	二元运算符，如==
TryConvert	转换（投射）到其他类型
TryInvoke	在对象本身上调用，例如，d("foo")

如果绑定成功，这些方法会返回true。如果这些方法返回false，那么DLR会退回语言绑定器，在DynamicObject（子类）上寻找匹配成员。如果没有找到匹配成员，那么抛出RuntimeBinderException。

我们使用一个类说明TryGetMember和TrySetMember，通过这个类能够以动态方式访问XElement(System.Xml.Linq)中的属性：

```
static class XExtensions
{
  public static dynamic DynamicAttributes (this XElement e)
  {
    return new XWrapper (e);
  }

  class XWrapper : DynamicObject
  {
    XElement _element;
    public XWrapper (XElement e) { _element = e; }

    public override bool TryGetMember (GetMemberBinder binder,out object result)
    {
      result = _element.Attribute (binder.Name).Value;
      return true;
    }

    public override bool TrySetMember (SetMemberBinder binder,object value)
    {
      _element.SetAttributeValue (binder.Name, value);
      return true;
    }
  }
}
```

下面列出了使用它的方法：

```
XElement x = XElement.Parse (@"<Label Text=""Hello"" Id=""5""/>");
dynamic da = x.DynamicAttributes();
Console.WriteLine (da.Id);            // 5
da.Text = "Foo";
Console.WriteLine (x.ToString());     // <Label Text="Foo" Id="5" />
```

下面这段代码执行的操作与System.Data.IDataRecord执行的操作类似，它使得数据阅读器变得更容易使用：

```
public class DynamicReader : DynamicObject
{
  readonly IDataRecord _dataRecord;
  public DynamicReader (IDataRecord dr) { _dataRecord = dr; }

  public override bool TryGetMember (GetMemberBinder binder,out object result)
  {
    result = _dataRecord [binder.Name];
    return true;
  }
}
...
using (IDataReader reader = someDbCommand.ExecuteReader())
{
  dynamic dr = new DynamicReader (reader);
  while (reader.Read())
  {
    int id = dr.ID;
    string firstName = dr.FirstName;
    DateTime dob = dr.DateOfBirth;
    ...
```

```
    }
  }
```

下面这段代码演示了TryBinaryOperation和TryInvoke：

```
static void Main()
{
  dynamic d = new Duck();
  Console.WriteLine (d + d);                  // foo
  Console.WriteLine (d (78, 'x'));            // 123
}

public class Duck : DynamicObject
{
  public override bool TryBinaryOperation (BinaryOperationBinder binder,
                                           object arg, out object result)
  {
    Console.WriteLine (binder.Operation);     //增加
    result = "foo";
    return true;
  }

  public override bool TryInvoke (InvokeBinder binder,
                                  object[] args, out object result)
  {
    Console.WriteLine (args[0]);              // 78
    result = 123;
    return true;
  }
}
```

为了获取动态语言的优势，DynamicObject还提供了一些虚方法。尤其是通过重写GetDynamicMemberNames，可以返回动态对象提供的所有成员名称的列表。

提示： 另一个实现GetDynamicMemberNames的原因是Visual Studio的调试器使用这个方法显示动态对象的视图。

20.4.2 ExpandoObject

DynamicObject的另一个简单应用是编写动态类，使用这种动态类可以通过字符串关键字将对象存储在字典中，还可以在字典中检索它们。然而，ExpandoObject对象已经提供了这个功能：

```
dynamic x = new ExpandoObject();
x.FavoriteColor = ConsoleColor.Green;
x.FavoriteNumber = 7;
Console.WriteLine (x.FavoriteColor);              // 绿色
Console.WriteLine (x.FavoriteNumber);             // 7
```

ExpandoObject实现了IDictionary<string,object>，因此我们可以继续编写示例并执行以下操作：

```
var dict = (IDictionary<string,object>) x;
Console.WriteLine (dict ["FavoriteColor"]);       // 绿色
Console.WriteLine (dict ["FavoriteNumber"]);      // 7
Console.WriteLine (dict.Count);                   // 2
```

20.5 通过动态语言交互操作

尽管C# 5.0支持通过dynamic关键字进行动态绑定，但是它无法在运行时执行字符串中描述的表达式：

```
string expr = "2 * 3";
//无法执行expr
```

提示：这是因为这段代码要将字符串转换为表达式树，需要词汇和语义分析程序。C#编译器内置了这些功能，但在运行时无法获得这些服务。在运行时，C#仅会提供绑定器，绑定器会告诉DLR对已创建的表达式树的编译方法。

真正的动态语言（如IronPython和IronRuby）确实允许执行随机字符串，而且该功能对一些任务（如编写脚本、动态配置和实现动态规则引擎）很有用。因此，尽管使用C#编写应用程序中的大部分代码，为了完成这类任务调用动态语言仍旧很有用。而且，当在.NET库中没有可用的同等功能时，可能需要增强使用动态语言编写的API的功能。

下面的示例使用了IronPython评估C#中在运行时创建的表达式。这段脚本可以用于编写计算器。

提示：要运行这段代码，应下载IronPython（可在Internet搜索*IronPython*），然后在C#应用程序中引用*IronPython*、*Microsoft.Scripting*和*Microsoft.Scripting.Core*程序集。

```
using System;
using IronPython.Hosting;
using Microsoft.Scripting;
using Microsoft.Scripting.Hosting;

class Calculator
{
  static void Main()
  {
    int result = (int) Calculate ("2 * 3");
    Console.WriteLine (result); // 6
  }

  static object Calculate (string expression)
  {
    ScriptEngine engine = Python.CreateEngine();
    return engine.Execute (expression);
  }
}
```

因为要将字符串传递到Python中，所以该表达式会根据Python的规则评估，而不是根据C#的规则评估。这也意味着可以使用Python的语言功能，如下列代码所示：

```
var list = (IEnumerable) Calculate ("[1, 2, 3] + [4, 5]");
foreach (int n in list) Console.Write (n); // 12345
```

在C#和脚本之间传递状态

要将变量从C#传递到Python，需要增加几个步骤。下面的示例说明了这些步骤，而且这段代码可以成为规则引擎的基础：

```
//下面的字符串可以来自文件或数据库:
string auditRule = "taxPaidLastYear / taxPaidThisYear > 2";

ScriptEngine engine = Python.CreateEngine ();

ScriptScope scope = engine.CreateScope ();
scope.SetVariable ("taxPaidLastYear", 20000m);
scope.SetVariable ("taxPaidThisYear", 8000m);

ScriptSource source = engine.CreateScriptSourceFromString (
                      auditRule, SourceCodeKind.Expression);

bool auditRequired = (bool) source.Execute (scope);
Console.WriteLine (auditRequired);    // True
```

还可以通过调用GetVariable收回变量:

```
string code = "result = input * 3";

ScriptEngine engine = Python.CreateEngine();

ScriptScope scope = engine.CreateScope();
scope.SetVariable ("input", 2);

ScriptSource source = engine.CreateScriptSourceFromString (code,
                                  SourceCodeKind.SingleStatement);
source.Execute (scope);
Console.WriteLine (engine.GetVariable (scope, "result"));    // 6
```

注意，我们在第二个示例（而不是Expression）中指定SourceCodeKind.SingleStatement告诉引擎我们需要执行一条语句。

类型会在.NET和Python之间自动引导。甚至可以从脚本端访问.NET对象成员:

```
string code = @"sb.Append (""World"")";

ScriptEngine engine = Python.CreateEngine ();

ScriptScope scope = engine.CreateScope ();
var sb = new StringBuilder ("Hello");
scope.SetVariable ("sb", sb);

ScriptSource source = engine.CreateScriptSourceFromString (
                      code, SourceCodeKind.SingleStatement);
source.Execute (scope);
Console.WriteLine (sb.ToString());    // HelloWorld
```

第21章 安全

本章介绍.NET的两个主要安全组件：

- 权限
- 加密

.NET中的权限提供了一个独立于操作系统的安全层。其功能有两部分：

沙箱
　　限制不能完全可信的.NET程序集可以执行的操作类型。

授权
　　限制谁可以做什么。

通过.NET中支持的加密功能可以存储或交换机密、防偷听、检测信息篡改、为存储密码生成单向哈希表和创建数字签名。

本章涵盖的类型是在下列命名空间中定义的：

```
System.Security;
System.Security.Permissions;
System.Security.Principal;
System.Security.Cryptography;
```

第15章介绍了隔离存储的内容，在安全环境中隔离存储有另一个重要作用。

21.1 权限

Framework对沙箱和授权都使用权限。权限根据条件阻止代码的执行。沙箱使用代码访问权限；授权使用身份和角色权限。

尽管这两个组件都遵守类似的模型，但它们的使用方法有很大的差异。在某种程度上因为这个原因，它们通常会将你划分到不同的安全方：对于代码访问安全，你通常会是非置信方；对于身份和角色安全，你通常会是应用非置信方。代码访问安全最常通过CLR或托管环境（如ASP.NET或Internet Explorer）对你进行限制，而授权通常由你实现，以防止未授权的调用程序访问你的程序。

作为应用程序开发者，需要了解代码访问安全（CAS），以编写可以在权限受限环境中运行的程序集。如果要编写和销售组件库，用户很可能会从沙箱环境（如SQL Server CLR主机）调用库。

了解CAS的另一个原因是可能需要创建包含其他程序集的托管环境。例如，可能会编写允许第三方编写插件的应用程序。使用受限的权限在应用程序域中运行这些插件，可以降低插件对应用程序的稳定性和安全性的影响。

身份和角色安全主要用于编写中间层应用程序和网页应用服务。通常可以对一组角色进行决定，然后对于提供的每个方法，可以要求调用程序为特定角色。

21.1.1 CodeAccessPermission和PrincipalPermission

C#中有两种基本权限：

`CodeAccessPermission`

拥有所有代码访问安全（CAS）权限的抽象类，如`FileIOPermission`、`ReflectionPermission`或`PrintingPermission`。

`PrincipalPermission`

描述了身份和/或角色（例如Mary或“人力资源”）。

术语“权限”会稍微误导`CodeAccessPermission`的情况，因为它提出的某些操作已经被批准，这不是必要情况。`CodeAccessPermission`对象描述了特权操作。

例如，`FileIOPermission`对象描述了对特殊文件或目录执行读取、编写或附加操作的特权。可以通过各种方式使用这类对象：

- 验证所有的调用程序拥有执行这些行为（`Demand`）的权利
- 验证直接调用程序拥有执行这些行为（`LinkDemand`）的权利
- 暂时退出沙箱并且不论调用程序拥有哪种特权，都声明特定程序集执行这些行为的权利

提示：Framework 4.0不支持`Deny`、`RequestMinimum`、`RequestOptional`和`RequestRefuse`，而且强烈建议不使用`PermitOnly`。它新的透明模型还有效地降低了链接要求。

`PrincipalPermission`简单得多。它唯一的安全方法是`Demand`，该方法会根据当前的执行危险性检查特定的用户或角色是否合法。

IPermission

`CodeAccessPermission`和`PrincipalPermission`都实现了`IPermission`接口：

```
public interface IPermission
{
  void Demand();
  IPermission Intersect (IPermission target);
  IPermission Union (IPermission target);
  bool IsSubsetOf (IPermission target);
  IPermission Copy();
}
```

此处至关紧要的方法是Demand。它检查当前是否允许权限或特权操作，而且如果当前不允许权限或特权操作，它会抛出SecurityException。如果你是应用非置信方，就可以使用Demand提出要求。如果你是非置信方，那么调用的代码就会使用Demand对你提出要求。

例如，要确保只有Mary可以运行管理报告，应该编写下列代码：

```
new PrincipalPermission ("Mary", null).Demand();
// ...运行管理报告
```

相反，假设对程序集进行了沙箱化了（即I/O文件被禁止），那么下列代码会抛出SecurityException：

```
using (FileStream fs = new FileStream ("test.txt", FileMode.Create))
  ...
```

在这种情况中，Demand是由调用的代码创建的，换句话说，它是由FileStream的构造方法创建的：

```
...
new FileIOPermission (...).Demand();
```

提示：代码访问安全方法Demand会检查调用堆栈中的权限，以确保请求的操作被调用链中的每一方允许（在当前的应用程序域中）。事实上，Demand会检查这个应用程序域是否对此进行授权。

通过代码访问安全，GAC中运行的程序集（这些程序集被视为完全可信）会出现有趣的情况。如果这类程序集在沙箱中运行，那么它创建的任何Demand都仍旧遵守该沙箱的权限设置。然而，完全可信的程序集可以通过调用CodeAccessPermission对象上的Assert暂时逃出沙箱。这样操作之后，已断言权限的Demand永远都会成功。Assert不是在当前方法结束时结束，就是在调用CodeAccessPermission.RevertAssert时结束。

Intersect和Union方法将两个同样类型化的权限对象组合为一个。在使用Demand要求符合交叉权限时，Intersect结果的限制性更低。在使用Demand要求同时符合两种权限时，Union结果的限制性更高。

对于代码访问权限，在Demanded时“越大的”权限对象越严格，因为它必须满足更多数量的权限。

对于身份权限，在Demanded时“越大的”权限对象越不严格，因为只需要一个身份就足够满足要求。

如果特定的target至少含有其权限，那么IsSubsetOf会返回true：

```
PrincipalPermission jay = new PrincipalPermission ("Jay", null);
PrincipalPermission sue = new PrincipalPermission ("Sue", null);

PrincipalPermission jayOrSue = (PrincipalPermission) jay.Union (sue);
Console.WriteLine (jay.IsSubsetOf (jayOrSue));        // True
```

在这个范例中，调用jay和sue上的Intersect会生成空权限，因为它们没有交迭。

21.1.2 PermissionSet

PermissionSet代表以不同方式类型化的IPermission对象的集合。下列代码使用三种代码访问权限创建了权限设置，然后Demand将所有这些权限要求一次都执行了：

```
PermissionSet ps = new PermissionSet (PermissionState.None);
```

```
ps.AddPermission (new UIPermission (PermissionState.Unrestricted));
ps.AddPermission (new SecurityPermission (
                      SecurityPermissionFlag.UnmanagedCode));
ps.AddPermission (new FileIOPermission (
                      FileIOPermissionAccess.Read, @"c:\docs"));
ps.Demand();
```

PermissionSet的构造方法接收PermissionState枚举，用于指明该权限设置是否可以被视为“不受限制”。处理不受限制的权限设置时，可以将其视为含有所有可能应用的权限（即使它的集合是空的）。使用不受限制的代码访问安全执行的程序集可以称为“完全可信”的程序集。

当调用AddPermission时，权限设置检查是否已经出现了以相同方式类型化的权限。如果没有出现以相同方式类型化的权限，它会使用Union将新权限和已存在的权限合并；否则，它会将新权限添加到其集合中。调用不受限制的权限设置上的AddPermission不会有效果（因为在逻辑上，它已经拥有了所有可能使用的权限）。

可以像对IPermission对象那样，对权限设置应用Union和Intersect。

21.1.3 声明式和命令式安全

前面介绍了手动实例化权限对象和调用这些对象上的Demand的内容，这是“命令式安全”；通过向方法、构造方法、类、结构或程序集添加属性可以实现相同的结果，这是“声明式安全”。尽管命令式安全更加灵活，但声明式安全拥有以下三个优点：

- 它可以减少编码。
- 它允许CLR预先决定程序集需要哪些权限。
- 它可以改进性能。

例如：

```
[PrincipalPermission (SecurityAction.Demand, Name="Mary")]
public ReportData GetReports()
{
  ...
}

[UIPermission(SecurityAction.Demand, Window=UIPermissionWindow.AllWindows)]
public Form FindForm()
{
  ...
}
```

这段代码会起作用，因为每个权限类型都在.NET Framework中拥有对应的属性类型。PrincipalPermission对应PrincipalPermissionAttribute。属性构造方法的第一个参数永远是SecurityAction，用于指明一旦构造了权限对象，要调用哪个安全方法（通常为Demand）。剩余的已命名参数会对相应权限对象上的特性应用镜像。

21.2 代码访问安全（CAS）

表21-1至表21-6按照分类，列出了通过.NET Framework加强的CodeAccessPermission类型。总体来说，这些表格几乎涵盖所有对程序造成损害的方法！

表21-1：核心权限

类型	允许的操作
SecurityPermission	高级操作，如调用非托管代码
ReflectionPermission	使用反射
EnvironmentPermission	读取或写入命令行环境设置
RegistryPermission	读取或写入Windows注册表

SecurityPermission接收了一个SecurityPermissionFlag参数。该参数为枚举，它可以是下列任意组合：

```
AllFlags                 ControlThread
Assertion                Execution
BindingRedirects         Infrastructure
ControlAppDomain         NoFlags
ControlDomainPolicy      RemotingConfiguration
ControlEvidence          SerializationFormatter
ControlPolicy            SkipVerification
ControlPrincipal         UnmanagedCode
```

这个枚举中最重要的成分是Execution，没有该成分代码无法运行。其他成分仅在完全可信的情况下获得，因为它们启用了一个被授权者，以损害沙箱或从沙箱中逃出。ControlAppDomain允许创建新的应用程序域（请参阅第24章）；UnmanagedCode允许调用本地方法（请参阅第25章）。

ReflectionPermission接收了一个ReflectionPermissionFlag枚举，这个枚举包括MemberAccess和RestrictedMemberAccess成分。如果要沙箱化程序集，当API（如LINQ to SQL）要求批准反射情况时获得RestrictedMemberAccess成分更安全。

表21-2：2 I/O和数据权限

类型	允许的操作
FileIOPermission	读取或写入文件和目录
FileDialogPermission	向通过打开或保存对话框选中的文件应用读取/写入操作
IsolatedStorageFilePermission	向隔离存储的文件应用读取/写入操作
ConfigurationPermission	读取应用配置文件
SqlClientPermission、OleDbPermission、OdbcPermission	使用SqlClient、OleDb或Odbc类与数据库服务器通信
DistributedTransactionPermission	参与分布式转换

FileDialogPermission控制对OpenFileDialog和SaveFileDialog类的访问。这些类是在Microsoft.Win32（在WPF应用程序中使用）和System.Windows.Forms（在Windows Forms应用程序中使用）中定义的。为了使这个操作生效，还需要使用UIPermission。但是，如果通过调用OpenFileDialog或SaveFileDialog对象上的OpenFile访问选中的文件，那么就需要使用FileIOPermission。

表21-3：网络权限

类型	允许的操作
DnsPermission	DNS查找
WebPermission	基于Web请求的网络访问
SocketPermission	基于套接字的网络访问
SmtpPermission	通过SMTP库发送邮件
NetworkInformationPermission	使用类（如Ping和NetworkInterface）

表21-4：加密权限

类型	允许的操作
DataProtectionPermission	使用Windows的数据保护方法
KeyContainerPermission	公共密钥和签名
StorePermission	访问X.509证书存储文件

表21-5：UI权限

类型	允许的操作
UIPermission	创建窗口和剪贴板交互
WebBrowserPermission	使用Web浏览器控制
MediaPermission	在WPF中支持图像、音频和视频
PrintingPermission	访问打印机

表21-6：诊断权限

类型	允许的操作
EventLogPermission	读取或写入Windows事件日志
PerformanceCounterPermission	使用Windows的性能计数器

这些权限类型的Demand在.NET Framework中得到了增强。还有一些用于在代码中增强Demand的权限类。这些类中最重要的是与创建调用程序集身份有关的类，如表21-7所示。如果应用程序域在完全可信的条件下运行（请参阅下一节），那么应注意的情况（这种情况拥有所有CAS权限）是Demand会永远成功。

表21-7：身份权限

类型	允许的操作
GacIdentityPermission	程序集被加载到GAC中
StrongNameIdentityPermission	调用的程序集拥有特别强壮的名称
PublisherIdentityPermission	使用特殊证书为调用程序集添加Authenticode签名

21.2.1 应用代码访问安全的方式

当通过Windows Shell或命令提示符运行.NET可执行程序时，它会使用无限制权限运行，这称为“完全可信”。

如果通过另一个托管环境（如SQL Server CLR集成主机、ASP.NET、ClickOnce或自定义主机）执行程序集，那么主机会限制给予程序集哪种权限。如果主机以任意方式对权限进行限制，称为“部分可信”或“沙箱化”。

更确切地说，主机不会对程序集的权限进行限制。但它会使用应用了限制的权限创建应用程序域，然后将程序集加载到这个沙箱化的域中。这意味着加载到该域中的任何其他程序集（如引用的程序集）都会使用相同的权限设置在同一个沙箱中运行。但是有以下两种例外情况：

- 在GAC中注册的程序集（包括.NET Framework）
- 主机指定的完全可信的程序集

这两类程序集被视为完全可信，而且它们可以通过断言任何它们需要的权限，从沙箱中逃出。它们还可以调用其他完全可信程序集中带有[SecurityCritical]标记的方法、运行无法验证（unsafe）的代码、调用增强链接要求的方法（而且这些链接要求永远都会成功）。

因此，当我们说部分可信的程序集调用了完全可信的程序集时，是指运行在沙箱化应用程序域中的程序集调用了GAC程序集（或者由主机指定的完全可信的程序集）。

21.2.2 完全可信测试

可以使用下列代码测试是否拥有不受限制的权限：

```
new PermissionSet (PermissionState.Unrestricted).Demand();
```

如果应用程序域是沙箱化的，那么执行这一操作会抛出异常。然而，实际情况可能是程序集完全可信并且可以通过断言使其逃出沙箱。通过查询程序集上的IsFullyTrusted特性可以测试该情况。

21.3 允许部分可信的调用程序

允许程序集接受部分可信的调用程序可能会导致特权提升攻击，因此除非提出请求，否则CLR不会允许出现这种情况。要了解其原因，让我们首先了解特权提升攻击。

21.3.1 特权提升

假设CLR没有增强前面介绍的规则，而且编写了一个专门用于完全可信情况的库。其中的特性之一为下列代码：

```
public string ConnectionString
{
  get { return File.ReadAllText (_basePath + "cxString.txt"); }
}
```

如果部署了库的用户决定（正确地或错误地）将程序集加载到GAC中。然后该用户运行了在ClickOnce或ASP.NET中托管的位于受限制的沙箱中的完全无关的应用程序。现在沙箱化的应用

程序加载了完全可信的程序集，然后尝试调用ConnectionString特性。但是，它抛出了一个SecurityException，因为File.ReadAllText会要求使用File.ReadAllText，而调用程序不会拥有File.ReadAllText（注意Demand会沿着调用堆栈检查权限）。但是请看下列方法：

```
public unsafe void Poke (int offset, int data)
{
  int* target = (int*) _origin + offset;
  *target = data;
  ...
}
```

在没有使用隐式Demand的情况下，沙箱化的程序集可以调用这个方法，然后使用它造成损害，这是一种特权提升攻击。

这种情况的问题是永远无法专门让部分可信的程序集调用库。但是，在默认情况下CLR会帮助阻止发生这种情况。

21.3.2 APTCA和[SecurityTransparent

为了帮助避免特权提升攻击，默认情况下CLR不允许部分可信的程序集调用完全可信的程序集（注1）。

要允许这种调用，必须向X应用下列两种属性之一：

- 应用[AllowPartiallyTrustedCallers]属性（简称APTCA）
- 应用[SecurityTransparent]属性

应用这些属性意味着必须考虑成为应用非置信方（而不是非置信方）的可能性。

在CLR 4.0之前，CLR仅支持APTCA属性。而且该属性的所有作用是启用部分可信的调用程序。从CLR 4.0开始，APTCA也可以以隐含方式将程序集中的所有方法（以及函数）标记为“安全性透明”。下一节将详细介绍这部分内容；现在可以将这部分内容概括为安全性透明方法无法执行下列操作（不论是在部分可信环境中运行还是在完全可信环境中运行）：

- 运行无法验证的（Unsafe）代码。
- 通过P/Invoke或COM运行本地代码。
- 断言权限以提升它们的安全等级。
- 满足链接要求。
- 调用在.NET Framework标记为[SecurityCritical]的方法。基本上，这些方法包含在没有适当的安全措施或安全检查的情况下执行了上述4个操作之一的方法。

提示： 基本原理是无法执行上述操作的程序集通常易于受到特权提升攻击。

注1：在CLR 4.0之前，如果目标程序集使用了强命名，那么除非应用了APTCA，否则部分可信的程序集无法调用其他部分可信的程序集。这种限制无法真正帮助提高安全性，因此CLR 4.0取消了该限制。

[SecurityTransparent]属性应用了同一规则更强壮的版本。其差异在于对APTCA的使用，在这种情况可以在程序集中将选中的方法指定为非透明的，而在使用[SecurityTransparent]的情况中，所有方法必须是透明的。

提示：如果程序集可以与[SecurityTransparent]一起工作，执行的操作就会与库相同。可以忽略透明模型的细微差别，然后直接阅读后面“操作系统安全”的相关内容。

在介绍将选中的方法指定为非透明的之前，我们先介绍应用这些属性的情况。

第一种情况（同时也是较明显的情况）是编写将要在部分可信域中运行的完全可信的程序集。我们稍后将介绍一个这样的范例。

第二种情况（同时也是较不明显的情况）是在不知道将来部署情况的前提下编写库。例如，假设要编写与测绘仪有关的对象，而且要在Internet上销售它。用户可以通过以下三种方式调用库：

1. 通过完全可信的环境
2. 通过沙箱化的域
3. 通过沙箱化的域，但是将程序集视为完全可信（例如将其加载到GAC中）

很容易忽略第三种方式，而且这正是透明模型提供帮助之处。

21.4 CLR 4.0中的透明模型

提示：为了继续阅读下面的内容，需要回顾前面几节并且理解应用APTCA和[SecurityTransparent]的情况。

安全的透明模型可以使保护完全可信的程序集的安全性变得更加容易，然后可以通过部分可信的代码调用完全可信的程序集。

以此类推，假设部分可信的程序集就像被宣告有罪的罪犯，而且这个罪犯正要被送入监狱。在监狱中，该罪犯可以通过良好的表现获得一系列特权（权限）。凭借这些权限该罪犯可以执行一些行为，如看电视或打篮球。然而，该罪犯永远也不能执行某些行为，如获得进入电视房间的钥匙（或者监狱大门的钥匙），因为这类行为（方法）会损害整个安全系统。这些方法称为“安全关键”。

如果要编写完全可信的库，就需要保护这些安全关键方法。一种方式是要求调用程序是完全可信的。这种方式优先于CLR 4.0：

```
[PermissionSet (SecurityAction.Demand, Unrestricted = true)]
public Key GetTVRoomKey() { ... }
```

这样做会产生两个问题。第一个问题，Demand会变慢，因为它们必须沿调用堆栈检查权限；该情况的重点在于安全关键方法有时就是性能关键。如果循环（也许位于Framework中的另一个可信的程序集中）中调用了安全关键方法，那么Demand会变得特别浪费资源。使用这类方法的CLR 2.0会增强链接要求，而且仅会对直接调用程序检查这些要求。但是这样做同样需要付出代价。为了保持安全性，调用了带有链接要求的方法的方法必须本身执行链接要求，或者当从可信度较低方调用方法时通过审核确保不执行具有潜在威胁的操作。当调用关系比较复杂时，这类审核会变得难以实现。

第二个问题是很容易忘记在安全关键方法上执行要求或链接要求（同理，复杂的调用关系会加剧这个情况）。如果CLR能够以某种方式提供帮助会解决这个情况，同样强制安全关键函数不会无目的地公开于同类函数也会解决该情况。

这就是透明模型产生的结果。

提示： 透明模型的引入与删除CAS策略没有一点关系（请参阅本章后面的相关内容）。

21.4.1 透明模型的工作方式

在透明模型中，安全关键方法带有[SecurityCritical]属性：

```
[SecurityCritical]
public Key GetTVRoomKey() { ... }
```

所有"危险的"方法（含有CLR认为能够破坏安全性并且允许破坏者逃跑的代码）必须使用[SecurityCritical]或[SecuritySafeCritical]标记。这类方法包括：

- 无法验证的（Unsafe）方法
- 通过P/Invoke或COM交互操作调用非托管代码的方法
- Assert权限或调用带有链接要求方法的方法
- 调用[SecurityCritical]方法的方法
- 重写虚拟的[SecurityCritical]方法的方法

[SecurityCritical]表示这个方法允许部分可信的调用程序逃出沙箱。

[SecuritySafeCritical]表示这个方法起安全关键的作用，但是它拥有适当的保护，因此对于部分可信的调用程序它是安全的。

部分可信的程序集永远无法调用完全可信的程序集中的安全关键方法。[SecurityCritical]方法只能由下列方法调用：

- 其他[SecurityCritical]方法
- 带[SecuritySafeCritical]标记的方法

安全保护关键方法就像是安全关键方法的门卫（如图21-1所示），而且任何程序集（由基于权限的CAS要求管理的完全或部分可信的程序集）中的任何方法都可以调用安全保护关键方法。为了说明，假设一个犯人想要看电视，他调用的WatchTV（看电视）方法需要调用GetTVRoomKey（获得电视房间钥匙）方法，这意味着WatchTV方法必定是安全保护关键：

```
[SecuritySafeCritical]
public void WatchTV()
{
  new TVPermission().Demand();
  using (Key key = GetTVRoomKey())
    PrisonGuard.OpenDoor (key);
}
```

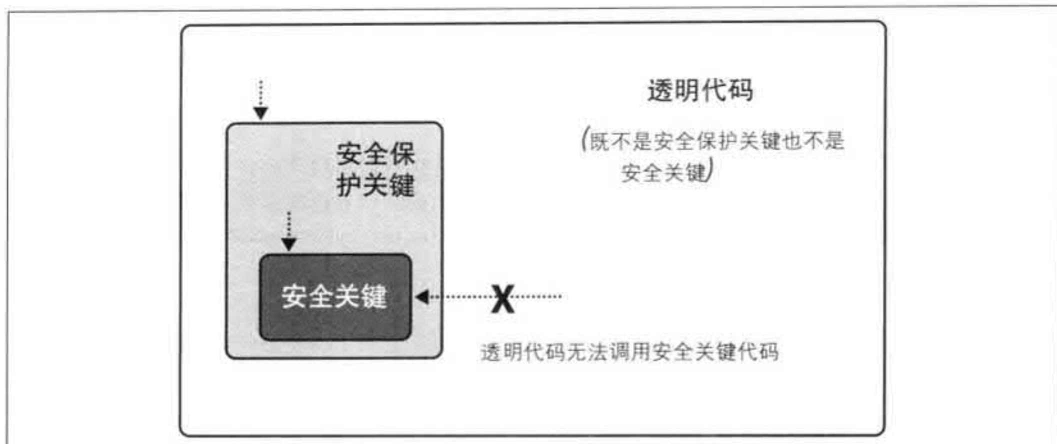


图21-1：透明模型：只有灰色区域需要安全审核

注意，我们使用Demand要求了一个TVPermission，以确保调用程序实际上拥有看电视的权限，而且小心地处理了我们创建的钥匙。我们要保护安全关键方法，使其对调用方都保持安全性。

提示：CLR将参与某些行为的方法视为“危险的方法”，但实际上这些方法并不危险。可以直接为这些方法添加[SecuritySafeCritical]标记，而不是添加[SecurityCritical]标记。Array.Copy方法就是一个这样的例子：为了提高效率它拥有非托管的执行方式，但也无法被部分可信的调用程序滥用。

UnsafeXXX模式

在看电视范例中有潜在的低效率情况，如果监狱警卫想要通过WatchTV方法看电视，那么他必须满足TVPermission要求（这是不必要的）。作为一种补救，CLR团队推荐为方法定义两种版本的模式。第一个版本是在安全关键中增加Unsafe前缀：

```
[SecurityCritical]
public void UnsafeWatchTV()
{
  using (Key key = GetTVRoomKey())
    PrisonGuard.OpenDoor(key);
}
```

第二个版本是安全保护关键，然后在满足了全部堆栈遍历要求后调用第一个版本：

```
[SecuritySafeCritical]
public void WatchTV()
{
  new TVPermission().Demand();
  UnsafeWatchTV();
}
```

1. 透明代码

在透明模型中，所有方法都分为以下三类：

- *安全关键*
- *安全保护关键*
- 不属于以上两类方法的方法（在这种情况中，它们称为*透明方法*）

透明方法之所以称为透明方法是因为当它们为了防止特权提升攻击而审核代码时，可以忽略它们。需要注意的是[SecuritySafeCritical]方法（门卫），它们通常仅包含程序集方法的一小部分。如果程序集包含整个透明方法，那么整个程序集都可以标记上[SecurityTransparent]属性：

```
[assembly: SecurityTransparent]
```

在这种情况中可以说这个程序集本身是透明的。透明的程序集无需为了防止特权攻击而进行审核，而且以隐含方式允许部分可信的调用程序进行调用（无需应用APTCA）。

2. 将透明设置为程序集的默认状态

概括前面的内容，可以通过两种方式将程序集等级设定为透明状态：

- 应用APTCA。除非添加其他标记，否则所有方法都会以隐含方式指定为透明的。
- 应用[SecurityTransparent]程序集属性。在不出现异常的情况下，所有方法都会以隐含方式指定为透明的。

第3种方式是不做任何操作。这仍旧符合透明规则，但是所有方法都会以隐含方式带有[SecurityCritical]（重写的虚拟[SecuritySafeCritical]方法除外，这些方法仍旧是安全关键）。其效果是（假设你的方法完全可信）你的方法可以调用任何方法，但是其他程序集中的透明方法无法调用你的方法。

21.4.2 通过透明模型编写APTCA库

要使用透明模型，首先应识别出程序集中带有潜在"危险"的方法（如上一节所述）。单元测试会找出这些方法，因为CLR会拒绝运行这类方法（即使是在完全可信的环境中）。这样这类方法会添加以下标记（Framework 4.0还自带一个*SecAnnotate.exe*工具，使用它可以为该操作提供帮助）：

- [SecurityCritical]，如果方法可能是不安全的或者是从可信度较低的程序集调用的。
- [SecuritySafeCritical]，如果方法执行了适当的检查/保护并且可以安全地从可信度较低的程序集调用。

要了解这一点，请看下面的方法，它在.NET Framework中调用了安全关键方法：

```
public static void LoadLibraries()
{
  GC.AddMemoryPressure (1000000);    // 安全关键
  ...
}
```

可信度较低的调用程序可能会通过重复调用该方法，从而滥用该方法。可以对其应用[SecurityCritical]属性，但是这样该方法只能由其他可信方通过关键或安全关键方法调用。一个更好的解决方案是固定该方法使其变得安全，然后应用[SecuritySafeCritical]属性：

```
static bool _loaded;
```

```
[SecuritySafeCritical]
public static void LoadLibraries()
{
  if (_loaded) return;
  _loaded = true;
  GC.AddMemoryPressure (1000000);
  ...
}
```

（这样做还可以使该方法对于可信的调用程序更加安全。）

1. 保护不安全的方法

假设我们拥有一个unsafe方法，如果由可信度较低的程序集调用它就会产生潜在的危险。可以简单地为其添加[SecurityCritical]：

```
[SecurityCritical]
public unsafe void Poke (int offset, int data)
{
  int* target = (int*) _origin + offset;
  *target = data;
  ...
}
```

提示：如果在透明模型中编写了不安全的代码，那么在执行这个方法前CLR会抛出VerificationException。

这样就需要保护上列方法，适当地为它们添加[SecurityCritical]或[SecuritySafeCritical]标记。

请看下面的unsafe方法，它过滤了一个位图。这实际上是无害的，因此可以为它添加SecuritySafeCritical标记：

```
[SecuritySafeCritical]
unsafe void BlueFilter (int[,] bitmap)
{
  int length = bitmap.Length;
  fixed (int* b = bitmap)
  {
    int* p = b;
    for (int i = 0; i < length; i++)
    *p++ &= 0xFF;
  }
}
```

相反，可以编写一个不执行任何CLR认为是“危险”操作的函数，但该函数可以引发危险。那么也可以为其添加[SecurityCritical]标记：

```
public string Password
{
  [SecurityCritical] get { return _password; }
}
```

2. P/Invokes和[SuppressUnmanagedSecurity]

请看下面的非托管方法，通过Point (System.Drawing)返回了一个窗口句柄：

```
[DllImport ("user32.dll")]
public static extern IntPtr WindowFromPoint (Point point);
```

注意，仅能通过[SecurityCritical]和[SecuritySafeCritical]方法调用非托管代码。

提示： 可以说所有extern方法都是隐含的[SecurityCritical]，尽管还存在细微的差别：显式地将[SecurityCritical]应用于extern方法，对将安全检查从运行时提早到JIT时间有些许效果。请看下面的方法：

```
static void Foo (bool exec)
{
  if (exec) WindowFromPoint (...)
}
```

如果通过false调用，那么该方法仅会在WindowFromPoint显式地带有[SecurityCritical]的情况下，由安全检查控制。

因为我们将该方法指定为全局方法，所以其他完全可信的程序集可以直接通过[SecurityCritical]方法直接调用WindowFromPoint。对于部分可信的调用程序，我们提供了下列安全版本，该版本可以通过Demanding UI排除危险并返回托管类（而不是IntPtr）：

```
[UIPermission (SecurityAction.Demand, Unrestricted = true)]
[SecuritySafeCritical]
public static System.Windows.Forms.Control ControlFromPoint (Point point)
{
  IntPtr winPtr = WindowFromPoint (point);
  if (winPtr == IntPtr.Zero) return null;
  return System.Windows.Forms.Form.FromChildHandle (winPtr);
}
```

现在还有一个问题：不论何时使用P/Invoke，CLR都会为非托管权限执行隐含的Demand。而且因为Demand会沿调用堆栈检查权限，如果调用程序是部分可信的，那么WindowFromPoint方法会失效。可以通过两种方式解决这个问题。第一个解决方案是在ControlFromPoint方法的第一行代码中为非托管代码断言权限：

```
new SecurityPermission (SecurityPermissionFlag.UnmanagedCode).Assert();
```

在此处断言特定程序集的非托管权限可以确保WindowFromPoint中隐含的Demand以后会成功。当然，如果程序集本身不是完全可信的（加载到GAC中或者由主机指定为完全可信的），那么该断言会失效。本章稍后会详细介绍断言。

第二个解决方案（性能更高）是向非托管方法应用[SuppressUnmanagedCodeSecurity]属性：

```
[DllImport ("user32.dll"), SuppressUnmanagedCodeSecurity]
public static extern IntPtr WindowFromPoint (Point point);
```

这样做可以指定CLR忽略耗费资源的非托管Demand的堆栈遍历操作（如果WindowFromPoint由其他可信的类或程序集调用，那么最优化操作会变得十分有用）。然后转储（dump）ControlFromPoint中的非托管权限断言。

提示：因为要使用透明模型，所以将这个属性应用于extern方法不会产生与CLR 2.0中类似的危险。这是因为事实上P/Invokes是隐含的安全关键，方法仍旧会受到保护，因此方法只能通过其他关键或安全关键方法调用。

21.4.3 完全信任情况中的透明

在完全信任的情况中，可能需要编写关键代码并避免安全属性和方法审核的操作。最简单的实现方式是不附加任何程序集安全属性，在这种情况中所有方法都是隐含的[SecurityCritical]。

如果所有参与的程序集都执行相同的操作，或者启用的透明的程序集位于调用关系的底部，那么这种方式会起作用。换句话说，仍旧可以调用第三方库（和.NET Framework）中的透明方法。

以反向方式处理会出现问题；然而，这种问题通常可以引出更好的解决方案。假设要编写程序集T，该程序集是部分或全部透明的，然后想要调用程序集X，这是个非属性化的程序集（因此是完全关键的）。可以使用下列方式调用它：

- 将自己的程序集完全关键化。如果域总是完全可信的，那么无需支持部分可信的调用程序。从而使缺失的显式支持具有意义。
- 将调用X的代码封装到[SecuritySafeCritical]中，可以突出显示安全弱点（尽管这样操作比较复杂）。
- 考虑应用透明。如果X没有执行任何关键的操作，这会像向X应用[SecurityTransparent]那样简单。如果X确实执行了关键函数，遵守透明模型的过程会强制认识到（如果不进行处理）X的弱点。

CLR 2.0中的安全策略

在CLR 4.0之前，CLR根据规则和映射的复杂设置，获得.NET程序集的默认权限。这称为CAS策略，而且该策略是在计算机的.NET Framework配置中定义的。通过策略提升、企业自定义、机器、用户和应用程序域等级产生了三类标准授权组：

- “完全信任”——对在本地硬盘驱动器上运行的程序集授权
- “本地内联网”——对通过网络共享运行的程序集授权
- Internet——对在Internet Explorer中运行的程序集授权

默认情况下，只有完全信任的程序集是完全可信的。这意味着如果通过网络共享运行可执行的.NET程序，那么该程序会通过受限的权限组运行，而且通常会失效。因此应该提供一些保护，但是实际情况是CLR没有提供任何保护，因为恶意代码可以使用非托管的并且不受权限限制的可执行程序轻易替换.NET可执行程序。这些限制仅会阻止通过网络共享在完全信任的情况下运行.NET程序集。

因此，CLR 4.0的设计者决定废除这些安全策略。现在所有程序集都在由托管环境完全定义的权限组中运行。不论是通过网络共享还是在本地硬盘驱动器上，通过双击或命令提示符运行的可执行程序总是以完全信任的方式运行。

换句话说，现在安全性完全由主机限定的权限决定，与机器的CAS策略无关。

如果仍旧需要使用CLR 2.0的安全策略（如果可执行程序的目标是Framework 3.5或更低版本），可以使用*mscorcfg.msc* MMC插件（“控制面板”→“管理工具”→“Microsoft .NET Framework配置”）或*caspol.exe*命令行工具查看和调整安全策略。MMC插件不再是.NET Framework的标准自带插件：必须自己安装.NET Framework 3.5 SDK。

安全配置最终存储在Framework配置文件夹中名为*security.config*的XML文件中。可以通过下列方式获取该文件：

```
string dir = Path.Combine
 (System.Runtime.InteropServices.RuntimeEnvironment
                .GetRuntimeDirectory(), "config");
string configFile = Path.Combine (dir, "security.config");
```

21.5 沙箱化程序集

假设需要编写允许用户安装第三方插件的应用程序。最可能的情况是需要阻止作为可信应用程序的插件利用授予它的特权，从而避免它破坏应用程序，或者最终用户的计算机。最佳的实现方式是让每个插件在其本身的沙箱化应用程序域中运行。

例如，假设一个插件被封装为名为*plugin.exe*的.NET程序集，而且要激活它仅需启动这个可执行程序（第24章介绍将库加载到应用程序域的方法以及通过更复杂的方式与库进行交互操作的方法）

下面列出主机程序的完整代码：

```
using System;
using System.IO;
using System.Net;
using System.Reflection;
using System.Security;
using System.Security.Policy;
using System.Security.Permissions;

class Program
{
  static void Main()
  {
    string pluginFolder = Path.Combine (
      AppDomain.CurrentDomain.BaseDirectory, "plugins");

    string plugInPath = Path.Combine (pluginFolder, "plugin.exe");

    PermissionSet ps = new PermissionSet (PermissionState.None);

    ps.AddPermission
      (new SecurityPermission (SecurityPermissionFlag.Execution));

    ps.AddPermission
      (new FileIOPermission (FileIOPermissionAccess.PathDiscovery |
                             FileIOPermissionAccess.Read, plugInPath));

    ps.AddPermission (new UIPermission (PermissionState.Unrestricted));
```

```
      AppDomainSetup setup = AppDomain.CurrentDomain.SetupInformation;
      AppDomain sandbox = AppDomain.CreateDomain ("sbox", null, setup, ps);
      sandbox.ExecuteAssembly (plugInPath);
      AppDomain.Unload (sandbox);
    }
  }
```

提示：还可以将想要完全信任的程序集的列表传递到CreateDomain方法中。下一节将介绍一个完整的范例。

首先创建受限制的权限组，用于描述授予沙箱的特权。这个权限组必须至少包括执行权限和插件读取其本身程序集的权限；否则，该插件将无法启动。在本例中，我们还授予了该插件不受限制的UI权限。然后构建一个新的应用程序域，设定自定义的权限组，可以将该权限组授予所有加载到新应用程序域中的程序集。这样就可以在新域中执行插件的程序集，然后在完成插件的执行后卸载这个域。

警告：本范例从名为*plugins*的子目录中加载了该插件的程序集。当完全可信的主机产生潜在的特权提升威胁时（在这种情况中，完全可信的域会隐式地加载和运行插件程序集中的代码，以分解类型），将这些插件放在同一目录中。可以说明这种情况发生原因的例子是，插件是否抛出了由其本身的程序集定义类型的自定义异常。当主机中出现这种异常时，如果可以找到该插件，那么主机会隐式地加载该插件的程序集，以尝试反序列化该异常。将这些插件放入隔离的文件夹中，阻止加载它们。

断言权限

当编写能够由部分可信程序集调用的方法时，权限断言很有用。通过权限断言可以使完全可信的程序集暂时逃出沙箱，以执行下行Demands禁止的行动。

提示：在CAS中断言无法对诊断或基于契约的断言起作用。实际上，调用Debug.Assert更像要求权限而不是断言权限。尤其是，在断言成功的情况下断言权限有副作用，而Debug.Assert没有副作用。

我们在前面编写了一个应用程序，它可以在受限制的权限组中运行第三方插件。假设需要通过调用插件的安全方法的库扩展这个应用程序。例如，可以禁止插件直接访问数据库，而仍旧允许它们通过我们提供的库中的方法执行某些查询；也可以为了编写日志文件提供方法，而无需给予这些插件任何基于文件的权限。

要实现这一操作，第一个步骤是创建独立的程序集（例如*utilities*），然后添加AllowPartiallyTrustedCallers属性。可以按照下列方式提供方法：

```
public static void WriteLog (string msg)
{
  // 写入日志
  ...
}
```

此处的难点是需要使用FileIOPermission向文件写入数据。即使*utilities*程序集会被完全信任，但是调用程序不会被完全信任，而且所有基于文件的Demand都会因此失效。解决方法是先断言该权限：

```
public class Utils
```

```
{
  string _logsFolder = ...;

  [SecuritySafeCritical]
  public static void WriteLog (string msg)
  {
    FileIOPermission f = new FileIOPermission (PermissionState.None);
    f.AddPathList (FileIOPermissionAccess.AllAccess, _logsFolder);
    f.Assert();

    // 写入日志
    ...
  }
}
```

提示： 因为要断言权限，所以必须要将方法标记为[SecurityCritical]或[SecuritySafeCritical]（除非使用Framework较早的版本）。在这种情况中，对于部分可信的调用程序该方法是安全的，因此可选择SecuritySafeCritical。当然，这意味着不能使用[SecurityTransparent]标记整个程序集，而必须使用APTCA代替它。

注意，Demand会执行抽查，然后在权限没有被满足的情况下抛出异常。然后Demand会遍历堆栈，以检查所有调用程序仍拥有权限（在当前的AppDomain中）。断言仅会检查当前程序集是否拥有必要权限，而且如果当前程序集拥有必要权限，它会在堆栈中添加一个标记，指明从现在开始调用程序的权限会被忽略，而且只有当前程序集会被视为遵守这些权限。当方法结束或者调用CodeAccessPermission.RevertAssert的时候，断言就会结束。

为了完成这个范例，剩下的步骤是创建完全信任*utilities*程序集的沙箱化应用程序域。然后实例化描述这个程序集的StrongName对象，将它传递到AppDomain的CreateDomain方法中：

```
static void Main()
{
  string pluginFolder = Path.Combine (
    AppDomain.CurrentDomain.BaseDirectory, "plugins");

  string plugInPath = Path.Combine (pluginFolder, "plugin.exe");

  PermissionSet ps = new PermissionSet (PermissionState.None);

  // 使用前面介绍的方法向ps中添加想要的权限
  // ...

  Assembly utilAssembly = typeof (Utils).Assembly;
  StrongName utils = utilAssembly.Evidence.GetHostEvidence<StrongName>();

  AppDomainSetup setup = AppDomain.CurrentDomain.SetupInformation;
  AppDomain sandbox = AppDomain.CreateDomain ("sbox", null, setup, ps,
                                              utils);
  sandbox.ExecuteAssembly (plugInPath);
  AppDomain.Unload (sandbox);
}
```

为了让这段代码生效，*utilities*程序集必须应用强命名签名。

提示：在Framework 4.0之前的版本中，像本例这样调用GetHostEvidence无法获得StrongName。解决方案是使用下面这段代码代替：

```
AssemblyName name = utilAssembly.GetName();
StrongName utils = new StrongName (
  new StrongNamePublicKeyBlob (name.GetPublicKey()),
  name.Name,
  name.Version);
```

当需要将程序集加载到主机域中时，原始方法仍旧有用。这是因为在不需要Assembly或Type的情况下就可以获得AssemblyName：

```
AssemblyName name = AssemblyName.GetAssemblyName(@"d:\utils.dll");
```

安全

21.6 操作系统安全

操作系统可以根据用户的登录特权，进一步限定应用程序的行为。Windows中有两种账户：

- 管理员账户可以不受限制地访问本地计算机。
- 权限受限制的账户在使用管理功能和查看其他用户的数据时受到限制。

Windows Vista引入了用户访问控制（UAC）功能，该功能是指在登录时管理员会收到两块令牌或“帽子”：一个管理员帽子和一个普通用户帽子。默认情况下，程序通过普通用户帽子运行（即使用受限制的权限），除非程序要求提升到管理员权限。这样用户必须通过出现的对话框的批准。

对于应用程序开发者来说，UAC意味着（在默认情况下）开发的应用程序将使用受限制的用户权限运行。这代表必须完成下列两个操作之一：

- 编写无需管理员权限即可运行的应用程序。
- 在应用程序清单中要求提升到管理员权限。

第一个选择比较安全，而且更加方便。在大多数情况中，将程序设计成无需管理员权限即可运行是很容易的：这类应用程序的限制比典型的代码访问安全沙箱少得多。

提示：使用下列方法可以查明程序是否在使用管理员账户运行：

```
[DllImport ("shell32.dll", EntryPoint = "#680")]
static extern bool IsUserAnAdmin();
```

在启用了UAC的情况下，只有在当前过程拥有提升的管理员权限时该方法才会返回true。

21.6.1 在标准用户账户中运行

使用Windows标准的用户账户无法实现下列关键操作：

- 向下列目录中写入数据：
  — 操作系统的文件夹（通常为*\Windows*）及其子目录

    — 程序文件文件夹（\Program Files）及其子目录
    — 操作系统驱动器的根目录（例如C:\）

- 向注册表中的HKEY_LOCAL_MACHINE分支写入数据
- 读取性能监听数据（WMI）

而且，操作系统可能会拒绝普通用户（甚至是管理员）访问其他用户的文件或资源。Windows使用访问控制列表（ACL）保护这类资源，通过System.Security.AccessControl中的类型可以查询和断言在ACL中的权限。ACL还可以应用于交叉处理的等待句柄，第22章详细介绍这部分内容。

如果操作系统拒绝程序访问操作系统安全的任何结果，程序会抛出Unautho-rizedAccessException。这与.NET权限要求不满足时抛出的SecurityException不同。

提示： .NET代码访问权限类独立于ACL。这意味着程序可以成功地要求FileIOPermission，但是当尝试访问文件时仍旧会由于ACL的限制得到UnauthorizedAccessException。

在大多数情况中，可以通过下列方式处理标准用户限制：

- 将文件写入用户指定的位置。
- 避免使用无法存储到文件中的注册表信息（可以对HKEY_CURRENT_USER附近的文件进行读/写访问）。
- 在安装过程中注册ActiveX或COM组件。

用户文档的指定位置是SpecialFolder.MyDocuments：

```
string docsFolder = Environment.GetFolderPath
                    (Environment.SpecialFolder.MyDocuments);

string path = Path.Combine (docsFolder, "test.txt");
```

配置文件的指定位置是SpecialFolder.ApplicationData（仅对于当前用户）或SpecialFolder.CommonApplicationData（所有用户），用户可能需要使用配置文件在外部修改应用程序。通常可以根据公司和产品的名称，在这些文件夹中创建子目录。

仅在应用程序中访问的数据的存储位置是隔离存储位置。

也许使用标准用户账户运行程序的最大不便之处在于程序无法对其文件进行写入访问，这使得实现自动更新系统变得困难。一种选择是使用ClickOnce进行部署：这样可以允许在不进行管理员权限提升的情况下应用更新，但是在安装过程中设置了大量限制（例如无法注册ActiveX控件）。通过ClickOnce部署的应用程序还可以根据它们的传输模式，通过代码访问安全进行沙箱化。本书前面的内容介绍过另一个更复杂的解决方案。

21.6.2 管理员权限提升和虚拟化

第18章介绍了部署应用程序清单的方法。通过应用程序清单，可以要求Windows提示用户在运行程序时进行管理员权限提升：

```
<?xml version="1.0" encoding="utf-8"?>
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1">
```

```
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges>
        <requestedExecutionLevel level="requireAdministrator" />
      </requestedPrivileges>
    </security>
  </trustInfo>
</assembly>
```

如果使用asInvoker替换requireAdministrator，那么它会指示Windows不需要进行管理员权限提升。除了虚拟化被禁用之外，其效果几乎与完全没有应用程序清单一样。虚拟化是Windows Vista引入的一个临时方法，以帮助旧应用程序在没有管理员权限的情况下正确地运行。在没有应用程序清单的情况下，使用requestedExecutionLevel元素可以激活这个向下兼容的功能。

当应用程序向程序文件、Windows目录或注册表中的HKEY_LOCAL_MACHINE区域中写入数据时，虚拟化就会起作用。没有抛出异常，此处的改变是重定向到无法影响原始数据的独立硬盘位置。这可以防止应用程序干扰操作系统或其他运行正常的应用程序。

21.7 身份和角色安全

在编写中间层服务器或ASP.NET应用程序时，基于身份和角色的安全非常有用，在这些情况中可能需要处理许多用户。可以根据验证的用户名或角色限定可以使用的功能。身份描述了用户名；角色描述了组。*Principal*是描述身份和/或角色的对象。因此，PrincipalPermission类增强了身份和/或角色安全。

在典型的应用程序服务器中，可以要求提供给需要增强安全的客户端的所有方法上的PrincipalPermission。例如，下列代码要求调用者是finance角色的成员：

```
[PrincipalPermission (SecurityAction.Demand, Role = "finance")]
public decimal GetGrossTurnover (int year)
{
  ...
}
```

为了限定只有特殊用户才能调用方法，可以指定Name进行代替：

```
[PrincipalPermission (SecurityAction.Demand, Name = "sally")]
```

（当然，必须使用硬编码的必要性使这段代码难以管理。）为了允许将身份或角色组合使用，必须使用强制安全进行代替。这意味着实例化PrincipalPermission对象，调用Union组合它们，然后在最后的结果上调用Demand。

分配用户和角色

在PrincipalPermission要求成功前，必须将IPrincipal对象附加到当前线程上。

可以根据是否需要影响整个应用程序域或仅影响当前线程，通过下面两种方式将当前使用的Windows用户指定为某种身份：

```
AppDomain.CurrentDomain.SetPrincipalPolicy (PrincipalPolicy.WindowsPrincipal);
// 或：
```

```
Thread.CurrentPrincipal = new WindowsPrincipal (WindowsIdentity.GetCurrent());
```

如果正在使用WCF或ASP.NET，它们的基础框架可以帮助模仿客户端的身份。还可以使用GenericPrincipal和GenericIdentity类实现这一功能。下列代码创建了一个名为Jack的用户并为他分配了3种角色：

```
GenericIdentity id = new GenericIdentity ("Jack");
GenericPrincipal p = new GenericPrincipal
  (id, new string[] { "accounts", "finance", "management" } );
```

为了使这段代码生效，应使用下列代码将其分配到当前线程：

```
Thread.CurrentPrincipal = p;
```

原则是基于线程的，因为应用程序服务器通常并行处理许多客户端请求，每个客户端请求都位于其本身的线程上。因为每个请求都来自不同的客户端，所以它们需要不同的原则。

可以子类化GenericIdentity和GenericPrincipal，也可以在自定义的类型中直接实现IIdentity和IPrincipal接口。下面的代码显示了这些接口的定义方式：

```
public interface IIdentity
{
  string Name { get; }
  string AuthenticationType { get; }
  bool IsAuthenticated { get; }
}
public interface IPrincipal
{
  IIdentity Identity { get; }
  bool IsInRole (string role);
}
```

关键方法是IsInRole。注意，这里没有返回角色列表的方法，因此应用程序仅会受到特定角色是否符合该原则的制约。这段代码可以成为更复杂的授权系统的基础。

21.8 加密综述

表21-8概括了.NET中的加密选项。本章后面的几节会详细介绍这些选项。

表21-8：.NET中的加密和散列法

选项	管理密钥	速度	强度	注释
File.Encrypt	0	快	由用户密码决定	以透明方式使用文件系统支持保护文件；密钥是隐式地从登录用户的证书得到的
Windows Data Protection	0	快	由用户密码决定	使用隐式获得的密钥加密和解密字节数组
Hashing	0	快	高	单向（不可逆的）的转换；用于存储密码、对比文件和检查数据错误

表21-8：.NET中的加密和散列法（续）

选项	管理密钥	速度	强度	注释
Symmetric Encryption	1	快	高	用于普通的加密/解密；使用相同的密钥加密和解密；可以用于保护传输中的信息
Public Key Encryption	2	慢	高	使用不同的密钥；用于交换传输消息中的对称密钥和数字签名文件

安全

为了创建和验证System.Security.Cryptography.Xml中基于XML的签名以及与System.Security.Cryptography.X509Certificates中的数字证书结合使用的类型，Framework还提供了更多的专门支持。

21.9 Windows数据保护

第15章“文件和目录操作”小节介绍了使用File.Encrypt要求操作系统以透明方式加密文件的方法：

```
File.WriteAllText ("myfile.txt", "");
File.Encrypt ("myfile.txt");
File.AppendAllText ("myfile.txt", "sensitive data");
```

这种情况中的加密使用了通过登录用户的密码得到的密钥。可以使用这种以相同的隐含方式获得的密钥，通过Windows数据保护API加密字节数组。数据保护API是通过ProtectedData类（带有两种静态方法的简单类型）指定的：

```
public static byte[] Protect (byte[] userData, byte[] optionalEntropy,
                              DataProtectionScope scope);

public static byte[] Unprotect (byte[] encryptedData, byte[] optionalEntropy,
                                DataProtectionScope scope);
```

提示：System.Security.Cryptography中的大多数类型位于*mscorlib.dll*和*System.dll*中。ProtectedData是一个例外，它位于*System.Security.dll*中。

不论在optionalEntropy中保护什么类型的数据，它们都会被添加到密钥中，从而增加其安全性。DataProtectionScope枚举参数允许使用两个选项：CurrentUser或LocalMachine。通过CurrentUser，可以从登录用户的证书获得密钥；通过LocalMachine，会使用所有用户常见的服务器端密钥。LocalMachine密钥提供的保护较低，但是它可以在需要使用各种账户的Windows服务或程序中运行。

下面的代码演示了简单的加密和解密过程：

```
byte[] original = {1, 2, 3, 4, 5};
DataProtectionScope scope = DataProtectionScope.CurrentUser;

byte[] encrypted = ProtectedData.Protect (original, null, scope);
byte[] decrypted = ProtectedData.Unprotect (encrypted, null, scope);
// 被解密为{1, 2, 3, 4, 5}
```

Windows数据保护提供中等的安全保护，它根据用户的密码强度，防止攻击者获取对计算机的完全访问权限。在使用LocalMachine的情况中，LocalMachine仅对一些有限的物理和电子访问有效。

21.10 散列法

散列法提供了一种加密方式。这种加密方式非常适用于存储数据库中的密码，因为不需要（或不想要）看到解密的版本。要进行验证，仅需散列用户输入的信息，然后将其与数据库中存储的信息相比较即可。

不论源数据的长度有多少，散列编码永远为较小的固定大小。这使其在比较文件或检查数据流（与校验和不同）时发挥重要作用。源数据中更改任何位置的单个位都会使得散列编码发生巨大的变化。

要进行散列操作，可调用HashAlgorithm某个子类（如SHA256或MD5）上的ComputeHash：

```
byte[] hash;
using (Stream fs = File.OpenRead ("checkme.doc"))
  hash = MD5.Create().ComputeHash (fs);          //散列长度为16字节
```

ComputeHash方法还可以接收字节数组，这对散列法密码非常方便：

```
byte[] data = System.Text.Encoding.UTF8.GetBytes ("stRhong%pword");
byte[] hash = SHA256.Create().ComputeHash (data);
```

提示：Encoding对象上的GetBytes方法将一个字符串转换为一个字节数组；GetString方法将该数组重新转换为字符串。然而，Encoding对象无法将加密的或散列的字节数组转换为字符串，因为编码数据通常会破坏文本编码规则。可以使用下列Convert.ToBase64String方法和Convert.FromBase64String方法代替。这些方法可以使字节数组和合法（与XML友好）的字符串相互转换。

MD5和SHA256是HashAlgorithm的两个子类型，它们是由.NET Framework提供的。下面按照安全等级的升序（和以字节为单位的散列长度）列出了主要算法：

```
MD5(16) → SHA1(20) → SHA256(32) → SHA384(48) → SHA512(64)
```

算法的长度越短，其执行速度就越快。MD5的执行速度比SHA512的执行速度快20多倍，而且非常适合计算文件的校验和。使用MD5每秒钟可以加密数百兆字节，然后将结果存储到Guid中（Guid的长度恰好为16字节，而且作为一个值类型它比字节数组更易于处理；例如，将Guid与大致同等的操作符比较能够获得有意义的结果）。然而，较短的散列会增加破解密码的可能性（两个不同的文件生成相同的散列）。

警告：在加密密码或其他区分安全等级的数据时，至少应使用SHA256。人们认为在这些情况中使用MD5、SHA1是不安全的，MD5和SHA1仅适用于防止意外破解，而无法防止有预谋的篡改。

提示：SHA384的执行速度并不快于SHA512的执行速度，如果需要获取比SHA256更高的安全性，可以使用SHA512。

较长的SHA算法适用于密码加密，但是它们需要增强密码策略的强度以减弱字典攻击的威胁（字典

攻击是指攻击者通过对字典中的每个词应用散列算法，创建密码查询表的攻击策略）。通过增加密码散列的长度，即重复应用散列算法从而获得计算强度更高的字节序列，可以针对这种攻击为密码提供额外的保护。如果应用散列算法100次，字典攻击原来可以花1个月破解的密码，现在需要8年才能破解。Rfc2898DeriveBytes和PasswordDeriveBytes类可以准确地执行这类增加密码长度的任务。

对抗字典攻击的另一种技术是通过随机数字生成器获得最初的一长串字节，然后在应用散列算法前将其与每个密码合并。这样做可以通过两种方式对抗攻击者：需要花更长的时间计算散列结果，而且攻击者无法访问获得的字节。

Framework还提供了160位的RIPEMD散列算法，其安全性比SHA1稍好。但是，它会受到.NET低效实现的影响，这使得其执行速度比SHA512的执行速度更慢。

21.11 对称加密

对称加密在加密和解密时使用相同的密钥。Framework提供了4种对称加密算法，这些算法中Rijndael是最方便的。Rijndael既快速又安全，而且拥有两个实现：

- Rijndael类，从Framework 1.0之后的版本可以使用它。
- Aes类，它是在Framework 3.5中引入的。

除了Aes不允许通过更改块尺寸消弱密码外，这两个类几乎相同。Aes是CLR安全团队推荐使用的类。

Rijndael和Aes类允许对称密钥的长度为16、24或32字节，这些长度当前都被视为是安全的。下面显示了使用16字节的密钥将一系列字节写入文件时加密它们的方式：

```
byte[] key = {145,12,32,245,98,132,98,214,6,77,131,44,221,3,9,50};
byte[] iv = {15,122,132,5,93,198,44,31,9,39,241,49,250,188,80,7};

byte[] data = { 1, 2, 3, 4, 5 };  // 这是要加密的内容。

using (SymmetricAlgorithm algorithm = Aes.Create())
using (ICryptoTransform encryptor = algorithm.CreateEncryptor (key, iv))
using (Stream f = File.Create ("encrypted.bin"))
using (Stream c = new CryptoStream (f, encryptor, CryptoStreamMode.Write))
  c.Write (data, 0, data.Length);
```

下面的代码解密了该文件：

```
byte[] key = {145,12,32,245,98,132,98,214,6,77,131,44,221,3,9,50};
byte[] iv = {15,122,132,5,93,198,44,31,9,39,241,49,250,188,80,7};

byte[] decrypted = new byte[5];

using (SymmetricAlgorithm algorithm = Aes.Create())
using (ICryptoTransform decryptor = algorithm.CreateDecryptor (key, iv))
using (Stream f = File.OpenRead ("encrypted.bin"))
using (Stream c = new CryptoStream (f, decryptor, CryptoStreamMode.Read))
  for (int b; (b = c.ReadByte()) > -1;)
    Console.Write (b + " ");                                // 1 2 3 4 5
```

本范例生成了一个由16个字节随机组成的密钥。如果在解密过程中使用了错误的密钥，CryptoStream就会抛出CryptographicException。捕捉该异常是测试密钥是否正确的唯一途径。

在生成密钥的同时，本范例还生成了IV（Initialization Vector，初始化向量）。这个16字节的序列是密

码的一部分（这与密钥类似），但它不是保密的。在传输加密消息时，可以以普通文本方式发送IV（也许在消息的首部），然后在每条消息中更改它。这会使得无法根据上一条消息识别每条加密消息，即使这些消息的未加密版本是类似或相同的。

提示： 如果不需要或不想要IV的保护，可以使用密钥和IV都使用的相同的16字节值废除IV。但是，使用相同的IV发送多条信息会消弱密码的安全性并使破解可能性增加。

各个类使用不同的密码系统。Aes使用数据密码系统，通过encryptor和decryptor转换应用密码算法；CryptoStream使用数据流加密算法，用于数据流加密。可以使用不同的对称算法替换Aes，而仍旧需要使用CryptoStream。

CryptoStream是双向的，因此可以根据是选择CryptoStreamMode.Read还是CryptoStreamMode.Write，读取数据流或向数据流中写入信息。加密机和解密机都是对读和写的理解，这生成了4种组合，这些选择可能使人感到茫然！将读取创建为"拉"模型和将写入创建为"推"模型可以帮助理解。如果仍旧有疑问，可以将加密的写入和解密的读取作为起点；这通常是最常见的方式。

使用System.Cryptography中的RandomNumberGenerator可以生成随机密钥或IV。实际上它生成的数字是无法预测的或具有密码强度的（System.Random类没有提供相同的保证）。请看下面的范例：

```
byte[] key = new byte [16];
byte[] iv = new byte [16];
RandomNumberGenerator rand = RandomNumberGenerator.Create();
rand.GetBytes (key);
rand.GetBytes (iv);
```

如果不指定密钥和IV，RandomNumberGenerator会自动生成具有密码强度的随机值。可以通过Aes对象的Key和IV特性查询这些数字。

21.11.1 在内存中加密

使用MemoryStream完全可以在内存中进行加密和解密。下面列出了通过字节数组实现这些操作点的方法：

```
public static byte[] Encrypt (byte[] data, byte[] key, byte[] iv)
{
  using (Aes algorithm = Aes.Create())
  using (ICryptoTransform encryptor = algorithm.CreateEncryptor (key, iv))
    return Crypt (data, key, iv, encryptor);
}

public static byte[] Decrypt (byte[] data, byte[] key, byte[] iv)
{
  using (Aes algorithm = Aes.Create())
  using (ICryptoTransform decryptor = algorithm.CreateDecryptor (key, iv))
    return Crypt (data, key, iv, decryptor);
}

static byte[] Crypt (byte[] data, byte[] key, byte[] iv,
                     ICryptoTransform cryptor)
{
  MemoryStream m = new MemoryStream();
  using (Stream c = new CryptoStream (m, cryptor, CryptoStreamMode.Write))
```

```
        c.Write (data, 0, data.Length);
      return m.ToArray();
    }
```

在上例中CryptoStreamMode.Write对加密和解密都有重要作用，因为不论在哪种情况中都要将其“压入”新的内存流中。

下面是重载该方法接收和返回的字符串：

```
public static string Encrypt (string data, byte[] key, byte[] iv)
{
  return Convert.ToBase64String (
    Encrypt (Encoding.UTF8.GetBytes (data), key, iv));
}

public static string Decrypt (string data, byte[] key, byte[] iv)
{
  return Encoding.UTF8.GetString (
    Decrypt (Convert.FromBase64String (data), key, iv));
}
```

下面的代码演示了对这些字符串的使用：

```
byte[] kiv = new byte[16];
RandomNumberGenerator.Create().GetBytes (kiv);

string encrypted = Encrypt ("Yeah!", kiv, kiv);
Console.WriteLine (encrypted);                  // R1/5gYvcxyR2vzPjnT7yaQ==

string decrypted = Decrypt (encrypted, kiv, kiv);
Console.WriteLine (decrypted);                  // Yeah!
```

21.11.2 链接加密流

CryptoStream是一个链接器，它可以将其他流链接起来。下面的范例将压缩的加密文本写入文件中，然后再将它读出：

```
// 使用默认的密钥/IV进行演示。
using (Aes algorithm = Aes.Create())
{
  using (ICryptoTransform encryptor = algorithm.CreateEncryptor())
  using (Stream f = File.Create ("serious.bin"))
  using (Stream c = new CryptoStream (f,encryptor,CryptoStreamMode.Write))
  using (Stream d = new DeflateStream (c, CompressionMode.Compress))
  using (StreamWriter w = new StreamWriter (d))
    w.WriteLine ("Small and secure!");

  using (ICryptoTransform decryptor = algorithm.CreateDecryptor())
  using (Stream f = File.OpenRead ("serious.bin"))
  using (Stream c = new CryptoStream (f, decryptor, CryptoStreamMode.Read))
  using (Stream d = new DeflateStream (c, CompressionMode.Decompress))
  using (StreamReader r = new StreamReader (d))
    Console.WriteLine (r.ReadLine());                // 小并且安全!
}
```

（最后，我们将调用WriteLineAsync和ReadLineAsync，然后等待结果，从而使程序变成异步程序。）

在这个范例中，所有单个字符的变量都是链的一部分。加密机构——算法、加密机和解密机帮助CryptoStream进行加密工作，如图21-2显示。

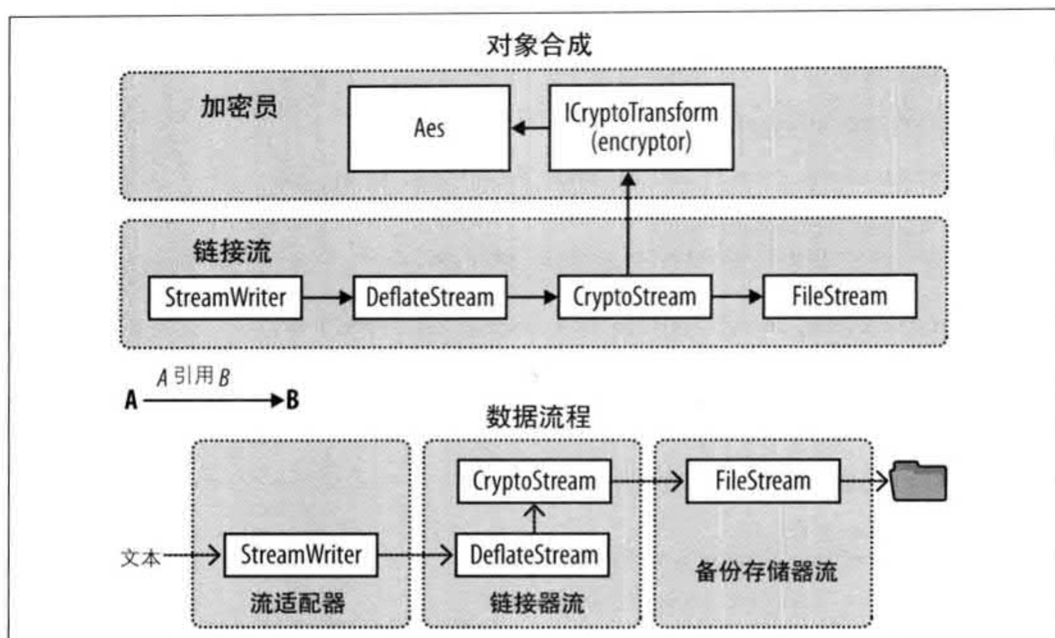


图21-2：链接加密和压缩流

不论最终流的大小如何，以上述方式链接的流都占用很少的内存。

提示： 嵌套多个using语句也是一种链接方法，可以使用下列代码构造一个链：

```
using (ICryptoTransform encryptor = algorithm.CreateEncryptor())
using
  (StreamWriter w = new StreamWriter (
    new DeflateStream (
      new CryptoStream (
        File.Create ("serious.bin"),
        encryptor,
      CryptoStreamMode.Write
    ),
    CompressionMode.Compress)
  )
)
```

然而，这个方法不如前面介绍的方法功能强，因为如果对象的构造方法（例如DeflateStream）中抛出了异常，那么任何已经实例化的对象（例如FileStream）都不会被布置。

21.11.3 处理加密对象

处理CryptoStream可以确保其内部缓存数据被基础流刷新。加密算法必须使用内部缓存，因为加密

算法以块为单位处理数据，而不是一次处理一个字节。

CryptoStream的Flush方法通常处于闲置状态。要刷新流（在不处理流的情况下），必须调用FlushFinalBlock。与Flush相反，FlushFinalBlock只能被调用一次，然后将无法再写入数据。

我们的范例还处理了Aes的数学算法和ICryptoTransform对象（加密机和解密机）。对于Rijndael转换实际上处理是可选的，因为Rijndael的实现是纯托管的。然而，处理还有一个重要作用：它会从内存中删除对称密钥和相关数据，防止计算机上运行的其他软件（后面将介绍的恶意软件）发现这些字节序列。不能依靠垃圾回收器完成这项工作，因为它仅会在可能的情况下在内存段中添加标志，而不会在每个字节前写入0。

除了使用using语句外，处理Aes对象的最简单方法是调用Clear。它的Dispose方法是通过显式实现隐藏的（以传递其不常用的布置语义）。

21.11.4 密钥管理

对加密密钥进行硬编码是不可取的，因为仅通过一点专门技术就可以使用流行的工具反编译程序集。一个更好的选择是为每次安装制作一个随机密钥，使用Windows数据保护安全地存储它（或者使用Windows数据保护加密整条消息）。如果要加密消息流，公共密钥加密仍是最佳选择。

21.12 公共密钥加密和签名

公共密钥加密是非对称的，因此加密和解密使用不同的密钥。

在对称加密过程中任何具有适当长度的字节序列都可以用作密钥，与对称加密不同，非对称加密需要专门制作的密钥对。密钥对包含公共密钥和私有密钥，它们一起完成下列工作：

- 公共密钥加密消息。
- 私有密钥解密消息。

制造密钥对的一方保留私有密钥的秘密，而可以自由地分发公共密钥。这种加密技术的特点是无法通过公共密钥计算出私有密钥。因此，如果丢失了私有密钥，就无法进行解密；相反，如果私有密钥被泄漏了，那么加密系统就无法起作用了。

公共密钥握手可以使两台计算机在公共网络中进行安全地通信，而且无需事先达成契约以及持有共享的密码。为了说明这种工作情况，假设计算机*Origin*想要向计算机*Target*发送一条机密信息：

1. *Target*生成一个公共/私有密钥对，然后将它的公共密钥发送给*Origin*。
2. *Origin*使用*Target*的公共密钥加密这条机密消息，然后将其发送给*Target*。
3. *Target*使用其私有密钥解密这条机密信息。

偷听者可以获取下列数据：

- *Target*的公共密钥
- 使用*Target*的公共密钥加密的机密信息

但是在没有*Target*的私有密钥的情况下，这条信息无法解密。

提示： 这样做无法阻止中间人攻击：换句话说，*Origin*无法知道*Target*是否是恶意攻击方。为了验证接收者，发送者需要先知道接收者的公共密钥，或者能够通过网站数字证书验证其密钥。

*Origin*发送给*Target*的机密信息通常含有用于以后对称加密的新密钥。这可以使以后的会话抛弃公共密钥加密，利用对称加密算法处理较大的消息。如果每次会话都使用新生成的公共/私有密钥对，那么这种协议会非常安全，因为无需在两端的计算机中存储密钥。

警告： 公共密钥加密算法的前提是消息比密钥小。这使得它们仅适于加密少量数据，如后续对称加密的密钥。如果尝试加密大于密钥尺寸一半的消息，那么加密程序就会抛出异常。

21.12.1 RSA类

.NET Framework提供了许多非对称算法，其中RSA是最流行的算法。下面显示了使用RSA进行加密的情况：

```
byte[] data = { 1, 2, 3, 4, 5 };   // 这是要加密的数据。

using (var rsa = new RSACryptoServiceProvider())
{
  byte[] encrypted = rsa.Encrypt (data, true);
  byte[] decrypted = rsa.Decrypt (encrypted, true);
}
```

因为我们没有指定公共或私有密钥，所以加密程序会使用默认长度为1,024的数据，自动生成一个密钥对；可以通过构造方法，以8字节的增量要求增加密钥的长度。对于关键的安全应用程序，它会谨慎地使用2,048位的数据：

```
var rsa = new RSACryptoServiceProvider (2048);
```

生成密钥对需要进行高强度的计算，可能会花100 ms。因此，在真正需要生成密钥（如调用Encrypt时）前，RSA实现会延迟该操作。这就提供加载已存在密钥（对）的机会（在有已存在密钥的情况下）。

ImportCspBlob和ExportCspBlob方法使用字节数组格式加载和保存密钥。FromXmlString和ToXmlString方法使用字符串格式加载和保存密钥，这些字符串中含有XML代码段。使用布尔型标志可以在保存时指明是否包括私有密钥。下面列出了制作密钥对并将它们保存到磁盘上的方法：

```
using (var rsa = new RSACryptoServiceProvider())
{
  File.WriteAllText ("PublicKeyOnly.xml", rsa.ToXmlString (false));
  File.WriteAllText ("PublicPrivate.xml", rsa.ToXmlString (true));
}
```

因为我们没有提供已存在的密钥，所以ToXmlString会强行制作新的密钥对（在第一次调用时）。下面的范例再次读取了这些密钥并使用它们加密和解密了一条信息：

```
byte[] data = Encoding.UTF8.GetBytes ("Message to encrypt");

string publicKeyOnly = File.ReadAllText ("PublicKeyOnly.xml");
string publicPrivate = File.ReadAllText ("PublicPrivate.xml");

byte[] encrypted, decrypted;
```

```
using (var rsaPublicOnly = new RSACryptoServiceProvider())
{
  rsaPublicOnly.FromXmlString (publicKeyOnly);
  encrypted = rsaPublicOnly.Encrypt (data, true);

  // 下一行代码会抛出一个异常，因为需要使用私有密钥解密：
  // decrypted = rsaPublicOnly.Decrypt (encrypted, true);
}

using (var rsaPublicPrivate = new RSACryptoServiceProvider())
{
  // 使用私有密钥可以成功解密：
  rsaPublicPrivate.FromXmlString (publicPrivate);
  decrypted = rsaPublicPrivate.Decrypt (encrypted, true);
}
```

安全

21.12.2 数字签名

公共密钥算法还可以用于消息或文档的数字签名。数字签名除了在制作时需要使用私有密钥因此无法伪造外，数字签名与散列类似。公共密钥用于验证数字签名。例如：

```
byte[] data = Encoding.UTF8.GetBytes ("Message to sign");
byte[] publicKey;
byte[] signature;
object hasher = SHA1.Create();         // 我们选择的散列算法。

// 生成新的密钥对，然后使用它为数据提供签名：
using (var publicPrivate = new RSACryptoServiceProvider())
{
  signature = publicPrivate.SignData (data, hasher);
  publicKey = publicPrivate.ExportCspBlob (false);    // 得到公共密钥
}

// 使用刚刚得到的公共密钥创建新的RSA，然后测试这个签名。
using (var publicOnly = new RSACryptoServiceProvider())
{
  publicOnly.ImportCspBlob (publicKey);
  Console.Write (publicOnly.VerifyData (data, hasher, signature)); // True

  // 处理数据，然后再次检查数字签名：
  data[0] = 0;
  Console.Write (publicOnly.VerifyData (data, hasher, signature)); // False

  // 当缺少私有密钥时，下列代码会抛出异常：
  signature = publicOnly.SignData (data, hasher);
}
```

第一次加密数据时签名会起作用，然后向合成的散列数据应用非对称算法。因为散列数据具有较小的固定尺寸，所以较大的文档添加签名的速度会较快（公共密钥加密的CPU工作强度比散列法的CPU工作强度高得多）。可以自己执行散列法，然后调用SignHash代替SignData：

```
using (var rsa = new RSACryptoServiceProvider())
{
  byte[] hash = SHA1.Create().ComputeHash (data);
  signature = rsa.SignHash (hash, CryptoConfig.MapNameToOID ("SHA1"));
  ...
}
```

SignHash仍旧需要知道使用了哪种散列算法；CryptoConfig.MapNameToOID通过友好的名称（如SHA1）使用正确的格式提供该信息。

RSACryptoServiceProvider生成的签名尺寸与使用的密钥尺寸相匹配。当前主要算法生成的安全签名都不会少于128字节（例如，适于生成激活码）。

提示： 为了更高效地添加数字签名，接收者必须知道并相信发送者的公共密钥。通过事先进行通信、预配置或颁发网站证书可以实现。网站证书是发送方公共密钥和名称的电子记录，它本身是由独立可信的权威机构签发的。命名空间System.Security.Cryptography.X509Certificates定义了证书运作的类型。

第22章 高级线程

第14章以任务和异步性作为先导知识，介绍了线程的基础知识。具体地说，前面已经介绍了如何启动/配置线程，并且涵盖了一些重要概念，如线程池、阻塞、自旋和同步上下文。此外，前面还介绍了锁和线程安全性，并且演示了最简单的信号发送结构：ManualResetEvent。

本章介绍前面未涉及的线程概念。前三节更深入地介绍同步、锁和线程安全性。

本章主要介绍：

- 非排他锁（信号机与读/写锁）
- 所有信号结构（AutoResetEvent、ManualResetEvent、CountdownEvent和Barrier）
- 延后初始化（Lazy<T>和LazyInitializer）
- 线程本地存储（ThreadStaticAttribute、ThreadLocal<T>和GetData/SetData）
- 抢占式线程方法（Interrupt、Abort、Suspend和Resume）
- 定时器

线程是一个很复杂的概念，所以我们还提供了一些线上学习资料，用于补充本章内容。请访问*http://albahari.com/threading/*，了解下面这些更复杂的概念：

- 用于发送特殊信号的Monitor.Wait和Monitor.Pulse
- 用于实现微优化的非阻塞同步技术（连锁、内存墙、不稳定性）
- 用于实现高并发的SpinLock和SpinWait

22.1 同步概述

同步（Synchronization）是指协调并发操作，实现可预测的结果。如果有多个线程访问相同的数据，那么同步就非常重要；这个应用领域很容易出现问题。

最简单和最实用的同步工具就是第14章介绍的延续和任务组合器。通过将并发程序统一为异步操作，再加上延续和组合器，就可以减少对锁和信号的依赖。然而，仍然有很多时候需要使用低层结构。

同步结构可以分成三类：

排他锁

排他锁结构只允许一个线程执行特定的活动，或者一次只允许执行一段代码。它们的主要目标是允许线程访问共享的写状态，但不会互相影响。排他锁结构包括lock、Mutex（和SpinLock）。

非排他锁

非排他锁只能实现有限的并发性。非排他锁结构包括Semaphore（Slim）和ReaderWriterLock（Slim）。

发送信号

这种方式允许线程保持阻塞，直至从其他线程接收到一个或多个通知。信号结构包括ManualResetEvent（Slim）、AutoResetEvent、CountdownEvent和Barrier。前三个就是所谓的事件等待处理器（event wait handles）。

在不使用锁的前提下，也可以通过使用非阻塞同步结构（nonblocking synchronization constructs）执行访问共享状态的特定并发操作（但是比较复杂）。这些结构包括Thread.MemoryBarrier、Thread.VolatileRead、Thread.VolatileWrite、volatile关键字和Interlocked类。我们提供了关于这些结构及Monitor的Wait/Pulse方法的线上资料，它们可用于编写自定义的信号发送逻辑。网站地址是：*http://albahari.com/threading/*。

22.2 排他锁

排他锁结构有三种：lock语句、Mutex和SpinLock。lock是最方便和最常用的结构

- Mutex可以跨越多个进程（计算机范围的锁）
- SpinLock实现了微优化，可以减少高度并发场景的上下文切换（参见*http://albahari.com/threading/*）

22.2.1 lock语句

下面这个类说明了必须使用锁的情况：

```
class ThreadUnsafe
{
  static int _val1 = 1, _val2 = 1;

  static void Go()
  {
    if (_val2 != 0) Console.WriteLine (_val1 / _val2);
    _val2 = 0;
  }
}
```

这个类不具有线程安全性：如果两个线程同时调用Go，则可能出现除零错误，因为一个线程可能将_val2设置为零，而另一个线程的执行过程可能正处于if语句和Console.WriteLine之间。下面是解决这个问题的方法：

```
class ThreadSafe
{
```

```
  static readonly object _locker = new object();
  static int _val1, _val2;

  static void Go()
  {
    lock (_locker)
    {
      if (_val2 != 0) Console.WriteLine (_val1 / _val2);
      _val2 = 0;
    }
  }
}
```

每次只有一个线程可以锁定同步的对象（这里是_locker），而其他竞争线程都会阻塞在这个位置，直至锁释放。如果有多个线程争夺这个锁，那么它们会在一个准备队列中排队，并且以先到先得的方式分配锁（注1）。排他锁有时候也称为对锁保护的对象添加序列化（serialized）访问控制，因为一个线程的访问不会与其他线程的访问重叠。在这个例子中，锁保护的是Go方法的逻辑，以及_val1和_val2域。

高级线程

22.2.2 Monitor.Enter和Monitor.Exit

事实上，C#的lock语句是Monitor.Enter和Monitor.Exit方法调用及try/finally语句块的简写语法。下面是前一个例子中Go方法（简化后）的实际操作：

```
Monitor.Enter (_locker);
try
{
  if (_val2 != 0) Console.WriteLine (_val1 / _val2);
  _val2 = 0;
}
finally { Monitor.Exit (_locker); }
```

如果未先调用同一个对象的Monitor.Enter，而直接调用Monitor.Exit，就会抛出异常。

lockTaken重载

前面演示的代码正是C# 1.0、2.0和3.0编译器转换lock语句得到的结果。

然而，这段代码有一个不易发现的漏洞。假设在方法调用和语句块之间抛出了异常（或许是线程调用了Abort，或者抛出了OutOfMemoryException异常——不过通常不可能发生），在这种情况中，锁不一定会被获得。如果锁已经获得，那么它也无法再释放——因为执行过程无法再进入try/finally语句块，这样就会导致锁泄漏。为了避免这个问题，CLR 4.0设计人员为Monitor.Enter添加以下重载方法：

```
public static void Enter (object obj, ref bool lockTaken);
```

当且仅当Enter方法抛出异常，并且锁未获得时，在这个方法执行之后lockTaken才可能是false。

下面是更完善的用法（这正是C# 4.0和C# 5.0转换lock语句的方式）：

```
bool lockTaken = false;
try
```

注1：Windows与CLR行为的细微区别意味着有时候会违反队列公平性。

```
{
  Monitor.Enter (_locker, ref lockTaken);
  // 执行其他代码...
}
finally { if (lockTaken) Monitor.Exit (_locker); }
```

2. TryEnter

Monitor也提供了一个TryEnter方法，它允许指定一个超时时间，可以是以毫秒为单位的时间值，也可以作为一个TimeSpan对象。然后，当获得锁时，这个方法会返回true；当因为方法超时而无法获得锁时，它就会返回false。TryEnter还可以不带参数直接调用，这样的作用是测试锁，如果锁无法马上获得，调用过程会立刻超时。与Enter方法一样，CLR 4.0也重载了TryEnter，可以接受参数lockTaken。

22.2.3 选择同步对象

各个参与的线程都可见的任意对象都可以作为一个同步对象，但是这里有一个硬性规定：它必须是引用类型。同步对象一般是私有的（因为这样有利于封装锁逻辑），并且通常是一个实例或静态域。同步对象可以就是所保护的对象，如下面例子的_list域：

```
class ThreadSafe
{
  List <string> _list = new List <string>();

  void Test()
  {
    lock (_list)
    {
      _list.Add ("Item 1");
      ...
```

作为锁的域（如前面例子的_locker）可以精确控制锁的范围和粒度。所包含的对象（this）或者它的类型都可以作为一个同步对象：

```
lock (this) { ... }
```

或者：

```
lock (typeof (Widget)) { ... }    // 保护静态域的方法
```

这种方式的锁有一个缺点：无法封装锁逻辑，因此更难避免死锁和排他锁。在一种类型上添加锁，还可能会破坏应用域的边界（在同一个进程中），参见第24章。

此外，使用lambda表达式或匿名方法，可以在局部变量上添加锁。

提示： 锁机制不会限制同步对象本身的访问方式。换而言之，x.ToString()不会阻塞，因为另一个线程已经调用lock(x)；这两个线程都必须调用lock(x)，才能够实现阻塞。

22.2.4 何时添加锁

基本原则是：为访问任意可写共享域的代码添加锁。即使是最简单的情况（如某个域的赋值操作），

也必须考虑同步问题。在下面的类中，Increment和Assign方法都不具有线程安全性：

```
class ThreadUnsafe
{
  static int _x;
  static void Increment() { _x++; }
  static void Assign()    { _x = 123; }
}
```

下面是Increment和Assign方法的线程安全版本：

```
static readonly object _locker = new object();
static int _x;

static void Increment() { lock (_locker) _x++; }
static void Assign()    { lock (_locker) _x = 123; }
```


如果不使用锁，则可能出现两个问题：

- 诸如变量增值等操作或者特定条件下的变量读/写操作，并不是原子操作。
- 为了提高性能，编译器、CLR和处理器都可能重新调整指令和缓存CPU寄存器的变量——只要这种优化不会改变单线程程序或者使用锁的多线程程序的行为。

使用锁可以解决第二个问题，因为它会在锁的前后创建内存屏障（memory barrier）。内存屏障是这些操作的“围栏”，它能防止跨线程的重排和缓存。

提示：这个方法不仅仅适用于锁，也适用于所有同步结构。例如，如果使用信号发送结构保证每次只有一个线程读/写一个变量，则不需要使用锁。因此，下面的代码并没有在x上添加锁，但是仍然具有线程安全性：

```
var signal = new ManualResetEvent (false);
int x = 0;
new Thread (() => { x++; signal.Set(); }).Start();
signal.WaitOne();
Console.WriteLine (x);    // 1（总是）
```

在*http://albahari.com/threading*的“非阻塞同步性”中，我们解释了为什么需要这种操作，以及内存屏障和Interlocked类为什么可以代替这些情况的锁操作。

22.2.5 锁与原子性

如果有一组变量总是在同一个锁中读写，那么可以认为这些变量是以原子方式进行读写。假设域x和y总是在对象locker的锁中读写：

```
lock (locker) { if (x != 0) y /= x; }
```

之所以说x和y以原子方式进行读写，是因为这段代码不可分割，也不可能被其他线程的操作占用，如修改x或y，或者破坏它的输出结果。这里不可能出现除零错误，因为x或y总是在相同的排他锁中访问。

警告：如果lock语句块中抛出异常，则可能破坏通过锁实现的原子操作。例如，假设有以下代码：

```
decimal _savingsBalance, _checkBalance;

void Transfer (decimal amount)
{
  lock (_locker)
  {
    _savingsBalance += amount;
    _checkBalance -= amount + GetBankFee();
  }
}
```

如果GetBankFee()抛出异常，那么银行就会丢失金钱。在这种情况中，提早调用GetBankFee，就可以避免这个问题。更复杂的解决方法是在catch或finally语句块中实现回滚操作。

指令级（Instruction）原子操作与原子操作是一个既相似又不同的概念：只有当指令以不可分割方式在底层处理器上执行时，它才是原子操作。

22.2.6 嵌套锁

线程可以用嵌套（重入）的方式重复锁住同一个对象：

```
lock (locker)
  lock (locker)
    lock (locker)
    {
      // 执行一些操作...
    }
```

或者：

```
Monitor.Enter (locker); Monitor.Enter (locker); Monitor.Enter (locker);
// 执行一些操作...
Monitor.Exit (locker); Monitor.Exit (locker); Monitor.Exit (locker);
```

在这些情况中，只有当最外层lock语句退出时，或者执行相同数量的Monitor.Exit语句，对象才会解除锁。

在一个锁中，如果一个方法调用另一个方法，则适合使用嵌套锁：

```
static readonly object _locker = new object();

static void Main()
{
  lock (_locker)
  {
    AnotherMethod();
    // 这里仍然获得锁，因为锁重入。
  }
}

static void AnotherMethod()
{
  lock (_locker) { Console.WriteLine ("Another method"); }
}
```

线程只能阻塞在第一个（最外层）锁上。

22.2.7 死锁

如果两个线程互相等待对方所占用的资源，就会形成死锁，使得双方都无法继续执行。演示这个问题的最简单方法是使用两个锁，执行以下操作：

```
object locker1 = new object();
object locker2 = new object();

new Thread (() => {
                    lock (locker1)
                    {
                      Thread.Sleep (1000);
                      lock (locker2);      // 死锁
                    }
                    }).Start();

lock (locker2)
{
  Thread.Sleep (1000);
  lock (locker1);                          // 死锁
}
```

使用三个或更多的线程，可能形成更复杂的死锁链。

警告： 在标准托管环境中，CLR与SQL Server不同，它不会自动检查和处理死锁（通过中止其中一个争夺线程）。除非指定锁的超时时间，否则线程死锁会使相关线程永远阻塞。（然而，SQL CLR整合主机会自动检测死锁，然后在其中一个线程中抛出可捕捉的异常。）

死锁是多线程中最难解决的问题——特别是其中涉及许多相关对象时。基本上，最难的问题是无法确定调用获取了哪些锁。

所以，代码可能锁住了类x中的一个私有域，但是不知道调用者（或者调用者的调用者）已经锁住了类y中的域b。同时，有另一个域执行相反操作，从而形成死锁。但是，面向对象设计模式会加剧这个问题，因为这些设计模式会创建只有在运行时才能确定的调用链。

最常见的建议是：通过一致的方式添加对象锁，避免出现死锁，虽然这适用于最前面的例子，但是这条建议很难应用于刚刚介绍的情况。更好的方法是避免在可能引用回本身对象的调用方法上添加锁。此外，要确认是否真正有必要在其他类的调用方法上添加锁（像“线程安全性”中介绍的一样，这种用法很常见，但是有时候它们可以采用其他方法实现）。由于更多地使用一些更高级的同步方法，如任务延续/组合器，所以数据并行性和不可变类型（本章后面会介绍）会减少对锁的使用。

提示： 还有一个方法可以发现这个问题：在获得锁时调用其他代码，锁的封装性就可能会泄漏。这并不是CLR或.NET框架的问题，而是一般锁都存在的基本问题。有许多研究性项目专门致力于解决锁的问题，其中包括软件事务级内存（Software Transactional Memory）。

当占有锁时调用Dispatcher.Invoke（在WPF应用程序中）或Control.Invoke（在Windows窗体应用程序中），就可能出现另一种死锁问题。如果UI运行在另一个等待相同锁的方法上，那么就会马上

发生死锁。解决这个问题，通常可以使用BeginInvoke代替Invoke（或者使用异步函数，它们会在出现的同步上下文中隐式执行相同的操作）。此外，也可以在调用Invoke之前释放所获得的锁，但是这种方法不适用于调用者本身占有锁的情况。

22.2.8 性能

锁的执行速度很快：在目前的计算机上，如果未出现争夺者，那么一般可以在80纳秒内获得和释放一个锁；如果出现争夺者，那么相应的上下文切换会将过载增加到毫秒级，但是这个时间远远小于线程的实际调度时间。

Mutex

Mutex类似于C#的锁，但是它可以支持多个进程。换而言之，Mutex可用于计算机范围或应用程序范围。

提示： 获得和释放一个无争夺的Mutex只需要几毫秒——时间比锁操作慢50倍。

使用一个Mutex类，就可以调用WaitOne方法获得锁，或者调用ReleaseMutex释放锁。关闭或去掉一个Mutex会自动释放互斥锁。与lock语句一样，Mutex只能在它所在的线程上释放。

跨进程Mutex的一个常见应用是保证每次只能运行一个程序实例。下面是实现方法：

```
class OneAtATimePlease
{
  static void Main()
  {
    // 给Mutex命名，使之在整个计算机范围有效。
    // 这个名称应该在公司和应用程序中保持唯一（例如，包含URL）。

    using (var mutex = new Mutex (false, "oreilly.com OneAtATimeDemo"))
    {
      // 如果遇到抢占，如程序的另一个实例仍在关闭过程中，则等待几秒钟时间。

      if (!mutex.WaitOne (TimeSpan.FromSeconds (3), false))
      {
        Console.WriteLine ("Another instance of the app is running. Bye!");
        return;
      }
      RunProgram();
    }
  }

  static void RunProgram()
  {
    Console.WriteLine ("Running. Press Enter to exit");
    Console.ReadLine();
  }
}
```

提示： 如果运行在终端服务下，计算机范围的Mutex通常只能由同一个终端服务器会话的应用程序访问。为了让所有终端服务器会话都可以访问它，需要在它的名称前面添加前缀*Global*。

22.3 锁与线程安全性

如果程序或方法可以在任意多线程场景中正常运行，那么它就具有线程安全性。线程安全性主要是通过锁和减少线程交互可能性而实现。

通用类型很少完全实现线程安全性，因为：

- 开发完整的线程安全性难度很大，特别是当一个类型拥有许多域时（每一个域都可能在任意多线程环境中出现交互）。
- 线程安全性可能增加性能开销（在一定程度上，开销与该类型是否被多个线程使用有关）。
- 具有线程安全性的类型不一定能够保证程序使用它的方式具有线程安全性，而且实现后者安全性的代价要高于前者。

因此，线程安全性通常只在需要的地方实现，用于处理特定的多线程场景。

然而，有一些方法可以"作弊"，让一些大型复杂类在多线程环境上安全执行。其中一种是通过将大量代码（即使是访问整个对象）封装在一个排他锁中，以牺牲粒度来保证高级序列化访问。事实上，如果想要在多线程环境中使用不具线程安全性的第三方代码（或者相关的大部分Framework类型），那么一定需要采用这种策略。这里的方法是使用相同的排他锁保护不具线程安全性的对象中所有属性、方法和域的访问。这种方法适用于对象方法都能够快速执行的情况（否则可能存在大量阻塞）。

警告：除了基本数据类型，.NET框架类型在初始化时都具有线程安全性，其中包括并发只读访问。开发者的责任是保证线程安全性（通常使用排他锁实现，第23章将介绍的`System.Collections.Concurrent`集合类型除外）。

另一个作弊方法是通过减少共享数据来减少线程交互。这是一种很好的方法，而且通常隐含用在无状态的中间层应用程序和网页服务器中。由于多个客户端请求可能同时到达，所以它们调用的服务器方法必须具有线程安全性。无状态设计（主要出于可伸缩性考虑）内在限制了交互可能性，因为这些类不会保存请求之间的数据。然后，线程交互仅限于线程中可能创建的静态域，如用于在内存中缓存常用数据，以及提供一些基础架构服务，如身份验证和审计。

另外一种解决方法（通常在富客户端应用程序中）是在UI线程上运行访问共享状态的代码。正如第14章所介绍，异步函数可以简化这种实现。

最后一种实现线程安全性的方法是使用自动锁机制。如果扩展`ContextBoundObject`，然后在类中应用`Synchronization`属性，那么.NET框架就会使用这种机制。然后，只要调用这个对象的方法或属性，该方法或属性的执行过程就会自动添加一个对象范围的锁。虽然这种方法减小了实现线程安全性的难度，但是它也有其自身的问题：可能会出现死锁、破坏并发性和意外重入。因此，手动锁通常是更好的选择——至少在出现合理自动锁机制之前是这样的。

22.3.1 线程安全性和.NET框架类型

使用锁可以将不具有线程安全性的代码转换为具有线程安全性的代码。.NET框架就是其中一种很好的应用：几乎所有非基本数据类型在实例化之后都不具有线程安全性（除了只读访问），但是如果使用锁保护特定对象的全部访问，它们也可以用在多线程代码中。在下面这个例子中，两个线程同时给同一个`List`集合添加项目，然后再列举这个列表：

```
class ThreadSafe
{
  static List <string> _list = new List <string>();

  static void Main()
  {
    new Thread (AddItem).Start();
    new Thread (AddItem).Start();
  }

  static void AddItem()
  {
    lock (_list) _list.Add ("Item " + _list.Count);

    string[] items;
    lock (_list) items = _list.ToArray();
    foreach (string s in items) Console.WriteLine (s);
  }
}
```

这个例子在`_list`对象上添加锁。如果有两个相关联的链表，则必须选择在一个通用对象添加锁（我们可以指定其中一个链表，或者更好的方法是使用一个独立的域）。

此外，.NET的集合遍历也不是线程安全操作，因为在遍历过程中，修改链表就会抛出一个异常。这个例子没有锁住整个遍历过程，而是将集合中的项目复制到一个数组中，这样就不需要在非常耗费时间的遍历过程中添加排他锁。（另一种方法是使用读/写锁；具体参见第22.4.2节“读/写锁”。）

1. 使用锁访问线程安全对象

有时候，我们还需要在访问线程安全对象的代码上添加锁。例如，Framework的`List`类是具有线程安全性的，我们需要在链表上添加一个项目：

```
if (!_list.Contains (newItem)) _list.Add (newItem);
```

无论链表本身是否具有线程安全性，这条语句肯定不具有线程安全性！整个`if`语句必须封装在一个锁中，才能防止判断包含关系和添加新项目的操作之间插入其他操作。然后，任何修改链表的代码都必须添加相同的锁。例如，下面的语句也需要封装在同一个锁中：

```
_list.Clear();
```

从而保证它不会在前面的语句之前执行。换而言之，必须为不具有线程安全性的集合类添加锁（使List类的假想线程安全变成多余的）。

提示： 在集合访问代码上添加锁可能会给高度并发环境增加过多的阻塞时间。为此，Framework 4.0提供了一个具有线程安全性的队列、堆栈和字典，它们将在第23章中介绍。

---

2. 静态成员

将对象的访问封装在自定义的锁语句中，仅适用于所有并发线程均能访问和使用锁的情况。如果对象范围较宽，那么情况可能就不一样。最坏的情况是设置为公共类型的静态成员。例如，如果`DateTime`结构体的静态属性`DateTime.Now`不具有线程安全性，而又有两个并发调用修改输出或抛出异常。那么使用外部锁的唯一解决方法是在调用之前在类型本身上添加锁——`lock(typeof(DateTime))`。

只有所有程序员都认可（这通常是不可能的），才能够使用这种方法。而且，为类型本身添加锁也存在一些问题。

因此，在使用DateTime结构体的静态成员实现线程安全性时，必须非常小心。这是整个.NET框架常用的模式：静态成员具有线程安全性；但是实例成员则不具有线程安全性。在编写公共访问的类型时，采用这种模式也很有意义，它可以防止出现无法创建线程安全性的问题。换言之，通过静态方法实现线程安全性，程序本身不会影响使用该类型的代码实现线程安全性。

提示：静态方法的线程安全性是必须显式编码的部分，因为静态方法本身无法自动实现！

---

3. 只读线程安全性

实现类型并发只读访问的线程安全性（这是可能的）是很有用的，因为这意味着使用者可以避免过多的锁。许多.NET框架类型都采用这种原则，例如，集合的并发读取操作都具有线程安全性。

采用这种原则也很简单：如果一个类型的并发只读访问只有线程安全性，则不要在使用者认为是只读的一些方法中添加域（或添加锁）实现相同效果。例如，在集合中实现一个ToArray()方法时，可能需要先压缩集合的内部结构。然而，这样做可能会破坏它的线程安全性，因为原本使用者只认为它是只读的。

只读线程安全性也是将枚举器与可枚举集合分离的原因之一：两个线程可以同时列举一个集合，因为它们分别使用独立的枚举器对象。

提示：如果没有文档，那么不要随意假定某些方法的只读性。Random类就是一个很好的例子：当调用Random.Next()时，它的内部实现会更新私有的种子值。因此，必须在使用Random类的代码上添加锁，或者在每一个线程上使用独立的实例。

22.3.2 应用服务器的线程安全性

应用服务器也需要实现多线程，才能并发处理多个客户端请求。WCF、ASP.NET和Web Services应用程序都隐含采用多线程实现；使用网络通道（如TCP或HTTP）的远程服务器应用程序也一样。这意味着，在编写服务器端代码时，如果处理客户端请求的线程可能发生交互，则一定要注意线程安全性。幸好，这种可能性非常低：典型的服务器类或者是无状态的（没有域），或者拥有一个激活模型，它可以为每一个客户端或请求创建独立的对象实例。线程交互通常只能通过静态域实现，它们有时候用于在内存中缓存部分数据库，从而提高性能。

例如，假设有一个查询数据库的RetrieveUser方法：

```
// 用户是一个使用域保存用户数据的自定义类
internal User RetrieveUser (int id) { ... }
```

如果这个方法频繁调用，那么将结果缓存在一个静态Dictionary中，就可以提高性能。下面是一种线程安全性的实现方法：

```
static class UserCache
{
  static Dictionary <int, User> _users = new Dictionary <int, User>();
```

```
  internal static User GetUser (int id)
  {
    User u = null;

    lock (_users)
    if (_users.TryGetValue (id, out u))
      return u;

    u = RetrieveUser (id);          // 查询数据库的方法
    lock (_users) _users [id] = u;
    return u;
  }
}
```

这里至少需要在读取和更新字典的代码上添加锁，才能保证线程安全性。这个例子在锁的简单性和性能上选择了折衷的方法。我们的设计实际上会对效率产生一定的影响：如果两个线程同时使用之前未查询到的id来调用这个方法，那么RetrieveUser方法可能会执行两次——因此字典会多更新一次。在整个方法上添加锁可以解决这个问题，但是可能会对效率造成更大影响：在调用RetrieveUser过程中，整个缓存被锁住，因此其他线程就可能被阻塞住，无法查询任何用户数据。

22.3.3 不可变对象

不可变对象是指状态不会发生变化的对象——包括外部或内部。不可变对象的域通常都声明为只读，而且全部构造方法初始化。

不可变性是函数式编程的特性——它不会修改对象，而是创建一个带有不同属性的新对象。LINQ采用这个编程范式。不可变性也适合用在多线程中，它可以避免共享写状态的访问问题——通过去除（或减少）可写性。

有一种模式是使用不可变对象封装一组相关域，减少锁的占用时间。下面是一个非常简单的例子，其中包含两个域：

```
int _percentComplete;
string _statusMessage;
```

我们希望实现它们的原子读/写操作。我们不在这些域上添加锁，而是定义了下面两个不可变类：

```
class ProgressStatus    // 表示某个进程的进度
{
  public readonly int PercentComplete;
  public readonly string StatusMessage;

  // 这个可能还有其他的域...

  public ProgressStatus (int percentComplete, string statusMessage)
  {
    PercentComplete = percentComplete;
    StatusMessage = statusMessage;
  }
}
```

然后，使用该类型定义一个域，以及一个锁对象：

```
readonly object _statusLocker = new object();
ProgressStatus _status;
```

现在，只需要在一个赋值语句上添加锁，就可以读/写该类型的值：

```
var status = new ProgressStatus (50, "Working on it");
// Imagine we were assigning many more fields...
// ...
lock (_statusLocker) _status = status;    // Very brief lock
```

为了读取这个对象的值，要先（在锁中）复制该对象。然后，不需要获取锁，就可以读取它的值：

```
ProgressStatus status;
lock (_statusLocker) status = _status;    // 同样，一个简短的锁
int pc = status.PercentComplete;
string msg = status.StatusMessage;
...
```

22.4 非排他锁

22.4.1 信号量

信号量就像是俱乐部：它有特定的容量，还有保镖保护。一旦满员之后，便不允许人再进入，人们只能在外面排队。然后，每当有人离开，队伍排头的人就可以进入。构造函数至少需要传入2个参数：俱乐部当前剩余位置数量和俱乐部的总容量。

容量为1的信号量类似于Mutex或lock，唯一不同的是信号量没有拥有者——它与线程无关。任何线程都可以调用Semaphore的Release，而对于Mutex和lock，只有获得锁的线程才可以释放锁。

提示：这个类有两个功能相似的版本：Semaphore和SemaphoreSlim。后者是在Framework 4.0引入的，它进行了一些优化，以满足并行编程的低延迟要求。此外，它也适用于传统多线程编程，因为它可以在等待时指定一个取消令牌（参见第14.6.1节“取消”）。然而，它不适用于进程间通信。

Semaphore在调用WaitOne或Release时需要消耗约1毫秒时间；SemaphoreSlim的延迟时间则只有前者的1/4。

信号量可用限制并发处理，防止太多线程同时执行特定的代码。在下面的例子，有5个线程试图进入一次只允许3个线程进入的俱乐部：

```
class TheClub      // 只有一个大门！
{
  static SemaphoreSlim _sem = new SemaphoreSlim (3);    // 容量为3

  static void Main()
  {
    for (int i = 1; i <= 5; i++) new Thread (Enter).Start (i);
  }

  static void Enter (object id)
  {
    Console.WriteLine (id + " wants to enter");
    _sem.Wait();
    Console.WriteLine (id + " is in!");       // 只有3个线程
    Thread.Sleep (1000 * (int) id);           // 可以同时访问
    Console.WriteLine (id + " is leaving");
```

```
      _sem.Release();
    }
  }
```

```
1 wants to enter
1 is in!
2 wants to enter
2 is in!
3 wants to enter
3 is in!
4 wants to enter
5 wants to enter
1 is leaving
4 is in!
2 is leaving
5 is in!
```

命名Semaphore可以像Mutex一样跨越多个进程。

22.4.2 读/写锁

经常，一个类型的实例的读操作是线程安全的，但是并发更新则不是线程安全的（并发读写操作也不是线程安全的）。一些资源访问也有相同的特点，如文件。虽然可以使用各种访问模式的简单排他锁保护这些类型的实例，但是如果读操作很多而写操作很少，那么限制并发访问就是不合理的。这种情况可能发生在业务应用服务器，它会将常用数据缓存在静态域中，用以加快访问速度。使用ReaderWriterLockSlim类，可以在这种情况中实现锁的最大可用性。

提示： ReaderWriterLockSlim是在Framework 3.5引入的，目的是代替旧的ReaderWriterLock类。后者具有相似的功能，但是执行速度要慢几倍，而且本身存在一些锁升级处理的设计缺陷。

与常规锁（Monitor.Enter/Exit）相比，ReaderWriterLockSlim的执行速度仍然要慢一倍。

使用这两个类，可以实现2种基本锁——读锁和写锁：

- 写锁是全局排他锁。
- 读锁兼容其他的读锁。

所以，获得写锁的线程会阻塞其他试图获得读锁或写锁的线程（反之则相反）。但是，如果没有线程获得写锁，那么任意数量的线程可以同时获得读锁。

ReaderWriterLockSlim定义了以下几个可用于获取和释放读/写锁的方法：

```
public void EnterReadLock();
public void ExitReadLock();
public void EnterWriteLock();
public void ExitWriteLock();
```

此外，所有Enter*XXX*方法都有相应的Try*XXX*，它们可以接受Monitor.TryEnter风格的超时参数（如果资源争夺情况严重，那么很容易出现超时情况）。ReaderWriterLock也提供了相似的方法，即Acquire*XXX*和Release*XXX*。如果发生超时，那么这些方法会抛出异常，而不会返回false。

下面的程序演示了ReaderWriterLockSlim。其中有3个线程不停地列举链表，同时另外2个线程每隔

1秒钟在链表后面附加1个随机数。代码使用1个读锁保护链表的读操作，同时使用1个写锁保护链表的写操作：

```
class SlimDemo
{
  static ReaderWriterLockSlim _rw = new ReaderWriterLockSlim();
  static List<int> _items = new List<int>();
  static Random _rand = new Random();

  static void Main()
  {
    new Thread (Read).Start();
    new Thread (Read).Start();
    new Thread (Read).Start();

    new Thread (Write).Start ("A");
    new Thread (Write).Start ("B");
  }

  static void Read()
  {
    while (true)
    {
      _rw.EnterReadLock();
      foreach (int i in _items) Thread.Sleep (10);
      _rw.ExitReadLock();
    }
  }

  static void Write (object threadID)
  {
    while (true)
    {
      int newNumber = GetRandNum (100);
      _rw.EnterWriteLock();
      _items.Add (newNumber);
      _rw.ExitWriteLock();
      Console.WriteLine ("Thread " + threadID + " added " + newNumber);
      Thread.Sleep (100);
    }
  }

  static int GetRandNum (int max) { lock (_rand) return _rand.Next(max); }
}
```

提示：在生产代码中，我们通常需要添加try/finally语句块，保证异常抛出时也能够释放锁。

下面是执行结果：

```
Thread B added 61
Thread A added 83
Thread B added 55
Thread A added 33
...
```

ReaderWriterLockSlim允许的并发Read活动比普通锁多。在Write方法的while循环开头插入下面

的代码，就可以说明这一点：

```
Console.WriteLine (_rw.CurrentReadCount + " concurrent readers");
```

这样的执行结果总是打印出“3 concurrent readers”（Read方法将大部分执行时间消耗在foreach循环中）。和CurrentReadCount一样，ReaderWriterLockSlim也提供了以下监控锁的属性：

```
public bool IsReadLockHeld                  { get; }
public bool IsUpgradeableReadLockHeld       { get; }
public bool IsWriteLockHeld                 { get; }

public int WaitingReadCount                 { get; }
public int WaitingUpgradeCount              { get; }
public int WaitingWriteCount                { get; }

public int RecursiveReadCount               { get; }
public int RecursiveUpgradeCount            { get; }
public int RecursiveWriteCount              { get; }
```

1. 可升级锁

有时候，最好将读锁封装在一个原子操作中。例如，当项目不在链表时将它添加到链表中。理想情况下，我们希望减小用于保持（排他）写锁的时间，所以可以采用以下操作步骤：

1. 获取一个读锁。
2. 判断该项目是否已经存在于链表中，如果已存在，则释放锁并返回。
3. 释放读锁。
4. 获取一个写锁。
5. 添加该项目。

问题是，另一个线程可能会在第3步和第4步之间插入并修改链表（例如，添加同一个项目）。ReaderWriterLockSlim可以通过第三种锁来解决这个问题，即可升级锁（upgradeable lock）。可升级锁与读锁类似，唯一不同的是它可以在将来升级为原子操作的写锁。

下面是使用可升级锁的步骤：

1. 调用EnterUpgradeableReadLock。
2. 执行基于读的活动（例如，判断该项目是否已经存在于链表中）。
3. 调用EnterWriteLock（将可升级锁转换为写锁）。
4. 执行基于写的活动（例如，将该项目添加到链表中）。
5. 调用ExitWriteLock（将写锁转换回可升级锁）。
6. 执行其他基于读的活动。
7. 调用ExitUpgradeableReadLock。

从调用者角度看，它与嵌套锁或递归锁很相似。但是，在功能上，第3步ReaderWriterLockSlim以原子方式释放读锁，然后获得一个新的写锁。

可升级锁和读锁还有另一个重要区别：虽然可升级锁可以与任意多个读锁共存，但是一次只能获取一

个可升级锁。这样可以将争夺转换序列化，从而防止转换死锁——和SQL Server的更新锁一样。

SQL Server	ReaderWriterLockSlim
共享锁Share lock	读锁Read lock
排他锁Exclusive lock	写锁Write lock
更新锁Update lock	可升级锁 Upgradeable lock

修改前一个例子的Write方法，就可以演示可升级锁。

```
while (true)
{
  int newNumber = GetRandNum (100);
  _rw.EnterUpgradeableReadLock();
  if (!_items.Contains (newNumber))
  {
    _rw.EnterWriteLock();
    _items.Add (newNumber);
    _rw.ExitWriteLock();
    Console.WriteLine ("Thread " + threadID + " added " + newNumber);
  }
  _rw.ExitUpgradeableReadLock();
  Thread.Sleep (100);
}
```

提示：ReaderWriterLock也可以执行锁转换；但是并不可靠，因为它不支持可升级锁。这正是ReaderWriterLockSlim设计人员重新编写这个新类的原因。

2. 锁递归

通常，ReaderWriterLockSlim禁止使用嵌套锁或递归锁。因此，下面的操作会抛出异常：

```
var rw = new ReaderWriterLockSlim();
rw.EnterReadLock();
rw.EnterReadLock();
rw.ExitReadLock();
rw.ExitReadLock();
```

然而，如果按照以下方式创建ReaderWriterLockSlim，则代码可以正常执行：

```
var rw = new ReaderWriterLockSlim (LockRecursionPolicy.SupportsRecursion);
```

这样可以保证，只有在需要时才会出现递归锁。递归锁可能会增加代码复杂性，因为它可能同时获得多种锁：

```
rw.EnterWriteLock();
rw.EnterReadLock();
Console.WriteLine (rw.IsReadLockHeld);     // True
Console.WriteLine (rw.IsWriteLockHeld);    // True
rw.ExitReadLock();
rw.ExitWriteLock();
```

基本的原则是，一旦获得了锁，后续的递归锁可以减少，但不能增多，等级变化如下：

读锁→可升级锁→写锁

然而，将可升级锁提升为写锁的请求总是合法的。

22.5 使用事件等待处理器发送信号

最简单的信号发送结构是事件等待处理器（event wait handles，与C#事件无关）。事件等待处理器有3种实现：AutoResetEvent、ManualResetEvent(Slim)和CountdownEvent。前两个基于通用的EventWaitHandle类，它们从基类继承了所有的功能。

22.5.1 AutoResetEvent

AutoResetEvent就像验票闸门：插入一张票据，则只允许一个人通过。类名中的“Auto”是指人通过之后打开的闸门会自动关闭或重置。通过调用WaitOne，线程可以在闸门前等待或阻塞（在“一个”闸门前等待，直至闸门开启）；调用Set方法，则插入一张票据。如果有多个线程调用WaitOne，那么闸门前面会排起一个队列（注1）。票据可以来自任意线程；换言之，访问AutoResetEvent对象的任意（非阻塞）线程都可以调用自己的Set方法，释放一个阻塞的线程。

有两种方法可以创建AutoResetEvent。第一种方法是使用它的构造方法：

```
var auto = new AutoResetEvent (false);
```

（在构造方法中传入true，相当于马上调用它的Set。）第二种方法是按照以下方式创建一个AutoResetEvent：

```
var auto = new EventWaitHandle (false, EventResetMode.AutoReset);
```

在下面的例子中，线程启动后，它的任务就开始等待，直至另一个线程发送信号（参见图22-1）：

```
class BasicWaitHandle
{
  static EventWaitHandle _waitHandle = new AutoResetEvent (false);

  static void Main()
  {
    new Thread (Waiter).Start();
    Thread.Sleep (1000);                // 暂停1秒钟...
    _waitHandle.Set();                  // 唤醒等待者Waiter
  }

  static void Waiter()
  {
    Console.WriteLine ("Waiting...");
    _waitHandle.WaitOne();              // 等待通知
    Console.WriteLine ("Notified");
  }
}

// 输出：
Waiting... (暂停) Notified.
```

注1：和使用锁一样，队列公平性有时候可能会因为操作系统的差异而受到影响。

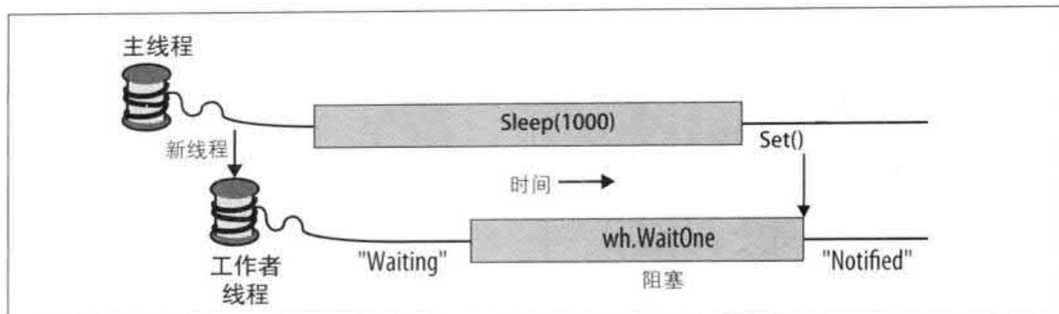


图22-1：使用EventWaitHandle发送信号

在没有线程等待时，如果调用Set，那么处理器会保持打开，直至有线程调用WaitOne。这种行为有利于避免线程进入闸门队列和线程插入一张票据的时间发生错乱（"在1毫秒内插入一张票据太早了，必须一直等待了！"）。然而，重复调用无人等待的闸门的Set方法，却无法让到达的全部人通过：只有下一个人可以通过，其他票据都无效。

调用AutoResetEvent的Reset，可以不等待或不阻塞，而直接关闭闸门（它原本处于开启状态）。

WaitOne接受一个可选的超时参数，如果由于超时引起等待结束，而不是接收到信号，那么它就会返回false。

提示： 使用超时时间0调用WaitOne，可以在不阻塞调用者的前提下测试等待处理器是否开启。但是，要记住，这样做会重置开启的AutoResetEvent。

撤销等待处理器

在完成等待处理器的工作时，调用它的Close方法，就可以释放操作资源。此外，也可以直接丢弃等待处理器的所有引用，使垃圾回收器能够在将来执行回收操作（等待处理器实现了撤销模式，它会在终结语句中调用Close）。这是少数几种允许使用这种后备方法的情况之一（有一定争议），因为等待处理器造成的操作系统负载并不高。

当应用域卸载时，等待处理器会自动释放。

双向信号

假设主线程需要向工作者线程连续发送三次信号。如果主线程以紧接方式多次直接调用等待处理器的Set方法，那么第2个或第3个信号可能会丢失，因为工作者可能需要一定时间才能完成各个信号的处理。

解决这个问题的方法是让主线程等待，直至工作线程完成之后再发送信号。这个过程可以通过另一个AutoResetEvent实现，具体方法如下：

```
class TwoWaySignaling
{
  static EventWaitHandle _ready = new AutoResetEvent (false);
  static EventWaitHandle _go = new AutoResetEvent (false);
```

```
static readonly object _locker = new object();
static string _message;

static void Main()
{
  new Thread (Work).Start();

  _ready.WaitOne();                          // 先等待工作者线程完成
  lock (_locker) _message = "ooo";
  _go.Set();                                 // Tell worker to go

  _ready.WaitOne();
  lock (_locker) _message = "ahhh";          // 向工作者线程传入另一个消息
  _go.Set();

  _ready.WaitOne();
  lock (_locker) _message = null;            // 发送信号，通知工作者退出
  _go.Set();
}

static void Work()
{
  while (true)
  {
    _ready.Set();                            // 表示已经准备好
    _go.WaitOne();                           // 等待启动...
    lock (_locker)
    {
      if (_message == null) return;          // 正常退出
      Console.WriteLine (_message);
    }
  }
}
}

// 输出：
ooo
ahhh
```

图22-2以可视化方式显示了这个过程。

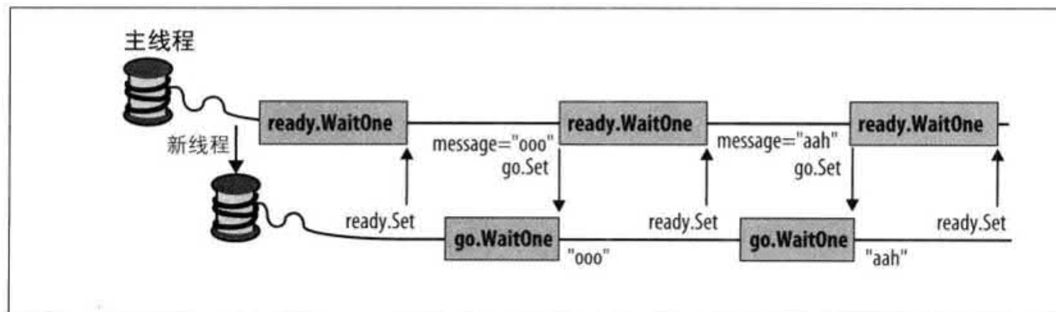


图22-2：双向信号

这里使用空消息表示工作者应该结束了。对于存在无限期执行的线程，一定要准备好退出策略！

22.5.2 ManualResetEvent

正如第14章所介绍，ManualResetEvent的作用就像是一扇大门。调用Set，可以打开大门，使任意线程可以调用WaitOne，然后获得进入大门的权限。调用Reset，则可以关闭大门。在已关闭大门上调用WaitOne的线程会进入阻塞状态；当大门再次打开时，这些线程会马上释放。除了这些区别，ManualResetEvent在功能上还与AutoResetEvent很相似。

和AutoResetEvent一样，有两种方法可以创建ManualResetEvent：

```
var manual1 = new ManualResetEvent (false);
var manual2 = new EventWaitHandle (false, EventResetMode.ManualReset);
```

提示： 从Framework 4.0开始，框架增加了另一个版本的ManualResetEvent，即ManualResetEventSlim。后者优化并缩短了等待时间——能够选择自旋一段时间。此外，它还拥有更高效的管理实现，允许通过CancellationToken取消一个等待（Wait）。然而，它不适用于进程间通信。ManualResetEventSlim并没有继承WaitHandle；然而，它包含一个WaitHandle属性，这个属性（使用传统等待处理器的性能配置）可以返回一个基于WaitHandle的对象。

信号结构与性能

（在无阻塞的情况下）等待或发送信号（AutoResetEvent或ManualResetEvent）都需要消耗1毫秒时间。

在短等待情况中，ManualResetEventSlim和CountdownEvent的速度要快50倍，因为它们不依赖于操作系统，并且优化了自旋结构的使用。

然而，在大多数情况中，信号发送类本身的过载并不会形成瓶颈，所以这很少会成为问题。

ManualResetEvent适用于在一个线程上释放多个其他线程。CountdownEvent则负责处理相反的情况。

22.5.3 CountdownEvent

CountdownEvent允许等待多个线程。这个类是在Framework 4.0引入的，也具有高效的管理实现。为了使用这个类，需要在初始化时传入希望等待的线程个数或计数：

```
var countdown = new CountdownEvent (3);  // 使用计数3进行初始化
```

调用Signal，可以增加计数；调用Wait，可以阻塞线程，直到计数减少为0。例如：

```
static CountdownEvent _countdown = new CountdownEvent (3);

static void Main()
{
  new Thread (SaySomething).Start ("I am thread 1");
  new Thread (SaySomething).Start ("I am thread 2");
  new Thread (SaySomething).Start ("I am thread 3");
  _countdown.Wait();   // 阻塞，直至Signal调用了3次
  Console.WriteLine ("All threads have finished speaking!");
}
```

```
static void SaySomething (object thing)
{
  Thread.Sleep (1000);
  Console.WriteLine (thing);
  _countdown.Signal();
}
```

提示： 有时候，第23章介绍的结构化并行结构（PLINQ和Parallel类）更适用于解决CountdownEvent的实际问题。

调用AddCount，可以再次增加CountdownEvent的计数。然而，如果它已经为0，那么它会抛出异常：我们无法通过调用AddCount来解除CountdownEvent。为了避免抛出异常，可以转而使用TryAddCount，它会在倒数到0时返回false。

要调用Reset，才可以解除倒数事件：解除该结构并将它的计数重置为原始值。

与ManualResetEventSlim类似，CountdownEvent也包含一个WaitHandle属性，它适用于需要接收一个基于WaitHandle的对象的其他类或方法。

22.5.4 创建跨进程的EventWaitHandle

EventWaitHandle的构造方法可以创建一个命名的EventWaitHandle，它可以运行在多个进程上。它的名称是一个普通字符串，理论上可以是任意值，但是不能与其他值发生冲突！如果这台计算已经使用了这个名称，那么它会返回已有的同一个EventWaitHandle；否则，操作系统才会创建一个新对象。例如：

```
EventWaitHandle wh = new EventWaitHandle (false, EventResetMode.AutoReset,
                                         "MyCompany.MyApp.SomeName");
```

如果有两个应用程序运行这段代码，那么它们就可以互相通信：等待处理器可以在两个进程的所有线程上运行。

22.5.5 等待处理器与延续

如果不想等待一个等待处理器和阻塞线程，那么可以调用ThreadPool.RegisterWaitForSingleObject，在线程上附加一个延续。这个方法接收一个代理对象，它会在等待处理器收到信号时执行：

```
static ManualResetEvent _starter = new ManualResetEvent (false);

public static void Main()
{
  RegisteredWaitHandle reg = ThreadPool.RegisterWaitForSingleObject
  (_starter, Go, "Some Data", -1, true);
  Thread.Sleep (5000);
  Console.WriteLine ("Signaling worker...");
  _starter.Set();
  Console.ReadLine();
  reg.Unregister (_starter);    // 清理前面的操作
}

public static void Go (object data, bool timedOut)
```

```
{
  Console.WriteLine ("Started - " + data);
  // 执行任务...
}

// 输出:
(延迟5秒钟)
Signaling worker...
Started - Some Data
```

当等待处理器接收到信号或者发生超时，那么代理对象就会在一个池化线程上运行。然后，可以调用Unregister，将自由的处理器释放回回调方法。

除了等待处理器和代理，RegisterWaitForSingleObject还可以接收一个“黑盒”对象（传给代理方法，与ParameterizedThreadStart不同）、一个毫秒为单位的超时时间（-1表示不超时）和一个布尔型标记（表示一次性请求，而非重复请求）。

高级线程

22.5.6 将等待处理器转换为任务

在实践中，ThreadPool.RegisterWaitForSingleObject并不好用，因为我们通常需要在回调方法上调用Unregister——在注册令牌可用之前。因此，需要按照下面的方式编写一个扩展方法，将一个等待处理器转换为可等待的Task：

```
public static Task<bool> ToTask (this WaitHandle waitHandle,
                                 int timeout = -1)
{
  var tcs = new TaskCompletionSource<bool>();
  RegisteredWaitHandle token = null;
  var tokenReady = new ManualResetEventSlim();
  token = ThreadPool.RegisterWaitForSingleObject (
    waitHandle,
    (state, timedOut) =>
    {
      tokenReady.Wait();
      tokenReady.Dispose();
      token.Unregister (waitHandle);
      tcs.SetResult (!timedOut);
    },
    null,
    timeout,
    true);
  tokenReady.Set();
  return tcs.Task;
}
```

然后，再按照如下方式给等待处理器附加一个延续：

```
myWaitHandle.ToTask().ContinueWith (...)
```

或者等待它：

```
await myWaitHandle.ToTask();
```

或者添加一个可选超时时间：

```
if (!await (myWaitHandle.ToTask (5000)))
  Console.WriteLine ("Timed out");
```

注意，在实现ToTask时必须使用另一个等待处理器（一个ManualResetEventSlim对象），才能避免出现争夺条件：回调方法在注册令牌分配给token变量之前运行。

22.5.7 WaitAny、WaitAll和SignalAndWait

除了Set、WaitOne和Reset方法，WaitHandle还有其他一些跟踪更复杂同步操作的静态方法。WaitAny、WaitAll和SignalAndWait方法可以在多个处理器上执行原子信号发送和等待操作。等待处理器可以是任意类型（包括Mutex和Semphore，因为它们还继承了抽象类WaitHandle）。此外，ManualResetEventSlim和CountdownEvent还可以通过WaitHandle属性使用这些方法。

警告：WaitAll和SignalAndWait与传统的COM架构存在奇怪的联系：这些方法要求调用者位于多线程环境，这种模型具有很差的互操作性。例如，在这种模式中，WPF和Windows窗体应用程序的主线程无法访问剪贴版。我们很快就介绍这个问题。

WaitHandle.WaitAny可以等待一组等待处理器中的任意一个；WaitHandle.WaitAll可以用原子方式等待指定的所有处理器。这意味着，如果等待两个AutoResetEvents：

- WaitAny最终无法同时封锁两个事件。
- WaitAll最终无法只封锁其中一个事件。

SignalAndWait会调用其中一个WaitHandle的Set，然后再调用另一个WaitHandle的WaitOne。原子操作可以保证在发送第一个处理器信号之后，它会转到等待第二个处理器的队头：可以认为它是从一个信号切换到另一个信号。在一对EventWaitHandles上使用这个方法，就可以创建两个同时会合的线程。AutoResetEvent或ManualResetEvent都一样有效。第一个线程会执行以下方法：

```
WaitHandle.SignalAndWait (wh1, wh2);
```

而第二个线程则执行相反操作：

```
WaitHandle.SignalAndWait (wh2, wh1);
```

WaitAll和SignalAndWait的替代选择

WaitAll和SignalAndWait不能运行在单线程环境中。幸好，它们有一些替代选择。对于SignalAndWait，它的原子操作保证并不是很必要：我们的会合例子就是一个很好的例子，直接调用第一个等待处理器的Set，然后再调用另一个的WaitOne。下一节将介绍另一种实现线程会合的方法。

对于WaitAny和WaitAll，如果不需要实现原子操作，则可以使用上一节的代码，将等待处理器转换为任务，然后使用Task.WhenAny和Task.WhenAll（第14章）。

如果一定需要原子操作，那么可以使用最低层的方法发送信号，然后使用Monitor的Wait和Pulse方法编写自定义逻辑。关于Wait和Pulse详细说明，请参见：*http://albahari.com/threading/*。

22.6 Barrier类

Barrier类可以实现一个线程执行屏障（thread execution barrier），它允许多个线程在同一时刻会合。这个类执行速度很快，也非常高效，它基于Wait、Pulse和自旋锁实现。

使用Barriet类的步骤是：

1. 创建它的实例，指定参与会合的线程数量（将来可以通过调用AddParticipants/RemoveParticipants来修改这个值）。
2. 当需要会合时，在每一个线程上调用SignalAndWait。

使用数值3创建一个Barrier实例，会使SignalAndWait进入阻塞状态，直至这个方法被调用3次。然后，它重新开始：再次调用SignalAndWait，进入阻塞状态，直至3次调用结束。这样就可以让各个线程都"步调一致地"执行。

在下面的例子中，3个线程都会打印出0至4的数字值，同时与其他线程保持一致的步调：

```
static Barrier _barrier = new Barrier (3);

static void Main()
{
  new Thread (Speak).Start();
  new Thread (Speak).Start();
  new Thread (Speak).Start();
}

static void Speak()
{
  for (int i = 0; i < 5; i++)
  {
    Console.Write (i + " ");
    _barrier.SignalAndWait();
  }
}

输出：0 0 0 1 1 1 2 2 2 3 3 3 4 4 4
```

在Barrier中，一个真正有用的特性是：在创建时还可以指定一个阶段后操作（post-phase action）。这是一个代理对象，它会在SignalAndWait被调用n次之后、线程解除阻塞之前执行（如图22-3的阴影区域所示）。在我们的例子中，如果按照以下方式创建一个Barrier实例：

```
static Barrier _barrier = new Barrier (3, barrier => Console.WriteLine());
```

那么输出结果就是：

```
0 0 0
1 1 1
2 2 2
3 3 3
4 4 4
```

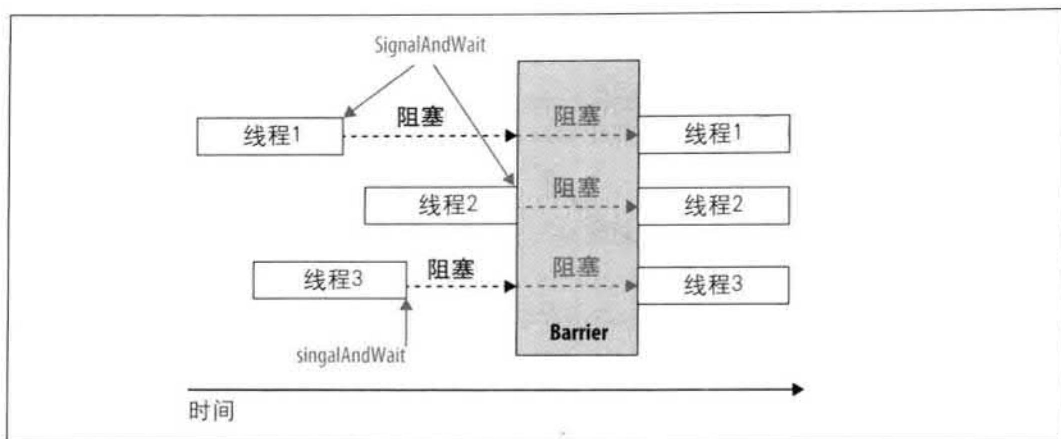


图22-3：Barrier

阶段后操作适用于从各个工作者线程获取数据。它不需要担心抢占问题，因为它在执行过程中会阻塞所有的工作者线程。

22.7 延后初始化

线程中有一个常见的问题，即如何以线程安全的方式延后初始化一个共享域。如果域的创建开销很大，那么就需要考虑这个问题。

```
class Foo
{
  public readonly Expensive Expensive = new Expensive();
  ...
}
class Expensive {  /* 假设这个类的构建开销很大 */  }
```

这段代码的问题是，实例化Foo包含实例化Expensive的性能开销——无论Expensive域是否会被使用。最直接的解决方法是按需创建这个实例：

```
class Foo
{
  Expensive _expensive;
  public Expensive Expensive                    // 延后初始化Expensive
  {
    get
    {
      if (_expensive == null) _expensive = new Expensive();
      return _expensive;
    }
  }
  ...
}
```

需要考虑一个问题：这段代码是否具有线程安全性？除了在不使用内存屏障的前提下在锁之外访问_expensive，考虑一下当两个线程同时访问这个属性时，会出现什么情况？它们可能都满足if语句

的断言，最终每一个线程都会获得完全不同的Expensive实例。由于这会导致一些微妙的错误，所以我们通常认为这段代码不具有线程安全性。

解决这个问题的方法是在检查和初始化对象的代码上添加锁：

```
Expensive _expensive;
readonly object _expenseLock = new object();

public Expensive Expensive
{
  get
  {
    lock (_expenseLock)
    {
      if (_expensive == null) _expensive = new Expensive();
      return _expensive;
    }
  }
}
```

高级线程

22.7.1 Lazy<T>

从Framework 4.0开始，我们可以使用Lazy<T>类实现延后初始化。如果实例化时传入参数值true，那么它就会实现刚刚描述的线程安全初始化模式。

提示： Lazy<T>实际上实现了这个模式的微优化版本，即所谓的双重检查锁（double-checked locking）。双重检查锁会执行另一个不稳定（volatile）读操作，避免在对象已经初始化之后获取锁的开销。

为了获得一个Lazy<T>，需要在创建这个类的实例时使用一个工厂代理值（用于指定如何初始化一个新值）和参数值true。然后使用Value属性访问它的值：

```
Lazy<Expensive> _expensive = new Lazy<Expensive>
  (() => new Expensive(), true);

public Expensive Expensive { get { return _expensive.Value; } }
```

如果在Lazy<T>构造方法中传入false，那么它会实现一个非线程安全的初始化模式，具体参见本节开头——如果要在单线程环境中使用Lazy<T>，则可以使用这种方法。

22.7.2 LazyInitializer

LazyInitializer是一个静态类，它的使用方法与Lazy<T>相似，但是：

- 它的功能是通过直接操作自定义类型域的静态方法实现。这样做可以避免一定程度的间接性，从而提高性能，适用于一些需要极致优化的场合。
- 它提供了另一种初始化模式，这时多个线程需要争夺实例化顺序。

在使用LazyInitializer时，需要在访问域之前调用EnsureInitialized，传入域的引用和工厂代理对象：

```
Expensive _expensive;
public Expensive Expensive
{
```

```
    get          // 实现双重检查锁
    {
      LazyInitializer.EnsureInitialized (ref _expensive,
                                         () => new Expensive());
      return _expensive;
    }
  }
```

此外，还可以传入另一个参数，要求竞争的线程争夺实例化顺序。这与初始的非线程安全例子很相似，但是第一个完成的线程就先实例化，所以最终只能得到一个实例。这种方法的优点是在多核处理器上它的速度比双重检查锁更快——因为它的实现完全不需要使用锁，而只需要使用非阻塞同步和延后初始化等高级技术，具体参见*http://albahari.com/threading/*。这是一种极致（很少使用）优化，它会带来以下开销：

- 如果争夺实例化的线程数量大于内核数量，那么速度会变慢。
- 它可能会将CPU资源浪费在执行多余初始化操作的过程中。
- 初始化逻辑必须具有线程安全性（在这个例子中，如果Expensive的构造方法需要写静态域，那么它就不具有线程安全性）。
- 如果初始化代码创建的对象需要执行清理操作，那么必须使用额外的逻辑代码清除“废弃的”对象。

22.8 线程本地存储

本章大部分内容都在介绍同步结构和线程并发访问数据的问题。然而，有时候也需要保持数据隔离，以保证每一个线程都拥有属于自己的副本。本地变量就可以实现这个目标，但是它们只适用于保存临时数据。

解决方法是使用线程本地存储（thread-local storage）。有一个需求可能并不常见：在线程上保持隔离的数据本身只是临时数据。远程本地存储主要用于存储带外数据——这是支持执行路径的基础架构，如消息传递、事务和安全令牌。在方法参数上传递这些数据的做法非常不灵活，而且仅适用于自己编写的方法；将这些信息存储在静态域中，又意味着将它们共享给所有线程。

提示： 线程本地存储还适用于优化并行模式。它允许每一个线程在不需要锁的前提下独立访问属于自己的非线程安全对象——不需要在方法调用过程之间重新创建这个对象。

然而，它不适合在异步代码中使用，因为有一些延续可能运行在另一个先运行的线程上。

有三种方法可以实现线程本地存储。

22.8.1 [ThreadStatic]

实现线程本地存储的最简单的方法是使用ThreadStatic修饰静态域：

```
[ThreadStatic] static int _x;
```

然而，每一个线程都可以使用独立的_x副本。

但是，[ThreadStatic]不适用于实例域——它不会产生任何作用；也不适用于域的初始化对象——它们只能在调用静态构造方法的线程上执行一次。如果需要处理实例域（或者使用非默认值），那么更适合使用ThreadLocal<T>。

22.8.2 ThreadLocal<T>

ThreadLocal<T>是Framework 4.0新增加的类。它支持创建静态域和实例域的线程本地存储，并且允许指定默认值。

下面的例子演示了如何为每一个线程创建一个默认值为3的ThreadLocal<int>：

```
static ThreadLocal<int> _x = new ThreadLocal<int> (() => 3);
```

然后，使用_x的Value属性，就可以获取或设置它的线程本地值。使用ThreadLocal的好处是它存储的值都会延后估值：工厂函数会先判断每一个线程第一个调用的值。

1. ThreadLocal<T>和实例域

ThreadLocal<T>也适用于实例域和获得的本地变量。例如，考虑多线程环境中生成随机数据的问题。Random类本身不具有线程安全性，所以必须在使用Random的代码上添加锁（限制并发性），或者为每一个线程生成独立的Random对象。ThreadLocal<T>简化了后一种情况：

```
var localRandom = new ThreadLocal<Random>(() => new Random());
Console.WriteLine (localRandom.Value.Next());
```

但是，这里创建Random对象的工厂函数有一些简单，其中Random的无参数构造方法使用系统时钟创建随机数种子。在10毫秒内创建的两个Random对象可能使用了相同的随机数种子。下面的方法可以解决这个问题：

```
var localRandom = new ThreadLocal<Random>
 ( () => new Random (Guid.NewGuid().GetHashCode()) );
```

下一章也会使用这种方法，参见第23章“PLINQ”的并行拼写检查例子。

22.8.3 GetData和SetData

第三种方法是使用Thread类的两个方法：GetData和SetData。这两个方法会将数据存储在线程独有的“插槽”中。Thread.GetData负责读取线程的独立数据存储；Thread.SetData则负责写入这些数据。这两个方法都需要使用LocalDataStoreSlot对象来获得这个存储插槽。所有线程都可以使用相同的插槽，但是它们保存相互独立的值。例如：

```
class Test
{
  // 所有线程可以使用同一个LocalDataStoreSlot对象。
  LocalDataStoreSlot _secSlot = Thread.GetNamedDataSlot ("securityLevel");

  // 这个属性保存各个线程的独立数据。
  int SecurityLevel
  {
    get
    {
      object data = Thread.GetData (_secSlot);
```

```
        return data == null ? 0 : (int) data;    // null == 未初始化
      }
      set { Thread.SetData (_secSlot, value); }
    }
    ...
```

这个例子调用Thread.GetNamedDataSlot，它会创建一个命名插槽，从而在整个应用程序中共享这个插槽。此外，调用Thread.AllocateDataSlot，获得一个未命名插槽，则可以控制插槽的范围：

```
class Test
{
  LocalDataStoreSlot _secSlot = Thread.AllocateDataSlot();
  ...
```

Thread.FreeNamedDataSlot将释放所有线程的命名数据插槽，但是只有当LocalDataStoreSlot的所有引用都已经删除并且都被垃圾回收器回收时才会释放。因此，在需要使用插槽时，只要线程保存了正确的LocalDataStoreSlot对象引用，就可以保证线程不会获得不属于自己的数据插槽。

22.9 Interrupt和Abort

Interrupt和Abort方法可以控制另一个线程。Interrupt并没有有效的用例，而Abort有时候是很有用的。

Interrupt可以强制释放处于阻塞状态的线程，同时在线程上抛出ThreadInterruptedException异常。如果该线程当前并未处于阻塞状态，那么执行过程会继续，直至它进入阻塞状态，然后再抛出ThreadInterruptedException异常。Interrupt用处不大，因为大部分情况更适合使用信号发送结构和取消令牌（或Abort方法）进行处理。此外，它内部还有一定的危险性，因为代码本身无法确定线程何时应该强制解除阻塞状态（例如，它可能位于.NET框架的内部）。

Abort会尝试强制中止另一个线程，然后它所在的线程上会马上抛出ThreadAbortException异常（非受控代码除外）。ThreadAbortException有些特殊，当它被捕捉时，除非在catch语句块中调用Thread.ResetAbort，否则它会在catch语句块末尾重新抛出（尝试永远中止该线程）。在这期间，线程获得了一个AbortRequested的ThreadState。

提示：未处理的ThreadAbortException异常是两种不会导致应用程序关闭的异常之一（另一种是AppDomainUnloadException）。

为了保持应用域的完整性，最好添加finally语句块，并且不要使用从不退出的静态构造方法。尽管如此，Abort也不适用于实现通用取消操作，因为它仍然可能给退出的线程带来问题，以及破坏应用域（甚至整个进程）。

例如，假设一个类型的实例构造方法获取非受控资源（例如，文件句柄），它会在Dispose方法中释放。如果线程在构造方法结束之前退出，那么创建未完成的对象就无法清除，因此非受控的句柄就会泄漏。（如果有终止化代码，那么它仍然会运行，但是必须在GC捕捉到它时才会运行。）.NET框架中许多类型有这样的问题，其中包括FileStream，因此Abort不适合在大部分情况中使用。关于中止.NET框架代码的不安全问题，更深入的讨论请参见“中止线程”：*http://www.albahari.com/threading/*。

如果只能使用Abort，那么在另一个应用域中运行线程，然后在中止线程之后重新创建这个域，可以

大大降低潜在的危害（这正是LINQPad在取消查询时所做的事情）。第24章将介绍应用域。

> 提示：在独立线程上调用Abort是有效且安全的，因为这时可以完全控制手中的代码。如果想要在每一个catch语句块之后重新抛出异常，那么就适合使用这个方法——调用Redirect时，ASP.NET就是样做的。

22.10 Suspend和Resume

Suspend和Resume可以冻结和解冻另一个线程。虽然在概念上与阻塞不同（可以通过它的ThreadState查询），但是冻结的线程就像进入了阻塞状态。与Interrupt一样，Suspend/Resume也缺少有效的用例，并且也可能存在危险：如果暂停一个获得了锁的线程，那么其他线程就无法获得这个锁（包括自己的锁），这使得程序很容易发生死锁。因此，Framework 2.0废弃了Suspend和Resume。

高级线程

然而，如果追踪另一个线程的堆栈信息，那么一定要暂停线程。这种操作有时候很适用于诊断程序，具体实现方法如下：

```
StackTrace stackTrace;   // 在System.Diagnostics中
targetThread.Suspend();
try { stackTrace = new StackTrace (targetThread, true); }
finally { targetThread.Resume(); }
```

但是，这种实现也很容易出现死锁，因为跟踪堆栈信息本身需要通过反射机制获取锁。解决方法是，如果它在200毫秒之后仍然处于暂停状态（这时，可以认为有一个线程已经出现死锁），那么就在另一个线程上调用Resume。当然，这会使堆栈跟踪信息失效，但是它肯定比造成应用程序死锁要好。

```
StackTrace stackTrace = null;
var ready = new ManualResetEventSlim();

new Thread (() =>
{
  // 在出现死锁时返回释放线程：
  ready.Set();
  Thread.Sleep (200);
  try { targetThread.Resume(); } catch { }
}).Start();

ready.Wait();
targetThread.Suspend();
try { stackTrace = new StackTrace (targetThread, true); }
catch { /* 死锁 */ }
finally
{
  try { targetThread.Resume(); }
  catch { stackTrace = null; /* 死锁 */  }
}
```

22.11 定时器

如果需要定期重复执行一些方法，最容易的方式是使用定时器。与下面这种方式相比，定时器在使用内存和资源方面既方便又高效：

```
new Thread (delegate() {
                         while (enabled)
                         {
          DoSomeAction();
                           Thread.Sleep (TimeSpan.FromHours (24));
                         }
                       }).Start();
```

这不仅永久性地关联到线程资源，而且如果不进行另外的编程，DoSomeAction过段时间每天都将执行。定时器解决了这些问题。

.NET Framework提供了四种定时器，以下两种是通用的多线程定时器：

- System.Threading.Timer
- System.Timers.Timer

其他两种是特殊用途的单线程定时器：

- System.Windows.Forms.Timer（Windows Forms 定时器）
- System.Windows.Threading.DispatcherTimer（WPF 定时器）

多线程定时器更加强大、精确和灵活，而在运行需要更新Windows Forms控件或WPF元素的简单任务时，单线程定时器更加安全和方便。

22.11.1 多线程定时器

System.Threading.Timer是最简单的多线程定时器，它只有一个构造方法和两个方法。在下面的示例中，定时器调用了Tick方法，该方法在5秒钟后写“tick…”，然后每隔一秒写一次，直到用户按下Enter键为止：

```
using System;
using System.Threading;

class Program
{
  static void Main()
  {
    // 首次时间间隔 = 5000ms; 后续时间间隔 = 1000ms
    Timer tmr = new Timer (Tick, "tick...", 5000, 1000);
    Console.ReadLine();
    tmr.Dispose();                  // 同时会停止定时器并进行清理
  }

  static void Tick (object data)
  {
    // 运行在一个池化线程上
    Console.WriteLine (data);              // 写"tick..."
  }
}
```

提示： 第12章的“定时器”一节介绍了如何清理多线程定时器。

调用Change方法可以修改定时器的时间间隔。如果指示定时器只触发一次，在构造方法的最后一个参数中指定Timeout.Infinite。

.NET Framework提供另一个与System.Timers命名空间中名称相同的定时器类。它简单地封装了System.Threading.Timer，在使用完全相同的底层引擎时更加方便。下面总结了它增加的功能：

- 一个Component实现，允许它嵌入在Visual Studio的设计器中。
- 一个Interval属性，而非一个Change方法。
- 一个Elapsed事件，而非一个回调委托。
- 一个Enabled属性，用于开始和停止定时器（其默认值为false）。
- Start和Stop方法，以防混淆Enabled属性的用法。
- 一个AutoReset标志，用于指示递归事件（默认值为true）。
- 一个带有Invoke和BeginInvoke方法的SynchronizingObject属性，用于安全地调用WPF元素和Windows Forms控件上的方法。

例如：

```
using System;
using System.Timers;                    // Timers而非Threading命名空间

class SystemTimer
{
  static void Main()
  {
    Timer tmr = new Timer();            // 不需要任何参数
    tmr.Interval = 500;
    tmr.Elapsed += tmr_Elapsed;         // 使用一个事件而非委托
    tmr.Start();                        // 开始定时器
    Console.ReadLine();
    tmr.Stop();                         // 停止定时器
    Console.ReadLine();
    tmr.Start();                        // 重启定时器
    Console.ReadLine();
    tmr.Dispose();                      // 永久性停止定时器
  }

  static void tmr_Elapsed (object sender, EventArgs e)
  {
    Console.WriteLine ("Tick");
  }
}
```

多线程定时器使用线程池来支持一些线程为多个定时器提供服务。这意味着每次被调用时，回调方法或Tick事件都可能在不同的线程上触发。此外，Tick始终会（近似）按时触发，无论前面的Tick是否已经结束执行。因此，回调或事件处理器必须是线程安全的。

多线程定时器的精度取决于操作系统，精度范围通常在10～20ms。如果需要更高的精度，可以使用本地的Interop，并调用Windows多媒体定时器。这个定时器的精度低至1ms，定义在*winmm.dll*中。首先调用timeBeginPeriod指示操作系统需要高定时精度，然后调用timeSetEvent启动一个多媒体定时器。完成之后，调用timeKillEvent停止定时器，并调用timeEndPeriod指示操作系统不再需要

高定时精度。第25章演示了如何使用P/Invoke调用外部方法。在Internet上搜索关键字*dllimport winmm.dll timesetevent*，便可找到使用多媒体定时器的完整示例。

22.11.2 单线程定时器

.NET Framework提供的定时器可用于解决WPF与Windows Forms应用程序的线程安全性问题：

- `System.Windows.Threading.DispatcherTimer`（WPF）
- `System.Windows.Forms.Timer` （Windows Forms）

警告： 单线程定时器不能在各自环境之外使用。例如，如果在Windows Service应用程序中使用Windows Forms定时器，Timer事件将不会触发！

这两种定时器在公开的成员方面与`System.Timers.Timer`相似（`Interval`、`Tick`、`Start`和`Stop`），使用方式也类似。然而，它们之间的差异在于内部工作原理不同。WPF与Windows Forms定时器依赖于其基本用户界面模型的消息泵机制，而不是使用线程池生成定时器事件。这意味着，`Tick`事件在原本创建定时器的线程上始终会触发——在常规的应用程序中，这个线程同样用于管理所有的用户界面元素与控件。这样做有很多好处：

- 可以忽略线程安全性。
- 一个新的`Tick`将永远不会触发，直到前面的`Tick`完成处理。
- 可以直接从`Tick`事件处理代码更新用户界面元素和控件，同时无需调用`Control.Invoke`或`Dispatcher.Invoke`。

因此，使用这些定时器的程序实际上并不是多线程的：最终得到的是与第14章在UI线程中用异步函数实现的伪并发性相同的效果。一个线程就可以处理所有定时器以及UI事件。这意味着，`Tick`事件处理器必须快速执行，否则用户界面的响应速度会受到影响。

这使得WPF和Windows窗体定时器适合处理小作业，通常是更新UI的特定部分（例如，显示时钟或倒数计时）。

在精度方面，单线程定时器与多线程定时器类似（几十毫秒），但它们通常不太精确，因为当处理其他用户界面请求或其他定时器事件时，它们可能延迟。

第23章 并行编程

在本章中，我们介绍多线程API和利用多内核处理器的结构：

- 并行的LINQ或*PLINQ*
- `Parallel`类
- *任务并行*结构
- *并发集合*
- `SpinLock`和`SpinWait`

这些是Framework 4.0的新功能，一般称为PFX（Parallel Framework，并行框架）。`Parallel`类和任务并行结构统称为任务并行库（Task Parallel Library，TPL）。

在阅读本章之前，需要熟练掌握第14章中的基础知识，特别是锁定和线程安全。

23.1 PFX

目前，CPU时钟速度已经停滞不前，制造商们把重点转移到了提高内核数量上。这就为我们程序员带来了问题，因为增加了这些额外的内核之后，标准单线程代码并不会自动提高运行速度。

利用多内核对于大多数服务器应用程序来说十分简单，因为每个线程都能独立处理一个单独的客户端请求，但在桌面上较难，因为这通常需要对计算密集型代码进行如下处理：

1. 将代码划分为多个小块。
2. 通过多线程并行执行这些代码块。
3. 结果变为可用后，以线程安全和高性能的方式整理这些结果。

尽管可以使用传统的多线程结构来实现所有这些功能，但难度颇高而且很不方便，特别是划分和整理的步骤。一个更深层次的问题是，当很多线程同时使用相同数据时，出于线程安全进行锁定的常用策略会引发大量竞争。

PFX库专门用于在这些应用场景中提供帮助。

提示：通过编程方式利用多内核或多处理器称为并行编程，它是多线程更宽泛概念的一个子集。

23.1.1 PFX概念

划分线程间工作有两种策略：数据并行和任务并行。

当一组任务必须基于很多数据值来执行时，我们可以让每个线程在一个数值子集上执行（同）一组任务，从而实现并行化。这称为数据并行，因为我们是在线程之间划分数据。相比之下，任务并行划分的是任务。换句话说，每个线程执行不同的任务。

一般而言，数据并行较为简单，更适合高度并行的硬件，因为它减少或消除了共享数据（从而减少竞争和线程安全问题）。此外，数据并行利用了数据值通常比离散任务要多这个特点，从而提高了并行性。

数据并行也有益于结构化并行，这意味着并行工作单元在程序中的相同位置启动和结束。相比之下，任务并行往往是非结构化的，因此并行工作单元的启动和结束可能分散在程序中的各个地方。结构化并行相对更加简单、不易出错，而且支持将划分和线程协作（甚至还有结果整理）的工作留给库去完成。

23.1.2 PFX组件

PFX包含两个层次的功能，如图23-1所示。高层由两个结构化数据并行API组成：PLINQ和Parallel类。底层包含任务并行类和一组另外的结构，可为并行编程提供帮助。

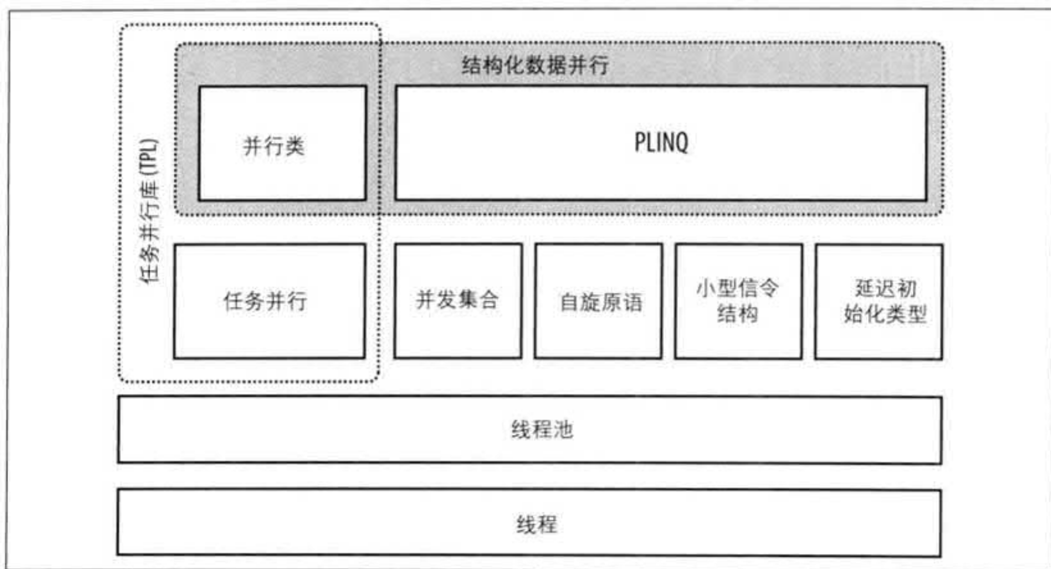


图23-1：PFX组件

PLINQ提供了最丰富的功能：它自动完成并行完成的所有步骤，包括将工作划分给任务、在线程上执行这些任务以及将结果整理为一个输出序列。它的调用是声明式的，因为只要声明将工作（结构化为一个LINQ查询）并行化，然后让Framework去处理实现细节即可。相比之下，其他方法是命令式

的，需要显式编写代码进行划分或整理。如果使用Parallel类，必须自己整理结果；而借助并行化结构，也必须自己划分工作：

	划分工作	整理结果
PLINQ	是	是
Parallel类	是	否
PFX的任务并行	否	否

并发集合和自旋原语通过底层的并行编程提供帮助。这些很重要，因为PFX不仅能够使用现在的硬件，而且也能使用下一代多内核处理器。例如如果要移动一堆砍好的木柴，一共有32名工人来完成这项工作，最大的难题是在移动木柴时，如何让工人们不会互相挡道。在32个内核之间划分一种算法也是如此：如果使用普通锁保护通用资源，结果造成阻塞，只有一部分内核是真正忙碌的。并发集合专门针对高度并发的访问进行了调整，重点是最小化或消除阻塞。PLINQ和Parallel类本身依赖于并发集合和自旋原语才能高效地管理工作。

PFX的其他用法

并行编程指令不仅可以调用多核线程，也适用于其他一些场景

- 并发集合有时候很适合用于创建线程安全的队列、堆栈或字典。
- BlockingCollection提供了一个实现生产者/消费者结构的简单方法，而且也是一种限制并发性的好方法。
- Task是异步编程的基础，具体参见第14章的介绍。

23.1.3 使用PFX的场合

PFX主要用于并行编程：利用多内核处理器加快计算密集型代码的执行速度。

利用多内核必须遵守Amdahl定律，根据此定律，并行化的最大性能改进取决于必须顺序执行的代码部分。例如，如果一种算法的执行时间只有三分之二是可并行化的，无论使用多少个内核所获得的性能提高绝不会超过3倍。

因此在执行之前，确认可并行化的代码。还要考虑代码是否需要是计算密集型的，优化通常是最简单和最有效的方法。但有些优化技术可能会让代码并行化变得更加困难。

最简单的示例就是易并行（embarrassingly parallel）问题，就是将一个任务容易地划分为可有效单独执行的任务（结构化并行非常适合解决这类问题）。示例包括很多图像处理任务、光线跟踪，以及数学或密码学中的粗略近似法。非易并行问题的一个示例是实现快速排序算法的一个优化版本，好的结果需要经过深思熟虑，可能需要非结构化的并行。

23.2 PLINQ

PLINQ将自动并行化本地的LINQ查询。PLINQ的优势是易于使用，因为它把工作划分和结果整理的任务转给了Framework。

要使用PLINQ，只要在输入序列上调用AsParallel()方法，然后继续执行LINQ查询。下面的查询计

算3与100,000之间的质数，它充分利用了目标计算机上的所有内核：

```
// 使用一种简单的（未经过优化的）算法计算质数

IEnumerable<int> numbers = Enumerable.Range (3, 100000-3);

var parallelQuery =
  from n in numbers.AsParallel()
  where Enumerable.Range (2, (int) Math.Sqrt (n)).All (i => n % i > 0)
  select n;

int[] primes = parallelQuery.ToArray();
```

AsParallel是System.Linq.ParallelEnumerable中的一个扩展方法。它基于Parall-elQuery<TSource>封装输入序列，使随后调用的LINQ查询运算符绑定到Parallel-Enumerable中定义的另一组方法。这为每个标准查询运算符提供了并行实现。基本上，它们的工作原理是将输入序列划分为在不同线程上执行的小块，然后将结果整理到一个输出序列中以供使用（见图23-2）。

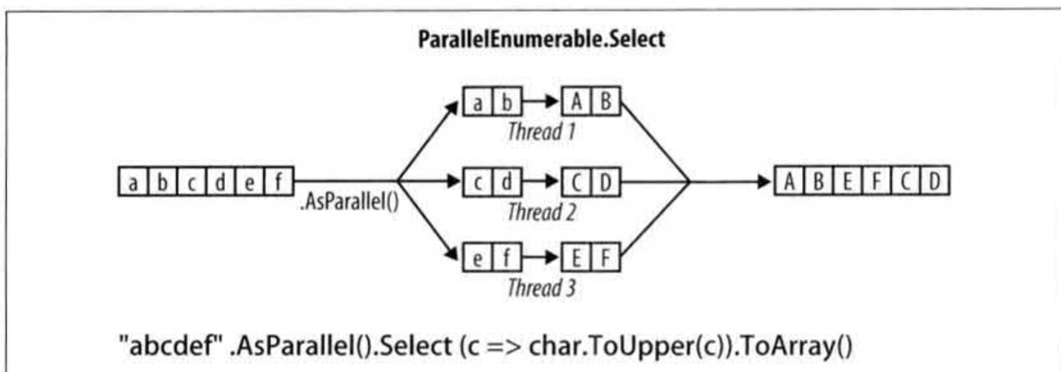


图23-2：PLINQ执行模型

调用AsSequential()方法可以解除对ParallelQuery序列的封装，这样后续的查询运算符便可绑定到标准的查询运算符上，然后继续执行。在调用有副作用或不是线程安全的方法之前，这是必要的。

对于接受两个输入序列的查询运算符（Join、GroupJoin、Concat、Union、Intersect、Except和Zip），必须对这两个输入序列应用AsParallel()方法，否则将抛出异常。但不需要在查询进行时一直对它应用AsParallel，因为PLINQ的查询运算符输出另一个ParallelQuery序列。事实上，再次调用AsParallel会降低效率，因为它会强制合并和重新划分查询：

```
mySequence.AsParallel()           // 封装ParallelQuery<int>中的序列
          .Where (n => n > 100)   // 输出另一个ParallelQuery<int>
          .AsParallel()           // 不必要并且效率低
          .Select (n => n * n)
```

并非所有的运算符都可以被有效并行化。对于那些不能有效并行化的运算符（见“PLINQ限制”一节），PLINQ还是以顺序方式实现它们。为了避免并行化的开销会实际降低特定查询的速度，PLINQ也可以按照顺序操作。

PLINQ仅用于本地集合：它不能与LINQ to SQL或 Entity Framework一起使用，因为在这些情况下，LINQ会转换为SQL，然后在数据库服务器上执行。然而，可以使用PLINQ基于从数据库查询获得的

数据集来执行另外的本地查询。

警告： 如果PLINQ查询抛出一个异常，它将以AggregateException形式被重抛，其InnerExceptions属性包含一个或多个真正的异常。见“处理AggregateException异常”一节以了解更多细节。

为什么 AsParallel 不是默认的？

如果AsParallel透明地并行化LINQ查询，问题出现了，“为什么Microsoft不简单地并行化标准查询运算符，并让PLINQ成为默认做法呢？”

默认不使用AsParallel有多条理由。首先，为了让PLINQ发挥作用，必须有一定数量的计算密集型工作可分配给工作者线程。大多数LINQ to Objects查询执行速度很快，不仅没有必要并行化，而且划分、整理和协调额外线程的开销实际上会降低执行速度。

另外：

- PLINQ查询的输出（默认）可能与用于元素排序的LINQ查询不同（见“PLINQ与排序”一节）。
- PLINQ将异常封装在AggregateException中，以便处理可能抛出多个异常的情况。
- 如果查询调用了非线程安全的方法，PLINQ的结果有可能不正确。

最后，PLINQ针对调整与调节提供了相当多的钩子（hook）。为了这些细微差别而给标准的LINQ to Objects API增加负担有点得不偿失。

并行编程

23.2.1 并行执行

和普通的LINQ查询一样，PLINQ查询也是延迟求值的。这意味着执行只在开始使用结果时触发——通常是通过一个foreach循环（但也可以通过像ToArray这样的转换运算符以及返回一个元素或值的运算符）。

但枚举结果时，与普通的顺序查询有区别。顺序查询完全由使用者通过“拉”方式来控制：当使用者请求时，从输入序列获取每个元素。并行查询通常使用独立线程来获取输入序列中的元素，而且时间上比使用者需要它们时要稍微提前一点（就像是新闻主播用的讲词提示机或者CD播放器中的防震缓冲）。接下来，它会通过查询链并行处理元素，将结果保存在一小块缓存中，以便让使用者按需取用。如果使用者很早就暂停或结束了枚举，查询处理器也会暂停或结束，目的是不浪费CPU时间或内存。

提示： 在AsParallel之后调用WithMergeOptions可以调节PLINQ的缓冲行为。AutoBuffered的默认值一般会给出最佳的总体结果。NotBuffered禁用缓存，用于需要尽快地看到结果时；FullyBuffered把整个结果集展现给使用者之前对其进行缓冲（OrderBy和Reverse运算符的工作方式也是如此，此外还有元素、聚合和转换运算符）。

23.2.2 PLINQ与排序

并行化查询运算符的副作用是当整理结果时，不必遵循提交它们的相同顺序（见图23-2）。换句话

说，LINQ不能保持序列的原始次序。

如果需要保持序列的原始次序，可以通过在AsParallel()之后调用AsOrdered()方法来强制实现：

```
myCollection.AsParallel().AsOrdered()...
```

调用AsOrdered时，因为PLINQ必须保持跟踪每个元素的原始位置，如果元素数量巨大就会导致性能损失。

通过调用AsUnordered，可在稍后的查询中抵消AsOrdered的效果：这引入了一个“随机洗牌点”，允许查询从这个点开始更加高效地执行。因此，如果只需对前两个查询运算符保持输入序列的次序，可以这样做：

```
inputSequence.AsParallel().AsOrdered()
  .QueryOperator1()
  .QueryOperator2()
  .AsUnordered()       // 从这里开始，次序不再重要
  .QueryOperator3()
  ...
```

AsOrdered并非默认的，因为对于大多数查询而言，原始的输入顺序并不重要。换句话说，如果AsOrdered是默认的，将不得不对绝大多数并行查询应用AsUnordered以获得最佳性能，而这会带来巨大的工作量。

23.2.3 PLINQ限制

目前，关于PLINQ能够并行化的内容还存在一些限制。在后续的服务补丁和Framework版本中，这些限制可能会减少。

以下查询运算符防止查询被并行化，除非源元素位于它们的原始的索引位置：

- Take、TakeWhile、Skip和 SkipWhile
- Select、SelectMany和ElementAt的索引版本

大多数查询运算符都会改变元素的索引位置，包括像Where这样移除元素的运算符。这意味着如需要前述运算符，它们通常需要位于查询的开始。

以下查询运算符是并行化的，但所使用的复杂划分策略有时可能比顺序处理的速度还要低：

- Join、GroupBy、GroupJoin、Distinct、Union、Intersect和Except

Aggregate运算符在标准形式中的预计负载不是可并行化的，PLINQ提供特殊负载来处理这个问题（见“优化PLINQ”一节）。

所有其他的运算符都是可并行化的，但使用这些运算符不能确保查询会被并行化。为了避免并行化的开销会降低特定查询的速度，PLINQ可以通过顺序方式运行查询。在AsParallel()之后调用如下方法可以重写这种行为并强制实现并行化：

```
.WithExecutionMode (ParallelExecutionMode.ForceParallelism)
```

23.2.4 示例：并行的拼写检查器

假定我们要编写一个可利用所有可用内核来快速处理非常大型文档的拼写检查器。通过在一个LINQ查询中写明算法，可以非常容易地让它实现并行化。

第一步是将一个英语字典下载到HashSet中，便于高效查找：

```
if (!File.Exists ("WordLookup.txt")) // 包含大约150,000个单词
  new WebClient().DownloadFile (
    "http://www.albahari.com/ispell/allwords.txt", "WordLookup.txt");

var wordLookup = new HashSet<string> (
  File.ReadAllLines ("WordLookup.txt"),
  StringComparer.InvariantCultureIgnoreCase);
```

接下来，我们将使用单词查找来创建一份测试“文档”，其中包含一个一百万随机单词的数组。构建这个数组之后，我们将会引入一些拼写错误：

```
var random = new Random();
string[] wordList = wordLookup.ToArray();

string[] wordsToTest = Enumerable.Range (0, 1000000)
  .Select (i => wordList [random.Next (0, wordList.Length)])
  .ToArray();

wordsToTest [12345] = "woozsh";       // 引入一些拼写错误
wordsToTest [23456] = "wubsie";
```

现在，我们可以基于wordLookup测试wordsToTest，从而执行并行拼写检查。PLINQ让这些操作变得十分简单：

```
var query = wordsToTest
  .AsParallel()
  .Select ((word, index) => new IndexedWord { Word=word, Index=index })
  .Where (iword => !wordLookup.Contains (iword.Word))
  .OrderBy (iword => iword.Index);

foreach (var mistake in query)
  Console.WriteLine (mistake.Word + " - index = " + mistake.Index);

// 输出：
// woozsh - index = 12345
// wubsie - index = 23456
```

IndexedWord是我们自定义的一个结构，如下所示：

```
struct IndexedWord { public string Word; public int Index; }
```

断言所使用的wordLookup.Contains方法有助于查询并行化。

提示：我们可以使用一个匿名类型来代替IndexedWord，从而略微简化查询。但这可能损害性能，因为匿名类型（是类，因此也属于引用类型）会引起基于堆的分配以及后续垃圾回收的开销。

这点区别不足以胜过顺序查询，但借助并行查询，使用基于堆栈的分配具有明显优势。这是因为基于堆栈的分配是高度并行的（每个线程都有自己的堆栈），而所有线程必须竞争同一个堆——由一个内存管理器和垃圾回收器托管。

使用ThreadLocal<T>

下面对这个示例进行扩展，并行化随机测试单词列表本身的创建过程。我们将这个过程组织为一个LINQ查询，因此很简单。下面给出了顺序版本：

```
string[] wordsToTest = Enumerable.Range (0, 1000000)
  .Select (i => wordList [random.Next (0, wordList.Length)])
  .ToArray();
```

但是，对random.Next的调用不是线程安全的，因此这不像将AsParallel()方法插入查询那么简单。一种解决方案是编写一个函数来锁定random.Next调用，但这又会限制并发性。更好的选择是使用ThreadLocal<Random>（见前一章中的"线程本地存储"一节）为每个线程创建一个独立的Random对象。然后可以这样并行化查询：

```
var localRandom = new ThreadLocal<Random>
  ( () => new Random (Guid.NewGuid().GetHashCode()) );

string[] wordsToTest = Enumerable.Range (0, 1000000).AsParallel()
  .Select (i => wordList [localRandom.Value.Next (0, wordList.Length)])
  .ToArray();
```

在用于实例化Random对象的工厂函数中，我们传入Guid的哈希代码，从而确保如果在很短时间内创建了两个Random对象，它们将生成不同的随机数序列。

> **何时使用PLINQ**
>
> 找到现有的LINQ应用程序并并行化它们的想法是无法实现的，因为LINQ能解决的大多数问题往往执行速度很快，因此无法从并行化中获益。一种更好的方法是找出CPU密集的瓶颈，然后考虑通过LINQ的形式来表达。（这类重构还有一个作用，即LINQ往往会让代码变得更少，而且可读性更强。）
>
> PLINQ十分适于易并行问题。它还可以很好地处理结构化的阻塞任务，例如同时调用几个Web services（见"调用阻塞或I/O密集的函数"一节）。
>
> PLINQ不适于镜像制作，因为将数百万像素整理为一个输出序列将带来瓶颈。相反，直接将像素写到一个数组或未托管内存块中，然后使用Parallel类或任务并行管理多线程是更好的选择。（然而，使用我们在"优化PLINQ"中讨论的ForAll可以解决结果整理的问题。如果镜像处理算法适合LINQ，这样做才有意义。）

23.2.5 纯功能

因为PLINQ在并行线程上运行查询，必须注意不能执行非线程安全的操作。特别地，写入变量具有副作用，因此是非线程安全的：

```
// 下面的查询将每个元素与其位置相乘。
// 给定输入为Enumerable.Range(0,999)，应该输出正方形。
int i = 0;
var query = from n in Enumerable.Range(0,999).AsParallel() select n * i++;
```

使用锁定或InterLocked（见前一章中的"非阻塞同步"一节）可以让增加的操作变为线程安全

的，但问题依然存在，i不一定对应于输入元素的位置。给查询增加AsOrdered无法解决后面这个问题，因为AsOrdered只能确保元素的输出顺序与它们的处理顺序一致，而它实际上并未按照顺序处理。

相反应该改写查询，使用Select的索引版本：

```
var query = Enumerable.Range(0,999).AsParallel().Select ((n, i) => n * i);
```

为了实现最佳性能，从查询运算符调用的任何方法都应该是线程安全的，因为没有写入字段或属性的操作（没有副作用，或者也称为纯功能）。如果它们由于锁定是线程安全的，查询的并行化潜力将受到限制——限制因素是根据在该函数中所花费总体时间而划分的锁定的持续时间。

1. 设置并行级别

默认情况下，PLINQ会选择最适合处理器使用的并行级别。在AsParallel之后调用WithDegreeOfParallelism，可以重写并行级别：

```
...AsParallel().WithDegreeOfPallelism(4)...
```

有一种情况可能需要将并行数增加到内核数量之上，即I/O密集作业（例如，同时下载许多个网页）。然而，在C# 5和Framework 4.5中，任务组合器和异步函数提供了更简单高效的方法（参见第14章的“任务组合器”）。与Task不同，PLINQ在执行I/O密集作业时一定会阻塞线程和池化线程（后者更糟糕）。

2. 改变并行程度

在一个PLINQ查询中，只能调用一次WithDegreeOfParallelism。如果需要再次调用它，必须在查询中再次调用IsParallel()方法，从而强制合并和重新划分查询：

```
"The Quick Brown Fox"
  .AsParallel().WithDegreeOfParallelism (2)
  .Where (c => !char.IsWhiteSpace (c))
  .AsParallel().WithDegreeOfParallelism (3)  // 强制合并与划分
  .Select (c => char.ToUpper (c))
```

23.2.6 取消

如果正在foreach循环中使用PLINQ查询的结果，取消这个PLINQ查询很容易：只要跳出foreach循环，查询就将自动被取消，因为枚举器被隐式释放。

对于以转换、元素或聚合运算符终止的查询，可以通过取消令牌（见前一章中的“安全取消”中对取消令牌的讨论）从另一个线程取消它。要插入一个令牌，在AsParallel之后调用WithCancellation，并传入CancellationTokenSource对象的Token属性。然后，另一个线程就能够在令牌源头调用Cancel，这将在查询的使用者上抛出一个OperationCanceledException异常：

```
IEnumerable<int> million = Enumerable.Range (3, 1000000);

var cancelSource = new CancellationTokenSource();

var primeNumberQuery =
  from n in million.AsParallel().WithCancellation (cancelSource.Token)
  where Enumerable.Range (2, (int) Math.Sqrt (n)).All (i => n % i > 0)
  select n;
```

```
new Thread (() => {
                    Thread.Sleep (100);      // 100毫秒后
                    cancelSource.Cancel();   // 取消查询
                  }
           ).Start();
try
{
  // 开始运行查询：
  int[] primes = primeNumberQuery.ToArray();
  // 永远不会执行到这里，因为其他线程将取消
}
catch (OperationCanceledException)
{
  Console.WriteLine ("Query canceled");
}
```

PLINQ不会优先终止线程，因为这样做很危险（见前一章中的“中断与终止”一节）。相反，取消时，它会在结束查询之前等待每个工作者线程完成它当前的元素。这意味着，查询调用的任意外部方法都将运行完成。

23.2.7 优化PLINQ

1. 输出端优化

PLINQ的优点之一是，它能够方便地把来自并行工作的结果整理到一个输出序列中。但有时，结束时要做的全部工作就是让序列在每个元素上运行一些函数：

```
foreach (int n in parallelQuery)
  DoSomething (n);
```

如果这是实情，而且可以忽略元素被处理的顺序，使用PLINQ的ForAll方法可以提高效率。

ForAll方法在ParallelQuery的每个输出元素上运行一个委托。它正确关联到PLINQ的内部，省略了整理和枚举结果的步骤。下面是一个常见的示例：

```
"abcdef".AsParallel().Select (c => char.ToUpper(c)).ForAll (Console.Write);
```

图23-3显示了这个过程。

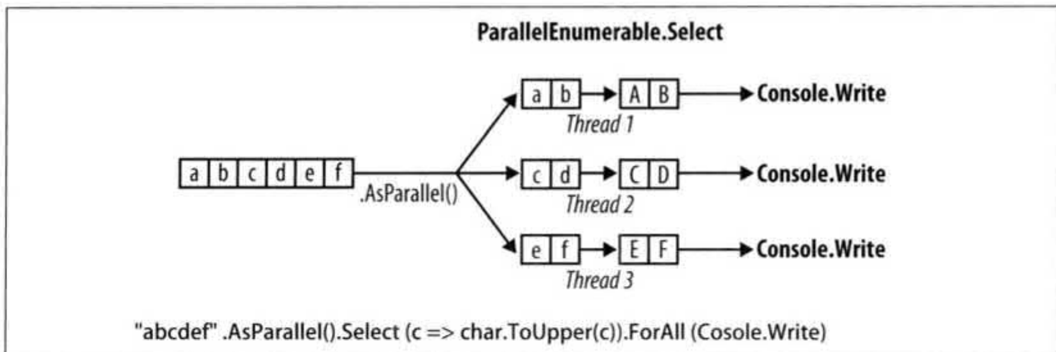


图23-3：PLINQ ForAll

提示：整理和枚举结果不是复杂的大型操作，因此当存在大量快速执行的输入元素时，ForAll优化能够获得最佳效果。

2. 输入端优化

PLINQ有三种用于给线程指派输入元素的划分策略：

策略	元素分配	相关性能
块划分	动态的	平均水平
范围划分	静态的	从极不好到极好
哈希划分	静态的	极不好

对于需要比较元素的查询运算符（GroupBy、Join、GroupJoin、Intersect、Except、Union和Distinct），PLINQ始终使用哈希划分策略。哈希划分效率相对较低，因为它必须预先计算每个元素的哈希代码，才能在同一线程上处理带有相同哈希代码的元素。如果觉得这样做太慢，唯一的选择就是调用AsSequential来禁用并行化。

对于其他所有的查询运算符，可以选择是使用范围还是块划分。默认地：

- 如果输入序列是有索引的（如果它是一个数组或者实现了Ilist<T>），PLINQ会选择范围划分策略。
- 否则，PLINQ会选择块划分策略。

概括地说，范围划分用于较长的序列，而且当每个元素花费的CPU时间大致相等时速度更快。否则，块划分的速度一般更快。

要强制使用范围划分：

- 如果查询以Enumerable.Range开始，使用ParallelEnumerable.Range代替它。
- 否则，在输入序列上简单地调用ToList或ToArray方法（这显然会增加性能成本，应该将这一点考虑在内）。

警告：ParallelEnumerable.Range不仅仅是调用Enumerable.Range(...).AsParallel()的一种快捷方式，而且通过激活范围划分改变了查询的性能。

要强制使用块划分，将输入序列封装在一次对Partitioner.Create（在System.Collection.Concurrent中）的调用中，如下所示：

```
int[] numbers = { 3, 4, 5, 6, 7, 8, 9 };
var parallelQuery =
  Partitioner.Create (numbers, true).AsParallel()
  .Where (...)
```

Partitioner.Create的第二个参数指示要对查询实现负载平衡，这是表明使用块划分策略的另一种方式。

块划分的工作方式是让每个工作者线程定期对输入序列中的小“块”元素进行处理（见图23-4）。PLINQ一开始分配非常小的块（每次一到两个元素），然后随着查询的进展增加数量：这确保小型序

列将被有效地并行化，而大型序列不会导致过度的双向传递。如果一个工作者恰好能够快速处理，它将获得更多元素。这个系统保持让每个线程同等忙碌（而且内核之间实现“平衡”）；唯一的缺点是，从共享输入序列获取元素需要同步（通常为一个独占锁定），而这可能导致一些开销与竞争。

范围划分省略了常规的输入端枚举，并预先给每个工作者分配了同等数量的元素，避免在输入序列上产生竞争。但如果有些线程对元素快速完成处理，它们就将无事可做，而余下的线程则不得不继续工作。如果使用范围划分，我们前面的质数计算器的性能可能会降低。范围划分适用的示例是计算前一千万个整数的平方根的总和：

```
ParallelEnumerable.Range (1, 10000000).Sum (i => Math.Sqrt (i))
```

ParallelEnumerable.Range返回一个ParallelQuery<T>，因此不需要随后调用AsParallel。

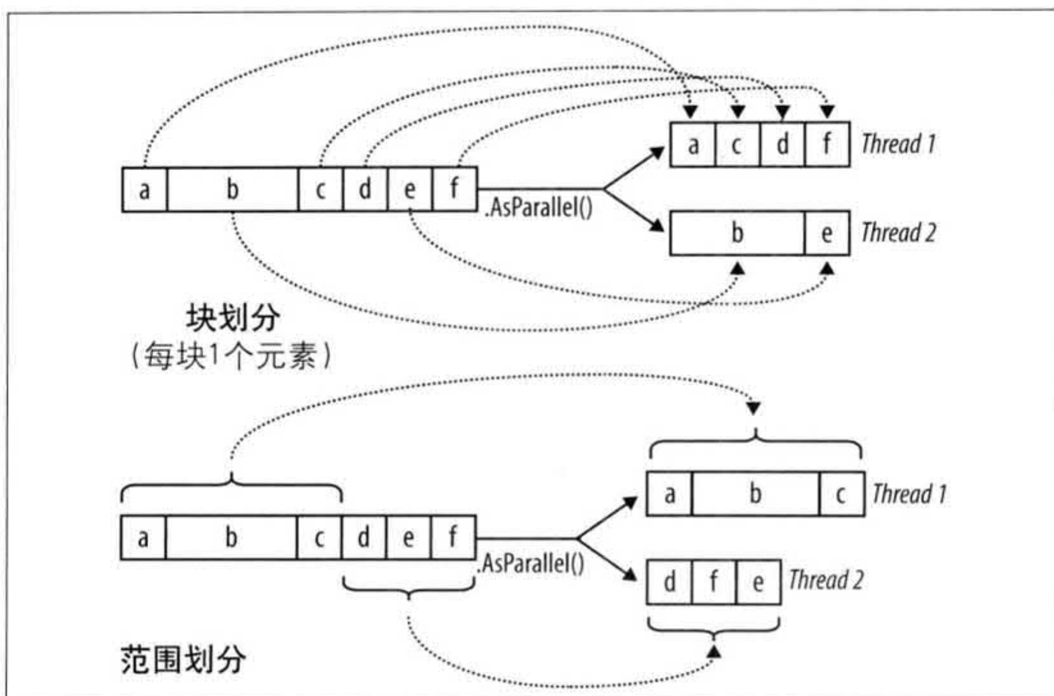


图23-4：块划分vs范围划分

> **提示：** 范围划分不一定以相邻块的形式分配元素范围——相反，它可能选择一种“条纹式”的策略。例如，如果存在两个工作者，一个工作者可能处理奇数编号的元素，而另一个工作者则处理偶数编号的元素。TakeWhile运算符使用条纹式策略，从而避免稍后不处理序列中元素的情况。

3. 优化自定义聚合

PLINQ在无需其他操作的情况下，可以有效地并行化Sum、Average、Min和Max运算符。但Aggregate运算符给PLINQ带来了特殊的挑战。正如第9章中介绍的那样，Aggregate执行自定义的聚合。例如，以下代码模仿Sum运算符对一个数字序列进行求和：

```
int[] numbers = { 1, 2, 3 };
int sum = numbers.Aggregate (0, (total, n) => total + n); // 6
```

我们在第9章中还看到了，对于没有种子的聚合而言，所提供的委托必须是联合的和可交换的。如果违反这条规则，PLINQ将给出错误的结果，因为它从输入序列得到了多个种子，从而同时聚合序列的几个部分。

显式提供种子的聚合看起来像是PLINQ的一种安全选择，但是，由于对单个种子的依赖，这些通常会顺序执行。为了减轻这个问题的影响，PLINQ提供了Aggregate的另一种重载，允许指定多个种子——或者更确切地说，一个种子工厂函数。每个线程都会执行这个函数来生成一个单独的种子，种子将变为一个用于本地聚合元素的线程本地累加器。

还必须提供一个函数来指示如何将本地累加器和主累加器结合起来。最后，这种Aggregate重载（有些毫无道理地）希望一个委托执行对结果的任意最终转换（之后在结果上运行一些函数，可以实现这个目的）。下面按照传递的顺序给出了4个委托：

seedFactory
: 返回一个新的本地累加器。

updateAccumulatorFunc
: 将一个元素聚合到一个本地累加器中。

combineAccumulatorFunc
: 将一个本地累加器与主累加器结合。

resultSelector
: 在最终结果上应用任意最终转换。

提示：在简单的应用场景中，可以指定一个种子值来代替种子工厂。当种子是希望转换的引用类型时，这种做法会失败，原因是同一个实例将被每个线程所共享。

举一个十分简单的示例，以下代码对numbers数组中的值求和：

```
numbers.AsParallel().Aggregate (
() => 0,                                      // seedFactory
 (localTotal, n) => localTotal + n,           // updateAccumulatorFunc
 (mainTot, localTot) => mainTot + localTot,   // combineAccumulatorFunc
 finalResult => finalResult)                          // resultSelector
```

这个示例是虚构的，有效使用更简单的方法（例如没有种子的聚合或者Sum运算符）时，可以获得相同的答案。举一个更加实际的示例，假设我们需要计算一个给定字符串中英语字母表中每个字母的出现频率。一种简单的顺序解决方案是：

```
string text = "Let's suppose this is a really long string";
var letterFrequencies = new int[26];
foreach (char c in text)
{
  int index = char.ToUpper (c) - 'A';
  if (index >= 0 && index <= 26) letterFrequencies [index]++;
};
```

提示： 输入文本可能很长的一个示例是基因序列分析。字母表将包含字母*a*、*c*、*g*和*t*。

要并行化这个解决方案，我们可以调用Parallel.ForEach（将在下一节中介绍）来代替foreach语句，但这需要自行处理共享数组上的并发性问题。围绕数组访问进行锁定几乎会消除并行化的可能性。

Aggregate提供一种精简的解决方案。在这个示例中，累加器是一个数组，就像前面示例中的letterFrequencies数组一样。下面给出了一个使用Aggregate的顺序版本：

```
int[] result =
  text.Aggregate (
    new int[26],                // 创建累加器
    (letterFrequencies, c) =>   // 将一个字母聚合到累加器中
    {
      int index = char.ToUpper (c) - 'A';
      if (index >= 0 && index <= 26) letterFrequencies [index]++;
      return letterFrequencies;
    });
```

而现在并行版本使用了PLINQ的特殊重载：

```
int[] result =
  text.AsParallel().Aggregate (
   () => new int[26],             // 创建一个新的本地累加器

   (localFrequencies, c) =>  // 聚合到本地累加器中
   {
     int index = char.ToUpper (c) - 'A';
     if (index >= 0 && index <= 26) localFrequencies [index]++;
     return localFrequencies;
   },
                                  // 聚合本地累加器到主累加器
   (mainFreq, localFreq) =>
     mainFreq.Zip (localFreq, (f1, f2) => f1 + f2).ToArray(),

   finalResult => finalResult       // 在结果上执行任意最终转换
 );
```

注意，本地累加函数修改了localFrequencies数组。执行这种优化的能力至关重要，而且是合理的，因为localFrequencies对于每个线程而言是本地的。

23.3 Parallel类

PFX通过Parallel类中的三个静态方法，提供了一种基本形式的结构化并行机制：

Parallel.Invoke

  并行执行一组委托。

Parallel.For

  执行C# for循环的并行等价循环。

Parallel.ForEach

  执行C# foreach循环的并行等价循环。

所有这三个方法都会阻塞直到所有工作完成为止。和PLINQ一样，在出现未处理的异常之后，余下的

工作者在它们当前的迭代之后停止，而异常将被抛回给调用者——封装在一个AggregateException中（见“处理AggregateException异常”一节）。

23.3.1 Parallel.Invoke方法

Parallel.Invoke方法并行执行一组Action委托，然后等待它们完成。该方法最简单的定义方式如下所示：

```
public static void Invoke (params Action[] actions);
```

下面说明了如何使用Parallel.Invoke方法一次性下载两个网页：

```
Parallel.Invoke (
 () => new WebClient().DownloadFile ("http://www.linqpad.net", "lp.html"),
 () => new WebClient().DownloadFile ("http://www.jaoo.dk", "jaoo.html"));
```

表面上看来，这就像是创建和等待两个Task对象（或异步委托）的一种捷径。但两者存在一个重要区别：如果传入一个包含一百万个委托的数组，Parallel.Invoke方法仍然能有效工作。这是因为它将大量元素划分为较小的块，然后指派给一些底层的Task，而不是为每个委托创建一个单独的Task。

和所有Parallel的方法一样，需要自行整理结果。这意味着需要时刻注意线程安全性。例如，以下代码是非线程安全的：

```
var data = new List<string>();
Parallel.Invoke (
 () => data.Add (new WebClient().DownloadString ("http://www.foo.com")),
 () => data.Add (new WebClient().DownloadString ("http://www.far.com")));
```

围绕增加列表进行锁定可以解决这个问题，但如果存在大量快速执行的委托，锁定会造成瓶颈。一种更好的解决方案是使用线程安全的集合，我们将稍后介绍。在这个示例中，使用ConcurrentBag是最理想的。

还可以将Parallel.Invoke重载为接受一个ParallelOptions对象：

```
public static void Invoke (ParallelOptions options,
                           params Action[] actions);
```

通过ParallelOptions可以插入取消令牌、限制最大并发性，并指定一个自定义的任务调度器。当执行的任务多于（大概）内核时，就需要用到取消令牌：取消时，任何未启动的委托都将被放弃。但所有执行中的委托都将继续完成。见“取消”一节以了解如何使用取消令牌。

23.3.2 Parallel.For和Parallel.ForEach

Parallel.For和Parallel.ForEach分别等价于C#中的for和foreach循环，但每次迭代是并行而非顺序执行。下面给出了它们最简单的形式：

```
public static ParallelLoopResult For (
  int fromInclusive, int toExclusive, Action<int> body)

public static ParallelLoopResult ForEach<TSource> (
  IEnumerable<TSource> source, Action<TSource> body)
```

以下顺序执行的for循环：

```
for (int i = 0; i < 100; i++)
  Foo (i);
```

被并行化为下面这样：

```
Parallel.For (0, 100, i => Foo (i));
```

或者更加简单的形式：

```
Parallel.For (0, 100, Foo);
```

而以下顺序执行的foreach循环：

```
foreach (char c in "Hello, world")
  Foo (c);
```

被并行化为下面这样：

```
Parallel.ForEach ("Hello, world", Foo);
```

举一个实际的示例，如果我们导入System.Security.Cryptography命名空间，就可以并行生成6个公共/私有的键对字符串，如下所示：

```
var keyPairs = new string[6];

Parallel.For (0, keyPairs.Length,
              i => keyPairs[i] = RSA.Create().ToXmlString (true));
```

和使用Parallel.Invoke一样，可以给Parallel.For和Parallel.ForEach提供大量的工作项，这些工作项将被有效地划分为一些任务。

提示： 后一个查询也可以使用PLINQ来完成：

```
string[] keyPairs =
  ParallelEnumerable.Range (0, 6)
  .Select (i => RSA.Create().ToXmlString (true))
  .ToArray();
```

---

1. 外部循环与内部循环

Parallel.For和Parallel.ForEach通常在外部循环上效果最好。这是因为使用前者时，要提供较大的工作块进行并行化，这将减少管理开销。同时并行化内部循环与外部循环一般没有必要。在下面的示例中，我们通常需要100个以上的内核才能从内部实现并行化：

```
Parallel.For (0, 100, i =>
{
  Parallel.For (0, 50, j => Foo (i, j));     // 对于内部循环，顺序执行的效果更好
});
```

2. 索引化的Parallel.ForEach

有时，循环的迭代索引很有用处。这在顺序执行的foreach循环中很容易实现：

```
int i = 0;
foreach (char c in "Hello, world")
  Console.WriteLine (c.ToString() + i++);
```

然而，增加一个共享变量的值在并行上下文中不是线程安全的。必须使用以下版本的ForEach作为代替：

```
public static ParallelLoopResult ForEach<TSource> (
  IEnumerable<TSource> source, Action<TSource,ParallelLoopState,long> body)
```

我们将在下一节中介绍ParallelLoopState。先介绍Action的第三个类型为long的类型参数，它用于指示循环索引：

```
Parallel.ForEach ("Hello, world", (c, state, i) =>
{
   Console.WriteLine (c.ToString() + i);
});
```

为了在实际环境中应用它，我们仍以前面使用PLINQ编写的拼写检查器为例。下面的代码加载一个字典和一个包含要测试的一百万个单词的数组：

```
if (!File.Exists ("WordLookup.txt"))          // 包含大约150,000个单词
  new WebClient().DownloadFile (
    "http://www.albahari.com/ispell/allwords.txt", "WordLookup.txt");

  var wordLookup = new HashSet<string> (
    File.ReadAllLines ("WordLookup.txt"),
    StringComparer.InvariantCultureIgnoreCase);

  var random = new Random();
  string[] wordList = wordLookup.ToArray();

  string[] wordsToTest = Enumerable.Range (0, 1000000)
    .Select (i => wordList [random.Next (0, wordList.Length)])
    .ToArray();

wordsToTest [12345] = "woozsh";       // 引入一些拼写错误
wordsToTest [23456] = "wubsie";
```

我们可以使用索引化的Parallel.ForEach来执行拼写检查，如下所示：

```
var misspellings = new ConcurrentBag<Tuple<int,string>>();

Parallel.ForEach (wordsToTest, (word, state, i) =>
{
  if (!wordLookup.Contains (word))
    misspellings.Add (Tuple.Create ((int) i, word));
});
```

注意，我们必须将结果整理到一个线程安全的集合中，这是与使用PLINQ时相比之下的缺点。胜过PLINQ之处在于，我们可以避免应用索引化的Select查询运算符，但比索引化ForEach的效率要低。

3. ParallelLoopState：提前跳出循环

因为并行For或ForEach中的循环体是一个委托，所以无法使用break语句提前退出循环。相反，必须调用ParallelLoopState对象上的Break或Stop方法：

```
public class ParallelLoopState
{
  public void Break();
  public void Stop();

  public bool IsExceptional { get; }
  public bool IsStopped { get; }
  public long? LowestBreakIteration { get; }
  public bool ShouldExitCurrentIteration { get; }
}
```

获得个ParallelLoopState对象很容易：所有For和ForEach都可以被重载为接受类型Action<TSource,ParallelLoopState>的循环体。因此，要并行化下面这段代码：

```
foreach (char c in "Hello, world")
  if (c == ',')
    break;
  else
    Console.Write (c);
```

需要这样做：

```
Parallel.ForEach ("Hello, world", (c, loopState) =>
{
  if (c == ',')
    loopState.Break();
  else
    Console.Write (c);
});

// 输出：Hlloe
```

从输出可以看出，循环体的完成方式是随机的。除了这个区别，调用Break会生成与顺序执行循环相同的元素。这个示例将始终以某种顺序输出字母*H*、*e*、*l*、*l*和*o*。相比之下，调用Stop代替Break会强制在当前迭代完成之后结束所有线程。在示例中，如果另一个线程落在后面，调用Stop将输出字母*H*、*e*、*l*、*l*和*o*的一个子集。当找到要查找的内容时，或者当出现错误而且不想查看结果时，便可调用Stop。

提示：Parallel.For和Parallel.ForEach方法返回一个ParallelLoopResult对象，该对象公开属性IsCompleted和LowestBreakIteration。这些属性表明循环是否结束，如果没有，指明循环将在哪个周期中断。

如果LowestBreakIteration属性返回null，表示在循环上调用了Stop（而非Break）。

如果循环体很长，当遇到一个Break或Stop时，可能想让其他线程中途退出。在代码中轮询ShouldExitCurrentIteration属性可以实现。在调用Stop或Break之后，此属性将立即变为true。

提示：ShouldExitCurrentIteration在取消请求或者循环中抛出异常后也会变为true。

IsExceptional用于指示另一个线程上是否出现异常。任何未处理的异常都将导致循环在每个线程完成当前迭代之后停止。为了避免这种情况，必须在代码中显式地处理异常。

4. 优化本地值

Parallel.For和Parallel.ForEach方法均提供一组接受TLocal普通类型参数的重载。这些重载方法用于帮助优化迭代密集型循环的数据整理。最简单的形式如下所示：

```
public static ParallelLoopResult For <TLocal> (
  int fromInclusive,
  int toExclusive,
  Func <TLocal> localInit,
  Func <int, ParallelLoopState, TLocal, TLocal> body,
  Action <TLocal> localFinally);
```

这些方法在实际中很少用到，因为它们的应用场景大多是在PLINQ中。

问题基本上是这样：假设我们需要对1～10,000,000之间数字的平方根求和。计算一千万个平方根这项工作很容易并行化，但对它们的值求和有点麻烦，原因是必须围绕更新总和进行锁定：

```
object locker = new object();
double total = 0;
Parallel.For (1, 10000000,
              i => { lock (locker) total += Math.Sqrt (i); });
```

并行化不仅抵消了获得一千万个锁定的开销，而且带来阻塞的开销。

实际上不需要一千万个锁。假设有一个志愿者小组收集了大量垃圾。如果所有工作者共享一个垃圾回收器，移动与竞争将使得收集过程异常低效。明显的解决方案是让每个工作者都有一个私有的或“本地的”垃圾回收器，它们偶尔将自己的垃圾卸空到主垃圾回收器中。

For和Foreach的TLocal版本工作方式完全相同。志愿者是内部的工作者线程，而本地值代表了本地的垃圾回收器。为了让Parallel来完成这项工作，必须提供两个另外的委托，分别用于指示：

1. 如何初始化一个新的本地值。
2. 如何将本地聚合与主值相结合。

另外，主体委托不返回void，而应该返回本地值的新委托。下面是经过重构的示例：

```
object locker = new object();
double grandTotal = 0;

Parallel.For (1, 10000000,

  () => 0.0,                        // 初始化本地值

  (i, state, localTotal) =>         // 主体委托。注意，它返回一个新的本地总和
     localTotal + Math.Sqrt (i),

  localTotal =>                     // 将本地值与主值相加
    { lock (locker) grandTotal += localTotal; }
);
```

我们仍然必须进行锁定，但只围绕将本地值聚合到总和，这让整个过程更加高效。

提示： 如前所述，PLINQ通常适用于这些应用场景。可以简单地使用PLINQ来并行化我们的示例，例如：

```
ParallelEnumerable.Range (1, 10000000)
                  .Sum (i => Math.Sqrt (i))
```

（注意，我们使用ParallelEnumerable强制实现范围划分。这可以提高这个示例的性能，因为所有数字的处理时间都是相等的。）

在更为复杂的应用场景中，可以使用LINQ的Aggregate运算符来代替Sum。如果提高一个本地的种子工厂，情况就有点类似于给一个本地值函数提供Parallel.For。

23.4 任务并行

任务并行是使用PFX进行并行化的最底层方法。工作在这一层次的类定义在System.Threading.Tasks命名空间中，这些类包括：

类	用途
Task	管理一个工作单元
Task<TResult>	管理一个带有返回值的工作单元
TaskFactory	创建任务
TaskFactory<TResult>	创建任务或者有相同返回值的延续任务
TaskScheduler	管理任务调度
TaskCompletionSource	手动控制一个任务的工作流

在第14章将介绍任务的基础概念；本节将介绍用于实现并行编程的一些高级任务特性。它们是：

- 调整任务调度
- 建立各个任务启动的父/子关系
- 延续的高级用法
- TaskFactory

警告：通过任务并行库可以用最小的开销来创建成百上千的任务，但是如果创建上百万个任务，就必须分割这些任务到更大的工作单元，以保持效率。Parallel类和PLINQ可以自动实现这一点。

提示：Visual Studio 2010提供一个新窗口用于监控任务（“调试”→“窗口”→“并行任务”），相当于用于任务的线程窗口。并行堆栈窗口也有一种针对任务的特殊模式。

23.4.1 创建与启动任务

和第14章介绍的一样，Task.Run可以创建和启动一个Task或Task<TResult>。这个方法实际上是Task.Factory.StartNew的简写方法，只是后者有更多的重载版本，所以也更加灵活一些。

1. 指定状态对象

Task.Factory.StartNew可以指定一个状态对象，它会传递给目标方法。目标方法的签名必须包含一个object类型的参数：

```
static void Main()
{
  var task = Task.Factory.StartNew (Greet, "Hello");
  task.Wait(); // 等待任务完成
}

static void Greet (object state) { Console.Write (state); } // Hello
```

这样可以避免执行lambda表达式（调用Greet）所需要的闭包开销。这是一个微优化，而且实践中很少使用，所以可以更好地使用状态对象，即给任务指定一个有意义的名称。然后使用AsyncState属性查询它的名称：

```
static void Main()
{
  var task = Task.Factory.StartNew (state => Greet ("Hello"), "Greeting");
  Console.WriteLine (task.AsyncState);        // Greeting
  task.Wait();
}
```

提示： Visual Studio在并行任务窗口中显示了每个任务的AsyncState属性，因此取一个有意义的名称可以让调试变得轻松许多。

并行编程

2. TaskCreationOptions

调用StartNew方法（或实例化Task）时，指定一个TaskCreationOptions枚举可以调整任务的执行。TaskCreationOptions是一个标志枚举，包含以下（可组合的）值：

```
LongRunning
PreferFairness
AttachedToParent
```

LongRunning指示调度器为任务指定一个线程，而正如我们在第14章介绍的，这样做对于I/O密集任务和长任务很有好处。如果不这样做，则会迫使短任务等待过长的时间才能进入调度队列。

PreferFairness会让调度器保证任务的调度顺序保持不变。它通常不会这样做，因为它在内部会使用本地作业抢断队列对任务调度进行优化——这个优化可以在不增加过载的前提下创建子任务，从而不需要出现另一个作业队列。子任务是通过指定AttachedToParent而创建的。

3. 子任务

当一个任务启动另一个任务时，可以选择通过指定TaskCreationOptions.AttachedToParent来建立父子关系：

```
Task parent = Task.Factory.StartNew (() =>
{
  Console.WriteLine ("I am a parent");

  Task.Factory.StartNew (() =>        // 分离的任务
  {
    Console.WriteLine ("I am detached");
  });

  Task.Factory.StartNew (() =>        // 子任务
```

```
    {
      Console.WriteLine ("I am a child");
    }, TaskCreationOptions.AttachedToParent);
});
```

子任务有一些特殊，因为必须等到所有子任务结束，父任务才会结束。而在这期间，某些子任务可能会抛出异常：

```
TaskCreationOptions atp = TaskCreationOptions.AttachedToParent;
var parent = Task.Factory.StartNew (() =>
{
Task.Factory.StartNew (() => // 孩子
{
Task.Factory.StartNew (() => { throw null; }, atp); // 孙子
}, atp);
});

// 下面的调用会抛出一个NullReferenceException异常
// （它封装在嵌套的AggregateExceptions异常中）：
parent.Wait();
```

如果子任务是一个延续，那么这样做非常有用，马上将会介绍这一点。

23.4.2 等待多个任务

在第14章介绍过，调用一个任务的Wait方法，或者访问它的Result属性（如果它是Task<TResult>），都可以等待一个任务。我们也可以同时等待多个任务，通过使用静态方法Task.WaitAll（等待所有指定任务完成）和Task.WaitAny（只等待其中一个任务完成）。

WaitAll类似于轮流等待每一个任务，但是它的效率更高，因为它（最多）只需要切换一次环境。此外，如果有一个或多个任务执行未处理的异常，那么WaitAll仍然会等待所有任务完成，然后再抛出一个AggregateException，它会累加每一个出错任务的异常（这是AggregateException真正发挥作用的时候）。这相当于：

```
// 假设有任务t1、t2和t3：
var exceptions = new List<Exception>();
try { t1.Wait(); } catch (AggregateException ex) { exceptions.Add (ex); }
try { t2.Wait(); } catch (AggregateException ex) { exceptions.Add (ex); }
try { t3.Wait(); } catch (AggregateException ex) { exceptions.Add (ex); }
if (exceptions.Count > 0) throw new AggregateException (exceptions);
```

调用WaitAny，相当于等待一个ManualResetEventSlim，每一个任务在完成时都会触发这个事件。

除了超时时间，还可以给Wait方法传入一个取消（cancellation）令牌：取消等待，而非取消任务本身。

23.4.3 取消任务

在启动一个任务时，可以选择传入一个取消令牌。然而，如果通过该令牌执行取消操作，那么任务本身会进入“已撤销（Canceled）”状态：

```
var cts = new CancellationTokenSource();
CancellationToken token = cts.Token;
cts.CancelAfter (500);
```

```
Task task = Task.Factory.StartNew (() =>
{
  Thread.Sleep (1000);
  token.ThrowIfCancellationRequested(); // 检查取消请求
}, token);

try { task.Wait(); }
catch (AggregateException ex)
{
  Console.WriteLine (ex.InnerException is TaskCanceledException);   // True
  Console.WriteLine (task.IsCanceled);                               // True
  Console.WriteLine (task.Status);                                   // Canceled
}
```

TaskCanceledException是OperationCanceledException的一个子类。如果想要显式抛出一个OperationCanceledException（而不是调用token.ThrowIf CancellationRequested），那么必须将取消令牌传给OperationCanceledException的构造方法。如果不能这样做，那么任务就不会进入TaskStatus.Canceled状态，也不会触发OnlyOnCanceled延续。

如果一个任务还没启动就已取消，不会为它安排调度，而是立即在该任务上抛出一个OperationCanceledException异常。

由于其他API也能识别取消令牌，可以将它们传入到其他结构中，而取消将无缝传播：

```
var cancelSource = new CancellationTokenSource();
CancellationToken token = cancelSource.Token;

Task task = Task.Factory.StartNew (() =>
{
  // 将取消令牌传入到一个PLINQ查询中：
  var query = someSequence.AsParallel().WithCancellation (token)...
  ... enumerate query ...
});
```

在这个示例中，调用cancelSource上的Cancel方法将取消PLINQ查询，这将在任务体上抛出一个OperationCanceledException异常，然后取消任务。

提示： 可以传入Wait和CancelAndWait这类方法中的取消令牌取消等待操作，而不是任务本身。

23.4.4 延续任务

有时可以在一个任务完成（或失败）之后立即启动另一个任务，Task类上的ContinueWith方法就是执行这件操作的：

```
Task task1 = Task.Factory.StartNew (() => Console.Write ("antecedant.."));
Task task2 = task1.ContinueWith (ant => Console.Write ("..continuation"));
```

当task1（前导任务）结束、失败或被取消时，task2（延续任务）就会自动启动。（如果task1在第二行代码运行之前已经完成，task2将立即被执行。）传递给延续任务的Lambda表达式的ant参数是一个对前导任务的引用。ContinueWithitself会返回一个任务，从而可以轻松地添加更多的延续。

默认情况下，先行和延续任务可以在不同线程上执行。在调用ContinueWith时指定TaskContinuationOptions.ExecuteSynchronously，就可以强制它们在同一个线程上执行：这样可以在非常精细的延续上通过减少转折来提升性能。

1. 延续任务与Task<TResult>

和普通任务一样，延续任务可以是Task<TResult>类型的返回数据。在下面的示例中，我们使用一系列链式任务来计算Math.Sqrt(8*2)，然后写出结果：

```
Task.Factory.StartNew<int> (() => 8)
  .ContinueWith (ant => ant.Result * 2)
  .ContinueWith (ant => Math.Sqrt (ant.Result))
  .ContinueWith (ant => Console.WriteLine (ant.Result));   // 4
```

我们的示例是有意设计成这么简单的。在实际应用中，这些Lambda表达式将调用计算密集的函数。

2. 延续任务与异常

通过查询前导任务的Exception属性，延续任务可以知道前导任务是否已失败——或者只是简单地调用Result/Wait并缓存相关的AggregateException。如果前导任务失败，而延续任务无法查询前导任务的Exception属性（而且前导任务没有被等待），该异常被认为是未处理的，而且稍后对该任务进行垃圾回收时会触发静态TaskScheduler.UnobservedTaskException事件。

一种安全的模式是重新抛出前导任务的异常。只要有任务在等待延续任务，该异常就会被传播并重新抛给等待的任务：

```
Task continuation = Task.Factory.StartNew (() => { throw null; })
                                .ContinueWith (ant =>
  {
    if (ant.Exception != null) throw ant.Exception;
    // 继续执行...
  });

continuation.Wait();          // 现在异常被抛回给调用者
```

另一种处理异常的方式是为异常和非异常的结果指定不同的延续任务，使用TaskContinuationOptions可以做到这一点：

```
Task task1 = Task.Factory.StartNew (() => { throw null; });

Task error = task1.ContinueWith (ant => Console.Write (ant.Exception),
                                 TaskContinuationOptions.OnlyOnFaulted);

Task ok = task1.ContinueWith (ant => Console.Write ("Success!"),
                              TaskContinuationOptions.NotOnFaulted);
```

这种模式在与子任务联合时特别有用，我们将在后面介绍。

以下扩展方法“吞下”了一个任务的未处理异常：

```
public static void IgnoreExceptions (this Task task)
{
  task.ContinueWith (t => { var ignore = t.Exception; },
    TaskContinuationOptions.OnlyOnFaulted);
}
```

增加代码记录异常可以改进此方法。下面给出了这个方法的用法：

```
Task.Factory.StartNew (() => { throw null; }).IgnoreExceptions();
```

3. 延续任务与子任务

延续任务有一个强大的特性，即只有所有子任务都完成之后它们才会开始（见图23-5）。这时，子任务抛出的任何异常都将被封送给延续任务。

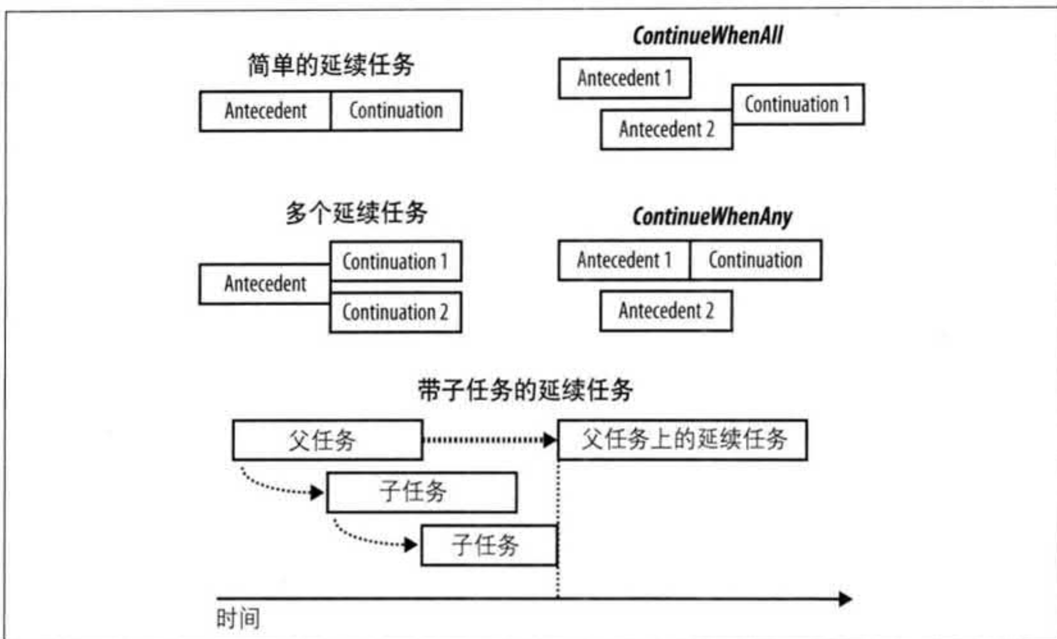


图23-5：延续任务

在下面的示例中，我们启动了3个子任务，每个子任务都抛出一个NullReferenceException异常。然后，通过父任务上的一个延续任务一次性捕获了所有这些异常：

```
TaskCreationOptions atp = TaskCreationOptions.AttachedToParent;
Task.Factory.StartNew (() =>
{
  Task.Factory.StartNew (() => { throw null; }, atp);
  Task.Factory.StartNew (() => { throw null; }, atp);
  Task.Factory.StartNew (() => { throw null; }, atp);
})
.ContinueWith (p => Console.WriteLine (p.Exception),
                    TaskContinuationOptions.OnlyOnFaulted);
```

4. 有条件的延续任务

默认地，延续任务的调度是无条件的——无论前导任务是完成、抛出异常还是被取消。通过

TaskContinuationOptions枚举中包含的一组（可组合）标志可以改变这种行为。控制有条件延续任务的3个内核标志包括：

```
NotOnRanToCompletion = 0x10000,
NotOnFaulted = 0x20000,
NotOnCanceled = 0x40000,
```

这些标志的值是递减的，即应用得越多，延续任务执行的可能性越小。方便起见，还提供了以下组合好的值：

```
OnlyOnRanToCompletion = NotOnFaulted | NotOnCanceled,
OnlyOnFaulted = NotOnRanToCompletion | NotOnCanceled,
OnlyOnCanceled = NotOnRanToCompletion | NotOnFaulted
```

组合所有Not*标志（NotOnRanToCompletion、NotOnFaulted、NotOnCanceled）是没有意义的，因为这会导致延续任务始终被取消。

“RanToCompletion”表示前导任务成功执行，既没有取消也不存在未处理的异常。

“Faulted”表示前导任务上抛出了一个未处理的异常。

“Canceled”表示以下两种情况之一：

- 前导任务通过其取消令牌被取消。换句话说，前导任务上抛出了一个OperationCanceledException异常，而前导任务的CancellationToken属性与启动前导任务时传递给它的值相匹配。
- 前导任务被隐式取消，因为它不满足一个有条件的延续断言。

当由于这些标志不执行延续任务时，延续任务并没有被遗忘或丢弃，而是被取消了，这一点必须要掌握。这意味着，延续任务本身的任意延续任务都将运行，除非使用NotOnCanceled对它们进行断言。例如，考虑下面的代码：

```
Task t1 = Task.Factory.StartNew (...);

Task fault = t1.ContinueWith (ant => Console.WriteLine ("fault"),
                              TaskContinuationOptions.OnlyOnFaulted);

Task t3 = fault.ContinueWith (ant => Console.WriteLine ("t3"));
```

实际上，t3将始终获得调度，即使t1没有抛出异常（见图23-6）。这是因为，如果t1成功，fault任务将被取消，t3上也就不会有任何延续限制，那么t3将无条件执行。

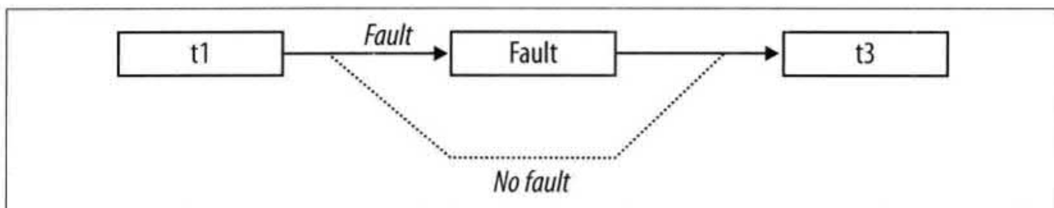


图23-6：有条件的延续任务

如果希望只有fault实际运行时，t3才会执行，必须这样做：

```
Task t3 = fault.ContinueWith (ant => Console.WriteLine ("t3"),
                              TaskContinuationOptions.NotOnCanceled);
```

另外，我们还可以指定OnlyOnRanToCompletion，区别在于如果fault中抛出异常，t3不会执行。

5. 具有多个前导任务的延续任务

延续任务的另一项特性是，可以根据多个前导任务的完成情况调度它们的执行。ContinueWhenAll当所有前导任务都完成时才调度执行，而 ContinueWhenAny当只要有一个前导任务完成时就调度执行。这两个方法都定义在TaskFactory类中：

```
var task1 = Task.Factory.StartNew (() => Console.Write ("X"));
var task2 = Task.Factory.StartNew (() => Console.Write ("Y"));
```

可以调度延续，使这两个任务按照下面的方式结束执行：

```
var continuation = Task.Factory.ContinueWhenAll (
  new[] { task1, task2 }, tasks => Console.WriteLine ("Done"));
```

下面是使用任务组合器WhenAll实现的相同结果：

```
var continuation = Task.WhenAll (task1, task2)
                       .ContinueWith (ant => Console.WriteLine ("Done"));
```

6. 前导任务上的多个延续任务

在同一任务上多次调用ContinueWith，将在一个前导任务上创建多个延续任务。当前导任务结束时，所有延续任务都将一起启动（除非指定了TaskContinuationOptions.ExecuteSynchronously，这时延续任务将按顺序执行）。

以下代码等待一秒种，然后写出“XY”或“YX”：

```
var t = Task.Factory.StartNew (() => Thread.Sleep (1000));
t.ContinueWith (ant => Console.Write ("X"));
t.ContinueWith (ant => Console.Write ("Y"));
```

23.4.5 任务调度器与用户界面

任务调度器负责给线程分配任务，由抽象类TaskScheduler表示。Framework提供了2个具体实现：与CLR线程池协同工作的默认调度器和同步环境调度器。后者（主要）用于处理WPF和Windows Forms的线程模型，它规定UI元素和控件只能从创建它们的线程访问（参见第14章的“富客户端应用的线程处理”）。有了任务调度器，我们就可以在这个环境上执行一个任务或延续：

```
// Suppose we are on a UI thread in a Windows Forms / WPF application:
_uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();
```

假设Foo是一个计算密集型方法，它会返回一个字符串，而lblResult是一个WPF或Windows Forms标签，那么可以按照下面的方式安全地在操作完成后更新标签内容：

```
Task.Run (() => Foo())
.ContinueWith (ant => lblResult.Content = ant.Result, _uiScheduler);
```

当然，更多的是使用C#的异步函数来实现相同的效果。

也可以编写我们自定义的任务调度器（绕过TaskScheduler），但这只适用于非常特殊的应用场景。对于自定义的调度，更加常用的是我们在下面讲到的TaskCompletionSource。

23.4.6 TaskFactory

调用Task.Factory时，实际调用的是Task上的一个静态属性，它返回一个默认的TaskFactory对象。任务工厂的用途是创建任务，明确分为三种任务：

- “普通”任务（通过StartNew）
- 具有多个前导任务的延续任务（通过ContinueWhenAll和ContinueWhenAny）
- 封装符合异步编程模型的方法的任务（通过FromAsync；见14.7节）

另一种创建任务的方法是初始化一个Task，然后调用Start。然而，这样只是创建了一个“普通”任务，而不是延续。

1. 创建自己的任务工厂

TaskFactory不是一个抽象工厂：可以真正地实例化这个类，而且当需要使用相同（非标准）的TaskCreationOptions、TaskContinuationOptions或TaskScheduler值重复创建任务时需要这么做。例如，如果想重复创建长期运行的有父任务的任务，可以像下面这样创建一个自定义工厂：

```
var factory = new TaskFactory (
  TaskCreationOptions.LongRunning | TaskCreationOptions.AttachedToParent,
  TaskContinuationOptions.None);
```

接下来，只要在工厂上调用StartNew就可以创建任务了：

```
Task task1 = factory.StartNew (Method1);
Task task2 = factory.StartNew (Method2);
...
```

调用ContinueWhenAll和ContinueWhenAny时选择使用自定义的延续任务。

2. 创建任务工厂

TaskFactory并不是一个抽象工厂：这个类可以实例化，如果要使用相同（非标准）TaskCreationOptions、TaskContinuationOptions或TaskScheduler值重复创建任务，那么它很好用。例如，如果想要重复创建长时间运行的父任务，那么可以按照下面的方式创建一个自定义工厂：

```
var factory = new TaskFactory (
TaskCreationOptions.LongRunning | TaskCreationOptions.AttachedToParent,
TaskContinuationOptions.None);
```

然后，调用工厂的StartNew，就可以轻松创建任务：

```
Task task1 = factory.StartNew (Method1);
Task task2 = factory.StartNew (Method2);
...
```

调用ContinueWhenAll和ContinueWhenAny时，自定义的延续选项就会生效。

23.5 处理AggregateException异常

正如我们看到的那样，PLINQ、Parallel类和Task自动将异常封送给使用者。这是必不可少的，考虑下面的LINQ查询，它在首次迭代时抛出一个DivideByZeroException异常：

```
try
{
  var query = from i in Enumerable.Range (0, 1000000)
              select 100 / i;
  ...
}
catch (DivideByZeroException)
{
  ...
}
```

如果我们要求PLINQ并行化这个查询而且忽略异常处理，很可能在一个单独的线程上抛出一个DivideByZeroException异常，从而不执行catch块并导致应用程序停止。

因此，异常将被自动捕获并重抛给调用者。但是，它不像捕获DivideByZeroException异常那么简单。因为这些库利用了很多线程，实际上有可能同时抛出两个以上的异常。为了确保报告所有的异常，因此将这些异常封装在一个AggregateException容器中，该容器公开的InnerExceptions属性中包含了所有捕获的异常：

```
try
{
  var query = from i in Enumerable.Range (0, 1000000)
              select 100 / i;
  // Enumerate query
  ...
}
catch (AggregateException aex)
{
  foreach (Exception ex in aex.InnerExceptions)
    Console.WriteLine (ex.Message);
}
```

提示：PLINQ和Parallel类在遇到第一个异常时就会结束查询或循环的执行，不会处理任何进一步的元素或循环体。但在当前周期完成之前，可能抛出更多的异常。AggregateException中的第一个异常在InnerExceptions属性中可见。

Flatten和Handle方法

AggregateException类提供一些方法来简化异常处理：Flatten和Handle。

1. Flatten

AggregateException经常包含其他的此类异常，一个示例是当子任务抛出异常时。通过调用Flatten方法，可以消除任意层次的嵌套以简化处理。这个方法返回一个AggregateException对象，其中包含一个简单的内部异常列表：

```
catch (AggregateException aex)
{
  foreach (Exception ex in aex.Flatten().InnerExceptions)
    myLogWriter.LogException (ex);
}
```

2. Handle

有时需要只捕获特定类型的异常，并重抛其他类型的异常。AggregateException对象上的Handle方法为此提供了一种快捷方式。它接受一个异常断言，并在每个内部异常上运行此断言：

```
public void Handle (Func<Exception, bool> predicate)
```

如果断言返回true，它认为该异常是“已处理的”。当委托已经在所有异常上运行之后，会出现以下情况：

- 如果所有异常都是已处理的（委托返回true），异常不会重抛。
- 如果存在任何委托返回false的异常（未处理的），就会构造一个新的AggregateException对象来包含这些异常，然后重抛这个对象。

例如，下面的代码最终重抛了包含NullReferenceException异常的另一个AggregateException对象：

```
var parent = Task.Factory.StartNew (() =>
{
  // 将使用3个子任务同时抛出3个异常：
  int[] numbers = { 0 };

  var childFactory = new TaskFactory
    (TaskCreationOptions.AttachedToParent, TaskContinuationOptions.None);
    childFactory.StartNew (() => 5 / numbers[0]);     // 除零
    childFactory.StartNew (() => numbers [1]);        // 索引超出范围
    childFactory.StartNew (() => { throw null; });    // 空引用
  });

  try { parent.Wait(); }
  catch (AggregateException aex)
  {
    aex.Flatten().Handle (ex =>       // 注意仍然需要调用Flatten
   {
      if (ex is DivideByZeroException)
      {
        Console.WriteLine ("Divide by zero");
        return true;                                    // 这个异常是已处理的
      }
      if (ex is IndexOutOfRangeException)
      {
        Console.WriteLine ("Index out of range");
        return true;                                    // 这个异常是未处理的
      }
      return false; //所有其他异常将被重抛
    });
  }
```

23.6 并发集合

Framework 4.0在System.Collections.Concurrent命名空间中提供了一组新的集合。所有这些集合都完全是线程安全的：

并发集合	非并发等价集合
ConcurrentStack<T>	Stack<T>
ConcurrentQueue<T>	Queue<T>
ConcurrentBag<T>	无
BlockingCollection<T>	无
ConcurrentDictionary<TKey,TValue>	Dictionary<TKey,TValue>

当需要线程安全的集合时，有时候可以在一般的多线程中使用并发集合。但这里要注意：

- 并发集合针对并行编程进行了调整。只有在高度并发的应用场景中，传统集合的性能才能胜过它们。
- 线程安全的集合不能确保使用它的代码也是线程安全的（见前一章的“线程安全”一节）。
- 如果在枚举一个并发集合的同时，另一个线程要修改它，不会抛出任何异常。相反，得到旧内容与新内容的混合。
- 不存在任何List<T>的并发版本。
- 并发的堆栈、队列和包类均使用链表内部实现。这让它们在内存利用方面没有非并发的Stack和Queue类高效，但对于并发访问的效果更好，因为链表实现不用锁或少用锁。（这是因为，在链表中插入一个节点需要更新几处引用，而在List<T>类的结构中插入一个元素可能需要移动数以千计的现有元素。）

换句话说，这些集合不仅仅只是为使用带锁的普通集合提供了快捷方式。为了进行演示，假设我们在一个线程上执行以下代码：

```
var d = new ConcurrentDictionary<int,int>();
for (int i = 0; i < 1000000; i++) d[i] = 123;
```

它的运行速度比下面这段代码慢3倍：

```
var d = new Dictionary<int,int>();
for (int i = 0; i < 1000000; i++) lock (d) d[i] = 123;
```

（但读取ConcurrentDictionary很快，因为读取操作是无需用锁的。）

并发集合与传统集合还有一个区别，即它们公开了特殊方法来执行原子的测试与行动操作，例如TryPop。大多数这种方法都可以通过IProducerConsumerCollection<T>接口统一起来。

23.6.1 IProducerConsumerCollection<T>

生产者/使用者集合有两个主要的用例：

- 添加一个元素（“生产”）

- 检索一个元素并删除它（“消费”）

传统的示例包括堆栈和队列。生产者/使用者集合对于并行编程的意义非凡，因为它们有利于高效的无锁实现。

IProducerConsumerCollection<T>接口代表一个线程安全的生产者/使用者集合。下面的类实现了这个接口：

```
ConcurrentStack<T>
ConcurrentQueue<T>
ConcurrentBag<T>
```

IProducerConsumerCollection<T>接口扩展了ICollection，增加了以下方法：

```
void CopyTo (T[] array, int index);
T[] ToArray();
bool TryAdd (T item);
bool TryTake (out T item);
```

TryAdd和TryTake方法用于测试一个添加/删除操作能否执行，如果可以，则执行添加/删除操作。测试与行动是以原子方式执行的，因此不需要对传统集合上锁：

```
int result;
lock (myStack) if (myStack.Count > 0) result = myStack.Pop();
```

如果集合为空，TryTake方法返回false。TryAdd方法在提供的三种实现中，始终成功执行并返回true。但如果编写禁止复制的并发集合，当元素已经存在时会让TryAdd方法返回false（一个示例是编写并发集）。

TryTake方法删除的特定元素由子类定义：

- 对于堆栈，TryTake删除最近添加的元素。
- 对于队列，TryTake删除最早添加的元素。
- 对于口袋，TryTake删除它可能最有效删除的任意元素。

以上3个具体的类主要显式实现了TryAdd和TryTake方法，并通过更加特殊的公共方法公开同样的功能，例如TryDequeue和TryPop方法。

23.6.2 ConcurrentBag<T>

ConcurrentBag<T>用于保存对象的无序集合（允许复制）。ConcurrentBag<T>适用于调用Take或TryTake时不关心获得哪个元素的情况。

ConcurrentBag<T>胜过并发队列或堆栈的原因在于，当很多线程同时调用一个包的Add方法时，不存在竞争。相比之下，在队列或堆栈上并行调用Add会引起一些竞争（但比围绕非并发集合进行锁定要少很多）。在并发包上调用Take方法也非常高效，只要每个线程获取的元素不超过它增加的元素数量。

在一个并发包中，每个线程都会获得自己的私有链表。元素被添加到调用Add的线程的私有列表中，从而消除了竞争。当枚举包中的元素时，枚举器将遍历每个线程的私有列表，依次生成它的每个元素。

调用Take时，包首先考虑当前线程的私有列表。如果至少存在一个元素，它可以在没有竞争的情况

下轻松完成任务。但如果列表为空，它必须从另一线程的私有列表“偷取”一个元素，这样就有可能引起竞争。

因此，准确地说，调用Take获得的是该线程上最近添加的元素；如果该线程上无任何元素，它会返回另一线程上最近添加的元素，而且这个元素是随机挑选的。

当集合上的并行操作大部分是添加元素时，或者当Add和Take操作在一个线程上达到平衡时，并发包是理想的选择。我们介绍过前者的一个示例，即使用Parallel.ForEach实现一个并行的拼写检查器：

```
var misspellings = new ConcurrentBag<Tuple<int,string>>();

Parallel.ForEach (wordsToTest, (word, state, i) =>
{
  if (!wordLookup.Contains (word))
    misspellings.Add (Tuple.Create ((int) i, word));
});
```

对于生产者/使用者队列，不能选择并发包，因为元素是由不同线程添加和删除的。

23.7 BlockingCollection<T>

如果在前一节介绍的任意生产者/消费者集合中调用TryTake：

```
ConcurrentStack<T>
ConcurrentQueue<T>
ConcurrentBag<T>
```

并且集合为空，那么该方法会返回false。有时候，这种情况适合于等待元素的出现。

PFX设计人员并没有通过重载TryTake方法来实现这个功能（在允许取消和超时时，这样会导致成员泄漏），而是将这个功能封装到一个包装类中，即BlockingCollection<T>。使用一个阻塞集合封装了所有实现IProducerConsumerCollection<T>的集合，并且允许从封装的集合中取出（Take）一个元素——如果没有元素，那么这个操作会阻塞。

此外，阻塞集合也可以限制集合的总大小，在超出大小时阻塞生产者。具有这种限制功能的集合称为有界阻塞集合（bounded blocking collection）。

要使用BlockingCollection<T>，必须：

1. 创建这个类的实例，选择指定需要封装的IProducerConsumerCollection<T>，以及集合的大小（边界）。
2. 调用Add或TryAdd，在底层集合上添加元素。
3. 调用Take或TryTake，删除底层集合的元素。

如果调用构造函数时不传入一个集合参数，那么这个类会自动创建一个ConcurrentQueue<T>实例。生产和消费方法可以指定取消令牌和超时时间。如果集合大小是有边界的，那么Add和TryAdd方法可能会阻塞；如果集合为空，那么Take和TryTake也会阻塞。

另一种消费元素的方法是调用GetConsumingEnumerable。这个方法会返回一个无限序列，它会根据需要创建元素。调用CompleteAdding，可以强制终止序列：这个方法也会阻塞继续添加元素。

BlockingCollection也包含静态方法：AddToAny和TakeFromAny，它们可以在同时指定几个阻塞集合时，添加或取回元素。然后，第一个能够响应请求的集合将执行这个操作。

编写生产者/消费者队列

生产者/消费者队列是一种实用结构，适用于并行编程和一般并发编程。其使用方法具体如下：

- 创建用于描述操作项目或操作数据的队列。
- 在需要执行一个任务时，要将它排列，然后调用者再执行其他操作。
- 在后台创建一个或多个工作者线程，取出和执行排队的项目。

使用生产者/消费者队列，可以精确控制同时执行的工作者线程，这样做不仅有利于限制CPU消耗，也有利于控制资源消耗。例如，如果执行磁盘I/O密集操作，那么限制并发可以避免响应操作系统和其他应用程序的性能。此外，在队列生命周期中，可以动态添加和删除工作者。CLR的线程池本身就采用一种生产者/消费者队列实现，非常适合运行短时间的计算密集型任务。

生产者/消费者队列通常会保存（同一个）任务的执行数据项目。例如，数据项目可能是文件名，而任务就是加密这些文件。然而，通过将数据项目变成代理，就可以编写更为通用的生产者/消费者队列，其中每一个项目都可以执行任意操作。

在*http://albahari.com/threading*上，我们介绍了如何从零开始使用AutoResetEvent（及Monitor的Wait和Pulse）编写生产者/消费者队列。但是，从Framework 4.0开始，我们根本不需要从零开始编写生产者/消费者队列，因为BlockingCollection<T>已经提供了大部分的功能。下面是它的使用方法：

```
public class PCQueue : IDisposable
{
  BlockingCollection<Action> _taskQ = new BlockingCollection<Action>();

  public PCQueue (int workerCount)
  {
    // 为每一个消费者创建并启动一个独立任务：
    for (int i = 0; i < workerCount; i++)
      Task.Factory.StartNew (Consume);
  }

  public void Enqueue (Action action) { _taskQ.Add (action); }

  void Consume()
  {
    // 如果没有可用元素，所列举的序列会阻塞，并且会在调用CompleteAdding时终止。

    foreach (Action action in _taskQ.GetConsumingEnumerable())
      action(); // 执行任务。
  }

  public void Dispose() { _taskQ.CompleteAdding(); }
}
```

因为这里没有给BlockingCollection的构造方法传递任意参数，所以它会自动创建一个并发队列。如果传入一个ConcurrentStack，则会得到一个生产者/消费者堆栈。

使用任务

前面编写的生产者/消费者队列并不灵活，因为我们无法跟踪排队之后的操作项目。如果能实现以下功能，则效果更好：

- 知道操作项目的完成时间（并且等待它完成）
- 取消一个操作项目
- 处理操作项目抛出的任意异常

理想的解决方法是创建Enqueue方法，返回一些完成上述功能的对象。Task类已经实现了这样的功能，我们可以使用TaskCompletionSource创建一个任务，或者直接实例化（创建一个未启动任务或冷任务）：

```
public class PCQueue : IDisposable
{
  BlockingCollection<Task> _taskQ = new BlockingCollection<Task>();

  public PCQueue (int workerCount)
  {
    // 为每一个消费者创建并启动一个独立的Task：
    for (int i = 0; i < workerCount; i++)
      Task.Factory.StartNew (Consume);
  }

  public Task Enqueue (Action action, CancellationToken cancelToken
                                            = default (CancellationToken))
  {
    var task = new Task (action, cancelToken);
    _taskQ.Add (task);
    return task;
  }

  public Task<TResult> Enqueue<TResult> (Func<TResult> func,
              CancellationToken cancelToken = default (CancellationToken))
  {
    var task = new Task<TResult> (func, cancelToken);
    _taskQ.Add (task);
    return task;
  }

  void Consume()
  {
    foreach (var task in _taskQ.GetConsumingEnumerable())
      try
      {
          if (!task.IsCanceled) task.RunSynchronously();
      }
      catch (InvalidOperationException) { }  // 竞争条件
  }

  public void Dispose() { _taskQ.CompleteAdding(); }
}
```

Enqueue将任务排队，然后返回给调用者一个之前创建但是没有启动的任务。

Consume在消费者线程上并发执行任务。捕捉InvalidOperationException异常，处理在检查任务是否取消和运行这两个操作期间执行取消操作的事件（不可能发生）。

下面是这个类的使用方法：

```
var pcQ = new PCQueue (2);    // 最大并发数为2
string result = await pcQ.Enqueue (() => "That was easy!");
...
```

因此，我们能够利用任务的优点——包括异常处理、返回值和取消操作，同时能够完全控制任务调度。

第24章 应用域

应用域是指运行中的.NET程序所在的独立区域。它提供了一个可控内存区域作为程序集和相关配置的容器，同时划定分布式程序的交互区域。

每个.NET进程通常拥有一个应用域：默认域。默认域在进程开始时由CLR自动创建。可以为应用程序建立额外的应用域，并且额外的应用域可以提供隔离，而且与单独的进程相比，降低额外系统开销和交互复杂性。它也可以应用于加载测试、应用程序补丁和运行稳定性错误恢复机制中。

警告：本章与Windows Metro应用无关，因为它们只能访问唯一一个应用域。

24.1 应用域架构

图24-1阐明了单应用域、多应用域和典型分布式客户端/服务器程序的应用域架构。通常情况下，进程的应用域是在用户双击可执行文件或者启动一个系统服务程序的时候，由操作系统建立的。但是，通过CLR的整合，互联网信息服务进程（IIS）和数据库服务进程（SQL）等也可以拥有应用域。

对于简单应用程序，进程和默认域同时结束运行。但是对于IIS和SQL，进程控制着.NET应用域的生命周期，在合适的时候生成应用域和销毁应用域。

24.2 创建和销毁应用域

在进程中，可以通过调用静态方法`AppDomain.CreateDomain`和`AppDomain.Unload`创建和销毁应用域。下面的示例中，*test.exe*在应用域中执行，随后紧毁该应用域。

```
static void Main()
{
  AppDomain newDomain = AppDomain.CreateDomain ("New Domain");
  newDomain.ExecuteAssembly ("test.exe");
  AppDomain.Unload (newDomain);
}
```

谨记：当由CLR在程序开始时创建的应用域即默认域销毁时，应用程序关闭并且销毁该程序其他所有

应用域。通过AppDomain属性IsDefaultDomain，可以确定应用域是否是默认域。

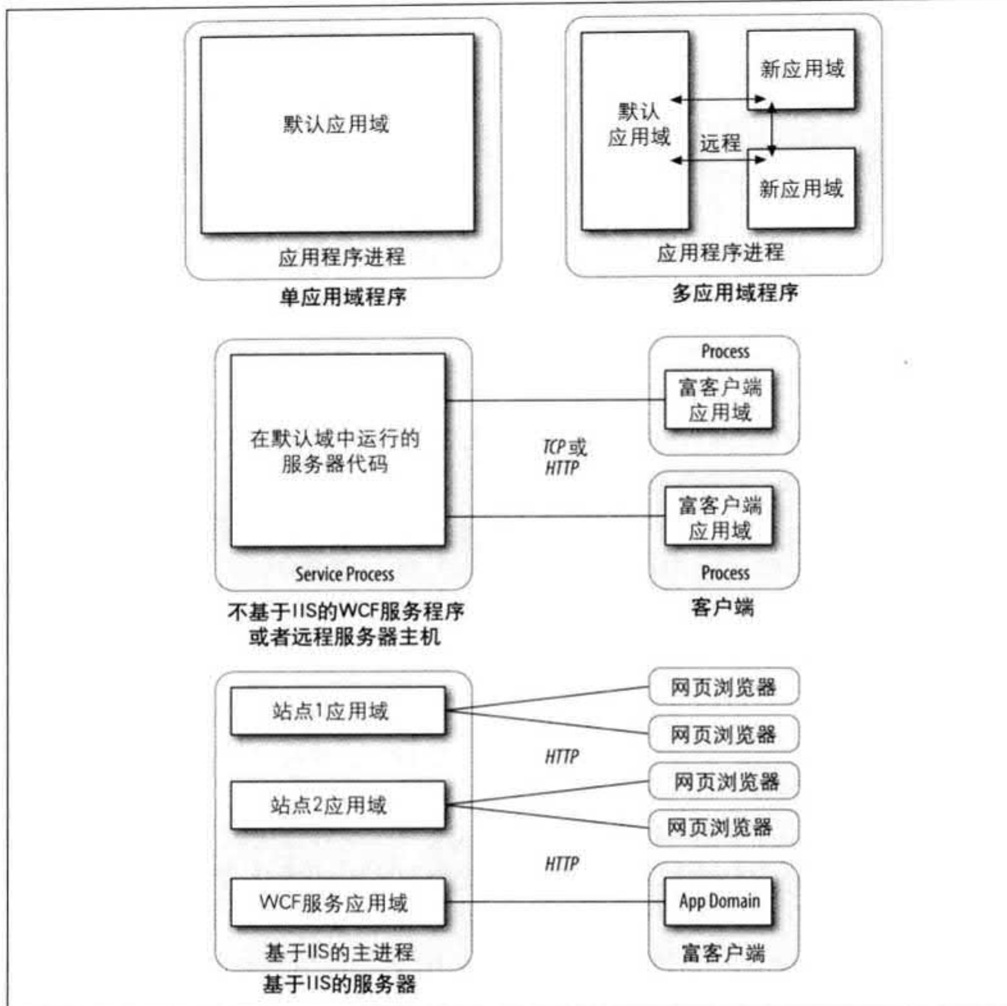


图24-1：应用域架构

AppDomainSetup类允许为一个新域指定选项。以下属性尤其有用：

```
public string ApplicationName { get; set; }        // 应用域名称
public string ApplicationBase { get; set; }        // 基础文件夹

public string ConfigurationFile { get; set; }
public string LicenseFile       { get; set; }

// 辅助自动程序集解析：
public string PrivateBinPath      { get; set; }
public string PrivateBinPathProbe { get; set; }
```

ApplicationBase属性控制应用域的根文件夹，该根文件夹指定了自动检测程序集时的范围。默认

域的根文件夹是主要的可执行文件所在的文件夹。对于创建的新应用域，其根文件夹可根据需要任意选取：

```
AppDomainSetup setup = new AppDomainSetup();
setup.ApplicationBase = @"c:\MyBaseFolder";
AppDomain newDomain = AppDomain.CreateDomain ("New Domain", null, setup);
```

允许专门定义一个新应用域去响应主体应用域中的程序集解析事件。

```
static void Main()
{
  AppDomain newDomain = AppDomain.CreateDomain ("test");
  newDomain.AssemblyResolve += new ResolveEventHandler (FindAssem);
  ...
}

static Assembly FindAssem (object sender, ResolveEventArgs args)
{
  ...
}
```

允许将事件处理函数定义成两个应用域均可以调用的静态方法，此时CLR可以在特定应用域中正确执行事件处理函数。本例中，尽管FindAssem函数是在默认域中定义，但是它将会在newDomain应用域中执行。

PrivateBinPath属性是应用域根文件夹下的以分号为间隔的子文件夹列表，该列表供CLR自动搜索程序集。与根文件夹属性一样，子文件夹属性也只能在应用域开始运行之前设定。例如下边这个文件夹的架构，根文件夹下只有一个可执行文件或者一个配置文件，所有程序集都在其子文件夹下。

```
c:\MyBaseFolder\                   ——启动应用程序
               \bin
               \bin\v1.23          ——最新的动态库程序集
               \bin\plugins        ——更多动态库
```

下面的代码显示了应用域如何使用这样的文件夹架构：

```
AppDomainSetup setup  = new AppDomainSetup();
setup.ApplicationBase = @"c:\MyBaseFolder";
setup.PrivateBinPath  = @"bin\v1.23;bin\plugins";
AppDomain d = AppDomain.CreateDomain ("New Domain", null, setup);
d.ExecuteAssembly (@"c:\MyBaseFolder\Startup.exe");
```

谨记：PrivateBinPath所列的子文件夹路径是相对于根文件夹路径来说的，并且子文件夹处于根文件夹内，因此指定为绝对路径是错误的。AppDomain同时也提供PrivateBinPathProbe属性，如果把该属性设置为非空字符串，根文件夹也被排除在程序集搜索范围以外。（指定PrivateBinPathProbe为字符串类型而非布尔类型的原因与COM的兼容性有关。）

销毁非默认域将会触发DomainUnload事件。可以在这个事件的处理过程中卸载逻辑，被销毁的应用域甚至可以让整个应用程序暂停下来等待直到事件处理函数完成。

应用程序退出将会触发所有已加载应用域（包括默认域）的ProcessExit事件。和DomainUnload事件不同，ProcessExit事件的处理是有时间限制的，在线程结束之前，CLR给每个应用域2秒钟的处理时间，所有应用域处理时间的和不能超过3秒钟。

24.3 多应用域的使用

多应用域有以下重要用途：

- 以最小的系统开销提供类似于进程之间的独立性。
- 没有重启进程时卸载程序集。

尽管额外的应用域产生在同一个进程内，但是CLR可以赋予它们一定的独立性，就好像运行在不同进程下。这意味着每个应用域有独立的内存区，应用域内的对象不会干扰其他应用域内的对象。此外，同一个类中的静态成员在每个应用域下有独立的值。ASP.NET正是使用这种方法使得许多站点在同一个进程下运行而且互不干扰。

ASP.NET的应用域是底层架构创建的，但是往往也可以从单进程多应用域的设计中获得很多便利。假设开发了一个客户认证系统，想通过模拟20个客户并发登录的情况，去测试服务器代码强度。有以下三种选择去模拟20个客户并发登录的情况：

- 通过调用20次`Process.Start`函数，开启20个独立的进程。
- 在同一个进程和应用域下开启20个线程。
- 在同一个进程中开启20个线程，每个线程有独立的应用域。

第一种选择是笨拙的，而且十分浪费资源。如果想要它们执行更多的操作也是很困难的。

第二种选择的客户端程序具有基于线程的安全性，当认证状态是通过一个静态变量存储的时候，这种线程安全性是不可靠的。而且在客户端程序中添加锁的方式，会妨碍客户端线程的并行运行，导致不能对服务器进行强度测试。

第三种是比较理想的选择。它通过线程隔离保证了线程的独立状态，同时仍然能轻松地连接到主程序。

允许进程内创建独立应用域的另一个原因是为了方便在进程运行时卸载程序集。这是因为除了关闭加载了程序集的应用域，没有别的方法可以卸载程序集。但是如果程序集被加载在默认域就会有一个问题，默认域的关闭意味着进程的结束。程序集被加载后就被锁定了，不能对它打补丁或者替换它。将程序集加载在独立的应用域可以避免上述问题，而且对于需要加载大程序集的进程，这种独立应用域加载的方法可以减少进程的内存占用。

LoaderOptimization属性

默认情况下，程序集被加载到为其创建的应用域中，即时编译器（JIT）对其进行处理。这些程序集包括：

- 在调用者应用域中，已经被即时编译器编译过的程序集。
- 使用*ngen.exe*工具，已经为其创建本地化映像的程序集。
- 所有的.NET Framework程序集（除了*mscorlib*）

如果反复地加载和卸载.NET Framework的大型程序集，上述问题将成为影响性能的关键。权宜之计是将以下属性与程序的入口方法关联起来：

```
[LoaderOptimization (LoaderOptimization.MultiDomainHost)]
```

这指示CLR以非特定于域的方式共享加载全局程序缓存中的程序集，此时本地映像被提交并且JIT映像被跨域共享。这种方式是很理想的，因为全局程序缓存中包含.NET Framework的所有程序集，甚至程序中一些不变的部分。

如果进一步指定LoaderOptimization.MultiDomain特性，所有的程序集都将以非特定于域的方式共享加载（不包括被加载到正常程序集识别机制以外的程序集）。以非特定于域方式加载的程序被进程内的所有应用域共享，并且不会随着应用域的停止而卸载。直到它们所属的进程结束，它们才能够被卸载。

24.4 DoCallBack的应用

我们回顾一下最基本的多应用域方案：

```
static void Main()
{
  AppDomain newDomain = AppDomain.CreateDomain ("New Domain");
  newDomain.ExecuteAssembly ("test.exe");
  AppDomain.Unload (newDomain);
}
```

在新应用域上调用ExecuteAssembly函数是很方便的，但是该方法基本不提供与应用域的交互渠道。被调用的目标程序集必须是可执行的，并且将调用者提交给自身的唯一入口点。该方法与可执行程序交互的唯一渠道是传递给它一个字符串参数。

功能更强大的方法是使用AppDomain的DoCallBack方法，它在新创建的应用域中执行指定类的方法。被调用的方法所在的程序集被自动加载到新创建的应用域中（如果当前应用域能够引用它，CLR会知道程序集的位置）。在下面的示例中，正在执行的类中的方法将会在新的应用域中运行：

```
class Program
{
static void Main()
  {
    AppDomain newDomain = AppDomain.CreateDomain ("New Domain");
    newDomain.DoCallBack (new CrossAppDomainDelegate (SayHello));
    AppDomain.Unload (newDomain);
  }

  static void SayHello()
  {
    Console.WriteLine ("Hi from " + AppDomain.CurrentDomain.FriendlyName);
  }
}
```

该示例之所以有效，是因为委托指向一个静态方法，即指向一个类型而非一个实例。这使得这个委托具有应用域不可知性或者说应用域灵活性。它可以运行在任何应用域中，并且不依赖于原始应用域。同样可以使用DoCallBack通过委托调用一个实例方法。但是，这种情况下CLR会试图运用远程语法（后文介绍），导致事与愿违。

24.5 应用域的监视

从Framework 4.0开始，可以监视特定应用域的内存和CPU消耗情况。但是必须通过下列步骤提前启用应用域的监视功能：

```
AppDomain.MonitoringIsEnabled = true;
```

以上设置启用了当前应用域的监视功能。一旦功能被启用，此后将不能禁用这个功能，试图将属性赋值为假时会抛出异常。

提示：启用应用域监视功能的另外一条途径是通过应用程序配置文件。需要在配置文件中添加如下代码：

```
<configuration>
   <runtime>
     <appDomainResourceMonitoring enabled="true"/>
  </runtime>
</configuration>
```

这样做可监视所有应用程序域。

可以通过以下三个实例属性查询应用域的CPU和内存占用情况：

```
MonitoringTotalProcessorTime
MonitoringTotalAllocatedMemorySize
MonitoringSurvivedMemorySize
```

前两个属性返回从应用域启动至此的CPU消耗总和和内存占用总和（这两个数字一直增长不会缩减）。第三个属性返回上次资源回收后，应用域的内存实际占用量。

通过在当前应用域或者其他的应用域中调用该属性，可以监视当前定应用域的系统资源消耗。

24.6 应用域和线程

当在另一个应用域或者可执行块中调用一个方法，直到方法结束，同在当前应用域中调用该方法一样。上述状况往往还需要并发执行一个方法。可以通过多线程来解决这个问题。

前面我们介绍过通过多应用域模仿20个客户并发登录，用以测试认证系统。每个登录客户拥有相互独立的应用域，并且不可以通过类的静态成员进行交互。为了实现这个示例，我们需要在20个拥有独立应用域的并发线程中调用Login方法：

```
class Program
{
  static void Main()
  {
    //产生20个应用域和20个线程
    AppDomain[] domains = new AppDomain [20];
    Thread[] threads = new Thread [20];

    for (int i = 0; i < 20; i++)
    {
      domains [i] = AppDomain.CreateDomain ("Client Login " + i);
      threads [i] = new Thread (LoginOtherDomain);
    }
```

```
    // 将应用域传递给每个线程，并启动它们
    for (int i = 0; i < 20; i++) threads [i].Start (domains [i]);

    // 等待线程结束
    for (int i = 0; i < 20; i++) threads [i].Join();

    // 卸载应用域
    for (int i = 0; i < 20; i++) AppDomain.Unload (domains [i]);
    Console.ReadLine();
  }

  // 将线程的应用域作为启动参数，启动线程
  static void LoginOtherDomain (object domain)
  {
    ((AppDomain) domain).DoCallBack (Login);
  }

  static void Login()
  {
    Client.Login ("Joe", "");
    Console.WriteLine ("Logged in as: " + Client.CurrentUser + " on " +
      AppDomain.CurrentDomain.FriendlyName);
  }
}

class Client
{
  // 如果运行在同一个应用域中，有一个静态字段会干扰其他的客户端登录
  public static string CurrentUser = "";

  public static void Login (string name, string password)
  {
   if (CurrentUser.Length == 0)    // 如果没有成功登录...
    {
      //睡眠以模仿授权过程
      Thread.Sleep (500);
      CurrentUser = name;           // 记录下成功登录的名称
    }
  }
}

// 输出结果：
Logged in as: Joe on Client Login 0
Logged in as: Joe on Client Login 1
Logged in as: Joe on Client Login 4
Logged in as: Joe on Client Login 2
Logged in as: Joe on Client Login 3
Logged in as: Joe on Client Login 5
Logged in as: Joe on Client Login 6
...
```

想了解更多的多线程知识请查看第22章。

24.7 应用域间通信

24.7.1 管道通信

应用域可以通过命名管道共享数据，如下面示例所示：

```
class Program
{
  static void Main()
  {
    AppDomain newDomain = AppDomain.CreateDomain ("New Domain");

    // 将字符串写入名为"Message"的命名管道
    newDomain.SetData ("Message", "guess what...");

    newDomain.DoCallBack (SayMessage);
    AppDomain.Unload (newDomain);
  }

  static void SayMessage()
  {
    // 从Message命名管道中读取
    Console.WriteLine (AppDomain.CurrentDomain.GetData ("Message"));
  }
}
// 输出：
guess what...
```

管道在其第一次使用的时候被建立。交互的数据必须是可串行化（像示例中的guess what...）的或者基于MarshalByRefObject的。如果数据是可串行化的（就像示例中的字符串），就会被拷贝到其他应用域中。如果数据是基于MarshalByRefObject的，就会采用远程语法。

24.7.2 进程内远程化（Intra-Process Remoting）

远程化是指通过委托在其他应用域内实例化对象，这是与其他应用域交互最灵活的方法。

被远程化的类必须继承自MarshalByRefObject。客户端在远程域的AppDomain类中调用CreateInstance*XXX*方法远程地实例化一个对象。

在下面的示例中，Foo类被实例化在其他应用域内，其SayHello方法在当前应用域中被调用：

```
class Program
{
  static void Main()
  {
    AppDomain newDomain = AppDomain.CreateDomain ("New Domain");

    Foo foo = (Foo) newDomain.CreateInstanceAndUnwrap (
                      typeof (Foo).Assembly.FullName,
                      typeof (Foo).FullName);

    Console.WriteLine (foo.SayHello());
    AppDomain.Unload (newDomain);
    Console.ReadLine();
  }
}

public class Foo : MarshalByRefObject
{
  public string SayHello()
  {
    return "Hello from " + AppDomain.CurrentDomain.FriendlyName;
  }
```

```
    public override object InitializeLifetimeService()
    {
      // 确保对象存在以便于客户端对它的调用
      return null;
    }
  }
```

因为应用域是相互独立的，所以当foo被实例化在其他的应用域（称为远程域），我们无法得到对该对象的直接引用，只能得到一个透明委托（transparent proxy）。之所以说它透明，是因为它就像对远程对象的直接引用。当我们接下来调用foo实例的SayHello时，一个消息被暗中组建并传递给远程域，远程域操作真正的foo实例，返回值被装入消息中并传回调用者。

提示： WCF是在.NET 3.0中推出的，在此之前，远程化是两种最重要的分布式程序建立技术之一（另一种是网络服务技术）。在分布式远程程序中，需要在每个终端显式地建立一个HTTP或者TCP/IP信道，用以跨进程或者跨网络通信。

尽管对于分布式程序来说，WCF比远程化更高级，但是对于进程内应用域间的通信，远程化还是有其优势的，即它不需要任何配置、信道自动建立、不需要注册任何类、使用起来很简单。

Foo的方法会返回更多的MarshalByRefObject实例，当这些实例的方法被调用时会产生更多的透明委托。MarshalByRefObject实例可以作为Foo方法的参数，此时远程化发生逆转，调用者持有远程化的对象，被调用者持有委托。

应用域之间不但可以通过引用marshale对象，而且可以交换数据值和串行化对象。如果一个类具有Serializable属性或者执行ISerializable方法，那么它是可串行化的。当发生应用域间的数据交换时，会返回该对象的完整副本而非一个委托。一言以蔽之，该对象是通过值而非引用被marshale。

同一进程内的远程化是客户端激活模式的，意味着CLR不会试图在此客户端或者其他的客户端分享或者复用远程创建的对象。换句话说，如果客户端创建两个Foo对象，那么远程域将会创建两个Foo对象，客户端将会产生两个委托。这给予了对象最自然的语法特性，但是，这意味着远程域是依赖于垃圾回收器的，即只有当客户端垃圾回收器认为foo对象（其委托）不再被使用，远程域内的foo对象才被从内存中释放。如果应用域崩溃，Foo对象永远不会被回收。针对这种情况，CLR提供一种租期机制管理远程化对象的生存周期。默认情况下，如果五分钟之内未被使用，远程创建对象将会自动销毁。

本例中的客户端在默认域内运行，不会出现客户端崩溃而资源得不到释放的情况。它结束的时候，整个进程都会随着默认域的结束而终止。因此，示例中禁用了五分钟租期是可取的。这也是示例中重载InitializeLifetimeService函数的目的，令其返回空的租期，只有在客户端回收委托后，远程对象才能得到释放。

24.7.3 封闭的类和程序集

之前的示例中，Foo类实例化的过程如下：

```
Foo foo = (Foo) newDomain.CreateInstanceAndUnwrap (
                  typeof (Foo).Assembly.FullName,
                  typeof (Foo).FullName);
```

该方法的声明如下：

```
public object CreateInstanceFromAndUnwrap (string assemblyName,
                                           string typeName)
```

该方法接受程序集和类的名字而非一个对象，因此不必加载该类到调用域，就可以远程初始化该类的一个对象。当不想加载程序集到调用域时，这种方法是很有用的。

提示： AppDomain类也提供了一个CreateInstanceFromAndUnwrap方法。不同点如下：

- CreateInstanceAndUnwrap接受一个完全合格的程序集名（见第17章）。
- CreateInstanceFromAndUnwrap接受一个路径或者文件名。

为了说明这个问题，假设编写一个允许加载第三方插件的文字编辑器。在第20章的“沙箱化程序集”一节从安全的角度讨论过这个问题。当时加载插件是通过调用ExecuteAssembly函数。如今学习了远程化，我们可以通过更高级的途径与插件交互。

首先，写一个主程序和插件都要用的通用库文件。这个库定义了插件的接口，示例如下：

```
namespace Plugin.Common
{
  public interface ITextPlugin
  {
    string TransformText (string input);
  }
}
```

接着，写简单插件。假设接下来的代码编译到*AllCapitals.dll*中：

```
namespace Plugin.Extensions
{
  public class AllCapitals : MarshalByRefObject, Plugin.Common.ITextPlugin
  {
    public string TransformText (string input) { return input.ToUpper(); }
  }
}
```

最后，我们编写主程序，主要功能是加载*AllCapitals.dll*到独立的应用域，通过远程化调用TransformText函数，最后卸载独立的应用域：

```
using System;
using System.Reflection;
using Plugin.Common;

class Program
{
  static void Main()
  {
    AppDomain domain = AppDomain.CreateDomain ("Plugin Domain");

    ITextPlugin plugin = (ITextPlugin) domain.CreateInstanceFromAndUnwrap
      ("AllCapitals.dll", "Plugin.Extensions.AllCapitals");

    // 通过远程方式调用TransformText方法
    Console.WriteLine (plugin.TransformText ("hello"));   // "HELLO"
```

```
    AppDomain.Unload (domain);

    // 可以完全卸载并删除AllCapitals.dll 文件
  }
}
```

因为主程序与插件的交互仅仅是通过通用接口ITextPlugin，AllCapitals中的类未被加载到调用域（主程序）中。这维护了调用域的完整性，并且保证了插件所在的域卸载后，插件文件不被锁定。

类型识别

在我们之前的示例中，应用程序需要经过一些渠道得到插件类的名字，例如通过参数Plugin.Extensions.AllCapitals。

也可以通过在普通程序集中定义一个用于查找类名的类，基于反射的原理达到目的。示例如下：

```
public class Discoverer : MarshalByRefObject
{
  public string[] GetPluginTypeNames (string assemblyPath)
  {
    List<string> typeNames = new List<string>();
    Assembly a = Assembly.LoadFrom (assemblyPath);
    foreach (Type t in a.GetTypes())
      if (t.IsPublic
        && t.IsMarshalByRef
        && typeof (ITextPlugin).IsAssignableFrom (t))
    {
      typeNames.Add (t.FullName);
    }
    return typeNames.ToArray();
  }
}
```

关键在于Assembly.LoadFrom加载程序到当前应用域内，所以需要在插件所在应用域内调用该方法：

```
class Program
{
  static void Main()
  {
    AppDomain domain = AppDomain.CreateDomain ("Plugin Domain");

    Discoverer d = (Discoverer) domain.CreateInstanceAndUnwrap (
      typeof (Discoverer).Assembly.FullName,
      typeof (Discoverer).FullName);

    string[] plugInTypeNames = d.GetPluginTypeNames ("AllCapitals.dll");

    foreach (string s in plugInTypeNames)
      Console.WriteLine (s);              // 输出Plugin.Extensions.AllCapitals

    ...
```

提示：在*System.AddIn.Contract*程序集中，这种思想已经发展成一个完整的框架，增强了程序的可扩展性。该框架可处理隔离、版本化、查找、激活等问题。在*http://blogs.msdn.com*网站上搜索“CLR Add-In Team Blog”可以获得更多在线信息。

第25章 本地化和COM组件交互

本章主要介绍怎样集成非托管动态库和COM组件，除非特别说明，本章中所提到的类都存在于System或者System.Runtime.InteropServices命名空间中。

25.1 调用本地库

*P/Invoke*是平台调用服务（Platform Invocation Services）的简称，允许访问未托管DLL中的函数、构件和回调。例如，考虑Windows DLL *user32.dll*中定义的MessageBox函数：

```
int MessageBox (HWND hWnd, LPCTSTR lpText, LPCTSTR lpCation, UINT uType);
```

通过在该函数的定义中添加extern关键字和DllImport属性，可以将该函数定义成一个同名的静态方法，从而在程序中直接调用：

```
using System;
using System.Runtime.InteropServices;

class MsgBoxTest
{
  [DllImport("user32.dll")]
  static extern int MessageBox (IntPtr hWnd, string text, string caption,int type);
  public static void Main()
  {
    MessageBox (IntPtr.Zero,"Please do not press this again.", "Attention", 0);
  }
}
```

命名空间System.Windows和System.Windows.Forms中的MessageBox类调用类似的非托管方法。

CLR中包括一个封送器，可以实现.NET类型和非托管类型的相互转换。本例中，整数类型的参数被转换成函数所期望的4字节的整数，并且字符串类型参数被转换成以空字符结尾的双字节的Unicode字符。IntPtr是一个用来封装非托管句柄的结构，在32位平台下，它的位宽是32位；在64位平台下，它的位宽是64位。

25.2 类型封送

25.2.1 封送常见类型

在非托管代码中，一个给定的数据类型可以有很多种表示方法，例如，一个单字节的ANSI字符或者一个双字节的Unicode字符，可以带有长度前缀，也可以以空字符结尾，或者长度固定。利用MarshalAs属性，可以告诉CLR封送器当前的变化，使得封送拆收器可以正确地进行转换。如下面的示例所示：

```
[DllImport("...")]
static extern int Foo ( [MarshalAs (UnmanagedType.LPStr)] string s );
```

非托管类型枚举器包括了封送器所能接收的所有Win32和COM类型。这个示例中，指示封送器转化成LPStr类型，该类型是以空字符结尾的单字符ANSI类型字符串。本章最后一一列举所有非托管成员。

在.NET程序内，仍然有多种类型可以选择。以非托管句柄为例，可以映射为IntPtr类型、int类型、uint类型、long类型和ulong类型。

警告： 大多数情况下非托管句柄封装一个地址或者指针，因此必须转换成一个IntPtr类型以匹配32位和64位的系统。一个典型的示例是HWND句柄。

在Win32函数中，经常用到可以接收一系列常量的整型参数，这类常量一般定义在C++头文件*WinUser.h*中。可以将它们定义成一个枚举类型，相比于定义成简单的C#常量，代码更简洁，并能保证静态类型的安全性。本章在稍后的“共享内存”中给出了一个示例。

提示： 安装Microsoft Visual Studio时，就算不选择任何其他C++类，也必须安装C++头文件。这些头文件定义了Win32本地常量。安装后，可以通过在Visual Studio的程序安装目录中搜索*.h文件来查找所有头文件。

从非托管代码中接收字符串到.NET代码中，需要发生一些内存管理事务。如果使用了StringBuilder类而非String类的话，封送器会自动完成内存管理事务，如下所示：

```
using System;
using System.Text;
using System.Runtime.InteropServices;

class Test
{
  [DllImport("kernel32.dll")]
  static extern int GetWindowsDirectory (StringBuilder sb, int maxChars);

  static void Main()
  {
    StringBuilder s = new StringBuilder (256);
    GetWindowsDirectory (s, 256);
    Console.WriteLine (s);
  }
}
```

提示： 如果不能确定怎样调用某一个Win32方法，通常可以通过搜索方法的名字和*DllImport*，在网络上找到相关的示例。网站*http://www.pinvoke.net*是一个关于文档化所有Win32签名的维基网站。

25.2.2 类和结构的封送

有时需要传递一个结构到非托管方法，例如GetSystemTime函数——一个Win32的API函数，定义如下：

```
void GetSystemTime (LPSYSTEMTIME lpSystemTime);
```

LPSYSTEMTIME是一个C结构，定义如下：

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

为了调用GetSystemTime函数，我们需要定义一个.NET类或者结构来匹配该C结构：

```
using System;
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Sequential)]
class SystemTime
{
   public ushort Year;
   public ushort Month;
   public ushort DayOfWeek;
   public ushort Day;
   public ushort Hour;
   public ushort Minute;
   public ushort Second;
public ushort Milliseconds;
}
```

StructLayout属性指示封送器如何将结构中每个成员转换成对应成员。LayoutKind.Sequential意味着要求结构内成员达到封装尺寸的倍数边界（稍后了解），就像C结构一样。成员的名字是不重要的，重要的是成员的排列顺序。

现在我们调用GetSystemTime函数：

```
[DllImport("kernel32.dll")]
static extern void GetSystemTime (SystemTime t);

static void Main()
{
  SystemTime t = new SystemTime();
  GetSystemTime (t);
```

```
    Console.WriteLine (t.Year);
  }
```

C和C#中，每个成员都位于对象的起始地址附加一个确定的偏移量处。区别在于，CLR通过元素的名称查找这个偏移，而C元素被直接编译成一个偏移量。例如，在C结构中，wDay元素是一个符号表示了SystemTime实例的地址加上24个字节处的地址。

为了加快访问速度，每个元素都被放置在偏移量为元素尺寸整数倍的内存边界上。这个倍数被限制为包长度的最大值。在当前的实际应用中，缺省的来指定是8字节，所以一个由sbyte和long（8字节）型数据构成的结构体，占据了16个字节的空间，sbyte后的7个字节浪费了。可以通过修改StructLayout属性的Pack值，使字段偏移指定包长度的整数倍。如果设置封装尺寸为1，上述结构会占用9个字节。可以指定封装尺寸为1、2、4、8或者16个字节。

StructLayout属性也允许定义一个确切的元素偏移（见“模拟C共用体”）。

25.2.3 In和Out修饰词

之前的示例中，我们将SystemTime封装成一个类。也可以将其封装成一个结构体，前提是定义函数GetSystemTime时，其参数有ref或者out关键字修饰：

```
[DllImport("kernel32.dll")]
static extern void GetSystemTime (out SystemTime t);
```

大多数情况下，C#中的方向参数语义同外部方法相同。按值传递修饰的参数被复制到函数内，C#中ref修饰的参数被复制到函数内或从函数中复制出，C#中out修饰的参数从函数中复制出。但是，也有一些例外的转换情况，例如，当从一个函数中返回时，数组类和StringBuilder类需要复制，所以它们是复制到或复制出。使用In和Out修饰符重载这些方法，有时候是很有用的。例如，如果一个数组是只读的，即修饰符In限定只允许复制该数组到函数内，不允许从函数中复制出来。

```
static extern void Foo ( [In] int[] array);
```

25.3 非托管代码的回调函数

P/Invoke层作为在托管和非托管代码中一个固有的编程模型，对两者相关的结构映射起到了很大作用。C#不但可以调用C函数，而且可以作为C函数的回调函数，前提是P/Inovke层需要映射非托管函数指针到托管代码空间的合法结构。托管代码中的委托等同于一个指针，因此P/Invoke层会将C#中的委托与C中的指针相互映射。

例如，可以用*User32.dll*中以下方法，枚举所有顶层窗口的句柄：

```
BOOL EnumWindows (WNDENUMPROC lpEnumFunc, LPARAM lParam);
```

WNDENUMPROC是一个回调函数，返回为窗口句柄时，该函数会不断被调用，直到返回为空。该回调函数定义如下：

```
BOOL CALLBACK EnumWindowsProc (HWND hwnd, LPARAM lParam);
```

使用时，我们声明一个同名的委托，将委托的实例传递给外部函数：

```
using System;
```

```
using System.Runtime.InteropServices;

class CallbackFun
{
  delegate bool EnumWindowsCallback (IntPtr hWnd, IntPtr lParam);

  [DllImport("user32.dll")]
  static extern int EnumWindows (EnumWindowsCallback hWnd, IntPtr lParam);

  static bool PrintWindow (IntPtr hWnd, IntPtr lParam)
  {
    Console.WriteLine (hWnd.ToInt64());
    return true;
  }

  static void Main()
  {
    EnumWindows (PrintWindow, IntPtr.Zero);
  }
}
```

25.4 模拟C共用体

结构体中会被分配足够的空间用来存储它的每个元素。假设一个结构体包含一个int元素和一个char元素，int元素占据了从偏移0开始的4个字节内存，所以char元素会从偏移4开始，如果char元素从偏移2开始存储，就会破坏int元素。但是C语言中确实有这样子的数据类型，即共用体类型。在C#中可以通过LayoutKind.Explicit和FieldOffset属性模拟共用体结构。

假设要在外部电子合成器上播放一个录音，Windows多媒体API会提供一个基于MIDI协议的函数去做这些工作：

```
[DllImport ("winmm.dll")]
public static extern uint midiOutShortMsg (IntPtr handle, uint message);
```

第二个参数，即message介绍了将要播放的录音。这个32位无符号整数被分成多个字节，分别表示MIDI的通道\录音和打击速度。一个解决方案是借助<<、>>、&和|等运算符通过位移和按位运算将该打包的message拆成字节数据。更简单的方法是借助于一个具有显式布局的结构：

```
[StructLayout (LayoutKind.Explicit)]
public struct NoteMessage
{
  [FieldOffset(0)] public uint PackedMsg;    // 4字节长整型

  [FieldOffset(0)] public byte Channel;      // 偏移仍然为0
  [FieldOffset(1)] public byte Note;
  [FieldOffset(2)] public byte Velocity;
}
```

将Channel、Note和Velocity字段重合在32位的message参数上，这就允许直接读写message参数上的特定字节而不会影响其他字节：

```
NoteMessage n = new NoteMessage();
Console.WriteLine (n.PackedMsg);    // 0

n.Channel = 10;
```

```
n.Note = 100;
n.Velocity = 50;
Console.WriteLine (n.PackedMsg);    // 3302410

n.PackedMsg = 3328010;
Console.WriteLine (n.Note);         // 200
```

25.5 内存共享

内存映射文件和共享内存是Windows的特性，它允许一台电脑上多个进程共享数据，而不需要使用Remoting和WCF技术去消耗更多的系统资源。共享内存交互速度很快，不像管道通信那样提供随机数据访问。我们可以查询第14章来了解怎样使用Framework4.0，通过实例化一个MemoryMappedFile类去操作内存映射文件；除了这些，借助于P/Invoke直接操作Win32方法也是一个不错的选择。

Win32中的CreateFileMapping函数创建共享内存区。需要指定所需的内存大小和内存名称。通过内存名称，可以使用OpenFileMapping函数打开该内存共享区。两个方法都会返回一个句柄，可以通过MapViewOfFile将该句柄转换成指针。

下面这个类封装了共享内存的方法：

```
using System;
using System.Runtime.InteropServices;

public sealed class SharedMem : IDisposable
{
  // 这里我们使用枚举类型，因为它们比常数安全

  enum FileProtection : uint      // winnt.h中的常数
  {
    ReadOnly = 2,
    ReadWrite = 4
  }

  enum FileRights : uint          // WinBASE.h中的常数
  {
    Read = 4,
    Write = 2,
    ReadWrite = Read + Write
  }

  static readonly IntPtr NoFileHandle = new IntPtr (-1);

  [DllImport ("kernel32.dll", SetLastError = true)]
  static extern IntPtr CreateFileMapping (IntPtr hFile,
                                          int lpAttributes,
                                          FileProtection flProtect,
                                          uint dwMaximumSizeHigh,
                                          uint dwMaximumSizeLow,
                                          string lpName);

  [DllImport ("kernel32.dll", SetLastError=true)]
  static extern IntPtr OpenFileMapping (FileRights dwDesiredAccess,
                                        bool bInheritHandle,
                                        string lpName);

  [DllImport ("kernel32.dll", SetLastError = true)]
  static extern IntPtr MapViewOfFile (IntPtr hFileMappingObject
```

```
                                          FileRights dwDesiredAccess,
                                          uint dwFileOffsetHigh,
                                          uint dwFileOffsetLow,
                                          uint dwNumberOfBytesToMap);

  [DllImport ("Kernel32.dll", SetLastError = true)]
  static extern bool UnmapViewOfFile (IntPtr map);

  [DllImport ("kernel32.dll", SetLastError = true)]
  static extern int CloseHandle (IntPtr hObject);

  IntPtr fileHandle, fileMap;

  public IntPtr Root { get { return fileMap; } }

  public SharedMem (string name, bool existing, uint sizeInBytes)
  {
    if (existing)
      fileHandle = OpenFileMapping (FileRights.ReadWrite, false, name);
    else
      fileHandle = CreateFileMapping (NoFileHandle, 0,FileProtection.ReadWrite,
                                      0, sizeInBytes, name);
    if (fileHandle == IntPtr.Zero)
      throw new Win32Exception();

    // 获得整个文件的读/写映射
    fileMap = MapViewOfFile (fileHandle, FileRights.ReadWrite, 0, 0, 0);

    if (fileMap == IntPtr.Zero)
      throw new Win32Exception();
  }

  public void Dispose()
  {
    if (fileMap != IntPtr.Zero) UnmapViewOfFile (fileMap);
    if (fileHandle != IntPtr.Zero) CloseHandle (fileHandle);
    fileMap = fileHandle = IntPtr.Zero;
  }
}
```

本例中，这些Win32函数借助于SetLastError函数抛出错误代码，我们在DllImport方法上设置SetLast-Error=true，保证了当有错误发生时，Win32Exception被抛出时附有错误细节。也可以通过Marshal.GetLastWin32Error函数查询当前错误代码。

为了验证当前类，我们需要运行两个应用程序，第一个应用程序创建共享内存区，如下：

```
using (SharedMem sm = new SharedMem ("MyShare", false, 1000))
{
  IntPtr root = sm.Root;
  // 得到共享内存

  Console.ReadLine();          // 这里我们启动了第二个应用程序
}
```

第二个应用程序通过共享内存区的名字构建ShareMem对象从而订阅共享内存区，existing参数需要设置为true：

```
using (SharedMem sm = new SharedMem ("MyShare", true, 1000))
{
```

```
    IntPtr root = sm.Root;
    // 得到同一个共享内存区
    // ...
  }
```

每个.NET程序都有一个IntPtr变量，类似于非托管的指针。两个程序都需要借助于这个指针从共享区读取或者写入数据。一个方法是使用串行化类封装所有的共享数据，然后利用UnmanagedMemoryStream类将数据串行化写入到非托管内存区，或者从中读取。如果有很多数据的话，该方法效率低下。假设有1M字节的数据，我们只需要更新其中的一个整数。最佳方法是将共享数据定义成一个结构体，然后将其直接共享到内存，我们将在下节讨论。

25.6 映射结构体到非托管内存区

具有Sequential或者Explicit的StructLayout属性的结构体可以直接映射到非托管内存区。如下述结构：

```
[StructLayout (LayoutKind.Sequential)]
unsafe struct MySharedData
{
  public int Value;
  public char Letter;
  public fixed float Numbers [50];
}
```

通过fixed关键字可以定义定长的数组，将进入unsafe区域。该结构50个浮点数的空间是直接分配好的。不同于C#标准数组，NumberArray数组不是数组的引用，它本身就是数组。如果运行如下程序：

```
static unsafe void Main()
{
  Console.WriteLine (sizeof (MySharedData));
}
```

结果是208，即50个4字节浮点数加上4字节整数加上2字节字符，共206字节，又因为浮点数需要被放置在4字节的边界上，所以总计208字节（4字节是float型数据的长度）。

我们给出一种在非安全模式下请求MySharedData内存的范例，它基于栈分配并且更简单：

```
MySharedData d;
MySharedData* data = &d;        // 得到d的地址

data->Value = 123;
data->Letter = 'X';
data->Numbers[10] = 1.45f;
```

或者：

```
// 在堆栈中分配数组空间：
MySharedData* data = stackalloc MySharedData[1];

data->Value = 123;
data->Letter = 'X';
data->Numbers[10] = 1.45f;
```

当然，我们演示的功能非托管代码无法实现。但是，假设我们需要存储一个MySharedData实例到非托管堆中，已经超出了CLR垃圾回收器的管理区域，这时指针还是很有用的。

```
MySharedData* data = (MySharedData*)
  Marshal.AllocHGlobal (sizeof (MySharedData)).ToPointer();

data->Value = 123;
data->Letter = 'X';
data->Numbers[10] = 1.45f;
```

Marshal.AllocHGlobal方法在非托管堆上请求内存。以下是释放该内存的代码：

```
Marshal.FreeHGlobal (new IntPtr (data));
```

（忘记释放内存的结果是出现内存泄漏。）

正如其名称所指示的那样，现在我们将MySharedData与上一节中编写的SharedMem类结合起来。下面的程序创建了一个共享内存区，然后将MySharedData映射到该共享区。

```
static unsafe void Main()
{
  using (SharedMem sm = new SharedMem ("MyShare", false, 1000))
  {
    void* root = sm.Root.ToPointer();
    MySharedData* data = (MySharedData*) root;

    data->Value = 123;
    data->Letter = 'X';
    data->Numbers[10] = 1.45f;
    Console.WriteLine ("Written to shared memory");

    Console.ReadLine();

    Console.WriteLine ("Value is " + data->Value);
    Console.WriteLine ("Letter is " + data->Letter);
    Console.WriteLine ("11th Number is " + data->Numbers[10]);
    Console.ReadLine();
  }
}
```

提示：如果使用Framework4.0，除了SharedMem类，还可以使用MemoryMappedFile类：

```
using (MemoryMappedFile mmFile =
       MemoryMappedFile.CreateNew ("MyShare", 1000))
using (MemoryMappedViewAccessor accessor =mmFile.CreateViewAccessor())
{
  byte* pointer = null;
  accessor.SafeMemoryMappedViewHandle.AcquirePointer
   (ref pointer);
  void* root = pointer;
  ...
}
```

下面我们再演示一个程序，它连接第一个程序创建的共享内存区，读取第一个程序写入的数据。（当第一个程序正在等待ReadLine语句时，它必须运行，因为在执行using语句后共享内存区已被释放。）

```
static unsafe void Main()
{
  using (SharedMem sm = new SharedMem ("MyShare", true, 1000))
```

```
  {
    void* root = sm.Root.ToPointer();
    MySharedData* data = (MySharedData*) root;

    Console.WriteLine ("Value is " + data->Value);
    Console.WriteLine ("Letter is " + data->Letter);
    Console.WriteLine ("11th Number is " + data->Numbers[10]);

    // 更新共享内存区中的数据
    data->Value++;
    data->Letter = '!';
    data->Numbers[10] = 987.5f;
    Console.WriteLine ("Updated shared memory");
    Console.ReadLine();
  }
}
```

每个程序输出的内容如下：

```
// 第一个程序：

Written to shared memory
Value is 124
Letter is !
11th Number is 987.5

// 第二个程序：

Value is 123
Letter is X
11th Number is 1.45
Updated shared memory
```

不要忽略指针：C++编程者在所有的程序都会用到指针，并且可以借助它们执行任何操作。而且多数情况下，相对而言使用指针是比较简单的。

顾名思义，我们的示例是不安全的。因为我们没有考虑当两个程序同时访问同一块内存区时，它的线程安全性（更确切的说是进程安全性）。因此需要加入volatile关键字去修饰MySharedData结构中的元素Value和Letter，以防止字段被缓存在CPU寄存器。进一步讲，当我们同稍大一点的元素交互的时候，最好借助Mutex保护跨进程安全性，就像我们在多线程程序中通过锁保护元素的访问安全性。我们曾在第21章讨论过线程安全性。

fixed 和 fixed {...}

将结构直接映射到内存的一个缺点是结构中只能含有非托管类型。如果要共享字符串数据，必须使用一个固定长度的字符串数组。这意味着需要手动转换字符串类型。下面是具体操作：

```
[StructLayout (LayoutKind.Sequential)]
unsafe struct MySharedData
{
  ...
  // 请求200个字节的空间
  const int MessageSize = 200;
  fixed char message [MessageSize];

  // 有时把这些代码写入帮助类中
```

```
    public string Message
    {
      get { fixed (char* cp = message) return new string (cp); }
      set
      {
        fixed (char* cp = message)
        {
          int i = 0;
          for (; i < value.Length && i < MessageSize - 1; i++)
            cp [i] = value [i];

          // 添加空终止字符
          cp [i] = '\0';
        }
      }
    }
  }
```

提示： 虽然没有定长数组的引用，但是可以得到一个指针。当索引定长数组时，其实是对指针进行运算。

第一次使用fixed关键字时，我们直接为该结构分配了200字节的空间。接下来定义属性的时候，这个关键字的意义就发生了变化。它指示CLR需要在定长数组区放一个垃圾回收器，不要把底层结构放置在堆上，因为它的内容是通过内存指针直接迭代寻址的。假设MySharedData不在堆中，而是在垃圾回收器无法掌控的非托管代码区。其实编译器认为我们会在托管环境中使用MyShareData，因此它要求加入fixed关键字，保证代码在托管环境中的安全性。而且编译器没有指针，以下是将MySharedData放入堆中的代码：

```
object obj = new MySharedData();
```

这样做的结果是MySharedData被打包——放在堆上并且在资源回收时被传送。

这个示例阐明了怎样将一个字符串放在一个结构中并且映射到非托管内存区。对于更复杂的结构，可以选择采用已有的串行化代码。限制条件是被串行化的数据的长度不能超过结构体请求的空间，否则将会出现非预期的共用体——其后元素会被破坏。

25.7 COM交互

.NET程序对COM对象都有特殊的支持，使得COM程序可以在.NET程序中调用，反之亦然。C# 5.0和CLR 4.0增强了在.NET中部署和使用COM的功能。

25.7.1 COM的目的

COM（Component Object Model，组件对象模型）是微软1993年发布的应用程序接口的二进制标准。设计COM是为了让组件交互有更强的版本兼容性和更少的编程语言依赖性。COM出现以前，Windows通常发布动态链接库（DLL），它是使用C语言定义结构和函数的。不仅这种方法受限于编程语言，而且动态库本身也是很脆弱的。对于此种库中的类型的定义是与它的实现分不开的，即使更新一个结构中的元素也破坏了它的定义。

COM的优点在于使用COM结构实现了类型的定义与底层操作的分离。COM也允许调用有状态对象的

方法，而不仅仅是调用一个简单的过程函数。

提示：某种程度上来说，.NET程序是在COM规则上进化而来的：.NET平台有助于跨语言开发并且允许二进制组件的更新而不影响依赖于该组件的程序正常运行。

25.7.2 COM体系基础

COM体系是以接口为中心的。COM接口和.NET接口类似，但是更具普适性，因为COM接口只通过它的接口公开它的方法。例如，在.NET中，我们可以按如下方式定义一个类：

```
public class Foo
{
  public string Test() { return "Hello, world"; }
}
```

用户可以直接使用Foo类。并且如果以后修改了Test函数的实现，该函数的调用者并不需要重新编译。这就是说，.NET程序将接口和实现分离开来——没有用到接口。我们甚至可以在不打断调用者的情况下重载Test函数：

```
public string Test (string s) { return "Hello, world " + s; }
```

在COM中，Foo通过一个接口公开它的方法达到相同的效果。因此在Foo类型库中，会存在如下接口：

```
public interface IFoo { string Test(); }
```

（我们通过一个C#接口而非一个COM接口去说明这个问题。虽然应用场景不同，但是道理是一样的。）

调用者就可以使用IFoo而非Foo进行交互。

当我们需要重载Test方法时，COM要比.NET复杂一些，首先，我们需要避免修改IFoo接口，因为这会破坏以前版本的二进制兼容性（COM的原则就是一旦发布，接口不可更改）。其次COM不允许方法重载。解决方法是Foo类实现第二个接口：

```
public interface IFoo2 { string Test (string s); }
```

（我们再次将该接口介绍为比较熟悉的.NET接口。）

支持多接口是使得COM库兼容多版本的重要因素。

IUnknown和IDispatch接口

所有的COM接口都有一个GUID。

COM的根接口是IUnknown接口，所有的COM对象都必须实现它。这个接口有三个方法：

- AddRef
- Release
- QueryInterface

AddRef和Release方法控制着COM的生命周期，因为COM是基于引用计数而非垃圾回收的（COM被

设计运行在非托管区，自动垃圾回收是不可行的）。QueryInterface方法可查询类型，将返回该对象支持的所有COM接口。

为了实现动态编程（例如，脚本和自动化），一个COM对象还可能实现了IDispatch接口。该接口允许VBScript等动态语言通过后期绑定的方法调用COM对象，就像C#中的动态化一样。

25.8 在C#中调用COM对象

CLR内嵌对COM的支持意味着不需要直接使用IUnknown和IDispatch接口，而是使用CLR对象和通过运行时可调用封装器（RCW）引入到COM中。运行库也会通过AddRef和Release方法管理COM对象的生命周期并且管理着两种基本类型的转换问题。类型转换确保都能正确认识参数，例如整数和字符串都能按照熟悉的方式存储。

此外，我们还需要以一个静态类的方式操作RCW。这是COM互操作类型的工作。COM互操作类型会自动为每个COM成员生成一个委托类型——该类型将提供一个.NET成员。类型库导入工具（*tlimp.exe*）通过命令行生成一个所选COM库的互操作类型，并且将它们编译成COM互操作程序集。

提示： 如果COM组件包含多个接口，*tlimp.exe*工具将产生一个简单共用体类型，该共用体包含了所有的接口成员。

在Visual Studio中，可以通过添加引用对话框从COM列表中选择需要的COM库，创建COM互操作程序集。例如，如果已经安装Microsoft Excel 2007，添加对Microsoft Excel 12.0 COM对象的引用允许与Excel的COM对象进行互操作。下面的代码展示了怎样创建和显示一个工作簿，并且定义该工作簿中的一个单元格：

```
using System;
using Excel = Microsoft.Office.Interop.Excel;

class Program
{
  static void Main()
  {
    var excel = new Excel.Application();
    excel.Visible = true;
    Workbook workBook = excel.Workbooks.Add();
    excel.Cells [1, 1].Font.FontStyle = "Bold";
    excel.Cells [1, 1].Value2 = "Hello World";
    workBook.SaveAs (@"d:\temp.xlsx");
  }
}
```

Excel.Application类是一个COM互操作类。当我们访问工作簿和单元格的属性时，得到更多的互操作类型。

这段代码非常简单，因为C# 4.0引入了许多专门与COM相关的改进。如果没有这些改进，这个Main方法会变成：

```
var missing = System.Reflection.Missing.Value;

var excel = new Excel.Application();
excel.Visible = true;
```

```
Workbook workBook = excel.Workbooks.Add (missing);
var range = (Excel.Range) excel.Cells [1, 1];
range.Font.FontStyle = "Bold";
range.Value2 = "Hello world";

workBook.SaveAs (@"d:\temp.xlsx", missing, missing, missing, missing,
  missing, Excel.XlSaveAsAccessMode.xlNoChange, missing, missing,
  missing, missing, missing);
```

现在介绍具体有哪些语言改进，以及它们如何简化COM编程。

25.8.1 可选参数和具名实参

因为COM API不支持函数重载，所以经常会有一些函数包含许多个参数，而且其中有很多是可选参数。例如，下面的代码调用Excel工作薄的Save方法：

```
var missing = System.Reflection.Missing.Value;

workBook.SaveAs (@"d:\temp.xlsx", missing, missing, missing, missing,
  missing, Excel.XlSaveAsAccessMode.xlNoChange, missing, missing,
  missing, missing, missing);
```

幸好，C#的可选参数支持也适用于COM，所以可以这样做：

```
workBook.SaveAs (@"d:\temp.xlsx");
```

（我们在第3章讲过，编译器会把前者编译成后者。）

对于具名实参，可以很容易地定义额外的参数，而且不需要考虑它们的位置：

```
workBook.SaveAs (@"c:\test.xlsx", Password:"foo");
```

25.8.2 隐式参数引用

在一些COM库的API（尤其是Microsoft Word）中，无论函数是否会修改参数的值，函数中的参数都是声明为引用类型的。这是因为它的感知性能来自于非拷贝参数值。

在C#中调用这样的方法是很复杂的，因为必须使用ref关键字声明所有的参数，例如在C# 3.0中打开一个Word文档，必须编写如下代码：

```
object filename = "foo.doc";
object notUsed1 = Missing.Value;
object notUsed2 = Missing.Value;
object notUsed3 = Missing.Value;
...
Open (ref filename, ref notUsed1, ref notUsed2, ref notUsed3, ...);
```

为了解决这个问题，C# 4.0和C# 5.0编译器允许忽略COM函数调用的ref修饰符，从而允许使用可选参数：

```
word.Open ("foo.doc");
```

唯一的警告是，如果改变了一个参数的值，既不会得到相关编译错误也不会得到相关运行时错误。

25.8.3 索引器

省略ref修饰符的另一个好处是：COM中引用参数的索引器现在可以通过普通C#索引器访问。之前，这是被禁止的，因为ref/out参数不被C#索引器支持（变通方案是通过调用支持方法，例如get_*XXX*和set_*XXX*；为了向后兼容，这种变通方案还是合法的）。

C# 5.0加强了与索引器进行互操作的支持，因此可以使用参数调用COM属性。下面的示例中，Foo是接收一个整型参数的属性：

```
myComObject.Foo [123] = "Hello";
```

在C#中自定义代码实现这些属性是被禁止的，因为一个类只能公开自己的索引器（缺省索引器）。因此，如果想使上述代码合法，Foo必须返回提供给（默认）索引器的另一个类。

25.8.4 动态绑定

当调用COM组件时，有两种动态绑定方法。第一种情况是不想通过COM互操作类访问COM组件。此时使用COM组件的名字调用Type.GetTypeFromProgID去获得一个COM实例，紧接着使用动态绑定去调用组件中的成员。当然，这里不存在智能感知和编译检查。

```
Type excelAppType = Type.GetTypeFromProgID ("Excel.Application", true);
dynamic excel = Activator.CreateInstance (excelAppType);
excel.Visible = true;
dynamic wb = excel.Workbooks.Add();
excel.Cells [1, 1].Value2 = "foo";
```

（通过映射也可达到与动态绑定相同的效果，不过更加复杂一些。）

提示： 使用该策略的一个例外是调用仅仅支持IDispath接口的COM组件。不过这种组件是很少用的。

在一定程度上，动态绑定也同样适合用于处理COM变化类型。主要是由于它的必要性设计不好，所以COM API函数通常使用了很多这种类型，它差不多相当于.NET的object。如果在项目中启用"嵌入式互操作类型"（马上会详细介绍），那么运行时会将变化类型映射为dynamic，而不是将它们映射为object，从而不需要使用强制转换。例如，这样做是合法的：

```
excel.Cells [1, 1].Font.FontStyle = "Bold";
```

代替：

```
ver range=(Excel.Range)excel.Cells [1, 1];
range.Font.FontStyle = "Bold";
```

缺点是，你不能使用自动代码填充功能，因此你必须确切地知道属性Font的存在。因此，对比而言，将结果指定给已知的互操作类型比较简单一些。

```
Excel.Range range = excel.Cells [1, 1];
range.Font.FontStyle = "Bold";
```

如上所见，这只比旧版本省了五个字符！

将variant类型映射为动态类型是Visual Studio 2010的默认功能，该功能依赖于内嵌的互操作类型。

25.9 内嵌互操作类型

前面我们讲到C#一般通过*tlimp.exe*工具产生互操作类型，然后借助该类型（直接，或者通过Visual Studio）调用COM组件。

历史上唯一方法是像引用其他程序集一样引用互操作（interop）程序集。但这是很麻烦的，因为有复杂COM组件的加入，互操作程序集会变得很大。以一个为Microsoft Word端写的插件为例，它的互操作程序集会比自身大好几个数量级。

在C# 4.0中，不但可以使用互操作程序集，也可以直接连接到它。当对它进行直接连接时，编译器会分析应用程序中用到的类型和成员。然后将这些类型和成员直接定义在应用程序中。这时就不需要担心程序变大，因为只有真正用到的COM接口被包含在程序中。

互操作连接是Visual Studio 2010引用COM组件的默认功能。如果要禁用它，请选择工程浏览器的引用（reference）项，打开它的属性列表，将内嵌互操作类型设置为假。

如果想通过命令行的方式打开编译器的互操作连接功能，使用`/link`选项而非`/reference`（或者`/L`而非`/R`）去调用`csc`。

类型等价

CLR 4.0和4.5支持链接互操作类型等价。就是说如果两个程序集各自链接同一个COM类型，它们的类型被认为是等价的，哪怕它们链入的互操作程序集是相互独立的。

提示：类型等价依赖于`System.Runtime.InteropServices`命名空间中的`TypeIdentifierAttribute`属性。当链接互操作程序集时，编译器自动使用该属性。如果COM类型具有相同的GUID，它们被认为是同一类型。

类型等价往往需要主互操作程序集。

25.10 主互操作程序集

C# 4.0出现之前，不存在互操作链接和类型等价。因此当两个使用者都通过*tlimp.exe*工具针对同一个COM组件产生自己的互操作程序集时，这两个程序集的不兼容性会阻碍互操作。变通的方法是需要COM库发布一个互操作程序集，叫做主互操作程序集（PIA）。但是由于以下原因，PIA是个失败的解决方案：

PIA没有真正被用到

因为很多使用者都会使用库导入工具产生自己的版本，而不使用官方版本。有时这也是迫于无奈，因为COM并没有发布PIA。

PIA需要注册

PIA需要在全局程序集缓存（GAC）中注册。这加重了为COM组件编写小型插件的负担。

PIA膨胀开始

PIA例证前面我们讲到的互操作程序集会变大的问题。尤其是，Microsoft Office开发团队没有发布他们产品的PIA。

25.11 COM中调用C#对象

让COM调用C#中的对象也是允许的。CLR通过一种COM可调用封装器（COM-Callable Wrapper，简称CCW）技术实现这个功能。CCW在两种环境中封送相关类型，并且实现了COM协议所需的IUnknown接口。CCW是由COM端的引用计数控制生命周期的（而非由CLR端的垃圾回收器控制）。

可以提供任何类型给COM。要求是需要定义一个程序集GUID属性作为这个COM库唯一性标识：

```
[assembly: Guid ("...")]     // COM类独一无二的GUID
```

默认情况下，所有的类型对COM都是可见的。如果想设置某些类型不可见，需要使用[ComVisible(false)]属性。如果想要所有的属性都不可见，需要将整个程序集设置为[ComVisible(false)]。对于想要可见的类型设置为[ComVisible(true)]。

最后一步是调用*tlbexp.exe*工具：

```
tlbexp.exe myLibrary.dll
```

这样就生成了COM类型库文件（*.tlb*），注册后就可以在COM程序中使用它了。与可见类型的COM接口会自动生成。

第26章 正则表达式

正则表达式可以对字符串进行模式化识别。.NET中的正则表达式规范是基于编程语言Perl5的，并且支持查找替换功能。

正则表达式一般用于处理下列问题：

- 判定输入字符是否是密码或者手机号（ASP.NET提供的RegularExpressionValidator功能正是为了实现这个目的）
- 将文本数据转换成结构化形式（例如，从HTML文本中提取数据存入数据库）
- 替换文档中固定形式的文本（例如，全字匹配）

本章分成正则表达式基础概念和正则表达式语言。

所有的正则表达式类型都定义在System.Text.RegularExpressions中。

提示： 想要了解更多正则表达式相关内容，请登录网址*http://regular-expressions.info*，该网址提供了很多示例程序。另外认真阅读Jeffrey E. F. Friedl的文章*Mastering Regular Expressions*也会获得很多有价值的信息。

本章中的所有示例都被提前载入到LINQPad工具中。另一个很适用正则表达式交互分析工具是Expresso，它可以借助自身的正则表达式库，协助创建和可视化正则表达式。

26.1 正则表达式基础

一个常用的正则表达式运算符是量词。量词“?”表示前面的字符出现一次或者零次。换句话说，“?”表示前面的字符是可选的。前面的字符可以是单个字符，也可以是放在方括号内的由多个字符构成的复杂结构。例如，正则表达式“colou?r”匹配color和colour，但是不匹配colouur。

```
Console.WriteLine (Regex.Match ("color",   @"colou?r").Success);  // True
Console.WriteLine (Regex.Match ("colour",  @"colou?r").Success);  // True
Console.WriteLine (Regex.Match ("colouur", @"colou?r").Success);  // False
```

Regex.Match方法可以搜索大型字符串。它返回的对象具有匹配的长度、索引位和匹配的真实值等属性：

```
Match m = Regex.Match ("any colour you like", @"colou?r");

Console.WriteLine (m.Success);     // True
Console.WriteLine (m.Index);       // 4
Console.WriteLine (m.Length);      // 6
Console.WriteLine (m.Value);       // colour
Console.WriteLine (m.ToString());  // colour
```

可以将Regex.Match方法认为是字符串索引方法IndexOf的增强版。不同的是Regex.Match搜索的是一种模式而非普通字符串。

IsMatch方法是Match的一种捷径，它首先调用Match方法，然后判断返回对象的Success属性。

默认状态下，正则表达式引擎按照字符串从左到右的顺序进行匹配，所以返回的是左起第一个匹配字符串。可以使用NextMatch方法返回更多的匹配值：

```
Match m1 = Regex.Match ("One color? There are two colours in my head!",@"colou?rs?");
Match m2 = m1.NextMatch();
Console.WriteLine (m1);         // color
Console.WriteLine (m2);         // colours
```

Matches方法通过数组返回所有的匹配值。我们可以重写前一个例子：

```
foreach (Match m in Regex.Matches
          ("One color? There are two colours in my head!", @"colou?rs?"))
  Console.WriteLine (m);
```

另一个常见的正则表达式运算符是交替符，用一个竖线表示——“|”。交替符前后的表达式是可选的。以下例中的表达式可以匹配“Jen”、“Jenny”和“Jennifer”：

```
Console.WriteLine (Regex.IsMatch ("Jenny", "Jen(ny|nifer)?"));  // True
```

圆括号将可选的表达式同其他表达式分隔开。

提示： 从Framework 4.5开始，我们可以在匹配正则表达式时指定一个超时时间。如果匹配操作超过指定的时间间隔TimeSpan，就会抛出一个RegexMatchTimeoutException异常。如果程序需要处理一些很随意的正则表达式（例如在高级搜索对话框中），那么这个参数很好用，因为它可以防止一些错误正则表达式导致无限计算。

26.1.1 编译正则表达式

前面的示例中，我们基本上采用相同的模式——重复调用RegEx静态方法。我们还可以实例化一个Regex对象，然后调用该对象的方法：

```
Regex r = new Regex (@"sausages?");
Console.WriteLine (r.Match ("sausage"));   // sausage
Console.WriteLine (r.Match ("sausages"));  // sausages
```

这样不仅语法上方便：在底层，RegEx的实例使用了轻量级的代码生成器（Reflection.Emit中的DynamicMethod），针对特定的正则表达式编写和编译了代码。这提高了匹配速度，却只消耗了很少的代码初始编译时间——几十毫秒。

Regex实例是不可更改的。

提示： 正则表达式匹配引擎是很快的。就算没有编译，一个简单的匹配也用不了一毫秒。

26.1.2 RegexOptions属性

RegexOptions标志可以控制正则表达式匹配的行为。一个常见用法是不区分大小写的搜索：

```
Console.WriteLine (Regex.Match ("a", "A", RegexOptions.IgnoreCase)); // a
```

大多数RegexOptions标志也可以在正则表达式内通过选项字母被激活，例如：

```
Console.WriteLine (Regex.Match ("a", @"(?i)A"));                     // a
```

可以通过一个表达式打开和关闭这个选项：

```
Console.WriteLine (Regex.Match ("AAAa", @"(?i)a(?-i)a"));            // Aa
```

另一个选项是IgnorePatternWhitespace或者(?x)。这允许插入一些空白以增强正则表达式的可读性，不使用该选项的话，空白就按照真实字符去理解。

表 26-1 列出了所有的RegExOptions选项和与之对应的正则表达式中的选项字母。

表 26-1：正则表达式选项

枚举值	选项字母	介绍
None		
IgnoreCase	i	不区分大小写（默认时，正则表达式是区分大小写的）
Multiline	m	修改^和$，使得可以匹配一行的开始/终止，而不是一个 字符串的开始/终止
ExplicitCapture	n	捕获确定的名称或者确定编号的序列
Compiled	c	为正则表达式编译产生中间语言（IL）
Singline	s	用匹配所有字符，包括\n
IgnorePattern Whitespace	x	忽略模式中的所有空格
RightToLeft	r	从右至左搜索，这个选项不能在中间（midstream）指定
ECMAScript		符合ECMA标准（默认情况下，不按照ECMA标准实施）
CultureInvariant		关闭CurrentCulture行为（默认情况下CultureInfo.CurrentCulture影响正则匹配）

26.1.3 字符转义

正则表达式有以下几种元字符，它们是有特殊含义的：

```
\ * + ? | { [ () ^ $ . #
```

当要查找的串中含有元字符，需要在元字符前加反斜杠。例如下面的示例中，我们要查找元字符"?"，如果想要查找正确，则需要在其前面加上反斜杠，构成字符转义：

```
Console.WriteLine (Regex.Match ("what?", @"what\?")); // what? (正确)
Console.WriteLine (Regex.Match ("what?", @"what?"));  // what  (错误)
```

警告：如果这个字符在方括号（set）内，这个规则就不适用了，此时元字符就是其字符意思。我们将在下面的章节中介绍方括号。

Regex的Escape方法可以将字符串中的元字符转换成转义后的等价形式。Unescape刚好相反，例如：

```
Console.WriteLine (Regex.Escape   (@"?"));     // \?
Console.WriteLine (Regex.Unescape (@"\?"));    // ?>
```

本章中的所有正则表达式，我们都使用了@修饰。这就忽视了C#中使用反斜杠的字符转义机制。如果没有@修饰，一个字面意思上的反斜杠需要用四个反斜杠来表示：

```
Console.WriteLine (Regex.Match ("\\", "\\\\"));    // \
```

除非加入(?x)选项，正则匹配中的空白会被当作真正存在：

```
Console.Write (Regex.IsMatch ("hello world", @"hello world"));  // True
```

26.1.4 字符组（方括号[]）

字符组即方括号扮演着正则表达式中一组字符的通配符的角色。

表达式	含义	反义表达
[abcdef]	匹配括号内的某个字符	[^abcdef]
[a-f]	匹配a到f范围内的某一个字符	[^a-f]
\d	匹配一个十进制数字，例如0~9	\D
\w	匹配一个单词字符（默认情况下，根据CultureInfo.CurrentCulture的不同而变化，例如，英语环境下，同[a-zA-Z_0-9]）	\W
\s	匹配空白字符，例如[\n\r\t\f]	\S
\p{category}	匹配category中限定的字符类型	\P
.	（默认模式）匹配所有字符，但不包括\n	\n
.	（单行模式）匹配所有字符	\n

为了匹配一组字符中的某一个，可以将该组字符放到方括号内：

```
Console.Write (Regex.Matches ("That is that.", "[Tt]hat").Count);   // 2
```

为了在匹配时排除一组字符，可以将该组字符放入方括号内，字符前用^修饰：

```
Console.Write (Regex.Match ("quiz qwerty", "q[^aeiou]").Index);    // 5
```

可以通过连字符定义一组字符。下面的正则表达式匹配了棋子的移动：

```
Console.Write (Regex.Match ("b1-c4", @"[a-h]\d-[a-h]\d").Success);  // True
```

\d表示一个十进制数字，所以\d可以匹配任何数字。\D表示非数字。

\w表示一个单词字符，包括字母、数字和下划线。\W表示非单词字符，可以用于表示非英语字母，例如斯拉夫字母。

.匹配所有字符，除了\n（但是包括\r）。

\p匹配category所指定类型的字符，例如{Lu}表示大写字符，{P}表示标点符号（我们在随后的参考章节中列出了所有的类型定义）

```
Console.Write (Regex.IsMatch ("Yes, please", @"\p{P}"));   // True
```

如果将\d、\w、.与量词一起使用，可以得到很多的变化。

26.2 量词

量词定义了匹配特定字符的次数。

量词	含义
*	零次或者多次匹配
+	一次或者多次匹配
?	零次或者一次匹配
{*n*}	*n*次匹配
{*n*,}	至少*n*次匹配
{*n*,*m*}	匹配次数在*n*和*m*之间

量词*表示匹配之前的字符或者字符组零次或者更多次。下面这个正则表达式表示匹配*cv.doc*、或者它的任意版本（例如*cv2.doc*、*cv15.doc*）。

```
Console.Write (Regex.Match ("cv15.doc", @"cv\d*\.doc").Success);  // True
```

注意，我们需要在文件名后缀中的点号前加上反斜杠进行转义。

接下来的正则表达式允许在*cv*和*.doc*中插入任何值，它的作用等同于dir cv*.doc：

```
Console.Write (Regex.Match ("cvjoint.doc", @"cv.*\.doc").Success);  // True
```

量词+表示一次或者多次匹配之前的字符或者字符组，例如：

```
Console.Write (Regex.Matches ("slow! yeah slooow!", "slo+w").Count);  // 2
```

正则表达式

量词{}表示匹配确定的次数或者次数范围。接下来的正则表达式匹配了血压值：

```
Regex bp = new Regex (@"\d{2,3}/\d{2,3}");
Console.WriteLine (bp.Match ("It used to be 160/110"));  // 160/110
Console.WriteLine (bp.Match ("Now it's only 115/75"));   // 115/75
```

贪婪量词vs懒惰量词

默认情况下，量词都是贪婪的——相对于懒惰而言的。贪婪的量词会尽可能多地匹配目标字符串。懒惰的量词会尽可能少地匹配字符串。可以通过在量词后加?后缀，让量词变得懒惰。为了阐明这个问题，我们以下面的HTML语言片段为例：

```
string html = "<i>By default</i> quantifiers are <i>greedy</i> creatures";
```

加入我们要提取斜体部分的短语。我们可以按照下面的正则表达式进行提取：

```
foreach (Match m in Regex.Matches (html, @"<i>.*</i>"))
  Console.WriteLine (m);
```

结果如下，不是两处匹配，而是仅有一处匹配：

```
<i>By default</i> quantifiers are <i>greedy</i>
```

原因是量词*在匹配到字符</i>之前，重复尽可能多的（贪婪的）次数，所以它选择了右侧第一个</i>，在最后一个</i>处才停止（这个位置还是可以再进行匹配的），完成了第一次匹配。

如果我们使得量词懒惰：

```
foreach (Match m in Regex.Matches (html, @"<i>.*?</i>"))
  Console.WriteLine (m);
```

量词*停止在了第一次出现</i>处，其他字符仍然可以继续匹配，结果如下：

```
<i>By default</i>
<i>greedy</i>
```

26.3 零宽度断言

正则表达式允许在匹配处的前面或者后面放置约束条件，这些条件包括前瞻条件（lookahead）、后瞻条件（lookbehind）、锚点（anchors）和单词边界（word boundaries）。这些条件叫做零宽度断言（zero-width assertions），因为它们并没有增加匹配字符的宽度或者长度。

26.3.1 前瞻条件和后瞻条件

表达式(?=expr)表示匹配处前面的字符是否与expr表示的字符串相同，expr字符串本身并不作为匹配结果的一部分，这被称作正前瞻。下面的示例中，我们搜索“miles”后的数字：

```
Console.WriteLine (Regex.Match ("say 25 miles more", @"\d+\s(?=miles)"));
```

输出：25

注意，尽管字符串“miles”被用作匹配条件，但是它并没有被作为结果返回。

忽略前面的匹配，将上述正则表达式追加.*，再次进行匹配，代码如下：

```
Console.WriteLine (Regex.Match ("say 25 miles more", @"\d+\s(?=miles).*"));
```

输出：`25 miles more`

在校验一个强密码时，前瞻条件是很有用的。假设这个密码至少六个字符，并且至少含有一个数字。借助前瞻条件，我们可以通过如下代码实现密码校验：

```
string password = "...";
bool ok = Regex.IsMatch (password, @"(?=.*\d).{6,}");
```

代码中的前瞻条件首先保证了字符串中的某一处会出现一个数字。如果满足了这个条件，它会返回数字出现的位置，然后验证是否满足最少六个字符的要求。（在本章“正则表达式实例”一节，我们给出一个更加完整的密码验证示例。）

相反的表达是负前瞻条件(*?!expr*)。它表示匹配处跟随的字符不是*expr*所表示的字符串。下面的示例中，匹配了字符串“good”，但是它后面出现“however”或者“but”：

```
string regex = "(?i)good(?!.*(however|but))";
Console.WriteLine (Regex.IsMatch ("Good work! But...",  regex));  // False
Console.WriteLine (Regex.IsMatch ("Good work! Thanks!", regex));  // True
```

表达式(*?<=expr*)表示正后瞻，它对匹配处前面的字符加了正的限制条件，要求匹配处前面有expr表示的字符串。相反的表达式是(*?<!expr*)——负后瞻，它对匹配处前面的字符加了负的限制条件，要求匹配处前面不能出现expr表示的字符串。例如，下面的正则表达式中，匹配字符串是“good”，并且前面需要出现“however”：

```
string regex = "(?i)(?<!however.*)good";
Console.WriteLine (Regex.IsMatch ("However good, we...", regex)); // False
Console.WriteLine (Regex.IsMatch ("Very good, thanks!", regex));  // True
```

我们可以通过加入单词边界断言来改进这些示例，我们稍后介绍单词边界断言。

26.3.2 锚点

锚点^和$代表确定的位置。默认表示：

^

匹配字符串的开头

$

匹配字符串的结束

提示： ^根据上下文有两种不同的意思：首先，它可以作为一个锚点，其次作为字符类否定修饰符。

$根据上下文也会出现两种不同的意思：首先，它可以作为一个锚点，其次作为替代组的标志。

例如：

```
Console.WriteLine (Regex.Match ("Not now", "^[Nn]o"));   // No
Console.WriteLine (Regex.Match ("f = 0.2F", "[Ff]$"));   // F
```

如果定义了RegexOptions.Multiline或者在表达式中包含了(?m)：

- ^匹配字符串的开头或者行的开头（直接在\n之后）
- $匹配字符串的结束或者行的结束（直接在\n之前）

在多行模式下使用$需要注意：Windows下新的一行是通过\r\n而非\n来指示的。也就是说，如果想让$起作用，需要通过正前瞻同时匹配\r：

```
(?=\r?$)
```

正前瞻保证了\r不会成为结果的一部分。接下来的正则表达式匹配了以".txt"结尾的行：

```
string fileNames = "a.txt" + "\r\n" + "b.doc" + "\r\n" + "c.txt";
string r = @".+\.txt(?=\r?$)";
foreach (Match m in Regex.Matches (fileNames, r, RegexOptions.Multiline))
  Console.Write (m + " ");
```

输出： a.txt c.txt

下面的正则表达式匹配了字符串s的所有空行：

```
MatchCollection emptyLines = Regex.Matches (s, "^( ?=\r?$)",
                                            RegexOptions.Multiline);
```

接下来的正则表达式匹配了空行或者只包含空白的行：

```
MatchCollection blankLines = Regex.Matches (s, "^[ \t]*(?=\r?$)",
                                            RegexOptions.Multiline);
```

提示：因为锚点只表示一个位置，而不是一个字符，所以锚点的匹配得到的只是一个空串：

```
Console.WriteLine (Regex.Match ("x", "$").Length);   // 0
```

26.3.3 单词边界

单词边界断言\b匹配与单词（\w）毗邻的边界位置：

- 非单词字符（\W）
- 字符的开头/结束（^和$）

\b常用来匹配整个单词，例如：

```
foreach (Match m in Regex.Matches ("Wedding in Sarajevo", @"\b\w+\b"))
  Console.WriteLine (m);
```

输出：
Wedding
In
Sarajevo

下面的示例强调了单词边界的作用：

```
int one = Regex.Matches ("Wedding in Sarajevo", @"\bin\b").Count;  // 1
int two = Regex.Matches ("Wedding in Sarajevo", @"in").Count;      // 2
```

下面的查询通过使用正前瞻返回了“(sic)”前面的字符：

```
string text = "Don't loose (sic) your cool";
Console.Write (Regex.Match (text, @"\b\w+\b\s(?=\(sic\))")); // loose
```

26.4 分组

有时将正则表达式分成几个子表达式或者分组是很有效的。下面的正则表达式代表了美国的电话号码，例如206-465-1918：

```
\d{3}-\d{3}-\d{4}
```

假设我们希望将它分成两组：地区编号和本地号码。我们可以通过圆括号达到目的：

```
(\d{3})-(\d{3}-\d{4})
```

此时我们可以编写如下程序得到分组：

```
Match m = Regex.Match ("206-465-1918", @"(\d{3})-(\d{3}-\d{4})");

Console.WriteLine (m.Groups[1]);   // 206
Console.WriteLine (m.Groups[2]);   // 465-1918
```

第0组代表了整个匹配。换句话说，它和匹配的返回值相同：

```
Console.WriteLine (m.Groups[0]);   // 206-465-1918
Console.WriteLine (m);             // 206-465-1918
```

分组本身也可以成为正则表达式的一部分。也就是说可以在正则表达式中引用分组。\n语法允许在正则表达式中通过分组编号n索引分组。例如，正则表达式(\w)ee\1匹配deed和peep。在接下来的示例中，我们查找了字符串中开头字母和结尾字母相同的所有单词：

```
foreach (Match m in Regex.Matches ("pop pope peep", @"\b(\w)\w+\1\b"))
  Console.Write (m + " ");  // pop peep
```

\w两边的括号指示正则表达式引擎存储这个子匹配（在这个示例中，是一个字母）到一个分组，以便于后边的引用。我们通过\1引用这个分组，意味着引用了正则匹配中的第一个分组。

命名分组

在一个长且复杂的正则表达式中，通过名称引用分组要比通过索引值引用简单得多。下面我们重写了前面的示例，将其中一个分组命名为'letter'：

```
string regEx =
  @"\b"              + // 单词边界
  @"(?'letter'\w)"   + // 匹配第一个字母，命名分组为'letter'
  @"\w+"             + // 匹配中间的字母
  @"\k'letter'"      + // 匹配最后一个字母，引用分组名为 'letter'的分组
  @"\b";               // 单词边界

foreach (Match m in Regex.Matches ("bob pope peep", regEx))
  Console.Write (m + " ");  // bob peep
```

命名一个捕获到的分组：

正则表达式

```
(?'group-name'group-expr)  or  (?<group-name>group-expr)
```

引用这个分组：

```
\k'group-name'  or  \k<group-name>
```

接下来的示例通过查找节点的开头和结束标记，匹配了一个简单（非嵌套）的XML/HTML元素：

```
string regFind =
  @"<(?'tag'\w+?).*>"          +  // 匹配了开头标记，命名为'tag'
  @"(?'text'.*?)"              +  // 匹配了中间内容，命名为'text'
  @"</\k'tag'>";                  // 匹配了结束标记，命名为'tag'

Match m = Regex.Match ("<h1>hello</h1>", regFind);
Console.WriteLine (m.Groups ["tag"]);          // h1
Console.WriteLine (m.Groups ["text"]);         // hello
```

将XML结构的所有可能变化考虑在内，例如嵌套元素，所以它是很复杂的。.NET的正则表达式引擎也有叫做匹配平衡构造（matched balanced constructs）的复杂扩展去辅助嵌套标签。可以通过网络查找相关知识，另外Jeffrey E. F. Friedl的《Mastering Regular Expressions》一书中也有详细讲解。

26.5 文本替换和拆分

RegEx.Replace方法与string.Replace的功能类似，不过它使用正则表达式进行查找。

下面的示例将“cat”替换为“dog”。它不会像string.Replace一样，将“catapult”变成“dogapult”，因为我们匹配了单词边界：

```
string find = @"\bcat\b";
string replace = "dog";
Console.WriteLine (Regex.Replace ("catapult the cat", find, replace));
```

输出：catapult the dog

替换字符串可以通过替代结构$0访问原始匹配。下面示例中的正则表达式作用是将字符串中的数组用尖括号括起来：

```
string text = "10 plus 20 makes 30";
Console.WriteLine (Regex.Replace (text, @"\d+", @"<$0>"));
```

输出：<10> plus <20> makes <30>

可以通过$1、$2、$3等访问捕获到的分组，或者通过${name}访问捕获到的命名分组。为了说明这些功能的好处，以前面我们用来分析的简单XML元素的正则表达式为例。通过重新分组，我们可以创建一个用于替换的正则表达式，将元素的内容放入XML的属性中：

```
string regFind =
  @"<(?'tag'\w+?).*>"      +  // 匹配了开头标记，命名为'tag'
  @"(?'text'.*?)"          +  // 匹配了中间内容，命名为'text'
  @"</\k'tag'>";              // 匹配了结束标记，命名为'tag'

string regReplace =
  @"<${tag}"               +  // <tag
  @"value="""              +  // value="
```

```
    @"${text}"          + // text
    @"""/>";              // "/>

Console.Write (Regex.Replace ("<msg>hello</msg>", regFind, regReplace));
```

结果是：

```
<msg value="hello"/>
```

26.5.1 MatchEvaluator 委托

Replace方法有一个重载，重载中使用了MatchEvaluator委托，该委托在每次匹配时都会被调用。这就允许在正则表达式不能实现时，将替换字符串委托给C#代码。例如：

```
Console.WriteLine (Regex.Replace ("5 is less than 10", @"\d+",
                   m => (int.Parse (m.Value) * 10).ToString()) );
```

输出：50 is less than 100

在后面的示例中，我们演示了怎样使用MatchEvaluator为HTML适当的转义Unicode字符。

26.5.2 拆分单词

静态的Regex.Split方法是string.Split方法加强版。它使用了正则表达式替换了分隔符的模式。接下来的示例，我们以十进制数字为界限分割一个字符串：

```
foreach (string s in Regex.Split ("a5b7c", @"\d"))
  Console.Write (s + " ");      // a b c
```

结果中没有分隔符本身。使用正前瞻包装正则表达式可以将分隔符包含在结果中。下面的示例展示了将使用了骆峰命名法（camel-case）的字符串分割成单词的方法：

```
foreach (string s in Regex.Split ("oneTwoThree", @"(?=[A-Z])"))
  Console.Write (s + " ");    // one Two Three
```

26.6 正则表达式实例

1. 匹配美国社会保险号/电话号码

```
string ssNum = @"\d{3}-\d{2}-\d{4}";

Console.WriteLine (Regex.IsMatch ("123-45-6789", ssNum));      // True

string phone = @"(?x)
  ( \d{3}[-\s] | \(\d{3}\)\s? )
    \d{3}[-\s]?
    \d{4}";

Console.WriteLine (Regex.IsMatch ("123-456-7890",   phone));   // True
Console.WriteLine (Regex.IsMatch ("(123) 456-7890", phone));   // True
```

2. 提取“name=value”对（一行一个）

注意，这里需要多行指示符(?m)：

```
string r = @"(?m)^\s*(?'name'\w+)\s*=\s*(?'value'.*)\s*(?=\r?$)";

string text =
  @"id = 3
    secure = true
    timeout = 30";

foreach (Match m in Regex.Matches (text, r))
  Console.WriteLine (m.Groups["name"] + " is " + m.Groups["value"]);
```

输出：*id is 3 secure is true timeout is 30*

3. 强密码验证

接下来的代码用来校验密码是否是六位以上，是否含有一个数字、一个符号或者一个标点符号：

```
string r = @"(?x)^(?=.* ( \d | \p{P} | \p{S} )).{6,}";

Console.WriteLine (Regex.IsMatch ("abc12", r));     // False
Console.WriteLine (Regex.IsMatch ("abcdef", r));    // False
Console.WriteLine (Regex.IsMatch ("ab88yz", r));    // True
```

4. 每行至少80个字符

```
string r = @"(?m)^.{80,}(?=\r?$)";

string fifty = new string ('x', 50);
string eighty = new string ('x', 80);

string text = eighty + "\r\n" + fifty + "\r\n" + eighty;

Console.WriteLine (Regex.Matches (text, r).Count);   // 2
```

5. 转换日期/时间（N/N/N H:M:S AM/PM）

这个正则表达式处理各种形式日期的格式化——无论年份在前还是在后。(?x)指示符通过允许空白增强了可读性；(?i)关闭了区分大小写模式（因为AM/PM标识符有大写的可能）。可以通过分组访问每个匹配的部件：

```
string r = @"(?x)(?i)
 (\d{1,4}) [./-]
 (\d{1,2}) [./-]
 (\d{1,4}) [\sT]
 (\d+):(\d+):(\d+) \s? (A\.?M\.?|P\.?M\.?)?";

string text = "01/02/2008 5:20:50 PM";

foreach (Group g in Regex.Match (text, r).Groups)
  Console.WriteLine (g.Value + " ");
```

输出：*01/02/2008 5:20:50 PM 01 02 2008 5 20 50 PM*

6. 匹配罗马字符

```
string r =
  @"(?i)\bm*"             +
  @"(d?c{0,3}|c[dm])"     +
  @"(l?x{0,3}|x[lc])"     +
  @"(v?i{0,3}|i[vx])"     +
```

```
  @"\b";
Console.WriteLine (Regex.IsMatch ("MCMLXXXIV", r));   // True
```

7. 删除重复单词

这里我们捕获一个命名为dupe的分组：

```
string r = @"(?'dupe'\w+)\W\k'dupe'";

string text = "In the the beginning...";
Console.WriteLine (Regex.Replace (text, r, "${dupe}"));
```

输出：*In the beginning*

8. 单词数量

```
string r = @"\b(\w|[-'])+\b";

string text = "It's all mumbo-jumbo to me";
Console.WriteLine (Regex.Matches (text, r).Count);   // 5
```

9. 匹配GUID

```
string r =
  @"(?i)\b"              +
  @"[0-9a-fA-F]{8}\-"   +
  @"[0-9a-fA-F]{4}\-"   +
  @"[0-9a-fA-F]{4}\-"   +
  @"[0-9a-fA-F]{4}\-"   +
  @"[0-9a-fA-F]{12}"    +
  @"\b";

string text = "Its key is {3F2504E0-4F89-11D3-9A0C-0305E82C3301}.";
Console.WriteLine (Regex.Match (text, r).Index);              // 12
```

10. 解析XML/HTML标签

正则表达式在解析HTML片段时是非常有用的——尤其是当文档没有完美形成时：

```
string r =
  @"<(?'tag'\w+?).*>"+  // 匹配了开头标记，命名为'tag'
  @"(?'text'.*?)"    +  // 匹配了中间内容，命名为 it 'textd'
  @"</\k'tag'>";        // 匹配了结束标记，命名为 'tag'

string text = "<h1>hello</h1>";

Match m = Regex.Match (text, r)

Console.WriteLine (m.Groups ["tag"]);        // h1
Console.WriteLine (m.Groups ["text"]);       // hello
```

11. 分隔骆峰命名法单词

这里需要一个正前瞻将作为分隔符的大写字母包含在返回结果中：

```
string r = @"(?=[A-Z])";
```

```
foreach (string s in Regex.Split ("oneTwoThree", r))
  Console.Write (s + " ");     // one Two Three
```

12. 获得合法的文件名

```
string input = "My \"good\" <recipes>.txt";

char[] invalidChars = System.IO.Path.GetInvalidPathChars();
string invalidString = Regex.Escape (new string (invalidChars));

string valid = Regex.Replace (input, "[" + invalidString + "]", "");
Console.WriteLine (valid);
```

输出：**My good recipes.txt**

13. 为HTML转义Unicode字符

```
string htmlFragment = "  2007";

string result = Regex.Replace (htmlFragment, @"[\u0080-\uFFFF]",
                m => @"&#" + ((int)m.Value[0]).ToString() + ";");

Console.WriteLine (result);        // &#169; 2007
```

14. 反转义HTTP查询字符串中的字符

```
string sample = "C%23 rocks";

string result = Regex.Replace (
    sample,
    @"%[0-9a-f][0-9a-f]",
    m => ((char) Convert.ToByte (m.Value.Substring (1), 16)).ToString(),
    RegexOptions.IgnoreCase
);

Console.WriteLine (result);   // C# rocks
```

15. 从网站统计日志中解析谷歌搜索关键词

这里需要和前面用于查询字符串反转义的表达式一起使用：

```
string sample =
  "http://google.com/search?hl=en&q=greedy+quantifiers+regex&btnG=Search";

Match m = Regex.Match (sample, @"(?<=google\..+search\?.*q=).+?(?=(&|$))");

string[] keywords = m.Value.Split (
new[] { '+' }, StringSplitOptions.RemoveEmptyEntries);

foreach (string keyword in keywords)
  Console.Write (keyword + " ");       // 贪婪量词正则表达式
```

26.7 正则表达式语言参考

表26-2～表26-12总结了.NET中的正则表达式的语法和语法支持。

表26-2：转义字符

转义字符序列（元字符）	意义	等价16进制字符
\a	铃声（警报）	\u0007
\b	退格符	\u0008
\t	制表符	\u0009
\r	回车符	\u000A
\v	垂直制表符	\u000B
\f	换页符	\u000C
\n	换行符	\u000D
\e	转义符	\u001B
\nnn	十进制ASCII字符（例如\n052）	
\xnn	十六进制ASCII字符（例如\x3F）	
\cL	ASCII控制字符（例如\cG表示Ctrl+G）	
\unnnn	十六进制Unicode字符（例如\u07DE）	
\symbol	非转移符号	

特殊情况：在正则表达式中，\b表示一个单词边界，但是当\b位于方括号中时，代表退格符。

表 26-3：字符组

表达式	含义	反义表达
[abcdef]	匹配括号内的某个字符	[^abcdef]
[a-f]	匹配a到f范围内的某个字符	[^a-f]
\d	匹配一个十进制数字，例如0~9	\D
\w	匹配一个单词字符（默认情况下，根据CultureInfo.CurrentCulture的不同而变化，例如，英语环境下，同[a-zA-Z_0-9]）	\W
\s	匹配空白字符，例如[\n\r\t\f]	\S
\p{category}	匹配category限定的字符类型（详见表26-6）	\P
.	（默认模式）匹配所有字符，但不包括\n	\n
.	（单行模式）匹配所有字符	\n

表26-4：字符分类

量词	意义
\p{L}	字符
\p{Lu}	大写字符
\p{Ll}	小写字符
\p{N}	数字

表26-4：字符分类（续）

量词	意义
\p{P}	标点符号
\p{M}	变音符
\p{S}	符号
\p{Z}	分隔符
\p{C}	控制符

表 26-5：量词

量词	意义
*	零次或者多次匹配
+	一次或者多次匹配
?	零次或者一次匹配
{*n*}	*n*次匹配
{*n*,}	至少*n*次匹配
{*n*,*m*}	匹配次数在*n*和*m*之间

表 26-6：替换

表达式	意义
$0	用匹配字符串替换
$group-number	用索引值（group-number）所代表的匹配分组替换
${group-name}	用名称（group-name）所代表的匹配分组替换

只有在文本替换模式下，才允许定义替换表达式。

表 26-7：零宽度断言

表达式	意义
^	字符串开头（或者多行模式下行开头）
$	字符串结尾（或者多行模式下行结尾）
\A	字符串开头（忽略多行模式）
\z	字符串结尾（忽略多行模式）
\Z	行或者字符串的结尾
\G	搜索开始处
\b	单词边界
\B	非单词边界

表 26-7：零宽度断言（续）

表达式	意义
(?=expr)	匹配处右侧为字符串expr（正前瞻）
(?!expr)	匹配处右侧不为字符串expr（负前瞻）
(?<=expr)	匹配处左侧为字符串expr（正后瞻）
(?<!expr)	匹配处左侧不为字符串expr（负后瞻）
(?>expr)	子正则表达式expr匹配一次，并且不回溯

表26-8：分组构造

表达式	意义
(expr)	将匹配expr的字符串捕获到分组中
(?number)	将匹配子字符串捕获到特定编号（number）分组中
(?'name')	将匹配子字符串捕获到命名（名字为name）分组中
(?'name1-name2')	删除先前定义的name2组并在name1组中存储先前定义的name2组和当前组之间的间隔。如果未定义name2组，则匹配将回溯
(?:expr)	非捕获分组

表26-9：先后引用构造

参数语法	意义
\index	通过索引引用以前捕获值
\k<index>	通过名称引用以前捕获值

表26-10：替换构造

<table>
<tr><th>表达式</th><th>意义</th></tr>
<tr><td>|</td><td>逻辑与（译者注：可选符号）</td></tr>
<tr><td>(?(expr)yes|no)</td><td>如果表达式在此位置匹配，则与“yes”部分匹配；否则，与“no”部分匹配。“no”部分可省略</td></tr>
<tr><td>(?(name)yes|no)</td><td>如果命名捕获字符串有匹配，则与“yes”部分匹配；否则，与“no”部分匹配。“no”部分可省略</td></tr>
</table>

表26-11：混杂构造

表达式	意义
(?#comment)	插入到正则表达式内部的内联注释
#comment	注释以#开头，直到行的结尾（只工作在IgnorePatternWhitespace模式下）

表26-12：正则表达式选项

选项	意义
(?i)	不区分大小写
(?m)	多行
(?n)	捕捉确定名称或者数字的分组
(?c)	为正则表达式编译产生中间语言（IL）
(?s)	单行模式，修改了“.”的含义，使得它可以匹配所有字符包括\n
(?x)	消除非转义空白符
(?r)	从右至左搜索，这个选项不能在中间（midstream）指定

作者简介

Joseph Albahari是*C# 4.0 in a Nutshell*、*LINQ Pocket Reference*和*C# 5.0 Pocket Reference*的作者。他有15年的.NET及其他平台的大型企业应用开发经验，也是流行的LINQ数据库查询工具LINQPad的作者。Joseph的个人主页（*http://www.albahari.com*）。

Ben Albahari是Take On It的创始人。他在微软担任过5年的项目经理，期间参与过多个项目，包括.NET Compact Framework和ADO.NET。他是Genamics的共同创始人，这家公司专门开发C#和J++编程工具，以及用于DNA和蛋白质序列分析的软件。他是*C# Essentials*的作者之一，这本书是O' Reilly出版的第一本C#书籍，也是*C# in a Nutshell*的前身。

封面介绍

本书的封面动物是一只蓑羽鹤。蓑羽鹤（*Antropoides virgo*）因其体态优美和匀称而得名。这种鹤种原产于欧洲和亚洲，在冬天会迁徙到印度、巴基斯坦和非洲东北地区。

虽然蓑羽鹤是体型最小的鹤，但是它们对于其领土的捍卫与其他种类的鹤一样强悍，它们会大声警告其他的擅自闯入者。在必要的情况下，它们会飞起来。蓑羽鹤的巢会筑在高地而非湿地上，如果水源在200至500米内时，它们甚至会住在沙漠。它们有时候可以毫不费力地将巢筑好以便产卵，但是它们往往会将卵直接产在地面，然后只是用植被覆盖保护。

蓑羽鹤在有些国家被认为是好运的象征并且有时甚至通过法律对它们进行保护。