

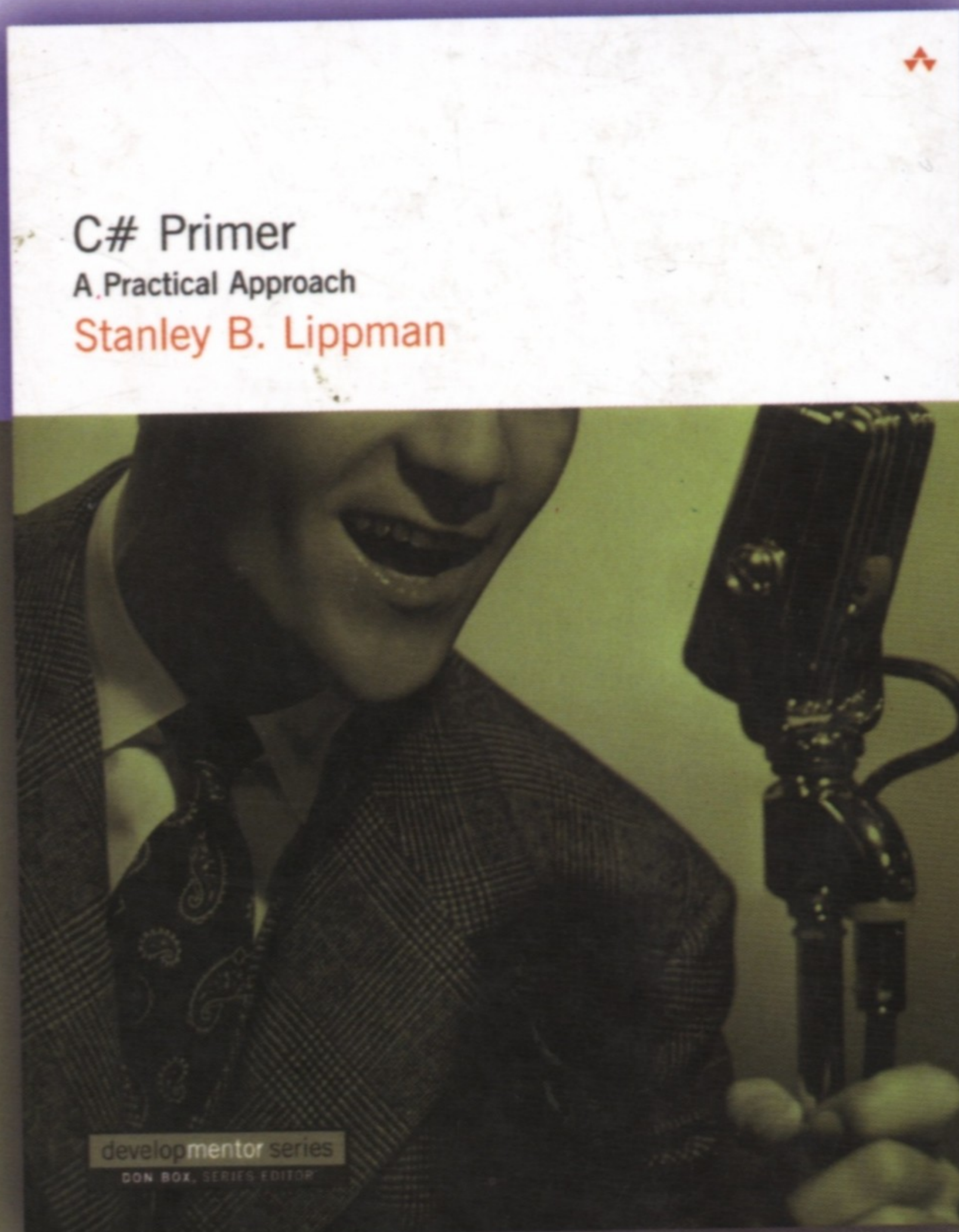


C# Primer 中文版

A Practical Approach

Stanley B. Lippman 著

侯捷 / 陈硕 合译



华中科技大学出版社

<http://press.hust.edu.cn>



TEAM TLFBOOK

译序 by 侯捷

历经前期的惨淡经营和后期数十年的蓬勃发展之后，OO（Object Oriented，面向对象）技术已成为学术界和工业界普遍支持与关注的一种软件技术。OO 技术细分为 OOP（Programming）、OOA（Analysis）、OOD（Design）……最基础的自然是 OOP。谈到 OOP 则不能不谈谈 OOP 语言（Object Oriented Programming Language，面向对象编程语言）。

目前软件业界存在三大主流 OOP 语言：C++，Java，C#。其中以 C++ 历史最久，C# 年纪最轻。就演化而言，愈新的 OOP 语言理应有着对 OO 观念的愈优秀支持（否则缔造者该打屁股㊄）。这个事实的确展现在上述三大 OOP 语言身上。当然，关键字（keywords）的多寡，以及关键字背后的语言机制的底层实现技术，并不能够总括一切价值——C++，Java，C# 各擅胜场，各有主攻。

在 C#（及其背后不可不提的 .NET Framework）问世不算太久的今天，我很高兴翻译完成这部由知名作家 Stanley B. Lippman 所写的《C# Primer》的中文版。这是一本轻量级小书，以 8 章 400 页的篇幅介绍了 C# 语法、OOP 概念，部分 .NET Framework、C#/.NET Framework 环境下的 Windows/Web 程序开发方法，以及 CLR（共通语言运行层）之中关于动态型别系统的一些技术内容。这些都是众多 C# 语言书很可触及的主题，谈不上独特（第 8 章也许稍微独特些）。本书也不是一本面面俱到的 C# 或 .NET Framework Library 百科全书。

是的，这本书就是一碟小菜。作者以其个人在技术写作上的丰富经验，为读者组织出一个容易亲近、容易学习的架构和一些有趣的样例。

这是我个人翻译过的第四本 Stanley B. Lippman 著作（前三本是《C++ Primer》、《Inside the C++ Object Model》和《Essential C++》）。Lippman 的写作向来有“大处着眼，小处不甚严谨”的特性，这个特性在本书依然存在。部分原因和 C# 的年轻

有关——或者说应该说主要和 .NET Framework Library 的年轻有关！由于规格 (spec.) 上的变化，导致书中内容偶与目前最新版本有所出入。译者在翻译过程中力求实际，脚踏书中样例细节，并将验证结果补充上来，弥补原书小节不矜的缺点，以及大环境激烈变动下的遗憾。当然啦，脚踏工作很难保证滴水不漏。

本书由我和陈硕先生合译。陈硕负责初译与术语转换，我负责其余一切。中译本（尤其是后半部）对原书有相当补充，以译注方式呈现，这些全归功于陈硕先生的勤奋和认真，以及游刃有余的实务能力。书中亦保留了对原书的某些勘误标记，例如“原书有误，已更改如上”等等说明，为的是方便可能的中英对照读者，使他们不至于一时困惑。本书勘误表由侯捷网站维护，网址见文末。

对于从未踏入 Object Oriented Programming（面向对象编程）领域的人们，任何一种 OOPL 都不易学习。因为相较于 Procedural-Based Programming（过程编程），这是一种思考模式上的大扭转。然而 OO 技术是大势所趋，早进入胜过晚进入，晚进入胜过不进入。路途多艰，唯勤是岸。

对于已经拥有 OOP 经验的人们，学习 C# 则是件轻松不过的事。C++、Java、C# 三者的语法和编程概念，绝大部分彼此兼容。精一而通三，我常这么说◎

网络论坛上，最容易引发众多讨论并且最容易撩拨民粹情绪的帖子，莫如侈言语言的优劣和前途。在 Java、C# 相继进入原本 C++ 擅专的领域之后，这样的讨论不时可见，日复一日。在一个容许并承认多元价值的世界里，这类口水之争实在没有意义，冗长而踊跃的发言涵盖不了本质的苍白。并非语言的比较没有价值，而是语言的价值没得比较。

人们总是“就已知学未知”。我自己，身为一个对 C++ 和 Java 都颇有经验的程序员，很希望看到诸如《C# for C++ Programmers》或《C# for Java Programmers》这样的书籍，可使我这般背景的人，藉由不同语言的特性之间的专注比较，快速而印象深刻地掌握新技术。当然，我的这种背景涵盖不了所有 C# 学习者，所以《C# Primer》自有其价值。

侯捷 2003.04.17 于台湾新竹

jjhou@jjhou.com

<http://www.jjhou.com> (繁体) <http://jjhou.csdn.net> (简体)

译序 by 陈硕

正如您所知道的，.NET 是 Microsoft 的战略核心技术，而 C# 又是 .NET 的核心语言。如何快速稳健地步入 .NET 开发领域？我想学习 C# 语言将是首选途径，而您手上这本《C# Primer 中文版》正是学习 C# 的优质教材。

本书作者 Stanley B. Lippman 拥有浓厚的 C++ 历史背景¹，既有深厚的技术实力，又有非凡的写作能力，善于运用平实的文字、精致的例子，将概念讲解透彻，让读者迅速领悟。这本新作也有 Lippman 的一贯特点，可视为《C++ Primer》的姊妹篇——当然本书的身材娇小得多。

本书虽名为“Primer”，却不只是一本初级入门书（姓高的人不一定长得很高 ☺），适合具备一定编程基础（至少理解循环、函数等基本概念）的读者。如果您正好具备这样的基础而又想踏进 C# 这个新领域，我想这本《C# Primer 中文版》是非常好的选择，它能引领您迅速步入实际的 C# 开发之中。当然，学习 C#/.NET 只凭一本书是不够的。

我本人便是凭借本书踏入 C#（或说 .NET）这番新天地。在阅读和学习的过程中，Visual Studio .NET 的随机帮助和本书的配套代码始终是我的最佳伙伴。建议您在阅读时千万别忘了它们，作者在序言中介绍了 C# 的学习方法和本书的内容组织，精彩内容不可错过 ☺。

书籍的翻译是我未曾有过的人生经历。这是一件辛苦的工作，特别是翻译

¹ Stanley B. Lippman 在贝尔实验室工作时，是第一代 C++ 编译器的主创人员之一，亦曾在迪斯尼梦工场从事动画设计，目前领导 Microsoft 新一代 C++ 编译器（VC8）的开发。曾经撰写数十篇 C++ 相关论文，出版数本深具影响的 C++ 专著，包括《C++ Primer》、《Inside the C++ Object Model》和《Essential C++》。

Lippman 的作品，更需打起十二分精神。曾经听朋友开玩笑说，读 Lippman 的原文书，如果看了十来页还没有发现错误，那一定是看得不够仔细 ☺。翻译过程中挑出的原书数十处错误，令我对此戏言体会颇深。不时出些无伤大雅的小错，简直成了 Lippman 书籍的一个特色。中文版读者朋友将比我幸运得多，因为本书将以比原文书更好的面貌呈现在您眼前。

翻译期间向侯捷先生请教讨论，受益匪浅。我是这本译作的第一个受惠者，侯先生是我首先要感谢的人。感谢您给我的机会，我十分荣幸！同时我要感谢华中科技大学出版社的周筠老师，感谢您时时刻刻的教导与关怀。我还要感谢学校实验室的周盛林老师，多谢您容忍我的电脑占用您实验室的空间达半年之久，让我置身清静的学习环境。我要特别感谢好友黄瑾和邹永强为本书的后期制作提供了诸多宝贵意见，多谢二位！最后，谢谢所有关心我的朋友。

愿您有一个良好的阅读体验！

陈硕 2003.01.31 于北京师范大学

chenshuo@chenshuo.com

目 录

译序 by 侯捷	iii
译序 by 陈硕	v
目录	vii
前言	xiii
C# 环境设置	xix
第 1 章 Hello, C#	1
1.1 你的第一个 C# 程序	1
1.2 命名空间 (Namespaces)	6
1.3 Main() 的另一种形式	10
1.4 编写一个语句 (Statement)	11
1.5 开启一个文本文件 (Text File) 以供读写	17
1.6 格式化输出	19
1.7 string 型别	21
1.8 局部对象 (Local Objects)	24
1.9 Value 型别和 Reference 型别	28
1.10 C# array (数组)	29
1.11 new 表达式	30
1.12 垃圾回收 (Garbage Collection)	32
1.13 动态 array: ArrayList collection class	33
1.14 统一型别系统 (The Unified Type System)	35
1.14.1 暗中装箱 (Shadow Boxing)	36
1.14.2 拆箱 (Unboxing) 与向下转型 (Downcast)	37
1.15 缺口型 (Jagged) array	39
1.16 Hashtable 容器	41
1.17 异常处理 (Exception Handling)	44
1.18 C# 语言简要手册	47
1.18.1 关键字 (Keywords)	47

1.18.2 语言内建的数值型别 (Built-in Numeric Types)	49
1.18.3 算术(Arithmetic)、关系(Relational) 和条件(Conditional) 操作符	51
1.18.4 操作符优先级 (Operator Precedence)	54
1.18.5 语句 (Statements)	55
第2章 Class 的设计	59
2.1 我们的第一个独立 Class	59
2.2 开启一个新的 Visual Studio 项目	63
2.3 声明数据成员 (Data Members)	66
2.4 Properties (属性)	67
2.5 Indexers (索引器)	69
2.6 成员初始化 (Member Initialization)	72
2.7 Class 的构造函数 (Constructor)	73
2.8 隐含的 (implicit) this Reference	76
2.9 static (静态) 成员	79
2.10 const 和 readonly 数据成员	81
2.11 enum (枚举) value 型别	83
2.12 delegate 型别	86
2.13 函数参数语义学 (Function Parameter Semantics)	92
2.13.1 传值 (Pass by Value)	94
2.13.2 传址 (Pass by Reference) : ref 参数	96
2.13.3 传址 (Pass by Reference) : out 参数	97
2.14 函数重载 (Function Overloading)	99
2.14.1 重载函数的决议 (Resolving)	100
2.14.2 寻求最佳匹配 (Best Match)	101
2.15 可变长度之参数列	103
2.16 操作符重载 (Operator Overloading)	107
2.17 转换式操作符 (Conversion Operators)	110
2.18 Class 的析构函数 (Destructor)	113
2.19 struct value 型别	113
第3章 面向对象程序设计	117
3.1 面向对象编程概念	117
3.2 实现一个“多态查询语言” (Polymorphic Query Language)	121
3.3 设计一个 Class 继承体系	124
3.4 关于 Object	128

3.5 设计一个抽象基类 (Abstract Base Class)	132
3.6 声明一个抽象基类 (Abstract Base Class)	133
3.7 抽象基类 (Abstract Base Class) 的 static 成员	137
3.8 混合型抽象基类 (Hybrid Abstract Base Class)	138
3.8.1 单一继承下的对象模型 (Object Model)	140
3.8.2 混合型抽象类 (Hybrid Abstract Class) 有何特别?	141
3.9 定义一个派生类 (Derived Class)	143
3.10 覆写继承而来的虚接口 (Virtual Interface)	145
3.11 覆写 Object 的虚函数 (Virtual Methods)	146
3.12 成员访问: new 修饰符和 base 修饰符	147
3.12.1 可达性 (Accessibility) 与可见性 (Visibility)	150
3.12.2 将“对基类 (Base Class) 的访问”封装起来	151
3.13 将 Class 密封起来	153
3.14 Exception 继承体系	154

第 4 章 接口继承 159

4.1 实现 System Interface: IComparable	160
4.2 访问业已存在的 Interface	163
4.3 定义一个 Interface	166
4.3.1 实现我们自己的 Interface: 概念验证	168
4.3.2 将我们的 Interface 整合进入 System Framework	174
4.4 Interface 成员的显式实现 (Explicit Implementation)	178
4.5 继承得来的 Interface 成员	180
4.6 重载? 掩盖? 抑或模棱两可?	183
Overloaded, Hidden, or Ambiguous?	
4.7 掌握 copy (拷贝) 语义: ICloneable	185
4.8 掌握 Finalize (终结) 语义: IDisposable	187
4.9 BitVector: 以组合 (Composition) 进行扩充	190

第 5 章 探访 System 命名空间 199

5.1 支持基本型别 (Fundamental Types)	199
5.2 所有 array 都是 System.Array	200
5.3 查询运行环境	203
5.3.1 Environment Class	204
5.3.2 访问所有环境变量 (Environment Variable)	205
5.3.3 Process Class	207

5.3.4 查找逻辑驱动器	208
5.4 System.IO	209
5.4.1 处理文件扩展名: Path Class	210
5.4.2 操控目录 (Directories)	212
5.4.3 操控文件 (Files)	215
5.4.4 读写文件 (Files)	216
5.5 System 杂项讨论	221
5.5.1 System.Collections.Stack 容器	221
5.5.2 System.Diagnostics.TraceListener Class	223
5.5.3 System.Math	225
5.5.4 DateTime Class	226
5.6 正则表达式 (Regular Expressions)	228
5.7 System.Threading	235
5.8 Web 的请求 / 响应模型 (Request / Response Model)	241
5.9 System.Net.Sockets	245
5.9.1 服务器端 (Server-Side) 的 TcpListener	246
5.9.2 客户端 (Client-Side) 的 TcpClient	248
5.10 System.Data	249
5.10.1 数据库表格 (Database Tables)	250
5.10.2 开启数据库: 选择一个数据供应器 (Data Provider)	252
5.10.3 DataTable 巡礼	254
5.10.4 建立 DataRelation	257
5.10.5 选取动作 (Selection) 与表达式 (Expressions)	258
5.11 System.Xml	259
5.11.1 在程序中使用 XML	260
5.11.2 XmlTextReader	265
5.11.3 Document Object Model (DOM, 文档对象模型)	272
5.11.4 System.Xml.Xsl	277
5.11.5 System.Xml.XPath	279
第 6 章 Windows Forms 设计器	283
6.1 我们的第一个 Windows Forms 程序	283
6.2 建立 GUI	285
6.3 实现“事件回调例程” (Event Callback Routines)	288
6.3.1 实现 TextBox Event	292
6.3.2 实现 Button Events: OK 按钮	293

6.3.3 实现 Button Events: Quit 按钮	294
6.4 检阅并添加 Control Events (控件相关事件)	295
6.4.1 可编程的 (Programmable) Labels	296
6.5 实现 MessageBox (弹出式对话框)	298
6.6 以 List Box 输出无格式数据	299
6.7 探究 File Dialog (文件对话框)	302
6.8 各式各样的 Buttons (按钮)	304
6.9 端上 Menus (菜单)	306
6.10 DataGrid 控件	308
6.11 添加 PictureBox 控件	310
第 7 章 ASP.NET 和 Web Forms 设计器	315
7.1 我们的第一个 Web Forms 程序	316
7.2 开启一个 ASP.NET Web 应用程序项目	316
7.2.1 修改文档的 Properties (属性)	318
7.2.2 在文档中添加控件: Label	319
7.3 在项目中加入页面	320
7.4 HyperLink 控件: 链接 (Linking) 其他页面	321
7.5 DataGrid 控件	321
7.6 理解页面事件 (Page Event) 的生命周期	323
7.7 数据供应器 (Data Provider)	325
7.8 管理 Web 状态	326
7.8.1 添加 TextBox 控件	328
7.8.2 添加 ImageButton 控件	329
7.8.3 添加 ListBox 控件	329
7.9 状态管理: Class Members	331
7.10 状态管理: Session Object	332
7.11 状态管理: Application Object	333
7.12 起验证作用的控件 (Validation Controls)	334
7.13 添加 DropDownList 控件	335
7.14 添加一组 RadioButton 控件	337
7.15 添加 CheckBoxList 控件	338
7.16 为控件添加验证器 (Validators)	340
7.17 添加 Calendar 控件	344
7.18 添加 Image 控件	345
7.19 编写 Web Server 控件	345

第 8 章 通用语言运行层	349
8.1 装配件 (Assemblies)	349
8.2 Reflection (运行期型别反射)	353
8.3 通过 BindingFlags 修改拣取策略 (Retrieval)	358
8.4 在运行期 (runtime) 调用某个成员函数	362
8.5 将测试委托 (Delegating) 给 Reflection	364
8.6 Attributes (特征属性)	367
8.6.1 固有型 Attribute: Conditional	367
8.6.2 固有型 Attribute: Serializable	369
8.6.3 固有型 Attribute: DllImport	370
8.7 实现我们自己的 Attribute class	372
8.7.1 位置 (Positional) 参数与具名 (Named) 参数	375
8.7.2 AttributeUsage	376
8.8 利用 Reflection 在运行期获取 Attributes	376
8.9 中间语言 (Intermediate Language)	378
8.9.1 检视中间语言	379
8.9.2 ildasm (IL 反汇编) 工具	381
索引	385

前 言

C# 是一门崭新的编程语言，它由 Microsoft 发明，并伴随着 Visual Studio.NET 开发工具而被引入人们视野。如今已有一百多万行 C# 代码被用于实现 .NET class framework。本书涵盖 C# 语言本身以及它在 .NET class framework 编程领域中的应用，并阐述其应用领域，如 ASP.NET 和 XML 等等。

书中素材通常以如下方式呈现：给定一个任务，然后以一二种方法实现出来，同时并介绍语言特性或 class framework 相关种种。这么做的目的在于示范如何利用 C# 语言和 class framework 来解决问题，而不单单只是条列语言特性和 class framework API。

C# 的学习可分为两个步骤：(1) 学习 C# 语言细节；(2) 熟悉 .NET class framework。这两个步骤体现于本书的组织结构上。

第一步，我们先学习语言，包括 class（类）、interface（接口）继承、delegates（委托）等语言机制，以及像“统一型别系统”（unified type system）、value 型别与 reference 型别、装箱（boxing）等底层概念。前四章涵盖这些内容。

第二步，熟悉 .NET class framework，特别是 Windows/Web 程序设计，以及对 XML 的支持。这是本书后四章的焦点所在。

这本书读下来，你的 C# 编程技能应该会有一个飞跃的进步。此外，你还会熟悉 .NET class framework 的部分精彩内容。书中所有程序代码可自本人公司的主页下载（www.objectwrite.com）。

您也可以直接给我电子邮件：slippman@objectwrite.com。

本书组织

本书由八个互有关联的长章节组成。前四章关注 C# 语言，着眼于语言内建特性、class 机制、class 继承、interface 继承等等。余下四章带你探究 .NET class framework 所支持的诸多应用领域。

第 1 章 涵盖语言基础及 .NET class framework 提供的一些基础 classes。本章的讨论以一个小程序的设计为主轴，介绍诸如命名空间 (namespaces)、异常处理 (exception handling)、统一型别系统 (unified type system) 等概念。

第 2 章 涵盖构建 class 所需的一些基本元素，包括访问许可 (access permission)、常量 (const) 成员和只读 (readonly) 成员之间的区别、特殊函数如索引器 (indexers) 和属性 (properties) 等等。我们还要学习“成员初始化”的不同策略、操作符重载 (operator overloading) 规则及转换操作符 (conversion operators)。最后还要看看 delegate (委托) 型别，这种型别用起来像是“用以指向函数”之万用指针。

第 3、4 章 涵盖 class 的继承和 interface 的继承。前者使我们得以定义一整族“覆写 (override) 某一公共接口”的特化型别，例如抽象的 `WebRequest` 基类和“与特定协议相关的”`HttpWebRequest` 子类。后者可以为彼此不相关的 classes 提供公共服务或共享的特征属性 (attribute)，例如 `IDisposable` 接口用于释放资源。持有数据库连接 (database connections) 的 class 和持有 window handles 的 class 往往都会实现 `IDisposable`，尽管它们在其他方面并无关联。

第 5 章 带你 .NET class library 做一次大范围巡礼，看看 I/O (包括文件和目录的操作)、正则表达式 (regular expressions)、sockets (网络套接口)、thread (线程)、`WebRequest` 和 `WebResponse` 等 classes 阶层体系，以及对 ADO.NET 和“建立数据库连接”的简短介绍、XML 的使用等等。

第 6、7 章 介绍 Windows Forms 和 Web Forms 的拖放式 (drag-and-drop) 开发。第 7 章聚焦于 ASP.NET 和 Web 页面的生命周期 (life cycle)。这两章有大量例子谈及如何使用预制控件 (prebuilt controls)，以及如何以事件处理器 (event handlers) 和用户互动 (互操作)。

第 8 章（最后一章）提供了一份“.NET 共通语言运行层（Common Language Runtime, CLR）开发人员指南”。本章主要着眼于装配件（Assemblies）、型别反射（type reflection）和 Attributes（特征属性）。本章末尾还摘要介绍了所有 .NET 语言的最终编译结果——中间语言（intermediate language）。

为程序员而写

本书并不假设你已经了解 C++、Visual Basic 或 Java，但假设你曾经以某种语言写过程序。也就是说，我假设你不知道 C# 的循环语句 `foreach` 的确切语法，但我认为你知道什么是“循环”。尽管我会阐述如何在 C# 中调用一个函数，我还是假设你听得懂“调用一个函数”是什么意思。本书不要求你有面向对象编程知识，你也不需要了解早期版本的 ASP 和 ADO。

有些人（特别是一些很聪明的人）认为，在 .NET 中，程序语言相对于（语言所附着的）底层 CLR（Common Language Runtime，共通语言运行层）来说，处于次要位置，就像陆地漂浮于地质构造板块之上一样。我不同意这种说法。我们借助语言来表达自己的思想，对语言的选择直接影响到我们的程序设计。本书实际上把 C# 设想为最佳的 .NET 编程语言。

本书由八个互有关联的长章节组成，前四章重点在于 C# 语言本身，着眼于内建的语言特性、class 机制、class 继承、interface 继承。后四章带你探究 .NET class framework 支援的各个应用领域，如正则表达式（regular expressions）、threading（线程）、sockets、Windows Forms、ASP.NET、Common Language Runtime（CLR，共通语言运行层）等等。

中文版字面规范

type（型别）名称、object（对象）名称、关键字等等，均以 Courier New 字体表示，例如 `int`（语言内建型别）、`Console`（Framework 中定义的一个 class）、`maxCount`（一笔数据成员或函数内的一个局部对象）、`foreach`（内建的循环语句之一）、`WriteLine()`（函数名称，后紧跟一对圆括弧）。某个概念第一次被引入时，以斜体表示，例如 *garbage collection*（垃圾回收）或 *data encapsulation*（数据封装）。这些安排都是为了让正文更加容易阅读。

致谢

本书在许多无形的帮助下得以完成。我首先要向我的妻子 Beth 和两个孩子 Daniel、Anna 表示最衷心的感谢。为了让本书顺利完成，我手上已经攒了一大把因推迟这次或那次家庭旅行而签下的欠条。感谢你们的忍耐和谅解，感谢你们不曾频繁地问我是否已经完成了这本书。

我也要感谢 you-niversity.com 的 Cato Segal 和 Shimon Cohen，他们赠给我一件大方的礼物——时间和鼓励。愿力量与你们同在。我还要向 Eric Gunnerson、Peter Drayton 和 Don Box 表示感谢，他们都曾扮演鞭策者的角色。

我愿向 Elena Driskill 深深地致谢两次。第一次感谢她的礼物——第 6 章出现的那几幅可爱素描；第二次感谢她慷慨允许我复制这些画。

自 1986 年《C++ Primer》第一版起，Deborah Lafferty 就一直担任我的编辑。她一直是判断力和洞察力的永恒来源，我要深深感谢她对本书写作过程中从头至尾的鼓励和督促。

我要特别感谢 Stephanie Hiebert 和 Steve Hall。近二十年来我的所有出版品的最后负责人都是 Stephanie，她使本书更出色。而当我被难以控制的版面弄得灰心丧气时，Steve 助我重新振作起来。在此谨向两位表示敬意。

以下这些同行在审阅本书原稿时提出了无数极具见地的意见和建议：Indira Dhingra（特别感谢他审阅了最终草稿）、Cay Horstmann、Eugene Kain、Jeff Kwak、Michael Lierheimer、Drew Nathanson、Clovis Tondo、Damien Watkins。

本书部分原稿曾在世界各地举办的课程和演讲中试用过，这些地点包括 Sydney、Amsterdam、Munich、Tel Aviv、Orlando、San Francisco、San Jose。感谢每一位反馈信息的人。

资源

Visual Studio.NET 的说明文档是供你时时查阅的最丰富资源。无论进行何种 C#/ .NET 程序开发, .NET Framework reference 都是必不可缺的资源。

《MSDN Magazine》的相关文章和专栏也是丰富的信息来源, 每期杂志都带给我深刻的印象。这本杂志可于 <http://msdn.microsoft.com/msdnmag> 在线阅读。

另一个丰富的信息来源是 DevelopMentor 主办的 DOTNET 邮件组, 可自 <http://discuss.develop.com> 订阅。

Jeffrey Richter、Don Box、Aaron Skonnard 和 Jeff Prosise 所写的关于 .NET (或 Aaron 写的关于 XML) 的任何作品, 都应该被视为“必要的基础读物”。截至目前, 他们的多数作品都还仅仅出现于《MSDN Magazine》上。

以下是我曾经参考或觉得有帮助的书:

- *Active Server Pages+*, by Richard Anderson, Alex Homer, Rob Howard, and Dave Sussman, Wrox Press, Birmingham, England, 2000.
- *C# Essentials*, by Ben Albahari, Peter Drayton, and Brad Merrill, O'Reilly, Cambridge, MA, 2001.
- *C# Programming*, by Burton Harvey, Simon Robinson, Julian Templeman, and Karli Watson, Wrox Press, Birmingham, England, 2000.
- *Essential XML: Beyond Markup*, by Don Box, Aaron Skonnard, and John Lam, Addison-Wesley, Boston, 2000.
- *Microsoft C# Language Specifications*, Microsoft Press, Redmond, WA, 2001.
- *A Programmer's Introduction to C#*, 2nd Edition, by Eric Gunnerson, Apress, Berkeley, CA, 2001.

Stanley Lippman

Los Angeles

November 18, 2001

www.objectwrite.com

C# 环境设置

文 / 图：陈硕

这篇短文说明 C# 开发工具 Visual C# .NET 的安装与环境设置，协助读者搭建一个学习的环境。少少的页数或许便可节省部分读者的摸索时间。

为完整使用本书内容，您的操作系统应为 Windows 2000 Professional、Windows 2000 Server 或 Windows XP Professional 其中之一。Windows XP Home 没有 IIS，无法执行 ASP.NET，故不在考虑之列。Windows 95/98/ME 和 Windows NT 都无法安装 Visual C# .NET。

您最好拥有一套 Visual C# .NET，任何版本（Standard、Professional、Enterprise）均可。因为本书主要以 Visual C# .NET 为开发环境。但如果您手头暂时没有 Visual C# .NET，也不会对学习 C# 造成太大妨碍，稍后详述。

请注意，以下叙述安装步骤时，为了与稍后的“具体安装方法”相对应，步骤编号有所跳跃。请首先确定采用何种安装方式（完整安装或简化安装），再依据安装顺序中的步骤号，至“具体安装方法”一节查询具体操作。如果您拥有 Visual C# .NET（或 Visual Studio .NET），建议采用“完整安装”。

完整安装

- 第 0 步：针对 Windows 2000 操作系统，请安装 Service Pack 2（含）以上版本。Windows XP 用户可省略本步骤。撰写本文时已可下载 Windows 2000 的 Service Pack 3，下载地址为（请注意选择语种）：

<http://www.microsoft.com/Windows2000/downloads/servicepacks/sp3/download.asp>

- 第 1 步：安装 IIS (Internet Information Services)。IIS 是执行 ASP.NET 的必备条件。IIS 位于 Windows 安装光盘上。
- 第 2 步：安装 Visual C# .NET。
- 第 5 步：安装 MSDE (Microsoft SQL Server Desktop Engine)。本书第 7 章谈到访问 SQL 数据库，如果您未曾安装 SQL Server 2000，可使用 MSDE。
- 第 6 步：安装 QuickStart 示例网页。本书第 5 章用到的 Northwind 数据库正是来自这个样例。

如果您手上没有 Visual C# .NET，可采取以下简化安装方式，这样也能学习到书中 IDE (集成开发环境) 之外的全部内容。

简化安装

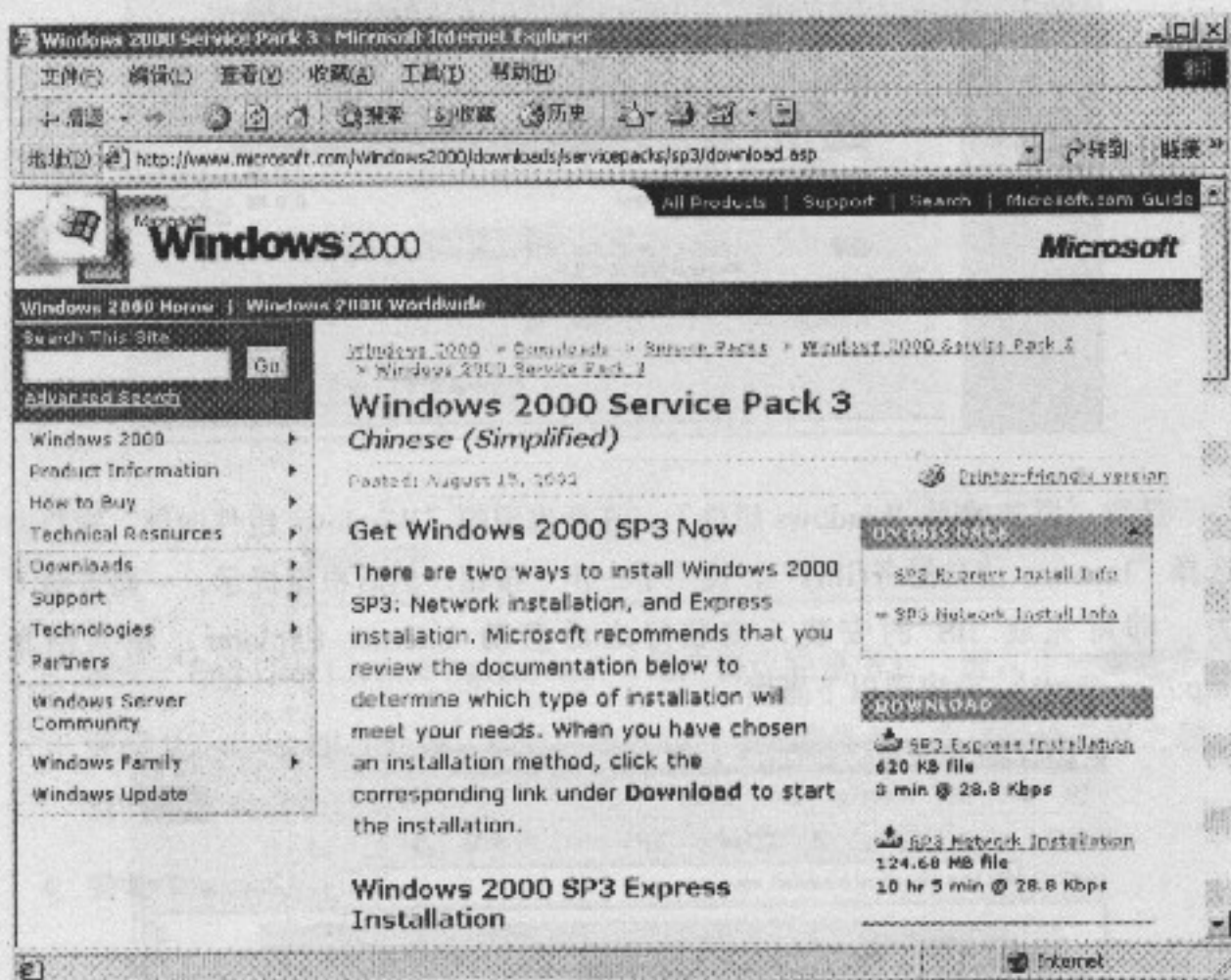
- 第 0 步：针对 Windows 2000 系统，安装 Service Pack 2 以上版本和 Internet Explorer 6 以上版本。Windows XP 用户可省略这一步骤，Internet Explorer 6 sp1 可从此处下载：
<http://www.microsoft.com/windows/ie/downloads/critical/ic6sp1/download.asp>
- 第 1 步：安装 IIS (Internet Information Services)。
- 第 3 步：安装 .NET Framework SDK。其简体中文版 (涵括简体中文之 help 文件) 可从这里下载：
<http://www.microsoft.com/china/msdn/downloads/msdncompositedoc.asp>
- 第 4 步：安装 MDAC (Microsoft Data Access Components) 2.6 以上版本。下载地址是：<http://www.microsoft.com/data/download.htm>。撰写本文时，MDAC 2.7 简体中文版已可下载。
- 第 5 步：安装 MSDE。MSDE 附属于 .NET Framework SDK，无须另外下载。
- 第 6 步：安装 QuickStart 示例网页。

具体安装方法

下面以 Windows 2000 为平台，以 Visual C# .NET Standard 为对象，讲解具体安装步骤，建议您泡上一杯好茶，安装过程可能要耗去一个多小时。

- 第 0 步：安装 Windows 2000 Service Pack 3，如果已经安装有 Service Pack 2，或如果您使用 Windows XP，可跳过这一步。

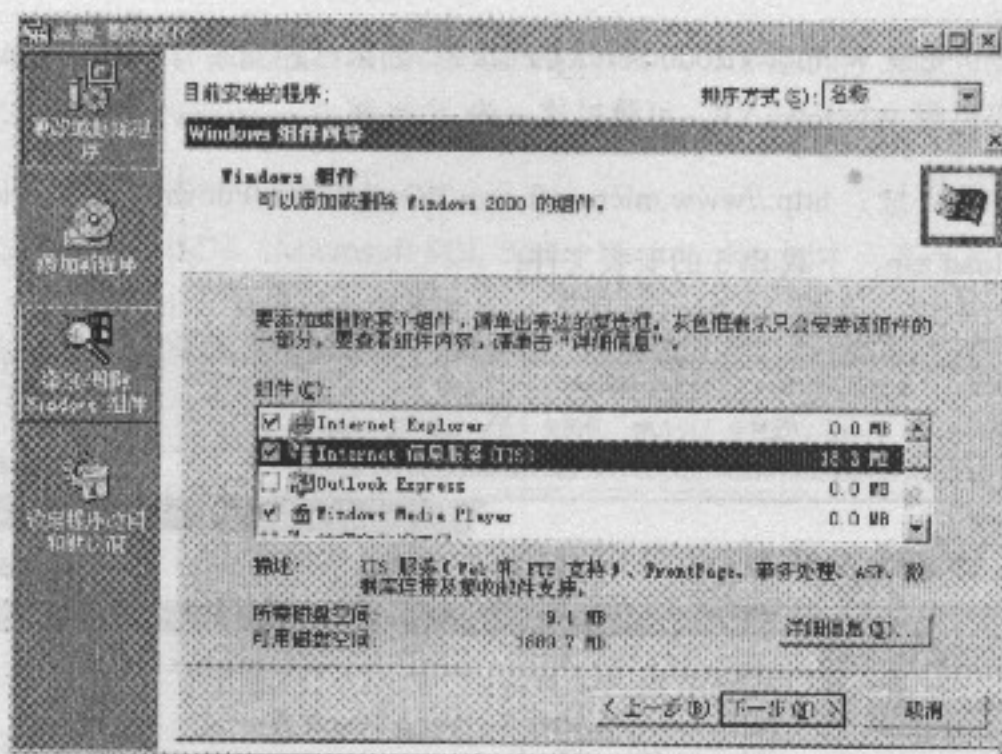
请访问网址：<http://www.microsoft.com/Windows2000/downloads/servicepacks/sp3/download.asp>，下载 SP3 的安装文件：



如果您用的是 Windows 2000，而且没有 Visual C# .NET，那么还需要安装 Internet Explorer 6.0。请访问网址：<http://www.microsoft.com/windows/ie/downloads/critical/ie6sp1/download.asp>，下载 IE6 的安装文件。

- 第 1 步：安装 IIS (Internet Information Services)

启动“开始”菜单\设置\控制面板\添加或删除程序，会出现如下窗口：

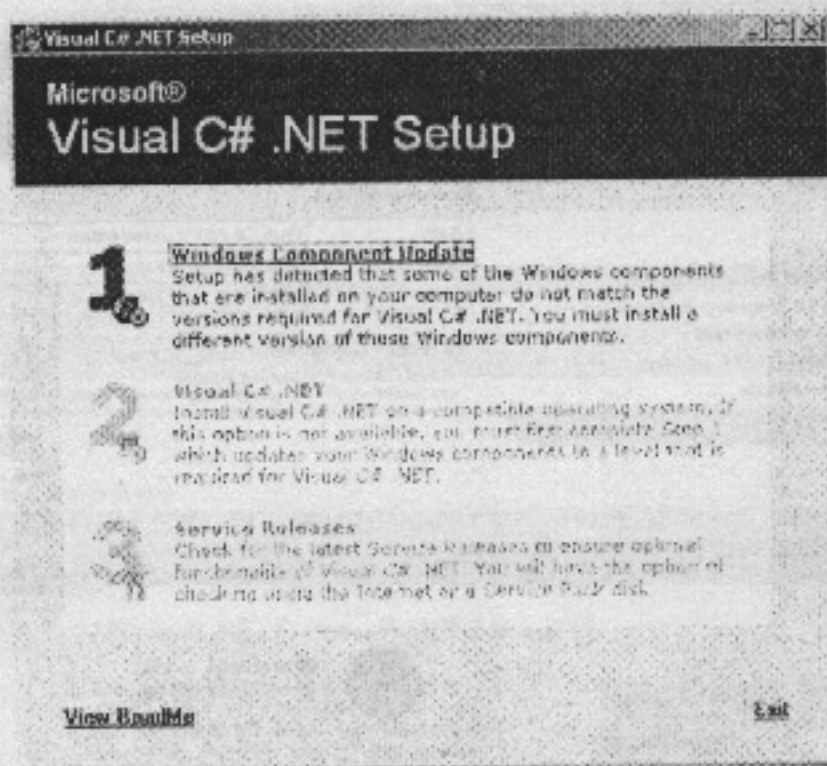


选取“添加/删除 Windows 组件”，在新出现的“Windows 组件向导”窗口中选择“Internet 信息服务(IIS)”，按“下一步”按钮。然后根据提示，一路安装下去，即可完成 IIS 的安装。安装好之后启动 Internet Explorer，输入网址 <http://localhost/>，会出现以下画面：



● 第2步：安装 Visual C# .NET

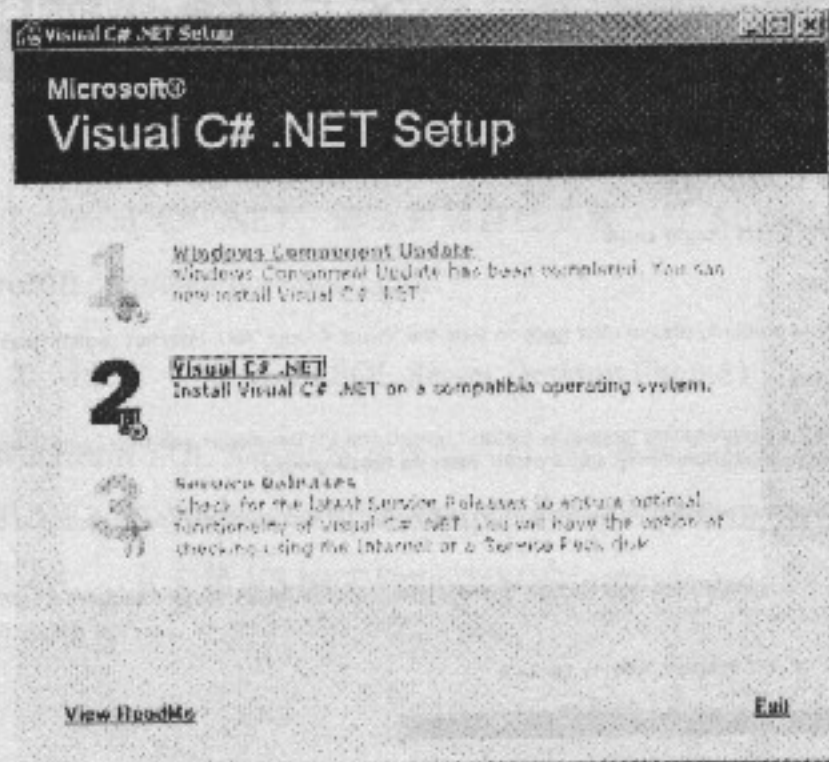
1. 启动 Visual C# .NET 安装程序，先进行 Windows Component Update (组件升级)：



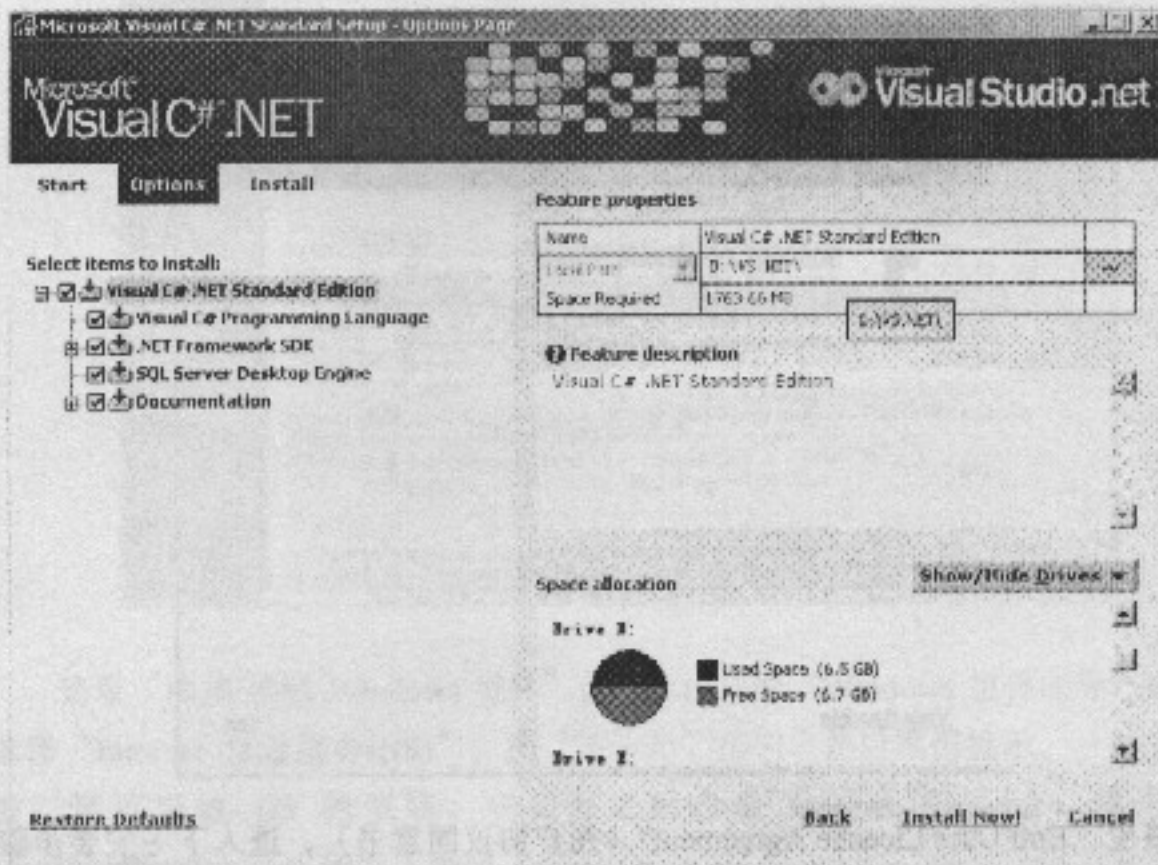
2. 接受 “End User License Agreement” (用户协议同意书)，进入下一安装步骤。

3. 进行 Windows 组件升级，这时可能要求更换安装光盘。完成组件升级之后，可能需要重新启动计算机。

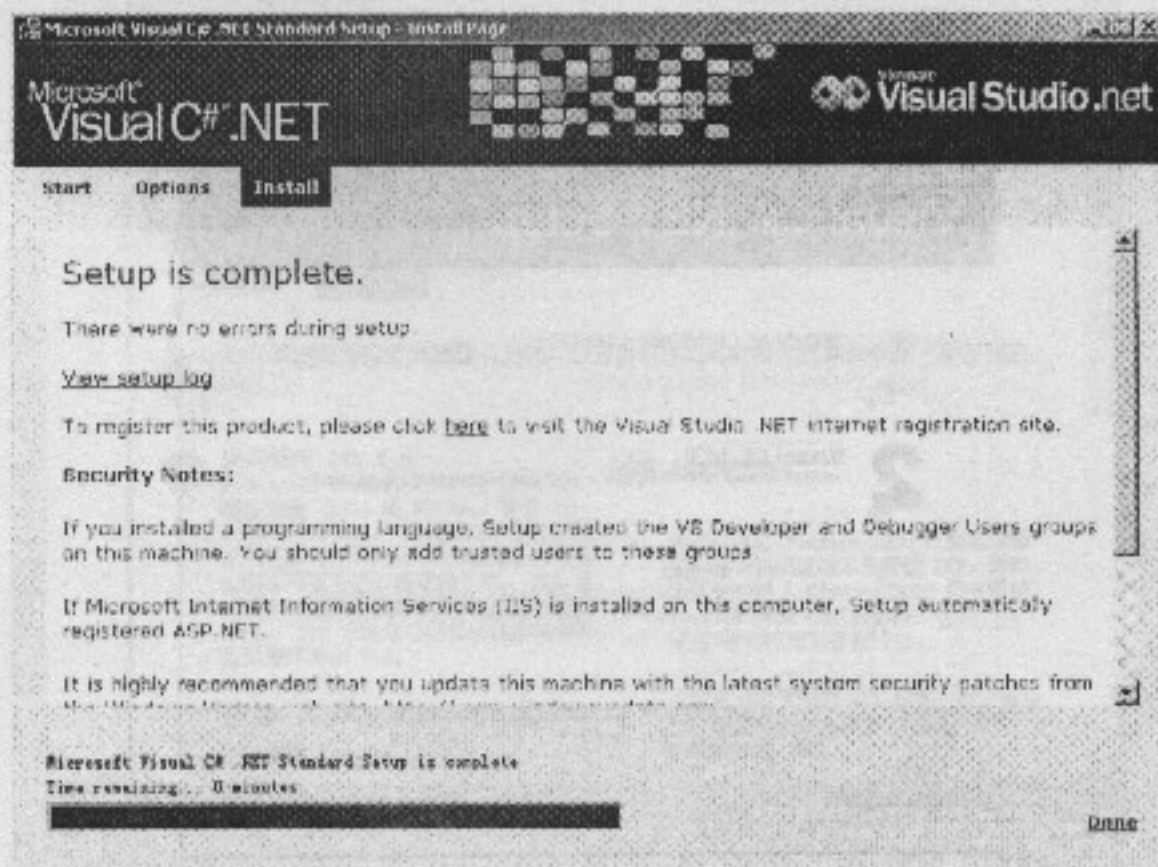
4. 安装 Visual C# .NET：

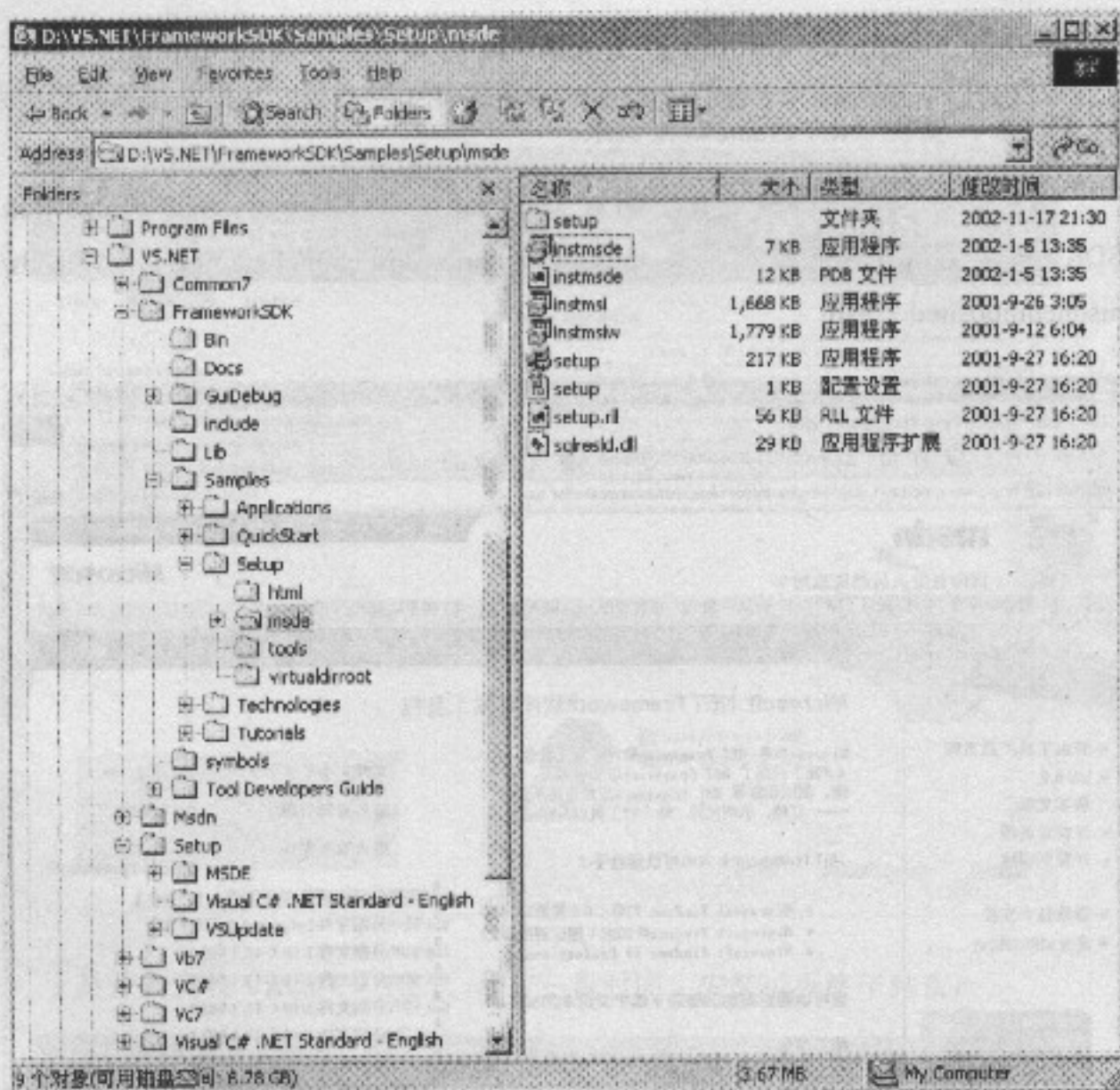


请输入您的 Product Key 并同意用户协议。然后选择要安装的部件（建议安装所有部件），并确定安装路径。我习惯将 Visual C# .NET 安装在 D:\VS.NET\

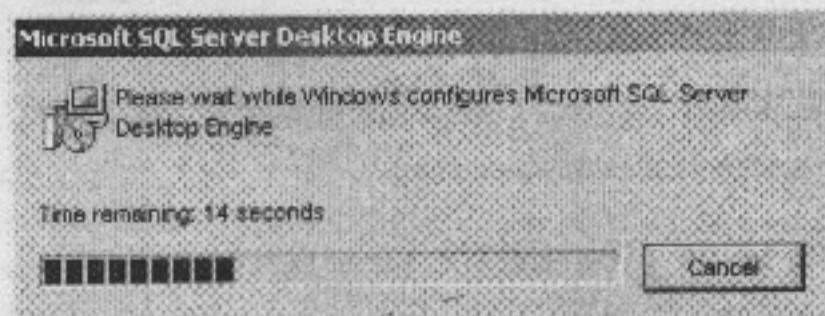


于是开始安装 Visual C# .NET。经过一段时间，安装过程终于结束：





安装进行中:

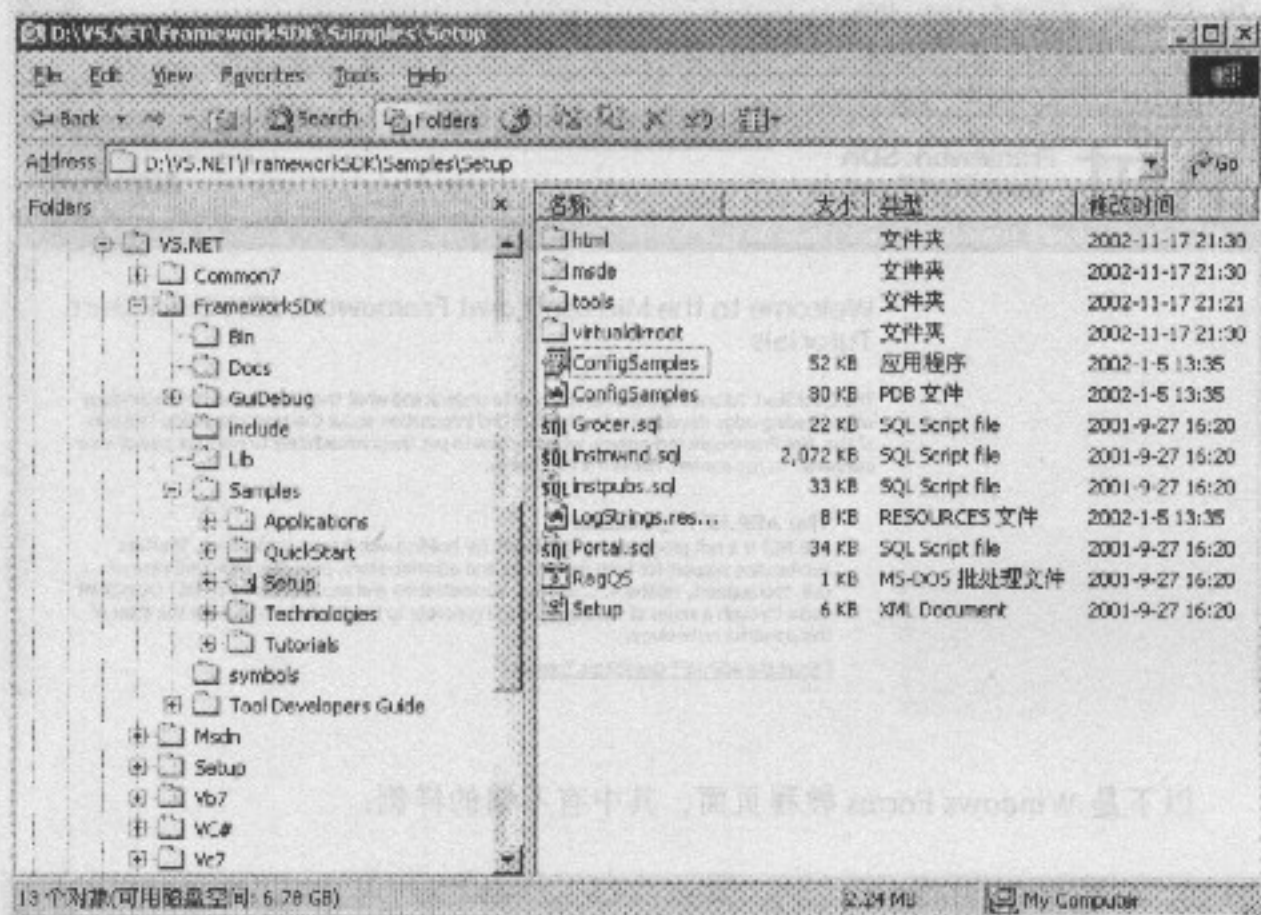


安装完成后, 重新启动计算机, 这时桌面右下角的 Tray 会出现 MSDE 图标:

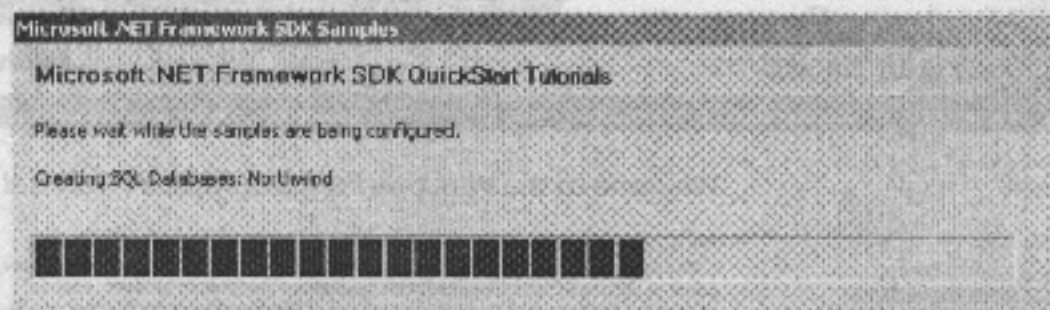


● 第 6 步：安装 QuickStart 示例网页

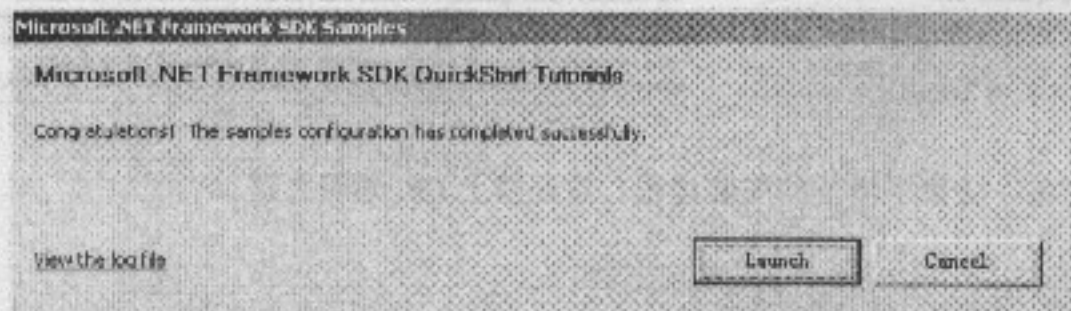
执行 VS.NET\FrameworkSDK\Samples\Setup\ConfigSamples.exe, 安装 QuickStart:



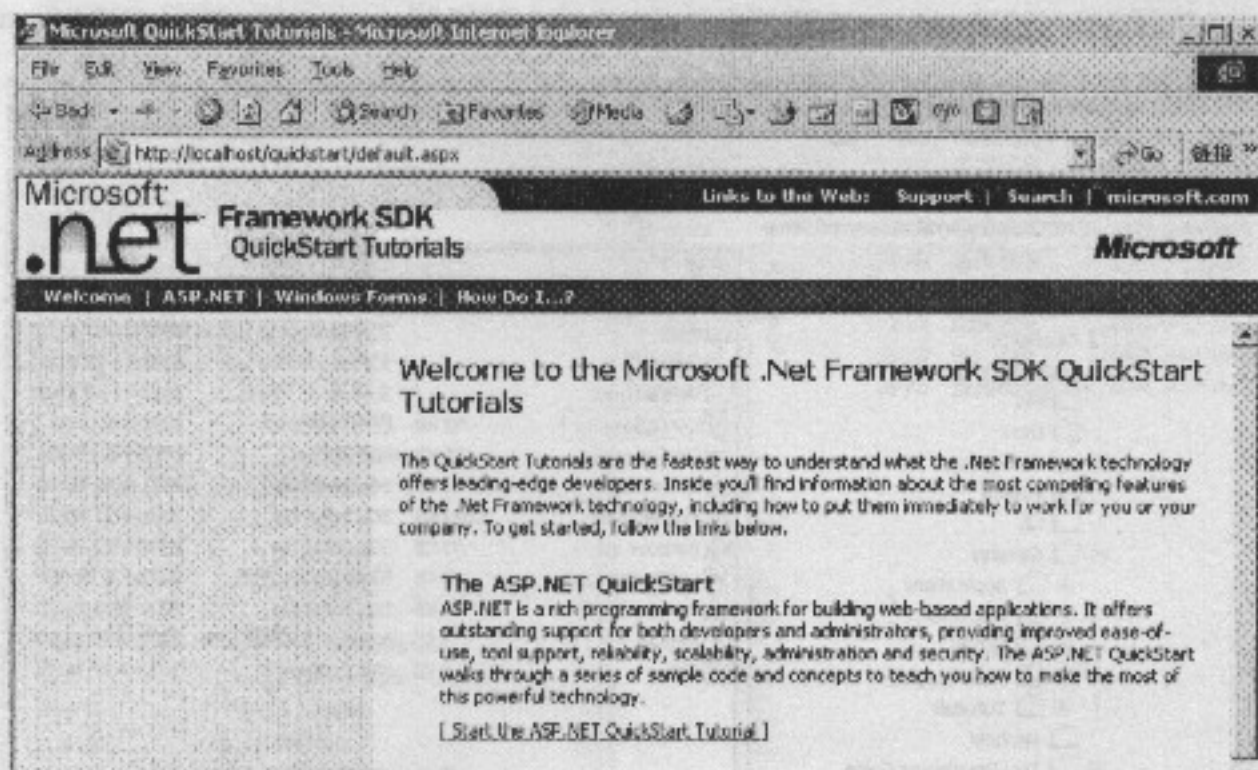
安装进行中:



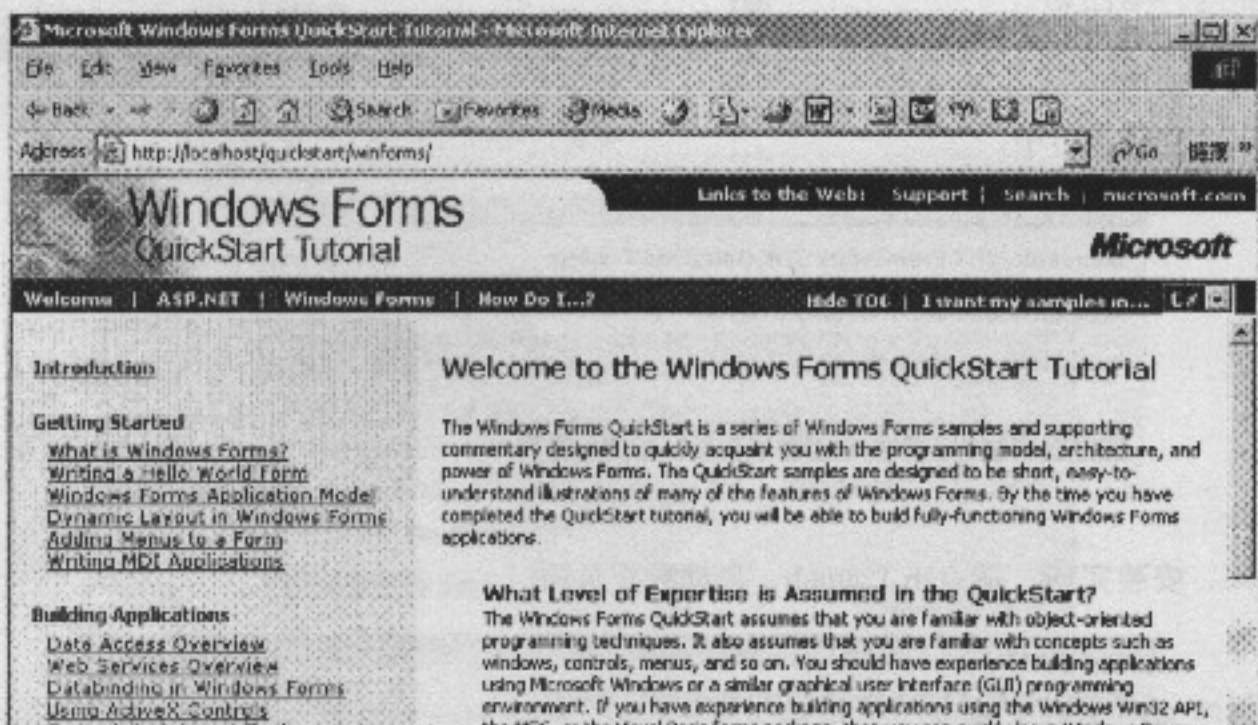
安装完毕, 请点击 Launch, 启动教程页面:



以下是 QuickStart 教程页面:



以下是 Windows Forms 教程页面, 其中有不错的样例:



现在, 结束了漫长的安装过程, 欢迎踏上 C# 学习之路。

1

Hello, C#

我的女儿来回学过好几种乐器，每拿到一样乐器，她总是渴望马上开始演奏一些世界名曲。噢不，不是古典的 Schubert（舒伯特，奥地利作曲家，1797-1828）和 Schoenberg（勋伯格，奥地利作曲家，1874-1951），而是 Backstreet Boys（后街男孩）和 Britney Spears（布兰妮）。她的多位老师为了让她在接受枯燥的基础训练时保持学习兴趣，往往迁就她演奏那些“名曲”。在某种意义上，本章展现 C# 时，试图把握相同的平衡。Web Forms（页面窗体）和 Type Inheritance（型别继承）就是我认为的名曲，基本音阶训练则是那些看起来很枯燥的“语言内建元素和机制”如生存空间（scope）规则、算术型别（arithmetic types）、命名空间（namespaces）等等。为了保持你的兴趣，我的做法是，只在程序实现过程中有必要时，才引入相应的语言元素。同时，遵循传统，本章末尾列出了 C# 语言内建元素的总整理。

C# 支持泛整数（integral）和浮点（floating-point）数值型别，以及布尔（Boolean）型别、Unicode 字符型别、高精度小数型别。以上被称为“简单型别”，与之相关的是一套操作符（operators），包括加法（+）、减法（-）、相等测试（==）、不等测试（!=）。C# 也预定义了一套语句（statements），如用以表示条件的 if 和 switch 语句，用以表示循环的 for、while、foreach 语句。所有这些，包括命名空间（namespace）和异常处理（exception handling）机制，本章都会提到。

1.1 你的第一个 C# 程序

遵循传统，学习一门新的程序语言，你所写的第一个程序通常是在控制台（console）上打印出“Hello, World!”。这样一个 C# 程序如下所示：

```
// our first C# program
using System;
class Hello
{
    public static void Main()
    {
        Console.WriteLine("Hello, World!");
    }
}
```

编译这个程序后加以执行，输出如下：

Hello, World!

这个程序由四个要素组成：(1) 由双斜线 `“//”` 引导的一行注释；(2) 一行所谓的 `using directive`（指令）；(3) 一份 `class` 定义式，(4) 一个名为 `Main()` 的 `class` 成员函数（member function，也称为 `class method`）。

C# 程序从 `class` 成员函数 `Main()` 开始执行，此函数称为程序入口（entry point）。`Main()` 必须被定义为 `static`，本例之中我将它声明为 `public static`。

关键字 `public` 标识出 `Main()` 被赋予的访问级别（access level）。凡被声明为 `public` 的 `class` 成员，皆可在程序的任何地点被访问。一般说来，`class` 的成员若不是函数（member function）就是数据（data member），前者执行其所属 `class` 的某个特定动作，常被声明为 `public`；后者内含一个与其所属 `class` 之状态（state）有关的某值，常被声明为 `private`。我将在开始设计 `class` 时再来谈成员的访问级别。

一般说来，`class` 的成员函数支持该 `class` 的相应行为。例如 `WriteLine()` 是 `Console` `class` 的 `public` 成员函数，它将字符串（string）输出于控制台（console）并在末尾加上一个换行符。`Console` `class` 还提供了功能相似的 `Write()` 函数，但它并不在输出末尾添加换行符。通常如果我们希望用户对出现于控制台上的问讯做出回应，就使用 `Write()`；如果仅仅只为了显示消息，就使用 `WriteLine()`。稍后很快会有相关例子。

作为一个 C# 程序员，我们的主要行为是设计并实现出 `classes`（类）。到底什么是 `classes` 呢？它通常代表应用域（application domain）中的某些物体（entities）。举个例子，如果要开发一个图书馆借阅系统，我们很可能需要诸如 `Book`（书籍）、`Borrower`（借阅人）、`DueDate`（最后归还日期）等等 `classes`。

Classes 从何而来？当然绝大多数来自像我们这样的程序员。有时候我们需要亲自动手实现 classes，本书主要就是希望让你成为这方面的专家。有些现成的 classes 可直接被我们拿来运用，例如 .NET System framework 提供的 `DateTime` class 就适合用来表现上述的 `DueDate`。要成为 C# 程序专家（而不是芸芸众生中平淡无奇的一员），你将遇到的挑战之一就是如何熟悉 .NET Framework 之中定义的上千个 classes。我不可能在书中提到所有这些 classes，但我会列出相当数量。这些 classes 分别支持正则表达式（regular expressions）、线程（threads）、sockets、XML、Web 程序设计、数据库以及用以建立 Windows 应用程序的一些新方法。

如何逻辑性地组织上千个 classes，让用户可以找到并弄清楚它们，而且还要防止名称冲突，是一个甚具挑战的问题。当然我们可以运用文件目录来组织这些 classes。例如将所有支持 ASP (Active Server Pages) 的 classes 都存放在 `System.NET` 根目录下的 `ASP.NET` 子目录里。这么做使 classes 的组织相当清晰，给试图“翻弄”这些文件目录结构的人带来相当的方便。

但是在程序之中我们该怎么办呢？我们不会当真使用文件目录来存储 classes。C# 有一个类似文件目录的组织机制，即所谓 *namespace*（命名空间）。.NET Framework 有一个几乎包罗万象的命名空间，名为 `System`。先前出现的 `Console` class 就是定义在 `System` 命名空间内。

用以表现共同抽象概念的各组 classes，被赋予各自独立的命名空间，这些命名空间（几乎）都被定义于 `System` 命名空间内（译注：.NET 之中还有类似 `Microsoft.CSharp` 这样并非隶属 `System` 的命名空间，故曰“几乎”），例如 `Xml` 命名空间被定义于 `System` 命名空间内。此时我们称 `Xml` 命名空间和 `System` 命名空间呈嵌套式（nested）分布。`Xml` 命名空间内又嵌套定义了数个子命名空间，除了 `Serialization` 之外，还有 `XPath`、`Xsl` 和 `Schema` 等等。这些“被嵌套定义

于外围 Xml 命名空间内”的数个独立命名空间，把整个 XML 领域的共通功能分为数块，分别封装起来。这样的安排让我们得以轻松找到我们需要的功能，例如 World Wide Web Consortium (W3C) 建议的 XPath (XML Path Language) 就可以在 .NET 的 XML 命名空间内找到。其他嵌套定义于 System 命名空间内的还包括 IO (含文件和目录相关 classes)、Collections、Threading、Web 等等。

在目录结构中，我们以反斜线 (\) 标示目录之间的包含与被包含关系（至少在 Windows 下是如此），例如：

```
System\Xml\XPath
```

在命名空间内，类似的包含与被包含关系以 scope (生存空间) 操作符 (.) 标示之，例如：

```
System.Xml.XPath
```

以上两种写法都表示 XPath 包含于 Xml 内，而 Xml 包含于 System 内。

C# 程序中凡提及一个“名称”，编译器都必须把这个名称决议 (resolve) 为我们程序某处的一个实物声明，例如当我们这么写：

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

编译器必须以某种方式了解 Console 是个 class 名称，并设法了解 WriteLine() 是 Console class 提供的一个成员函数。所谓成员函数是指：WriteLine() 函数在 Console class 的定义范围 (scope) 内被定义。由于此程序中我们只定义了一个 Hello class，所以编译器暂时无法对 Console 这一名称进行决议。一旦编译器无法决议某个名称代表何物，就会产生编译期错误，并中止编译。

```
C:\C#Programs\hello\hello.cs(7):
```

```
The type or namespace name 'Console' does  
not exist in the class or namespace
```

而我们的程序中的 using 指令：

```
using System;
```


便是告诉编译器到 `System` 命名空间中查找目前正处理的文件中尚无法决议的名称。本例中目前正被编译器处理的文件，就是含有 `Hello class` 定义式及其成员函数 `Main()` 的文件。

另一种做法是，明确告诉编译器到何处寻找某个名称：

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

某些人（更精确地说是某些很机灵或是很中规中矩的人）相信，使用上述的全饰名称（*full qualified name*）比使用先前的 `using` 指令更加可取。所谓 `class` 的全饰名称，就是像上面那样明确列出包含该 `class` 的全套命名空间。这些人认为全饰名称可以清楚表明在哪儿找到这个 `class`，并相信这些信息很有用，即便同一个 `class` 的全饰名称在邻近 20 行程序代码中重复出现 14 次也在所不惜。我并不认同这种看法，我也不喜欢打那么多字。在这本书里，除非为了消除型别名称运用上的歧义现象，否则我不会使用 `class` 全饰名称¹。1.2 节会告诉你什么时候需要全饰名称。

稍早我曾说过，绝大多数 `classes` 若非来自其他程序员，就是来自开发系统自身所提供的程序库（`libraries`），还有别的来源吗？是的，答案是 `C#` 语言本身。`C#` 预先定义了数种被大量使用的数据类型，包括泛整数（`integers`）、单精度和双精度浮点数，以及字符串（`strings`）。每种型别都有对应的 `C#` 型别指定词：`int` 代表整数型别；`float` 代表单精度型别；`double` 代表双精度型别；`string` 则是字符串型别。请参考 1.18.2 节的表 1.2 和表 1.3，其中列出了预定义的各种数值型别。

举个例子，先前那个小程序可以采用另一种做法：定义一个 `string object` 并以字符串字面值“`Hello, World!`”作为初值，然后打印这个 `string object`：

```
public static void Main() {  
    string greeting = "Hello, World!";  
    Console.WriteLine( greeting );  
}
```

¹ 虽然 Visual Studio wizards（例如 Windows Forms wizard 和 Web Forms wizard）生成的程序代码使用全饰名称，但因为那些名称是由机器自动产生的，所以并不能以此作为反例。

`string` 是个 C# 关键字 (译注: 其实是个 `class` 名称), 所谓关键字 (keyword) 是指 C# 保留并赋予特殊含义的单词。 `public`、`static` 和 `void` 都是 C# 关键字。上述的 `greeting` 我们称为标识符 (*identifier*) , 它是“隶属于 `string` 型别”的某个 `object` 的名称。C# 标识符必须以文字、数字或下划线 (`_`) 起头, 区分大小写。换句话说, `greeting`、`Greeting` 和 `Greeting1` 是不同的标识符。

标识符的组合名称究竟该以下划线分开 (例如 `xml_text_reader`) 还是应该令每个内部单词的首字母大写 (例如 `xmlTextReader`) , 这一直是程序员生活圈中的热门争论议题。至于代表 `class` 名称的标识符, 习惯上常常以大写字母起头 (例如 `XmlTextReader`) 。

在程序生存空间 (scope) 内, 标识符必须唯一。局部生存空间 (local scope) 内标识符的唯一性不是什么大问题, 因为函数内任何 `object` 的完整定义完全由我们控制。所谓“局部生存空间”是指函数体 (function body) 的左右两个花括弧之间, 例如先前我们所定义的 `Main()` 函数。随着生存空间 (scope) 的扩大, 标识符的唯一性愈来愈难控制, 因此命名空间 (namespace) 就有了用武之地。

1.2 命名空间 (Namespaces)

命名空间是程序之中控制名称可见度 (visibility) 的一种机制。利用命名空间可以使标识符之间的命名冲突降到最低, 这么一来“不同来源的程序组件”的协同工作就变得比较容易。在介绍命名空间机制之前, 我们最好先弄清楚命名空间所要解决的问题。

所有“未被置入某个命名空间”的名称, 都会被自动置入一个独一无二的“无名全局声明空间”内。这些名称在程序内从头到尾都能被看见, 不论它们出现在相同的程序文件或不同的程序文件内。全局声明空间中的每个名称都必须独一无二, 这么一来程序才能被成功地生成 (build) 出来。全局名称会给“将某些独立组件 (components) 合并到我们的程序中”带来相当难度。

举个例子, 想象你开发了一个二维 (2D) 图形组件, 并把一个 `global class` 命名为 `Point`。你使用你的组件, 一切正常。于是你向一些朋友谈起它, 他们自然也想试试看。

在此同时，我开发了一个三维 (3D) 图形组件，这回轮到我把我的一个 global class 命名为 Point。我用我的组件，也一切正常。我向朋友展示它，朋友们表示出极大的兴趣，也想试试这个组件。到目前为止，每个人都很开心——至少对于我们的项目是如此。

现在假设你我有一个共同的朋友，正在编写一个 2D/3D 游戏引擎，他愿意利用你我这两个“受到高度赞扬”的组件。不幸的是当他在程序中加入这两个组件时，两个原本独立的标识符 Point 导致其游戏引擎编译失败。由于这两个组件都不是他写的，修正这两个组件使它们协同工作并不是太轻松的事。

命名空间 (namespace) 给“全局名称冲突问题”提供了一个通解。所谓命名空间，就是取一个名称，并在这个名称内封装我们所定义的 classes 和其他 types²。也就是说，置于某个命名空间内的“名称(s)”，在程序中一般是不可见的。因此我们说，一个命名空间代表一个独立的声明空间 (declaration space) 或生存空间 (scope)。

现在，让我们帮助我们共同的朋友，将各人的 Point class 定义于各自的命名空间中：

```
namespace DisneyAnimation_2DGraphics
{
    public class Point { ... }
    // ...
}

namespace DreamWorksAnimation_3DGraphics
{
    public class Point { ... }
    // ...
}
```

² 只有命名空间和型别 (types) 可以被声明于全局命名空间内，函数仅能被声明为 class 的成员。纯数据对象 (data object) 若非作为 class 的成员，就是作为函数的局部对象 (local object)，就像我们先前声明的 greeting 对象那样。

命名空间 (namespace) 的定义由关键字 `namespace` 引导, 紧跟在关键字后面的是这个命名空间独一无二的标识符。如果我们用业已存在的命名空间标识符来作为新命名空间的名称, 编译器会认为我们想在原先存在的命名空间内加入新声明。两个名称相同的命名空间并不会导致冲突, 因此, 我们可以把命名空间的声明分散到不同的程序文件去。

命名空间的内容被封在一对花括弧 "`{ }`" 之间, 我们的两个 `Point` classes 对整个程序来说不再是可见的, 它们分别嵌套定义于各自的命名空间。我们说每个 `class` 是其命名空间内的一个成员。

`using` 指令在这种情况下就显得有点儿画蛇添足了。一旦我们的那位朋友在他的程序中同时“开启” (曝光) 两个命名空间:

```
using DisneyAnimation_2DGraphics;  
using DreamWorksAnimation_3DGraphics;
```

那么, 使用无修饰标识符 `Point` 仍会导致编译错误。为了明确指出是这个或那个 `Point` class, 我们必须运用全饰名称, 像这样:

```
DreamWorksAnimation_3DGraphics.Point origin;
```

由右向左读这句话, 意思就是: 声明一个名为 `origin` 的 object, 其型别为 `Point`, 定义于命名空间 `DreamWorksAnimation_3DGraphics` 内。

面对“出现于同一个命名空间内的名称冲突”和“出现于不同命名空间之间的名称冲突”, 编译器的处理方式不尽相同, 取决于编译器自身的能力。最简单的情况是, 如果在单一声明空间内发生重复名称 (亦即重复声明), 那么第二次声明会立刻触发编译错误。我们假设, 看到这类错误消息的程序员, 都有能力修改或重新命名“发生命名冲突的那个声明空间内”的标识符。

然而一旦发生跨命名空间的名称冲突, 事情就不再那么简单。举个例子, 我们“开启” (曝光) 两个独立命名空间, 每个命名空间内都用上了同一个标识符, 就像前述的 `Point` 那样。这时如果再显式运用 (explicit use) 被多重定义的标识符 `Point`, 就会发生错误。编译器不会试图区分两个标识符的优先次序, 或试图消除其所代表之 object 的混淆现象。一个解决办法是, 像我们以前所做的那样, 明确指出每个标识符的访问路径。另一个办法是, 为一个 (或所有) 多重定义的实体(s) 取

一个别名 (alias)。也就是说，运用下面这种形式的 using 指令：

```
namespace MyApp
{
    // exposes the two instances of Point
    using DisneyAnimation_2DGraphics;
    using DreamWorksAnimation_3DGraphics;

    // OK: create unique identifiers for each instance
    using Point2D = DisneyAnimation_2DGraphics.Point;
    using Point3D = DreamWorksAnimation_3DGraphics.Point;

    class MyClass
    {
        Point2D thisPoint;
        Point3D thatPoint;
    }
}
```

别名 (alias) 仅在当前的声明空间内有效。也就是说，所谓的“别名”并不会带给这个 class 另一个永久有效的型别名称。如果我们试图跨越命名空间来使用这个别名，例如下面这样：

```
namespace GameEngine
{
    class App
    {
        // error: not recognized
        private MyApp.Point2D origin;
    }
}
```

编译器会不知所措，因而发出如下消息：

```
The type or namespace name 'Point2D' does not exist in the class or
namespace 'GameApp'
```

使用命名空间时，我们并不知道其中已经定义了多少标识符。程序每多开启（曝光）一个命名空间，就多增加发生混乱的可能性，我们得重新编译，看看有没有爆发灾难。因此，C# 语言尽量使“开启命名空间所造成的程序混乱”的可能性降到最低。

如果多个 `using` 指令使得“某个标识符所代表的两个或多个实体”在当前的声明空间中变得可见（曝光了），就像先前两个 `Point` classes 那样，那么，只有“直接使用未经资格修饰（unqualified）的标识符”才会触发错误。如果我们不使用那个标识符，歧义（ambiguity）的状况会潜伏并保持下去，编译器不会给出任何错误或警告。

如果“通过 `using` 指令而曝光的标识符”和“吾人所定义的标识符”重复，那么我们所定义的那个标识符拥有优先权。一个未经资格修饰（unqualified）的标识符总是被编译器决议为我们自己定义的那个实体。可以说，我们定义的那个实体，遮掩（hide）了命名空间内的那个同名标识符。程序将会完全依照“添加命名空间前”的方式运行。

命名空间该有怎样的名称？诸如 `Drawing`、`Data`、`Math` 之类的一般性名称，其保持唯一的可能性很小。一般推荐的策略是在名称中使用“足以代表你的组织或项目”的前缀字。

命名空间（namespaces）是组件开发的必备要素。正如我们所看到的，有了它，其他程序才会较容易地复用（reuse）我们的软件。至于对不那么有“雄心壮志”的程序来说——例如本章一开始的那个 `Hello` 程序，倒是没什么必要为它编写一个命名空间。

1.3 `Main()` 的另一种形式

本章剩余篇幅中，我要探讨 C# 语言的预定义元素（predefined elements），同时完成一个名为 `WordCount` 的小型程序。此程序开启用户指定的文本文件，统计其中每个单词的出现次数，并以字典排序方式写至一个输出文件内。这个程序接受两个命令行选项（command-line options）：

1. `-t` 令程序开启“追踪（trace）”功能：缺省情况下此功能是关闭的。
2. `-s` 令程序统计读取文件、处理单词、输出结果所耗费的时间，并通报给用户知道。缺省情况下不做如此计算。

我们的第一个任务是修改 `Main()`，取得传人的命令行参数（如果有的话）。我采用 `Main()` 的第二种形式（一个单参数的函数标记式）：

```
class EntryPoint
{
    public static void Main( string [] args ) { }
```

其中 `args` 被定义为字符串数组 (string array)，它会被自动填入用户指定的命令行参数。如果用户这样调用我们的程序：

```
WordCount -s mytext.txt
```

那么 `args` 的第一个元素就是 `-s`，第二个元素是 `mytext.txt`。

不论哪一种形式，`Main()` 都可以选择返回一个 `int` 值：

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

返回值用以反映程序的结束状况。习惯上返回 0 表明程序运行完全顺利，返回非零值则表明程序有某种形式的故障。“返回值型别 (return type) 为 `void`”显得有点似是而非，但它实际上表示返回状态为 0，也就是说操作系统总是认为程序顺利运行完毕。下一节我们会考虑如何使用另一种形式的 `Main()`。

1.4 编写一个语句 (statement)

我们需要做的第一件事是检测用户是否指定了任何引数。为此我询问 `args` 所含的元素个数³。一旦用户未提供必要的命令行参数，我就让程序结束。（你可以自己练习，重写这个程序，允许用户交谈式地输入所需选项，这样的程序当然更加亲和一些。）

³ 在 C# 中，我们不能以 `if (!args.Length)` 来测试 array 是否为空，因为 0 并不代表 false（伪值）。

在我的程序里，如果 `args` 为空，程序就打印出一份“WordCount 正确执行办法”的说明，然后以 `return` 语句结束程序。`return` 语句会导致它所在的函数结束生命，返回 (`return`) 当初调用此函数的地点。

```
public static void Main( string [] args )
{
    if ( args.Length == 0 )
    {
        display_usage();
        return;
    }
}
```

上述的 `Length` 是 `array` 的一个属性 (*property*)，代表存储于 `array` 之内的元素个数。这个测试动作置于 C# 条件测试句 `if` 之中。如果测试值评估为真 (`true`)，那么紧接着 `if` 的那些语句就会被执行起来；否则就会跳过那些语句。如果有多个语句需要执行，像上述例子那样，就得将这些语句以花括弧围起来，被花括弧围起来的语句(s) 统称为语句块 (*statement block*)。

关于 `if` 语句，初学者常犯的一个错误是，当他希望执行两条或更多条语句时，忘了使用语句块⁴：

```
// this is an incorrect usage of the if statement
if ( args.Length == 0 )
    display_usage();
    return;
```

上述 `return` 语句的缩排，虽然表现出程序员的意图，却无法反映到程序行为上。如果没有使用语句块，一如上例，那么当测试条件获得满足 (亦即测试值为真) 时，只有 `display_usage()` 函数会被执行，至于 `return` 语句，则不论 `array` 是否为空都会被执行。

`return` 语句也可以返回一个值，该值将成为该函数的返回值 (*return value*)。例如：

⁴ 这些程序片段全部出现在 `Main()` 函数内，为了节省版面，我没有列出完整的 `Main()`。


```
public static int Main( string [] args )
{
    if ( args.Length == 0 )
    {
        display_usage();
        return -1;           // indicate failure
    }
}
```

这里存在一个规则: return 语句之后的值必须与函数返回型别 (function return type) 兼容。“兼容”有两个含义, 最简单的情况是“返回值的型别”与“函数所声明的返回型别”完全吻合, 例如数值 -1 的型别是 int, 函数声明的回传型别也是 int。“兼容”的第二个意思是, 在“实际返回值”与“函数的返回型别”之间存在隐式转换 (implicit conversion)。

if-else 语句使我们得以“依据某个特定条件的成立与否”来选择执行不同语句 (块)。如果测试条件为伪 (false), else 子句所代表的语句 (块) 就会被执行起来。如果我们打算“一旦发现 args array 为空, 不立即返回”, 可使用以下 if-else 语句替换原来的代码:

```
if ( args.Length == 0 )
    display_usage();
else { /* do everything else here */ }
```

为了取得个别的命令行选项, 我们可以采用 foreach 循环来遍历 args array, 依次读取每个元素。以下循环便是把每个命令行选项打印至控制台 (console):

```
foreach ( string option in args )
    Console.WriteLine( option );
```

其中 option 是一个只读的 (read only) string object, 只在 foreach 语句范围 (scope) 内可见。上述循环每进行一次迭代, option 就代表 args array 内的下一个元素。

在这个程序里, 我拿每一个 string 元素和程序所支持的选项进行比对。如果这个字符串与所有选项都不吻合, 就检查它是否代表一个文本文件名称。测试一组互斥条件时 (正如我们现在所处状况), 往往将这些测试以一连串的 if-else-if 语句链接在一起, 例如:

```
bool traceOn = false;
bool spyOn = false;

foreach ( string option in args )
{
    if ( option.Equals( "-t" ) )
        traceOn = true;
    else
        if ( option.Equals( "-s" ) )
            spyOn = true;
        else
            if ( option.Equals( "-h" ) )
                { display_usage(); return; }
            else
                check_valid_file_type( option );
}
```

关键字 `bool` 表示布尔 (Boolean) 数据类型, 它可以接受字面常量 `true` 或 `false`。上述例中, `traceOn` 和 `spyOn` 代表初值为 `false` 的两个 `bool` objects。

`Equals()` 是 `string` class 的一个 *non-static* 成员函数——这东西 (也称为 *instance* 成员函数) 必须由其 class 实体 (亦即 object) 调用, 此处的实体是 `option`。以下表达式:

```
option.Equals( "-t" );
```

要求编译器调用 `string` 的 *non-static* 成员函数 `Equals()`, 用以比对“`option` 所代表的字符串”和“字符串常量 `“-t”`”。如果两者相等就返回 `true`, 否则返回 `false`。

如果互斥条件是一种常量表达式 (constant expressions)⁵, 我们可以将“if-else-if 语句链”变成更具可读性的 `switch` 语句, 像这样:

⁵ 所谓常量表达式 (constant expressions) 代表一个可于编译期被评估 (核定) 的值。这通常意味该表达式中没有包含数据对象 (data object) ——因为数据对象的实际值只有在运行期才能被评估 (核定) 出来。


```

foreach ( string option in args )
{
    switch ( option )
    {
        case "-t":
            traceOn = true;
            break;

        case "-s":
            spyOn = true;
            break;

        case "-h":
            display_usage();
            return;

        default:
            check_valid_file_type( option );
            break;
    }
}

```

switch 语句可用来测试泛整数型别、char 型别、枚举型别 (enumeration)、string 型别。关键字 switch 之后紧跟着圆括弧围起来的一个表达式，另有一系列 case 标签 (labels) 跟在关键字 switch 之后，每个标签指定一个常量表达式。每个 case 标签所对应的值必须独一无二。（译注：标签后的冒号不可省略）

switch 圆括弧内的表达式，其计算结果依次被拿来和每个 case 标签比对，如果吻合，就执行 case 标签后的语句。如果全都不吻合，就轮到 default 标签，于是与 default 标签相关联的语句就会被执行起来。如果既没有任何标签吻合，又没有 default 标签，那么就不发生任何事情。每个 switch 语句块至多只能有一个 default 标签。

每一个不为空的 case 标签必须跟着一个 break 语句，或其他“起终结作用”的语句，例如 return 或 throw 等等，否则就会触发编译错误。throw 语句会将程序的控制权从“当下正在运行的函数”交给“运行期异常处理机制” (runtime exception-handling mechanism)。1.17 节对异常处理有一些介绍。break 语句的作用则是将程序控制权交给 switch 语句末尾大括弧之后的第一条语句。

空的 case 标签可以不带 break 语句。这种（空）标签往往用于“多个值对应同一个回应动作”的情况，例如：

```

switch ( next_char )
{
    case 'a':
    case 'A':
        acnt++;
        break;

    // to illustrate an alternative syntax ...
    case 'e': case 'E': ecnt++; break;

    // ... the other vowels

    case '\0': return; // OK

    default: non_vowel_cnt++; break;
}

```

case 标签彼此代表互斥条件，因此，如果我们希望执行两条 case 标签（而且第一条不为空），就必须使用 goto 语句，跳转到 default 标签或某个 case 标签去：

```

switch ( text_word )
{
    case "C#":
    case "c#":
        csharp_cnt++;
        goto default;

    case "C++":
    case "c++":
        cplusplus_cnt++;
        goto default;

    case "Java":
    case "java": goto case "C#";

    default:
        word_cnt++;
        break;
}

```


1.5 开启一个文本文件 (Text File) 以供读写

假设用户向程序提供了一个有效的文本文件名, 我们接下来要做的是开启该文件, 读取其内容, 并在处理之后将结果写入另一个文件, 该文件须由我们加以创建, 让我们看看这该怎么做。

支持“文件输入与输出 (I/O)”的 classes 被封装于 `System.IO` 命名空间中。因此我们要做的第一件事就是对编译器说: 请开启这个命名空间:

```
using System.IO;
```

我打算通过 `StreamReader` class 和 `StreamWriter` class 来读写文本文件。有多种做法可以创建这两个 classes 的实体 (object), 例如:

```
string file_name = @"C:\fictions\gnome.txt";  
  
StreamReader freader = File.OpenText( file_name );  
StreamWriter fwriter =  
    File.CreateText( @"C:\fictions\gnome.diag" );
```

其中 `OpenText()` 可返回 `StreamReader` 的一个实体, 该实体代表了“`OpenText()` 所接受之 string 引数”所指示的那个文件。本例之中它开启 c: 盘的 `fictions` 目录下的一个文本文件。注意, 引数字符串所表示的那个文件必须存在, 并且用户必须拥有其开启许可, 否则 `OpenText()` 会抛出一个异常。

字符 `@` 表示, 其后的字符串是个“逐字字符串” (verbatim string)。在一般的字符串字面常量中, 反斜线 (`\`) 被视为特殊字符, 例如当我们写 “`\n`” 时, 反斜线和 `n` 一起被解释为换行 (new-line) 字符, 这便是所谓的“转义序列” (escape sequence)。如果希望字符串中出现反斜线本身, 我们必须在它前面再加一个反斜线, 像这样:

```
string fname1 = "c:\\programs\\primer\\basic\\hello.cs";
```

但是对于逐字字符串字面常量 (verbatim string literal), 我们不需要使用“转义序列”就可以指定反斜线之类的特殊字符。

“逐字字符串”与一般字符串的另一个不同点是, 它能够跨越数行。这数行之

内的空白字符 (white space), 如换行字符 (new-line) 或制表字符 (tab) 皆保留, 这样便能允许存储和生成带有格式的文本块, 例如下面是 `display_usage()` 的一种可能实现方式:

```
public static void display_usage()
{
    string usage =
        @"usage: WordCount [-s] [-t] [-h] textfile.txt
        where [] indicates an optional argument
        -s prints a series of performance measurements
        -t prints a trace of the program
        -h prints this message";
    Console.WriteLine( usage );
}
```

回到上页程序, `CreateText()` 返回 `StreamWriter` 的一个实体, 如果函数引数 (某个字符串) 所代表的文件确已存在, 该文件会被覆盖 (overwritten)。如果你想将已有的文本文件尾端附加内容 (而不是覆盖原内容), 请使用 `AppendText()`。

`StreamReader` 提供了全套读取方法, 我们可以读取单一字符、一大块文本, 或使用 `ReadLine()` 读取一整行文本, 另外也有个 `Peek()`, 可读取下一字符但不取出 (即不改变数据流的当前位置)。`StreamWriter` 则提供了 `WriteLine()` 和 `Write()` 函数。

以下程序代码依次读取文本文件的每一行文本, 并将结果赋值给 `string` object `text_line`。如果返回的是一个 `null` 字符串, 表示已到达文件末尾。由于操作符优先级 (operator precedence, 详见 1.18.4 节) 的因素, 我们一定得在 `text_line` 的赋值动作周围加上圆括弧:

```
string text_line;
while (( text_line = freader.ReadLine() ) != null )
{
    // write to output file
    fwriter.WriteLine( text_line );
}

// must explicitly close the readers
freader.Close();
fwriter.Close();
```


一旦使用 `StreamReader` 完毕，我们必须调用相应的成员函数 `Close()`，这样才能释放它们占用的相关资源⁶。

1.6 格式化输出

除了将每一行文本写至输出文件，让我们对原先的程序代码作一些扩充，让它也把每一行文本写到控制台（console），并且除了输出文本本身，还标示行号和每一行的长度。此外，空行不做任何输出。下面是修改后的相关代码：

```
string text_line;
int line_cnt = 1;

while ( ( text_line = freader.ReadLine() ) != null )
{
    // don't format empty lines
    if ( text_line.Length == 0 )
    {
        Console.WriteLine();
        continue;
    }

    // format output to console:
    // 1 {42}: Master Timmothy Gnome left home one morning
    Console.WriteLine( "{0} {({2}): {1}",
                       line_cnt++, text_line, text_line.Length );
}
```

`continue` 语句会造成循环体的剩余部分“短路”（骤死）。本例之中如果文本行为空，就向控制台写出一个换行字符，并提前结束此次迭代（iteration）。`continue` 语句可使循环的下一轮迭代立即开始。

⁶ 如果你是一名 C++ 程序员，通常会利用 class 的析构函数（destructor）自动释放资源。然而在垃圾回收环境（garbage-collected environment）下，析构函数不能起“释放资源”的作用。为此我们提供了一个 `Dispose()` 函数（4.8 节），`Close()` 则是 `Dispose()` 的另一种形式。

类似情况，`break` 语句可令循环体提前结束。当我们为了查找某个特定值而遍历某个群集（collections）时，便可能用到 `break` 语句，因为我们希望一旦找到该特定值，就跳出循环。这个关键字的一个极端运用实例是无穷循环的终止条件的测试，例如：

```
while ( true )
{
    // process until some condition occurs

    // ...

    if ( condition_occurs )
        break;
}
```

这里假设 `classes` 或程序的某个内部状态最终会导致执行 `break` 语句，并因而终结循环。

我们可以在传给 `WriteLine()` 的字符串常量中放入位置引数（position argument），例如：

```
Console.WriteLine( "Hello, {0}! Welcome to C#", user_name );
```

`WriteLine()` 会将“字符串常量中以花括弧围起来的数字”视为该字符串后的“参数列相应值”的占位符（placeholder），0 代表第一参数，1 代表第二参数，依此类推。这些编了号码的占位符可以以任意顺序出现，例如：

```
Console.WriteLine( "{0} ({1}): {1}",
    line_cnt++, text_line, text_line.Length );
```

相同的占位符也可以出现多次，例如：

```
Console.WriteLine( "Hello, {0}! Everyone, please welcome {0}",
    user_name );
```

我们可以通过这种附加的格式字符（format characters）控制内建型别（基础型别）的数值格式。例如 `c` 代表“当地货币”（local currency），`F` 代表定点（fixed-point）

格式，E 代表指数形式（科学计数法），G 则允许系统选取最紧凑的输出形式。每个格式字符都可以附带一个用以表示精度的数值。例如，假设有个 double d：

```
double d = 10850.795;
```

那么以下的 WriteLine() 语句：

```
Console.WriteLine("{0} : {0:C2} : {0:F4} : {0:E2} : {0:G}", d);
```

将产出如下输出：

```
10850.795 : $10,850.80 : 10850.7950 : 1.09E+004 : 10850.795
```

格式字符 x 或 X 表示以 16 进制来表现数值；x 表示输出的 16 进制符号为大写的 A~F，X 表示输出的 16 进制符号为小写的 a~f。

1.7 string 型别

读取一行文本之后，我们需要把它拆成一个个单词。最简单的办法是利用 string 的 Split() 函数，像这样：

```
string text_line;
string [] text_words;
int line_cnt = 1;

while ((text_line = freader.ReadLine()) != null)
{
    text_words = text_line.Split( null );
    // ...
}
```

Split() 会返回一个 array，其中元素都是“根据用户指定之一组字符而分割开来”的字符串元素。如果将 null 传给 Split()，就像上例那样，它便以“空白字符”（white space）分割原始字符串。所谓“空白字符”是指空格（blank）或制表符（tab）。例如以下字符串：

```
A beautiful fiery bird, he tells her, magical but untamed,
```

便因此被分割为一个“含有 10 个字符串元素”的 array。其中三个单词（bird, her 和 untamed）带有它们身后的标点符号。如果要除去这些标点符号，策略之一是直接为 Split() 提供“用以分割字符串”的一串字符，例如：

```
char [] separators = {  
    ' ', '\n', '\t', // white space  
    '.', '\"', '\'', ',', '?', '!', ')', '(', '<', '>', '[', ']'  
};  
  
text_words = text_line.Split( separators );
```

任何字符字面常量 (character literal) 都应该放在一对单引号里。new-line (换行) 字符和 tab (制表) 字符分别由转义序列 (escape sequence) `\n` 和 `\t` 表示, 每个转义序列代表一个字符。双引号也必须以特殊方式表示 (`\"`), 这样它才能被解释为一个字符, 而不是一个字符串字面常量 (string literal) 的起始标志。

string 型别支持 subscript (下标) 操作符 (`[]`), 但它仅支持读取 (而非涂写) 动作。下标索引以 0 开始, 最大为 `Length-1` 个字符, 例如:

```
for ( int ix = 0; ix < text_line.Length; ++ix )  
    if ( text_line[ ix ] == '.' ) // OK: read access  
        text_line[ ix ] = '_'; //error: no write access
```

string 型别并不支持“以 foreach 遍历字符串中的所有字符”, 原因是 string object 具有不可变性 (immutable), 这和我们的直觉稍有差异⁷。在搞清楚这个意思之前, 让我们先大致理解 C# 的 for 循环语句。

for 循环语句由以下元素组成:

```
for ( init-statement; condition; expression )  
    statement
```

其中 init-statement 在循环执行之前就先被执行一次。上述例子里, 循环开始执行之前先把 ix 初始化为 0。

上式中的 condition 代表循环控制条件。它在循环每次迭代之前被评估 (求值), 只要 condition 被评估为 true, 就会执行一次 statement。statement 可以是单一语句或语句块。如果 condition 第一次评估结果就是 false, 那么 statement 永远不会被执行。在我们的例子里, condition 用来测试“ix 是否小于 text_line.Length”, 其中 text_line.Length 是字符串内含的字符个数。只

⁷ 技术上的原因是, string class 并未实现 IEnumerable 接口。第 4 章对于 IEnumerable 接口有一个完整的讨论, 对接口 (interface) 也有一般性的讨论。

要 `ix` 指向一个有效的字符元素，循环就继续运行。

`for` 循环语句中的 `expression` 会在每一次迭代完成后被评估求值。一般而言它用来修改“在 `init-statement` 中被初始化并在 `condition` 中被测试”的那个 `object`。如果 `condition` 第一次评估结果就是 `false`，那么 `expression` 永远不会被执行。本例中每次迭代之后，`ix` 累加 1。

我们不可以对字符串字面常量 (`string literal`) 里的个别字符进行涂写动作，原因是 `string object` 是不可变的 (`immutable`)。有时候看起来好象我们改变了 `string object` 的内容，其实真正发生的是“一个新的 `string object` 被创建出来，内含我们对原 `string` 的改动”。

举个例子，如果我们要对文本文件中的单词出现次数做出正确统计，就应该把 `A` 和 `a` 视为同一个单词。要实现这个目的，办法之一是在存储每个字符串之前，先把它们全部变成小写：

```
while (( text_line = freader.ReadLine() ) != null )
{
    // oops: this doesn't work as we intended ...
    text_line.ToLower();
    text_words = text_line.Split( null );

    // ...
}
```

其中 `ToLower()` 将 `text_line` 内的所有大写字母正确转为小写（另有一个名为 `ToUpper()` 的成员函数）。问题是，转换后的内容系被存储于“调用 `ToLower()` 后，返回的那个新的 `string object`”里。由于我们没有把这个新的 `string object` 赋值给任何 `object`，因此它也就这样被扔掉了。`text_line` 依旧没有改变，除非我们把返回值重新赋值给它，像这样：

```
text_line = text_line.ToLower();
```

如果你希望尽量减少“因多次修改某个 `string` 而造成的新 `string object`”的个数，请使用 `StringBuilder class`，详见 4.9 节。

1.8 局部对象 (Local Objects)

Data object (数据对象) 必须定义于函数或 classes 之内。它不能作为一个独立 object 存在于命名空间 (namespace) 或全局声明空间 (global declaration space) 内。定义于函数体内的 object, 称为 local objects (局部对象), 它们自“它们所在的外围函数开始运行”时形成, 并在函数运行结束后不复存在。Local objects 没有缺省初始值。

读写一个 local objects 之前, 必须让编译器确信它已经赋予这个 object 初值。打消编译器顾虑的最简单办法就是, 定义任何 local objects 时就一并将它初始化, 例如:

```
int ival = 1024;
```

这条语句定义了一个整数 ival, 并将其初值设为 1024。

有时候, 让某个 object 拥有初值, 似乎不是很合理, 因为我们并不考虑在对它赋值之前使用它。例如, 考虑以下程序片段中的 user_name:

```
static int Main()
{
    string      user_name;
    int          num_tries = 0;
    const int    max_tries = 4;

    while ( num_tries < max_tries )
    {
        // generate use messages ...

        ++num_tries;
        user_name = Console.ReadLine();

        // test whether entry is valid
    }

    // compiler error here!
    // use of unassigned local variable user_name
    Console.WriteLine( "Hello, {0}", user_name );
    return 0;
}
```


检测后发现, `user_name` 总是会在 `while` 循环内被赋予一个值——因为 `num_tries` 被初始化为 0, 所以 `while` 循环至少总要运行一次, 但编译器却认为我们在 `WriteLine()` 语句中使用 `user_name` 是非法的。这是不是大出你的意料?

是这样的, 每次访问一个 `local object` 前, 编译器都会先检查它是否已被明确赋予某值, 编译器依赖静态流程分析 (static flow analysis, 编译期分析) 做出判断, 所以它不知道 `non-const object` (例如 `num_tries`) 的值——即便这些值对我们来说显而易见, 编译器的静态流程分析假设 `non-const object` 可能持有任何值; 基于这个假设, `while` 循环不保证会被执行, 因此 `user_name` 不保证会被赋予某值, 因此编译器发出错误消息。

编译器仅能完全评估“常量表达式” (例如字面值 7 或 'c') 和只读的 (read only) `const object` (例如上例 `max_tries`) 的值, `non-const object` 的值和表达式的值只在运行期才能完全确定下来, 这就是为什么编译器认为 `non-const object` 可能持有任意值, 这不失为最保守 (也最安全) 的做法。

对于上述程序, 改正办法之一就是提供一个“轻松带过”的初始值:

```
string user_name = null;
```

另一种解法是利用 C# 的第四种循环语句 `do-while`, `do-while` 在评估循环条件之前会先执行循环体一次, 如果以 `do-while` 重写程序, 就连编译器都能认可“`user_name` 肯定会被赋值”这一事实, 因为其赋值动作和 `non-const object` `num_tries` 的值毫不相干。

```
do
{
    // generate user message ...
    ++num_tries;
    user_name = Console.ReadLine();
    // test whether entry is valid
} while ( num_tries < max_tries );
```

Local objects 和其他 objects 有点不同，它们的使用有着次序依赖性（order dependent）：除非先做声明，否则你无法使用 local objects。这条规则有一个隐微的扩充解释：某个名称一旦在某个局部生存空间（local scope）内被用过，那么如果你试图改变既有名称的含义（例如引入该名称的一个新声明），会导致错误。看看这个例子：

```
public class EntryPoint
{
    private string str = "hello, field";
    public void local_member()
    {
        // OK: refers to the private member
        /* 1 */ str = "set locally";

        // error: This declaration changes the
        // meaning of the previous statement
        /* 2 */ string str = "hello, local";
    }
}
```

标示 /* 1 */ 之处所赋值的 str 被决议（resolved）为 EntryPoint class 的 private 成员。标示 /* 2 */ 之处则因为声明了 local object string str，造成 str 的意义有所变化。是的，C# 不允许改变 local 标识符的含义，所以上例 str 的 local 定义式出现点会触发编译错误。

如果我们把 private 数据成员 str 的声明移到上述函数定义之后，会怎样？这么做并不能改变编译器的行为。classes 的整个定义式在“每一个成员函数的函数体被评估”之前已检查完毕。每个成员的名称和型别都记录在 class 声明空间中供编译器做下一步检查。成员声明式的先后顺序无关紧要，例如：

```
public class EntryPoint
{
    // OK: let's place this first
    public void local_member()
    {
        // still refers to the private class member
        /* 1 */ str = "set locally";

        // 译注：C++ 上的相应规则，可参考 C++ Primer, 3e, p666, 简体版 p546
        // still the same error
        /* 2 */ string str = "hello, local";
    }
}
```



```
// position of member does not change its visibility  
private string str = "hello, field";  
}
```

注意，每个 local 区块都维护/支持一个声明空间 (declaration space)。此区块内声明的名称，在区块之外是不可见的。当然啦，如果你有一个区块嵌套定义于某区块之内，那么外区块的名称在内区块中可见。例如：

```
public void example()  
{ // top-level local declaration space  
    int ival = 1024;  
  
    // ival is still in scope here  
  
    double ival = 3.14; // error: reuse of name  
    string str = "hello";  
}  
  
{  
    // ival still in scope, str is not!  
    double str = 3.14159;  
}  
  
// what would happen if we define a str object here?  
}
```

如果我们在上述函数的末尾添加一个 local object str 的声明，会发生什么事？最后加上的这个声明式将出现在局部声明空间 (local declaration space) 的最顶层。尽管先前分别在两个嵌套定义之 local 区块内使用标识符 str 是合法的，但是最后加上的这个位于最顶层 (top-level) 的声明式将会破坏合法性，结果产生编译错误。

为何有如此严厉的措施用以制止局部声明空间 (local declaration space) 内的名称被重复运用？某种程度上是因为局部声明空间由程序员全权管理。也就是说，严厉的政策性强制措施并不会带给程序员繁重的负担，反而可以让他们一下子就发现错误，并就地修改造成冲突的标识符。做这些修改的同时，程序员会思考并改善程序的清晰度。

1.9 Value 型别和 Reference 型别

C# 提供的型别 (types) 分为两类: value 型别和 reference 型别。这两种型别的 object 复制行为和修改行为有着相当大的差异。

value object 本身存储其相关数据。你对这些数据做任何改动都不会影响其他 object。语言内建的算术型别如 `int` 和 `double`, 都是一种 value 型别。当我们写:

```
double pi = 3.14159;
```

3.14159 这个值将直接存放于变量 `pi` 内。

当我们以一个 value object 初始化另一个 value object, 或是将一个 value object 赋值给另一个 value object 时, 前者所含数据将被复制一份给后者; 两个 objects 依旧保持独立。举个例子, 当我们这么写:

```
double shortPi = pi;
```

尽管此时 `pi` 和 `shortPi` 持有相同的值, 但这些值是两个独立实体, 分别包含于彼此独立的两个 objects 中。我们称此为“深拷贝” (deep copy)。

如果改变存储于 `shortPi` 内的值:

```
shortPi = 3.14;
```

`pi` 的值不会随之而变。这些介绍尽管似乎显而易见, 简直就是枯燥说教了, 但这一切都不会在我们复制和修改 reference object 时发生!

reference object 分为两部分:

1. 一个具名的 (named) handle, 供我们直接操控。
2. 一个不具名 (unnamed) object, 其型别与 handle 相应, 存储在被称为 *managed heap* (受控堆) 处。这个 object 必须由表达式 `new` 创建出来 (详见 1.11 节)。

上述 handle 若非持有 heap object 的地址, 就是被设为 `null`, 表示它目前并未代表任何 object。当我们以一个 reference object 初始化或赋值给另一个 reference object 时, 只有存储于 handle 内的地址会被复制。此时这两个 reference objects 代表的是 heap 内的同一个 object。如果你修改其中一份实体所代表的 object, 会影响另一份实体。我们称此为“浅拷贝” (shallow copy)。

所有 classes 都被 C# 视为 reference 型别。当我们写:

```
class Point
```



```
{  
    float x, y;  
    // ...  
}
```

```
Point origin;
```

其中的 Point 是个 reference 型别, origin 会反映出 reference 的行为。

struct 定义式允许我们引入用户自定义的 value 型别, 例如:

```
struct Point  
{  
    float x, y;  
    // ...  
}
```

```
Point origin;
```

此时的 Point 是个 value 型别, origin 会反映出 value 的行为。从性能上讲, value 型别通常更高效, 至少对小型、反复大量运用的 object 来说是如此。关于这个主题, 请见 2.19 节的详细讨论。

预定义的 C# array 是一种 reference 型别, 下一节对 array 的讨论应该能够说明 reference 型别的特性。

1.10 C# array (数组)

C# 语言内建的 array 主要用来保存单一型别的多个元素, 是一种固定尺寸 (fixed-size) 的容器。虽说是固定容量, 但当声明一个 array object 时, 其实际尺寸并不在声明式中出现, 事实上直截了当地写出容器大小反而会产生编译错误, 例如:

```
string [] text; // OK  
string [ 10 ] text; // error
```

声明一个多维 (multidimensional) array 时, 每一维度之间应以逗号隔开:

```
string [,] two_dimensions;  
string [,,] three_dimensions;  
string [,,,] four_dimensions;
```

如果我们这么写:

```
string [] messages;
```

这就表示 messages 是个 handle, 代表一个 array, 其中各元素的型别都是 string。

请注意, `messages` 本身并不是一个 `array object`. 缺省情况下 `messages` 被设为 `null` (无物). 在对 `array` 存储元素之前, 我们需得先以 `new` 表达式产生一个 `array object`, 那时才是我们指明尺寸的地方:

```
messages = new string[ 4 ];
```

现在 `messages` 代表拥有 4 个字符串元素的一个 `array`, 第一元素的序号 (下标) 为 0, 第二元素的序号为 1, 依此类推:

```
messages[ 0 ] = "Hi, Please enter your name: ";  
messages[ 1 ] = "Oops, Invalid name. Please try again: ";  
// ...  
messages[ 3 ] = "Well, that's enough. Bailing out!";
```

如果试图访问未定义的元素, 例如访问 `messages array` 中尚未定义的第 5 个元素:

```
messages[ 4 ] = "Well, OK: one more try";
```

会导致抛出一个运行期异常 (runtime exception) 而非一个编译错误:

```
Exception occurred: System.IndexOutOfRangeException:  
An exception of type System.IndexOutOfRangeException was thrown.
```

1.11 new 表达式

`new` 表达式不仅可以在程序的 `managed heap` (受控堆) 上分配单一 `object`:

```
Hello myProg = new Hello(); // 注意, () 必不可少
```

也可以用来在程序的 `managed heap` (受控堆) 上分配以 `object` 组成的 `array`:

```
messages = new string[ 4 ]; // 译注: messages 声明于本页最上
```


关键字 `new` 之后的型别名称, 其后若非紧跟一对圆括弧(用以表明单一 `object`), 就是紧跟一对方括弧(表明这是个 `array`)。单个 `reference object` 的分配将在 2.7 节谈及 `class` 构造函数 (`constructors`) 时再讨论。本节剩余篇幅的重点在于 `array` 的分配。

除非为 `array` 内的每个元素指定初值, 否则每个元素皆被初始化为缺省值(任何数值型别的缺省值都是 0, `reference` 型别的缺省值则是 `null`)。如果你想为 `array` 提供初值, 你必须指定一个“以逗号隔开”的常量表达式 (`constant expressions`), 并以花括弧围起来, 放在 `array` 维度之后:

```
string[] m_message = new string[4]
{
    "Hi, Please enter your name: ",
    "Oops, Invalid name. Please try again: ",
    "Hmm. Wrong again! Is there a problem? Please retry: ",
    "Well, that's enough. Bailing out!",
};

int [] seq = new int[8] { 1,1,2,3,5,8,13,21 };
```

初值的个数一定要和 `array` 的长度(尺寸)精确吻合。多或少了都会出现编译错误:

```
int [] ial;

ial = new int[128] { 1, 1 }; // error: too few
ial = new int[3] { 1,1,2,3 }; // error: too many
```

提供初值时, 也可以不指明 `array` 尺寸, 那么实际尺寸将按照初值个数自动计算出来:

```
ial = new int[] { 1,1,2,3,5,8 }; // OK: 6 elements
```

如果声明的是 `local array`, 有一个简短表示法可以使我们不必再明确调用 `new` 表达式, 例如:

```
string[] m_message =
{
    "Hi, Please enter your name: ",
    "Oops, Invalid name. Please try again: ",
```

```
"Hmm. Wrong again! Is there a problem? Please retry: ",  
"Well, that's enough. Bailing out!",  
};
```

```
int [] seq = { 1,1,2,3,5,8,13,21 };
```

尽管 `string` 是一种 *reference* 型别,其分配却可以不使用 `new` 表达式。`string` object 的初始化可采“适用于 *value* 型别”的语法,像这样:

```
// a string object initialized to literal Pooh  
string Winnie = "Pooh";
```

1.12 垃圾回收 (Garbage Collection)

我们不必明确删除 `new` 表达式分配得来的 object,留给运行期环境 (runtime environment) 去清理即可。垃圾回收算法能识别出 *managed heap* (受控堆) 上不再被引用的 objects,并为这些 objects 打上“可回收”标记。

当程序在 *managed heap* (受控堆) 分配一个 *reference object* 时,例如下面的 array object:

```
int [] fib = new int[6] { 1,1,2,3,5,8 };
```

将被视为拥有一个 “active reference”。本例中的 `fib` 代表这个 array object。

好,现在我们以 `fib` 代表的 object 来初始化第二个 array handle:

```
int [] notfib = fib;
```

结果发生了浅拷贝 (shallow copy),也就是说 `notfib` 并未指向另一个 array object 并于其中拥有原本六个整数元素的复本, `notfib` 只是代表“`fib` 所指的那个” array object。

如果我们通过 `notfib` 更改 array 内的某个元素,例如:

```
notfib [ 0 ] = 0;
```

这个更动对 `fib` 来说也是可见的 (有影响的)。如果我们不能容忍这类间接改动 (有时称为 *aliasing*, 别名),就必须写出一个深拷贝动作:

```
// allocate a separate array object  
notfib = new int [6];
```



```
// copy the elements of fib into notfib
// beginning at element 0 of notfib
fib.CopyTo( notfib, 0 );
```

这样 notfib 和 fib 所指的就不再是同一个 array object。如果我们现在通过 notfib 修改它所代表的 array 内的某个元素，fib 所代表的那个 array 将不受影响。这就是浅拷贝和深拷贝的语义区别。

如果重新赋值 fib，让它指向一个新的 array object，例如一个包含 Fibonacci（费伯纳契）数列前 12 个数的 array：

```
fib = new int[12]{ 1,1,2,3,5,8,13,21,34,55,89,144 };
```

那么，fib 先前所指的那个 array object 就不再有所谓的 "active reference"。此后它可能被做上记号，并在垃圾回收器（garbage collector）活动期间被清理掉。

1.13 动态 array: ArrayList Collection Class

有时我们从文件中读取一行一行文本，并不直接处理，而是先把它们存储起来。我们可以选用 string array 作为容器来完成此项工作，但 C# array 是个大小固定的容器。我们认为实际需求大小应该随着被开启的文本文件而有所改变，所以 C# array 称不上灵活。

命名空间 System.Collections 提供了 ArrayList 这样一个容器类，当我们插入或删除元素时，其大小可以动态成长。下面是原先的读取循环（p.18）的修订版，将元素添加到容器中：

```
using System.Collections;
private void readFile()
{
    ArrayList text = new ArrayList();
    string text_line;

    // 译注：以下的 m_reader 相当于 p.18 的 freader
    while ( ( text_line = m_reader.ReadLine() ) != null )
    {
        if ( text_line.Length == 0 )
            continue;

        // insert the line at the back of the container
        text.Add( text_line );
    }

    // let's see how many we actually assed ...
```

```
Console.WriteLine( "We inserted {0} lines", text.Count );  
}
```

对着 `ArrayList` 插入单一元素, 最有效率的做法就是使用 `Add()` 函数, 它会在 `ArrayList` 末尾插入一个新元素:

```
text.Add( text_line );
```

至于 `Count`, 代表 `ArrayList` object 所持有的元素个数:

```
Console.WriteLine( "We inserted {0} lines", text.Count );
```

就像面对所有 `reference` 型别一样, 我们以 `new` 表达式在 `managed heap` (受控堆) 上创建一个 `ArrayList` object:

```
ArrayList text = new ArrayList();
```

`ArrayList` 的元素存储于一块连续内存中。一旦这块内存被填满, 它就分配一块更大的连续内存 (通常是原先的两倍大), 并把既有元素复制到新空间去。我们将这块“用以存储 object”的内存的大小称为 `ArrayList` 的容量 (`capacity`)。

`ArrayList` 的容量代表“有必要另分配一块新内存之前”, 可允许的元素总数。`ArrayList` 的 `Count` 表示当前存储于此 `ArrayList` object 内的元素个数。缺省情况下一个空的 `ArrayList` object 一经诞生便拥有 16 个元素容量。

创建 `ArrayList` object 时如有必要改变缺省容量, 可传给它一个容量新值, 例如:

```
ArrayList text = new ArrayList( newCapacity );
```

其中 `newCapacity` 必须代表一个合理的整数值。`Capacity` 代表 `ArrayList` object 的当前容量:

```
Console.WriteLine( "Count {0} Capacity {1}",  
                    text.Count, text.Capacity );
```

一旦完成了元素插入工作, 我们可以运用 `TrimToSize()` 调整 (缩小) `ArrayList` 的容量, 使它刚好吻合实际元素数量:


```
text.TrimToSize();
```

调整 ArrayList object 容量, 并不会限制我们后来再插入其他元素。如果插入新元素, ArrayList object 的容量会再次成长。

1.14 统一型别系统 (The Unified Type System)

定义任何一个 object, 都得指明其型别。object 的型别决定了它所能持有的数值种类, 以及它所允许的取值范围。例如 byte 是不带正负号的整数型别, 由 8 个 bits 组成, 因此以下定义式:

```
byte b;
```

便是声明 b 可持有整数值, 且其值必须位于 0 ~ 255 之间。如果我们尝试将一个浮点数赋值给 b:

```
b = 3.14159; // compile-time error
```

或是将一个字符串 (string) 赋值给 b:

```
b = "no way"; // compile-time error
```

抑或是将一个超出范围的整数赋值给 b:

```
b = 1024; // compile-time error
```

编译器会将以上各赋值动作都标上错误记号。这条规则对 C# array 也成立。如此说来, 为何 ArrayList 容器有能力持有任何类型的 object 呢?

这要归功于“统一型别系统 (unified type system)”。C# 预定义了一个名为 object 的 reference 型别。任何 reference 型别或 value 型别, 不论是语言预定义或程序员自定义, 都“是一种”object。这意味着我们所处理的任何类型的 object 都可以赋值给一个 object object。如果有以下 object:

```
object o;
```

那么下面的每一个赋值动作都合法:

```
o = 10;  
o = "hello, object";  
o = 3.14159;  
o = new int[ 24 ];  
o = new WordCount();  
o = false;
```

`ArrayList` 之所以可以持有任何类型的 `object`，正因为其元素型别被声明为 `object`。

`object` 提供数个 `public` 成员函数，其中最被频繁使用的是 `ToString()`，它返回一个“代表 `object` 实际型别”的字符串，例如：

```
Console.WriteLine( o.ToString() );
```

1.14.1 暗中装箱 (Shadow Boxing)

尽管不是那么一目了然，“将一个 `int` `object` 赋值给一个 `object` `object`”还是有些蹊跷，毕竟 `object` 是 *reference* 型别，而 `int` 是 *value* 型别。

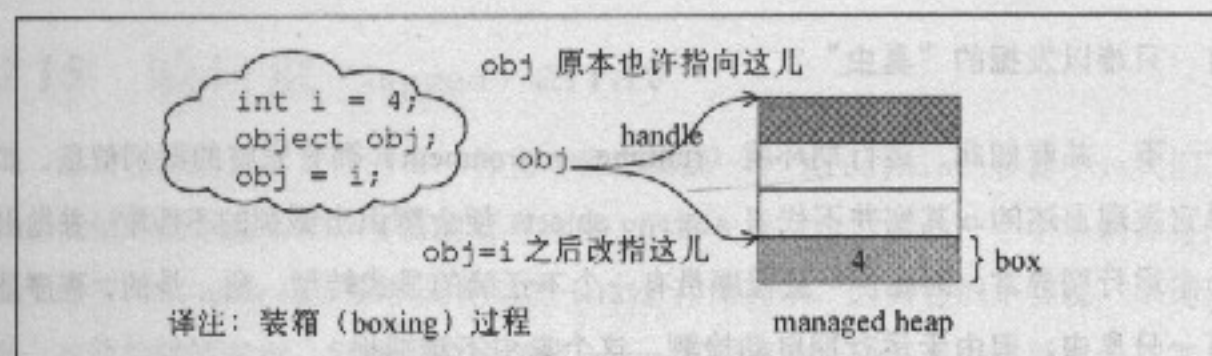
如果你不记得了，让我提醒你，*reference* 型别有两个成分：一是供我们在程序中操控的“具名 *handle*”，一是由 `new` 表达式在 *managed heap*（受控堆）上分配的“不具名 `object`”，当我们以一个 *reference* 初始化另一个 *reference*，或将一个 *reference* 赋值给另一个 *reference* 时，两个 `object handles` 就指向了 *heap* 内的同一个不具名 `object`。这就是先前介绍的“浅拷贝”（*shallow copy*）。

value 型别并不以一对 *handle/object* 来表现，*value object* 内直接包含其数据。*value* 型别并不在 *managed heap*（受控堆）上分配空间。它既非引用计数形式（*referenced-counted*），也非垃圾回收形式（*garbage-collected*），*value* 型别的生命期与其所处环境的存在时间相等，换句话说，*local object*（局部对象）的生命期等于“定义这一 `object`”之函数的运行时间。*class* 成员的生命期与其所属 `object` 的生命期相等。

“将 *value object* 赋值给一个 `object object`”的怪异之处，现在我们应该能够看得稍微清楚些了，`object object` 隶属于 *reference* 型别，那其实是一对 *handle/object*。*value* 型别则仅仅持有实值（而无 *handle*），而且不存储于 *heap* 之内，那么，一个 `object object` 的 *handle* 如何指向一个 *value object* 呢？

嗯，通过所谓 *boxing*（装箱）暗中转换步骤，编译器得以分配一个 *managed heap* 地址，并将该地址赋值给 `object object`。当我们为一个 `object object` 赋值一个字面常量或一个 *value object* 时，发生如下步骤：(1) 在 *managed heap* 内分配一个 `object box`（一块空间），准备用来持有 *value object* 值；(2) 将 *value object* 值复制到箱内；(3) 将箱子的“*managed heap* 地址”赋予 `object object`。

译注：*handle* 的功能相当于 C/C++ 指针，但其提领（*dereference*）动作是自动的，而且不允许指针算术运算。下图补充说明 *reference*、*handle/object*、*box* 的意义。



1.14.2 拆箱 (Unboxing) 与向下转型 (Downcast)

面对 `object object`, 我们能做的事情不多, 唯一可做的就是调用其 `public` 成员函数。我们不能访问原型别的任何成员 (包括 `properties` (属性) 和函数)。举个例子, 当我们把一个 `string object` 赋值给 `object object` 后:

```
string s = "cat";
object o = s;

// error: string property Length is not available
//         through the object instance ...
if ( o.Length != 3 )
```

对编译器来说, 原型别 (`string`) 的所有型别信息都遗失了。要是我们想访问 `string` 的 `Length` 属性, 须得先把 `object` 转型为 `string`。然而 `object` 不会自动转换为另一种型别:

```
// error: no implicit conversion of an object type
//         to any other type ...

string str = o;
```

唯有保证安全, 转换才会自动执行。编译器要做出判断, 就必须既知道原始型别, 也知道目标型别。对 `object object` 而言, 所有型别信息都不存在——至少对编译器来说是如此。当然, 程序运行期间, 这些“型别信息”和“环境信息”对运行期环境和对我们程序员来说, 都是可用的。第 8 章将告诉你如何访问这些信息。

对于“编译器无法保证安全”的任何转换, 用户 (程序员) 就必须自己做显式型别转换, 像这样:

```
string str = ( string ) o;
```

显式转型 (`explicit cast`) 用来告诉编译器, 即使编译期分析指出这个转换动作有潜在的不安全性, 也要执行之。万一程序员错了怎么办? 是否意味着我们的程序

有一只难以发掘的“臭虫”？

不，并非如此。运行期环境 (runtime environment) 拥有完整的型别信息，如果它发现上述的 `o` 其实并不代表 `string` object，便会辨识出型别的不匹配，并抛出一个运行期异常。因此，一旦程序员有一个不正确的显式转型，唔，是的，程序是有一只臭虫，但由于运行期自动检测，这个臭虫不难捕捉。

两个操作符可以协助我们检测转型的正确性：`is` 和 `as`。`is` 操作符用来询问某个 `reference` 型别实际上是否为某个特定型别，例如：

```
string str;

if ( o is string )
    str = ( string ) o;
```

`is` 操作符在运行期被评估 (evaluated) 其值。如果 `object` 的实际型别的确是该特定型别，就返回 `true`，但之后的显式转型仍旧必要，毕竟编译器并不评估我们的程序逻辑 (译注：也就是说编译器不会因为 `is` 返回 `true` 就自动做向下转型)。

第二种方法是，我们可以在运行期利用 `as` 操作符完成转型——如果实际 `object` 正是属于我们所讨论的某个特定型别。例如：

```
string str = o as string;
```

但如果上述的 `o` 不是相应型别 (此处为 `string`)，便不实施转换，于是 `str` 被设为 `null`。要察觉向下转型是否真的实施了，我们必须测试转换后的结果：

```
if ( str != null )
    // OK: o does reference a string ...
```

将一个 `object` 转换为特定之 `reference` 型别，需要的工作仅仅是把 (目标物的) `handle` 设为 `object` 的 `heap` 地址。将一个 `object` 转换为特定之 `value` 型别，需要做的工作稍微多一点，因为 `value` `object` 直接内含其数据。

把一个 `reference` 型别转回 `value` 型别，所做的附加工作称为拆箱 (unboxing)，原先被复制到箱内的数据又被复制回到 `value` `object` 内。同时，`managed heap` (受控堆) 上的数据箱 (box) 的引用计数 (reference count) 也相应减 1。

1.15 缺口型 (Jagged) array

既然已经把文本文件的每一行文本都存储到一个 `ArrayList` 容器中，我们下一步要做的便是遍历这个容器的所有元素，把每一行文本拆解为一个个单词并存储到对应的某个 `array` 内。我们要把这些 `arrays` 存储起来，因为它们是实现“单词计数”功能所需的素材。但是要这样存储 `arrays` 有些困难，唔，至少是困扰。别管是困难还是困扰，缺口型 `array` 提供了解决方案。

如果仅仅只是从容器中读取元素，`foreach` 循环是最佳的迭代/遍历手法，可以省下“将 `object` 元素显式转型为其实际（原本）型别”的工作，其他遍历方式的“元素赋值动作”均需实施强制转型：

```
for( int ix = 0; ix < text.Count; ++ix ) {  
    string str = ( string )text[ ix ];    // 需强制转型  
}  
  
// read-only access ...  
foreach ( string str in text ){ ... }    // 不需强制转型
```

把一行文本分解为单词所组成的 `array`，很容易：

```
string [] words = str.Split( null );
```

下一步也很简单，但一开始可能令人困惑：我们要把这些 `arrays` 存入另一个 `array` 内，也就是说我们需要一个“以 `arrays` 为元素”的 `array`。

外围的 `array` 代表实际文本，索引为 1 的元素（也是个 `array`）代表第一行文本，索引为 2 的元素（也是个 `array`）代表第二行文本，依此类推。C# 语言的多维 `array` 不能满足我们的需求，因为它要求每一维的大小都必须确定。

在我们所处理的问题中，第一维的大小是确定的，就是文本文件的行数，第二维的大小随着每一行所包含的单词多寡而变化，这正是“缺口型 `array`”用以对付的情形。以 `array` 为元素，每个作为元素的 `array` 可以拥有各自独立的大小，这就是缺口型 `array`，其语法是“每一维对应一对空的方括弧”。例如我们即将用到的“`array` 的 `array`”是个二维 `array`，因此其声明看起来像这样子：

```
string [][] sentences;
```

我们分两个步骤来初始化这个 array。第一步骤先为第一维分配空间，其长度是原本存储于 ArrayList object 内的文本行数：

```
sentences = new string[ text.Count ][];
```

这个语句表示 sentences 是个长度为 text.Count 的 array，其元素是“以 string 为元素”的一维 array。唔，这正是我们想要的。

接下来以实际的 string array 逐一初始化 sentences 的每个元素。我们遍历 ArrayList，取出每一行文本，执行 Split()，并将其返回值赋值给 sentences 内的相应元素：

```
string str;
for( int ix = 0; ix < text.Count; ++ix )
{
    str = ( string )text[ ix ];
    sentences[ ix ] = str.Split( null );
}
```

我们可以通过 sentences 的第一维来访问个别的 string array。举个例子，如果要打印 sentences 内每个 string array 的元素个数和内容，可以这么写：

```
// returns length of first dimension ...
int dim1_length = sentences.GetLength( 0 );
Console.WriteLine( "There are {0} arrays stored in sentences",
    dim1_length );

for( int ix = 0; ix < dim1_length; ++ix )
{
    Console.WriteLine( "There are {0} words in array {1}",
        sentences[ ix ].Length, ix+1 );

    foreach ( string s in sentences[ ix ] )
        Console.Write( "{0} ", s );

    Console.WriteLine();
}
```

所有 C# array 都可以访问定义于 System 命名空间内的 Array class 的 public 成员——本例用到的 GetLength() 就是其中一个。不过，Array 提供的大部分函数如 Sort(), Reverse() 和 BinarySearch() 等等，仅支持一维 array。

1.16 Hashtable 容器

`System.Collections` 命名空间提供了一个 `Hashtable` (散列表) 容器。`Hashtable` 用来表现一组 `key/value`, 其中 `key` 可用来快速查找。`Hashtable` 在其他程序语言里可能被称为 `map` 或 `Dictionary`。现在我将运用 `Hashtable` 来保存每个单词的出现次数。

我把单词当做 `key`, 把单词的出现次数当做 `value`。如果单词尚未存入 `Hashtable`, 就添加一组 `key/value`, 并将出现次数 (`value`) 设为 1。否则就以 `key` 找出它所对应的“单词出现次数”, 并累加 1。看起来应该如此这般:

```
Hashtable words = new Hashtable();
int dim1_length = sentences.GetLength( 0 );

for( int ix = 0; ix < dim1_length; ++ix )
{
    foreach ( string st in sentences[ ix ] )
    {
        // normalize each word to lowercase
        string key = st.ToLower();

        // is the word currently in Hashtable?
        // if not, then we add it ...

        if ( ! words.Contains( key ) )
            words.Add( key, 1 );

        // otherwise, we increment the count
        else
            words[ key ] = (int) words[ key ] + 1;
    }
}
```

为了找出某个 `key` 是否出现于 `Hashtable` 内, 我们调用 `Hashtable` 的 *predicate method* `Contains()`, 它如果返回 `true` 便表示找到了目标物。(译注: 所谓 *predicate method* 系指返回值为 `bool` 的函数)。欲为 `Hashtable` 加入一组 `key/value`, 可以这么做:

```
words[ key ] = 1;
```

也可以通过 `Add()` 完成:

```
words.Add( key, 1 );
```

典型情况下 key 应该是 string 或某种内建数值型别, value (译注: 请注意这里指的是 key/value 而非前述的 reference/value) 用来保存实际数据, 其型别与具体应用有关, 通常是一个用以支持题域 (problem domain) 的 class. Hashtable 能够支持各种不同型别的 key 和 value, 因为它把它们统统声明为 object 型别。

当然啦, 由于 Hashtable 将 value 存储为 object 型别, 所以我们必须将它显式 (explicitly) 转回其原本型别。还记得吗? 如果面对的是 value 型别, 这种转型需要拆箱过程 (unboxing)。对拆箱后得到的值所做的任何改动, 都不会自动反映回 Hashtable 存储的原值身上, 因此需得“重设” Hashtable 内的实体:

```
words[ key ] = (int) words[ key ] + 1; // 译注: 转型、改变内容、重设
```

实际应用上, 统计诸如 a, an, if, but 等极普通的单词的出现次数, 可能有用途, 也可能没啥用。如果不想统计这些单词, 一个办法是为这些单词创建第二个 Hashtable, 像这样:

```
Hashtable common_words = new Hashtable();
```

```
common_words.Add( "the", 0 );
```

```
common_words.Add( "but", 0 );
```

```
// ...
```

```
common_words.Add( "and", 0 );
```

然后在我们将单词加入主表 (先前说的第一个 Hashtable) 前, 先检查它是否出现于此第二个 Hashtable 内:

```
foreach ( string st in sentences[ ix ] )
```

```
{
```

```
    string key = st.ToLower();
```

```
    if ( common_words.Contains( key ) )
```

```
        continue;
```

```
}
```

至此, 程序的剩余工作是: 把单词的出现次数按字典顺序输出至某个文件内。这该如何实现? 第一个想法是利用 foreach 遍历整个 Hashtable。欲完成这件事, 可通过 DictionaryEntry object, 逐次访问每一组 key/value. DictionaryEntry object 提供了一对属性 (properties): Key 和 Value:

```
foreach ( DictionaryEntry de in words ) // 译注: 以下 fwriter 见 p.17
    fwriter.WriteLine( "{0} : {1}", de.Key, de.Value );
```


但是,就像编程上的众多解法一样,它“既有效又不太有效”。好消息是它的确打印出 Hashtable 容器内的各个单词及其出现次数,坏消息是它并非以字典顺序打印,例如前数条输出如下:

```
lime-tinted : 1
waist : 1
bold : 1
```

关键在于 Hashtable 内的那些 keys 是基于其 *hash value* (散列值) 放入的,而我们不能直接改变该值。一个解决办法是取出 key, 放入可排序容器内,然后排序,然后遍历该容器。针对已排好序的每个 key, 我们再取回其相应的 value:

```
ArrayList aKeys = new ArrayList( words.Keys );
aKeys.Sort();

foreach ( string key in aKeys )
    fwriter.WriteLine( "{0} :: {1}", key, words[ key ] );
```

这样便解决了我们的最后一个问题。前数条输出如下:

```
apron :: 1
around :: 1
blossoms :: 3
```

接口 (interface) IDictionary 为存储和取回 key/value 提供了一个抽象模型,第4章对此有详细介绍。Hashtable 实现这一接口,并将 key 和 value 定义为 object 型别。命名空间 System.Collections.Specialized 内定义有数个强型 (strongly typed) 且特化 (specialized) 的 classes, 包括:

- **StringDictionary**: key 的型别被限定为 string, 并区分大小写。
- **ListDictionary**: 这是 IDictionary 的一个实现,以单向链表完成。如果元素个数不超过 10, 它比 Hashtable 更小更快。
- **NameValueCollection**: 一个已序 (sorted) 且 key/value 均为 string 的数据群集 (collection)。其中的 value 可通过 key 或 index (索引序号) 来访问。NameValueCollection 可以在单个 key 之下存储多个 string values。

1.17 异常处理 (Exception Handling)

我们的 WordCount 程序即将大功告成，至少功能方面如此。余下的工作重点主要是错误检测。举个例子，如果用户指定了一个并不存在的文件，该当如何？如果开启的文件的格式不为我们的程序所支持，又该怎么办？检测出这种错误往往很容易。例如要检测一个文件是否存在，可以调用 File class 的 Exists() 函数，并把用户指定的文件名称以 string 传递过去：

```
using System.IO;
if ( ! File.Exists( file_name ) )
    // oops ...
```

难点在于处理方式和报告方式的选择。 .NET 环境下通常我们会通过异常处理 (exception handling) 方式来报告程序中的所有异常。这正是我接下来要讲的主题。

异常处理主要由两部分构成：(1) 经由 throw 表达式，辨识并发出一个异常，(2) 在 catch 子句内处理异常。以下是 throw 表达式的写法：

```
public StreamReader openFile( string file_name )
{
    if ( file_name == null )
        throw new ArgumentNullException();

    // reach here only if no ArgumentNullException thrown
    if ( ! File.Exists( file_name ) )
    {
        string msg = "Invalid file name: " + file_name;
        throw new ArgumentException( msg );
    }

    // reach here if file_name not null and file exists
    if ( ! file_name.EndsWith( ".txt" ) )
    {
        string msg = "Sorry. ";
        string ext = Path.GetExtension( file_name );

        if ( ext != string.Empty )
            msg += "We currently do not support " + ext + "files.";
    }
}
```



```
        msg = "\nCurrently we only support .txt files.";
        throw new Exception( msg );
    }
```

```
    // OK: here only if no exceptions thrown
    return File.OpenText( file_name );
}
```

throw 表达式抛出的 object 一定隶属于 Exception classes 阶层体系中的某个 class (第 3 章讨论面向对象编程时, 我会谈到所谓的继承以及 classes 阶层体系)。Exception class 定义于 System 命名空间内, 以某个字符串 (用来指出异常具体情况) 进行初始化。ArgumentException 是 Exception 的一个子类, 可以更加精准地指明异常种类。ArgumentNullException 又是 ArgumentException 的子类, 是这三个 objects 中最特定而精准的一个。

一旦抛出一个异常, 程序的正常运行便告暂停。异常处理设施会在函数调用链 (call chain, 译注: 亦即 call stack) 中“逆向搜索”有能力处理这一异常的 catch 子句。

程序中应该以“和异常 object 之型别相吻合”的一个个 catch 子句来处理异常。catch 子句由三部分构成: 关键字 catch、圆括弧内的异常型别声明、大括弧内用以实际处理异常的一组语句。

catch 子句通常和 try 区块相关联。try 区块以关键字 try 起头, 后跟着一系列被封入大括弧内的程序语句。catch 子句位于 try 区块末尾。例如以下程序代码:

```
StreamReader freader = openFile( fname );
string textline;
```

```
while ( ( textline = freader.ReadLine() ) != null )
```

我们知道, openFile() 可能抛出三种异常, ReadLine() 可能抛出一种异常 (就是 IOException)。正如上述所写的那样, 这些程序代码并未处理这四种异常中的任何一种。如果要改变它, 我们应该把上述程序代码置入 try 区块内, 并与相应的一组 catch 子句关联起来:

```
try
{
    StreamReader freader = openFile( fname );
    string textline;

    while (( textline = freader.ReadLine() ) != null )
    {
        // do the work here ...
    }
}
catch ( IOException ioe)
{ ... }

catch ( ArgumentNullException ane )
{ ... }

catch ( ArgumentException ae )
{ ... }

catch ( Exception e )
{ ... }
```

现在，如果抛出一个异常，会发生什么？异常处理机制会环顾 throw 表达式四周，并询问：“此异常发生于 try 区块内吗”？如果是，异常 object 的型别就会被拿来依次和每个相应的 catch 子句内的异常声明进行比较。一旦比较成功（吻合），该 catch 子句内的所有语句就被执行起来。

以上是异常被完整处理的情况，程序得以继续运行下去。如果被执行起来的 catch 子句内没有使用 return 语句，程序运行权就交给“最后一个 catch 子句”的下一行语句。程序并不会回到“异常被抛出处”继续运行。

如果异常 object 的型别与所有 catch 子句都不吻合，或是“异常抛出处”并不位于 try 区块内，又该如何？那么就会中止当前执行的函数（假设为 A），并由异常处理机制往 A 的调用者（另一个函数）继续查找动作，以求寻找匹配的 catch 子句。

本页上端及 p.44 的例子中，如果 openFile() 内的三个 if 语句中的某一个抛出异常，freader 赋值动作以及 try 区块内的剩余部分都不会执行，改由异常处理机制取得程序控制权，依次检查每个相应的 catch 子句，试着找到吻合的异常型别。

如果函数调用链 (function call chain) 辗转开解 (unwound) 至程序进入点 `Main()`, 仍没能够找到相应的 `catch` 子句, 又该如何? 此时程序本身会被中止, 未经处理的异常会被传递给运行期调试器 (runtime debugger), 用户可以选择对它进行调试, 或是让它径赴黄泉。

我们可以在最末一个 `catch` 子句之后放置一个 `finally` 语句块, 作为 `catch` 子句组的最后附属品。 `finally` 语句块内的程序代码总是会在函数退离 (exit) 前被执行:

- 如果有异常发生并被处理, 那么首先是相应的 `catch` 子句, 然后是 `finally` 子句, 会在“程序恢复正常运行状态前”依次被执行起来。
- 如果有异常发生, 但没找到足堪匹配的 `catch` 子句, 那么 `finally` 子句会被执行起来, 函数剩余部分弃之不用。
- 如果没有异常发生, 函数正常终止, 那么 `finally` 子句会在最后一个 `non-return` 语句之后 (译注: 亦即 `return` 之前) 被执行起来。

`finally` 语句块的主要用途是减少程序代码的重复。由于“异常”和“非异常”的退离点 (exit points) 不同, 如果都必须执行某段程序, 而又没有 `finally` 区块, 那么就可能形成重复的程序代码。

1.18 C# 语言简要手册

欲完成 `WordCount` 程序, 我们还有三件事未进行: (1) 计时诊断 (timing diagnostics); (2) 条件输出追踪 (tracing) 语句; (3) 将代码包装到一个 `WordCount` class 内。最后一条十分重要, 值得以独立的一章 (第 2 章) 完整介绍。

5.5.2 节将讨论 `TraceListener` class 阶层体系, 届时我们再来看看如何生成追踪输出 (tracing output)。 8.6.3 节将讨论与 Win32 API 之间的相互操作性 (interoperability), 其中将介绍 timing (计时) 方面的支持。本节剩余篇幅将以一组带有注释的表格, 为你提供一份 C# 语言简要手册。

1.18.1 关键字 (Keywords)

所谓关键字, 是语言保留的一些标识符 (identifiers)。它们代表了基本 C# 语言的某种概念或设施。例如 `abstract`, `virtual`, `override` 详细描述了动态函数 (用以支持面向对象编程) 的不同类型。 `delegate`, `class`, `interface`, `enum`,

event, struct 代表多种可由我们自定义的复杂型别，全套关键字列于表 1.1。

表 1.1 C# 关键字

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

C# 中的标识符（名称）不能以数字起头，例如 1_name 是非法的，name_1 则合法，任何名称都不能与关键字雷同，除了一个例外：我们可以在 C# 关键字之前面加上 '@' 而重用它，例如：

```
class @class
{
    static void @static(bool @bool)
    {
        if (@bool)
            Console.WriteLine( "true" );
        else
            Console.WriteLine( "false" );
    }
}
```



```
class Class1
{
    static void M() { @class.@static( true ); }
}
```

前缀字符 '@' 在与其他语言进行接口整合 (interfacing) 时很有用, 更明确地说是在某个 C# 关键字被其他语言视为标识符时很有用, '@' 字符并非标识符的一部分, 只是提供一个“将此关键字视为标识符”的脉络背景。

1.18.2 语言内建的数值型别 (Built-in Numeric Types)

整数字面常量如 42 和 1024, 隶属于 int 型别。如果你要指定一个无正负号 (unsigned) 字面值, 必须以大写或小写的 'u' 作为后缀, 例如 42u 或 1024U。如果要指定型别为 long 的字面值, 必须以大写或小写的 'L' 为后缀, 例如 42L 或 1024L (为强化可读性, 最好采用大写 'L')。如果要指定字面值为 unsigned long, 可把两个后缀合在一起, 例如 42UL 或 1024LU。

型别的“大小”(无论带不带正负号)决定了它可以持有的数值范围, 例如 signed byte 的范围是 -128~127, unsigned byte 的范围是 0~255, 依此类推。

不过, 某些场合如果使用小于 int 的型别, 可能不符合直觉。除了偶尔用于 class 成员之外, 我尽量避免使用小于 int 的型别。例如以下代码:

```
sbyte s1 = 0;      // 译注: sbyte 小于 int
s1 = s1 + 1;       // error!
```

被视为错误, 编译器发出以下消息:

```
Cannot implicitly covert type 'int' to 'byte'
```

这里存在的一个规则是, 内建数值型别可以隐式 (implicitly) 转换为同等大小或更大的型别。因此, int 可以隐式晋升为 double。但如果从大型别转换为小型别, 就需程序员做显式 (explicitly) 转换。例如编译器就不允许从 double 到 int 的隐式转换, 所以以上转换必须显式进行:

```
s1 = (sbyte) ( s1 + 1 ); // OK
```

你可能要问,为何只是简单地将 `s1` 加上 1 竟需要显式转换?在 C# 里,整数运算的最低限度是以“带正负号的 32-bit 数值”完成。这意味着 `s1` 被用于算术用途时,其值需晋升为 `int`。当我们将不同型别的操作数 (operands) 混合使用时,两个操作数都必须晋升为“最小共通型别” (smallest common type)。例如 `s1+1` 运算结果为一个 `int`。

当我们对一个 `object` 赋值时 (例如此处对 `s1` 的赋值动作),右侧值的型别必须与 `object` 所属型别吻合。如果不是这样,右侧值必须能够被转为 `object` 所属型别,否则这一赋值动作会被编译器视为错误。

缺省情况下,浮点字面常量 (floating-point literal constant) 如 3.14 会被视为 `double` 型别。如果在其后加上大写 'F' 或小写 'f',可令它成为单精度的 `float` 型别。字符 (character) 字面常量须置于单引号中,例如 'a'。十进制数值 (decimal) 字面常量须有大写或小写的 'M' 作为后缀字符。大多数读者或许并不熟悉 `decimal` 型别,4.3.1 节有详细介绍。表 1.2 列出 C# 支持的各种内建数值型别。

表 1.2 C# 数值型别 (Numeric Types)

关键字	型别	用法
<code>sbyte</code>	signed 8-bit int	<code>sbyte sb = 42;</code>
<code>short</code>	signed 16-bit int	<code>short sv = 42;</code>
<code>int</code>	signed 32-bit int	<code>int iv = 42;</code>
<code>long</code>	signed 64-bit int	<code>long lv = 42, lv2 = 42L, lv3 = 42L;</code>
<code>byte</code>	unsigned 8-bit int	<code>byte bv = 42, bv2 = 42U, bv3 = 42u;</code>
<code>ushort</code>	unsigned 16-bit int	<code>ushort us = 42;</code>
<code>uint</code>	unsigned 32-bit int	<code>uint ui = 42;</code>
<code>ulong</code>	unsigned 64-bit int	<code>ulong ul = 42, ul2 = 4ul, ul3 = 4UL;</code>
<code>float</code>	单精度	<code>float f1 = 3.14f, f3 = 3.14F;</code>
<code>double</code>	双精度	<code>double dd = 3.14;</code>
<code>bool</code>	布尔值 (boolean)	<code>bool b1 = true, b2 = false;</code>
<code>char</code>	Unicode 字符	<code>char c1 = 'e', c2 = '\0';</code>
<code>decimal</code>	十进制数值	<code>decimal d1 = 3.14M, d2 = 7m;</code>

代表“C# 内建各型别”的关键字，其实是 `System` 命名空间内各种型别的别名。例如 `int` 其实就是 .NET 环境下的 `System.Int32`，`float` 其实就是 .NET 环境下的 `System.Single`。

C# 内建型别之底层表述，和其他所有 .NET 语言提供的型别是相同的。这意味着虽然简易型别 (simple type) 可以像数值一样地被简易操作，但我们也可以把它们当做“带有一组定义明确之 `public` 函数”的 `class` 来运用。这使我们的程序能更直接地和其他 .NET 语言结合在一起。第一个好处是我们不必为了“让其他语言也能识别”而翻译或修改型别。第二个好处是我们可以直接引用或扩充其他 .NET 语言所建立的型别 (译注：意指我们的 `classes` 甚至可以继承自 VB `classes`)。System 底层型别列于表 1.3。

表 1.3 System 底层型别

C# 型别	System 型别	C# 型别	System 型别
<code>sbyte</code>	<code>System.SByte</code>	<code>byte</code>	<code>System.Byte</code>
<code>short</code>	<code>System.Int16</code>	<code>ushort</code>	<code>System.UInt16</code>
<code>int</code>	<code>System.Int32</code>	<code>uint</code>	<code>System.UInt32</code>
<code>long</code>	<code>System.Int64</code>	<code>ulong</code>	<code>System.UInt64</code>
<code>float</code>	<code>System.Single</code>	<code>double</code>	<code>System.Double</code>
<code>char</code>	<code>System.Char</code>	<code>bool</code>	<code>System.Boolean</code>
<code>decimal</code>	<code>System.Decimal</code>		
<code>object</code>	<code>System.Object</code>	<code>string</code>	<code>System.String</code>

1.18.3 算术 (Arithmetic)、关系 (Relational) 和条件 (Conditional) 操作符

C# 预定义了一套算术、关系、条件操作符，可运用于内建的数值型别身上。算术操作符列于表 1.4，关系操作符列于表 1.5，条件操作符列于表 1.6，三个表都附含运用示例。

二元数值操作符 (binary numeric operator) 只接受“型别相同”的两个操作数。如果一个表达式由不同型别的操作数混合组成，这两个操作数的型别都必须隐式晋

级为最小共通型别。例如 `double` 和 `int` 相加会导致 `int` 首先晋级为 `double`，然后两个 `double` 相加。`int` 和 `unsigned int` 相加会导致两个操作数都先晋级为 `long`，然后再执行 `long` 的加法。

表 1.4 C# 算术操作符 (Arithmetic Operators)

操作符	意义描述	用法
*	乘法	<code>expr1 * expr2;</code>
/	除法	<code>expr1 / expr2;</code>
%	取模 (余数, Remainder)	<code>expr1 % expr2;</code>
+	加法	<code>expr1 + expr2;</code>
-	减法	<code>expr1 - expr2;</code>
++	累加 (increment by) 1	<code>++expr1; expr2++;</code>
--	递减 (decrement by) 1	<code>--expr1; expr2--;</code>

两个整数相除，结果仍为整数，余数将被截去，不进行四舍五入。余数可以由“取模 (余数) 操作符” (%) 取得：

5 / 3 的结果为 1, 5 % 3 的结果为 2
 5 / 4 的结果为 1, 5 % 4 的结果为 1
 5 / 5 的结果为 1, 5 % 5 的结果为 0

整数算术运算可以在受检 (checked) 或非受检 (unchecked) 环境中进行。在受检环境中，如果运算结果超出目标型别的储值范围，会抛出 `OverflowException` 异常。在非受检环境中不会如此。

浮点运算永远不会抛出异常，例如“除以 0”只会导致正无穷大或负无穷大。其他无效的浮点运算可能导致 `NAN` (not a number)。

关系操作符的评估结果为布尔值 (Boolean) `true` 或 `false`。布尔操作数和数值操作数不能混合使用，因此关系操作符不支持链接操作 (concatenation)。例如已知三个 `int` 变量 `a, b, c`，下面这样的“复合型不等比较式 (compound inequality)”：

```
// illegal
a != b != c;
```

是不合法的，因为 `a != b` 的运算结果是个布尔值，而 `int c` 无法与布尔值比较是否不等。

表 1.5 C# 关系操作符 (Relational Operators)

操作符	意义描述	用法
<	小于	<code>expr1 < expr2;</code>
>	大于	<code>expr1 > expr2;</code>
<=	小于或等于	<code>expr1 <= expr2;</code>
>=	大于或等于	<code>expr1 >= expr2;</code>
==	相等与否 (equality)	<code>expr1 == expr2;</code>
!=	不等与否 (inequality)	<code>expr1 != expr2;</code>

表 1.6 C# 条件操作符 (Conditional Operators)

操作符	意义描述	用法
!	logical NOT	<code>! expr1;</code>
	OR (短路式评估)	<code>expr1 expr2;</code>
&&	AND (短路式评估)	<code>expr1 && expr2;</code>
	logical OR (bool), 两端评估	<code>expr1 expr2;</code>
&	logical AND (bool), 两端评估	<code>expr1 & expr2;</code>
?:	条件式 (conditional)	<code>cond_expr ? expr1 : expr2;</code>

条件操作符的一般形式如下:

```
expr
    ? execute_if_expr_is_true
    : execute_if_expr_is_false;
```

如果 `expr` 被评估为 `true`, 问号 (?) 之后的那个表达式便被执行。如果 `expr` 被评估为 `false`, 冒号 (:) 之后的表达式便被执行。两条表达式的运行结果必须有相同的型别。以下示范如何运用条件操作符, 根据 `last_elem` 的值决定“紧跟在空白字符之后”要打印空白字符或逗号:

```
Console.Write( last_elem ? " " : ", " );
```

由于 assignment 操作符 (=) 会返回接受端 (左侧值), 于是我们可以把多个赋值动作链接起来。例如以下语句把 1024 同时赋予 val1 和 val2:

```
// sets both to 1024  
val1 = val2 = 1024; // 译注: 相当于 val1 = (val2 = 1024);
```

当运算结果必须赋值给某个 object, 而该 object 本身同时也是运算的参与者时, “复合式 assignment 操作符” 提供了一种简短表示法。我们不再需要这么写:

```
cnt = cnt + 2;
```

可以改写为这样:

```
// add 2 to the current value of cnt  
cnt += 2;
```

是的, 每个算术操作符都有相应的复合式 assignment 操作符:

```
+=      -=      *=      /=      %=
```

C# 程序员可使用递增操作符 (++) 和递减操作符 (--) 将 object 递增或递减 1:

```
cnt++; // add 1 to the current value of cnt  
cnt--; // subtract 1 from the current value of cnt
```

这两个操作符都有前置版本和后置版本, 前置 (prefix) 版返回“运算后”的值, 后置 (postfix) 版返回“运算前”的值, 无论哪一个版本, 执行之后, 被操作之 object 的值相同, 但返回值不同 (相差 1)。

1.18.4 操作符优先级 (Operator Precedence)

认清操作符优先级, 才能“抓得住”操作符的运用。当多个操作符组合于单一表达式内, 表达式的评估 (计算) 顺序取决于每个操作符的优先等级, 例如 $5+2*10$ 的结果总是 25 而非 70, 因为乘法比加法有较高的优先级, 因此表达式中的 2 总是先乘以 10 再加 5。

我们也可以改变操作符的优先级——只要在“希望被优先运行的操作符”周围放置圆括弧即可。例如 $(5+2)*10$ 的运行结果为 70。

以下是一些常用操作符的优先级。列出来的每一个操作符都比其后续列出的操作符的优先级高。同一行的各个操作符的优先级相等。对于优先级相同的操作符，表达式评估顺序由左而右进行（译注：但赋值运算由右而左进行）。

```

逻辑运算 Logical NOT (!)
算术运算 *, /, %
算术运算 +, -
关系运算 <, >, <=, >=
关系运算 ==, !=
逻辑运算 Logical AND (&)
逻辑运算 Logical OR (|)
条件运算 AND (逻辑与) (&&)
条件运算 OR (逻辑或) (||)
赋值运算 =

```

例如，考虑以下语句：

```
if (textline = Console.ReadLine() != null) ... // error!
```

我的意图是测试 `textline` 获得的是个 `string` 或是个 `null`。不幸的是 `inequality`（不等测试）操作符比 `assignment`（赋值）操作符的优先级高，导致评估结果与预期不同。其中的子表达式：

```
Console.ReadLine() != null
```

会先被评估，结果为 `true` 或 `false`，然后再把这一结果赋予 `textline`，这会导致错误，因为 C# 并不支持由 `bool` 到 `string` 的隐式转换。

要想正确评估上述表达式，我们必须使用圆括弧明确设定评估顺序：

```
if ((textline = Console.ReadLine()) != null) ... // OK!
```

1.18.5 语句 (Statements)

C# 支持四种循环语句：`while`、`for`、`foreach`、`do-while`。此外 C# 支持条件语句 `if` 和 `switch`。细节列于表 1.7、表 1.8、表 1.9 中。

表 1.7 C# 的循环语句

语句 (Statement)	用法 (Usage)
while	<pre>while (ix < size) { iarray[ix] = ix; ix++; }</pre>
for	<pre>for (int ix = 0; ix < size; ++ix) iarray[ix] = ix;</pre>
foreach	<pre>foreach (int val in iarray) Console.WriteLine(val);</pre>
do-while	<pre>int ix = 0; do { iarray[ix] = ix; ++ix; } while (ix < size);</pre>

表 1.8 C# 的 if 条件句

语句 (Statement)	用法 (Usage)
if	<pre>if (usr_rsp=='N' usr_rsp=='n') go_for_it = false; if (usr_guess == next_elem) { // begins statement block num_right++; got_it = true; } // ends statement block</pre>
if-else	<pre>if (num_tries == 1) Console.WriteLine(" ... "); else if (num_tries == 2) Console.WriteLine(" ... "); else if (num_tries == 3) Console.WriteLine(" ... "); else Console.WriteLine(" ... ");</pre>

表 1.9 C# 的 switch 语句

语句 (Statement)	用法 (Usage)
switch	<pre>//equivalent to if-else-if clauses above switch (num_tries) { case 1: Console.WriteLine(" ... "); break; case 2: Console.WriteLine(" ... "); break; case 3: Console.WriteLine(" ... "); break; default: Console.WriteLine(" ... "); break; } // can use string as well switch (user_response) { case "yes": // do something goto case "maybe"; case "no": // do something goto case "maybe"; case "maybe": // do something break; default: // do something break; }</pre>

2

Class 的设计

一个 class (类) 代表一种抽象概念, 这种抽象概念通常来自我们的应用域 (application domain)。例如在计算机图形学 (Computer Graphics) 领域, class 可以代表光源、镜头、几何形体 (球体、圆锥体、立方体)、曲线和曲面; 也可以代表数学上的矩阵 (matrices) 和向量 (vectors)。在 Windows 应用程序设计领域, 我们大量运用 classes 来表现 text boxes (文本框)、buttons (按钮)、label (标签)、message boxes (消息框)。本章的主要目的就是协助你熟悉 C# 语言对于“设计和实现 classes”所提供的支持。

一般而言, class 由两部分构成: 一部分是公开的操作 (operations) 和属性 (properties), 统称为 public interface (公开接口), 另一部分是非公开的实现细节 (private implementation)。身为 classes 用户, 我们是其 public interface 的消费者。例如我们知道, 要取得当前存放于 ArrayList object 中的元素个数, 就要访问其 Count property, 这就是其 public interface。Count 具体如何取得。这是实现细节 (也许 Count 可以作为数据成员存储起来, 或必要时才被计算并保存); 实现细节对我们用户来说是隐蔽的。本章的次要目的是带大家看看, 当我们设计自己的 classes 时, 如何将 interface (接口) 和 implementation (实现) 分开来。

2.1 我们的第一个独立 Class

Class 可以代表一个独立的抽象性, 或代表某种泛化 (general) 抽象性的一个特化 (specialization) 版本。例如 FileStream 和 MemoryStream 都是“位于 System.IO 命名空间内的 Stream class”的特化定义。Stream 代表一般性的“数据流” (流入或流出我们的程序), 是一个抽象的 (abstract) class, 因为虽然它定义了 stream

的行为（也就是 `public interface`），但并没有完整实现出来。`stream` 的完整实现留待更特化的 `file stream classes` 和 `memory stream classes` 去完成，这二者定义了 I/O 媒介。`file stream classes` 和 `memory stream classes` 都被称为 `stream class type` 的子型别（`subtypes`）。这种 `type/subtype` 的关系正是面向对象编程（`object-oriented programming`, `OOP`）的核心精神。第 3 章将详细探讨 `OOP`。

定义 `classes` 间的关系之前，我们首先要能够轻松地创建 `classes`。这正是本章目的：介绍如何制作独立的 `classes`。所谓独立，意指完整实现其功能性（`functionality`）。`System` 命名空间内的 `DateTime class` 和 `Buffer class` 可视为样例。

在第 1 章里，我们基本实现了一个用以统计“文件内的单词个数”的程序。本节我们要让那些已完成的工作成为设计 `WordCount class` 的素材。

从哪儿开始好呢？

我们需要做的第一件事情是确定 `WordCount class` 能够执行的操作集。这些操作将成为 `class` 的成员函数。总是会有人争论：两个或多个函数是否应当组合到一块儿？是不是该把某个函数分解为多个函数？一般说来，函数最好只执行单一任务。面对 `WordCount class`，我确定了以下四个操作：

1. `openFiles()`，确认用户提供之文本文件的有效性，如果有效，就开启该文件。此外还开启一个输出文件用以保存单词计数，再开启一个输出文件用以保存追踪记录（可有可无）。
2. `readFile()`，读取文本并存储起来，以备下一步处理。
3. `countWords()`，把文本分解为一个个单词，并统计单词出现次数。
4. `writeWords()`，按字典顺序将单词的出现次数输出到指定文件中。

此外，第一次创建 `WordCount class object` 时，还有一个初始化任务要完成；不再需要这一 `object` 时，也有一个清理作业要完成。我们将在后续各节中探讨这四个操作的实现手法。

一旦决定了成员函数的设置, 我们还需要确定每个成员函数的 interface(接口)。对每个成员函数而言, 其 interface 由两部分组成: (1) 函数的返回型别(return type); (2) 函数的参数列(又称 signature, "标记")。

“参数列”使我们得以将 object 传入函数内; 这些参数要么用作运算, 要么用来对外提供“从函数里攫取的信息”。“返回型别”表示函数传回之 object 的类型, 这一 object 通常代表函数内部的计算结果, 当然也可以表示操作状况(成功或失败)。请记住, 在 C# 领域(或更一般地说在 .NET 编程领域), 出现错误后的通常做法是抛出异常, 而不是返回一个示误状态码(例如 HRESULT)。

设计 class 成员函数时, 函数返回值和参数列往往可以省略。之所以可以这么做, 是因为一个 class object 可以通过 class 数据成员来维护自身状态。我们大可不必向成员函数传入数值, 或从成员函数返回数值。成员函数可以在“调用此函数的那个 class object”的内部数据成员上施行运算。这有助于简化编程模型(programming model)。

一旦确定成员函数名称、返回型别和参数列, 下一步我们要确定每个函数的访问级别(access level)。也就是说, 是否将某个函数声明为 public, 使整个程序都可以访问; 或是将它声明为 private, 只能由此 class 的其他成员函数调用。(面向对象编程还引入了另一个访问级别: protected。本书第 3 章详细介绍。)

乍看之下, 每个成员函数都似乎应该声明为 public, 如此一来用户便可以以任意顺序通过 WordCount object 开启(open)、读取(read)、计数(count)、写入(write)。然而若要达到这样的灵活性, 我们的实现会变得更复杂, 因为函数的调用顺序是相互依存的。例如, 我们很难想象用户在文件尚未被开启之前就要求做单词计数。因此, 我们的另一个策略是: 将整个调用序列封装为单个 public 函数, 例如命名为 processFile(), 再由它调用上述 4 个成员函数, 于是那 4 个成员函数便可以声明为 private。

让我们看看目前已有的成果。这是 WordCount class 定义式的第一个版本:


```
using System;
public class WordCount
{
    public void processFile()
    {
        openFiles();
        readFile();
        countWords();
        writeWords();
    }

    private void countWords()
    {
        Console.WriteLine( "!!! WordCount.countWords()" );
    }

    private void readFile()
    {
        Console.WriteLine( "!!! WordCount.readFile()" );
    }

    private void openFiles()
    {
        Console.WriteLine( "!!! WordCount.openFiles()" );
    }

    private void writeWords()
    {
        Console.WriteLine( "!!! WordCount.writeWords()" );
    }
}
```

成员函数必须定义于 class 定义式内。各函数的声明顺序并不重要。在某函数被调用之前，编译器无需先看到其声明。每个成员函数都明白指出其访问级别（access level）。缺省情况下，如果某个成员函数（或 class 的数据成员）未明白带有访问级别，便被视为 private 级别。

定义于某个命名空间（namespace）或全局声明空间内的 class，可被整体声明为 private 或 internal（后者意味着这个 class 仅在它出现的那个装配件（assembly）之内可见，第 8 章将详细介绍所谓的装配件）。一个 class 若没有指明其访问级别，将被视为 internal 级别。

尽管上述 class 的如此实现并没有展现什么功能，但它毕竟是完整的。继续下一步之前，我们最好先看到这个程序通过编译，并成功调用 public 成员函数 processFile()。我发现，对我自己而言，一步一步地在一个能够动作的程序上累加功能，比一口气编写一大堆代码而不知道它们是否可执行，生产力高得多。

要执行这个程序，我们需要提供一个入口 (entry point)。下面是个“剥离版本”，生成一个 WordCount object 并调用 processFile()：

```
using System;
public class WordCountEntry
{
    static public void Main()
    {
        Console.WriteLine( "Beginning WordCount program ... " );

        WordCount theObj = new WordCount();
        theObj.processFile();

        Console.WriteLine( "Ending WordCount program ... " );
    }
}
```

这个 class 和前述的 WordCount class 共同构成了一个完整的 C# 程序。

继续探讨 C# class 机制之前，让我们先开启一个 Visual Studio 项目 (project) 并运行这个程序。

2.2 开启一个新的 Visual Studio 项目

译注：以下各功能项的英文名称之后的括弧内，是 VS.NET 中文版所用名称。

现在让我们在 Visual Studio 中以 "Console Application" Templates (模板) 建立一个 C# 项目，并在其中容纳上述程序。假设 Visual Studio Start Page (起始页) 已经出现在你的屏幕上 (如图 2.1)，请选按 New Project (新建项目) 按钮，在 Project Types (项目类型) 下单击 Visual C# Projects (Visual C# 项目)。在 Templates (模板) 之下点击 Console Application (控制台应用程序)。将项目名称改为 WordCount。你可以采用缺省目录，也可以另选一个目录来存储你的整个项目。一切就绪后，请点击 OK (确定)。

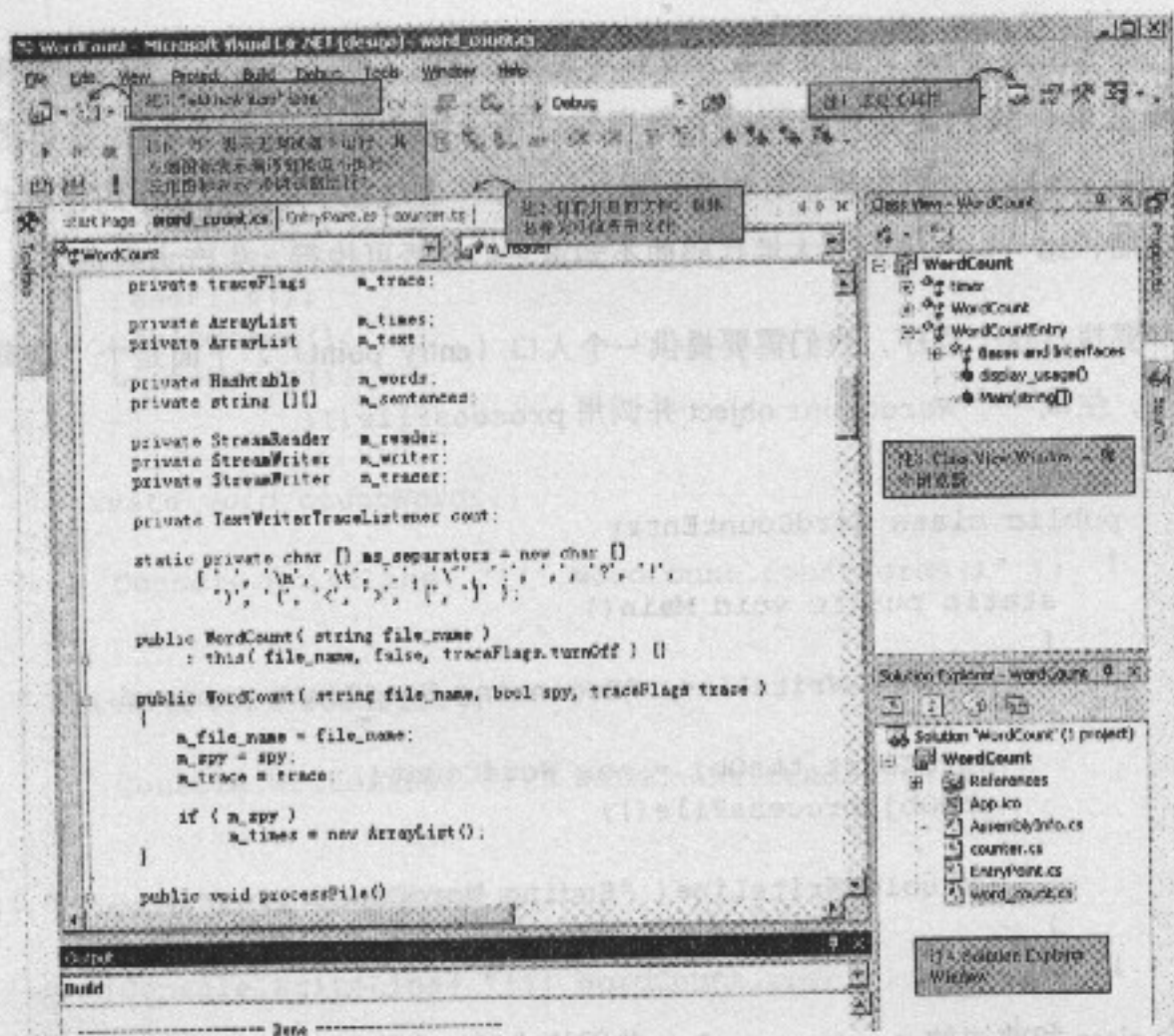


图 2.1 Visual Studio

程序文件的缺省名称是 `Class1.cs`。为它取个助忆名称如 `word_count.cs`，应该是个不错的主意。将来你肯定常常会在 Visual Studio 中为某个 project file 重新命名。你也可以在 **Solution Explorer**（解决方案资源管理器，图 2.1 右下侧标号 4 的那个窗口）中为文件重新命名。此处列出所有 project files（图 2.1 中的文件已经重新命名）。“航行”于各个 source files 之间的办法之一是：单击此窗口中的文件名称。

如果你的 **Solution Explorer** 窗口尚未打开，那么请单击代表 **Solution Explorer** 的那个“三色 Mobius 带”图标。它位于屏幕右上方的第一个工具栏处（图 2.1 标号 1 的注解处）。开启 **Solution Explorer** 窗口后，请以右键单击 `Class1.cs`，并选 **Rename**（重命名）。

Visual Studio 会为我们生成缺省的 namespace 声明和 class 骨架。我一向都删除这些东西，并以一个空文件白手起家。请在空文件中键入上一节的 `WordCount` class 定义式。注意，你一边输入程序代码，集成环境就一边分析（parse）它，甚至在编译之前就提醒你留心可能的错误。

我们还需要输入 `WordCountEntry` class，并将这一部分放入另一个独立文件。要新增一份 C# 文件，请单击 **Add New Item**（添加新项）图标（图 2.1 中窗口左上方标号 5 的注解处）。在 **Local Project Items**（本地项目项）之下选择 **C# Code File**（代码文件），而后将它命名为 `EntryPoint.cs`。做好之后，点击 **Open**。

我们通常以 **Build**（生成）命令来编译程序（图 2.1 标号 6 的两个图标中左边那个），而后修改编译报告所列出的任何错误。所有编译错误（Errors）和警告（Warnings）都经由 Visual Studio 下方的窗口呈现。如果双击其中某一条编译错误，`program text window`（程序文本窗口）便会显示发生错误的那行代码。

“已开启文件”列于 `program text window`（程序文本窗口）顶部，其下显示的是当前查看的 class，以及光标所指的 class 成员。在图 2.1 中，当前查看的 class 是 `WordCount`，目前所指的成员是 `m_reader`。

利用 **Class View**（类视图）窗口（图 2.1 标号 3 的注解处），我们可以快速切换存储于不同文件内的 classes 或 class members。此窗口展示项目中的所有 classes。每个 class 之下有其所有成员（members）的列表。单击某个 class 或某个 class member，就能使相应的程序文本显示于主窗口。

既然这个项目没有任何编译错误或链接错误，我们就来执行它。按下 **Ctrl+F5**，会在不启动调试器（debugger，也就是图 2.1 标号 6 的图标中的那个感叹号，译注：缺省情况下这个图标并不显现，需自行调出。）的情况下执行这个程序。程序运行时会弹出一个控制台窗口，显示程序中各个函数的 `WriteLine()` 输出结果。

2.3 声明数据成员 (Data Members)

数据成员所表示的是：某个 class 的实体 (instance) 的相关状态信息，例如某个文件的名称或某个 ArrayList 的容量等等。class 的数据成员可以隶属任何型别。我们应该如何确定某个 class 需要加入哪些数据成员呢？

有一种数据成员是“用户建立 class 实体时所提供的数据集”。例如我们的 WordCount class 就需要用户提供文本文件的名称和可选项（指出是否需要“追踪”或“性能计时”等功能）。我们很可能需要把这些数值存入相应的 class 数据成员中。

第二类数据成员是“供多个成员函数使用”的对象集 (objects set)。如果某个 object 只在单一成员函数内才需要，我们应该将它声明为该函数的 local object。然而如果此后其他成员函数也需要访问这一 object，我们可能需要考虑是否将它改为 class 数据成员。

例如，考虑 readFile() 成员函数的部分实现。它用了 local object 和两个 class 数据成员：

```
private void readFile()
{
    m_text = new ArrayList();
    string text_line;

    while (( text_line = m_reader.ReadLine() )
           != null )
    {
        if ( text_line.Length == 0 )
            continue;

        m_text.Add( text_line );
    }
}
```

text_line 被声明为 local object，用以临时存储从“用户所指定之文本文件”读入的每一行文本。一旦文件读取完毕，这个 object 也就结束了它的使命。

`m_reader` 和 `m_text` 被声明为 `private` 数据成员,前者指向(代表)在 `openFile()` 内建立的 `StreamReader` object,后者是个 `ArrayList` object,持有文本的非空行,会被接下来的 `countWords()` 访问。

以下是 `WordCount` 的一部分数据成员的声明:

```
public class WordCount
{
    private bool        m_spy;
    private bool        m_trace;

    private string      m_file_name;
    private string      m_file_output;

    private StreamReader m_reader;
    private StreamWriter m_writer;

    private ArrayList   m_text;
    private Hashtable    m_words;

    private string [][] m_sentences;
    // ...
}
```

每个数据成员都必须指明其访问级别 (access level)。一个未明确指出访问级别的数据成员,将被缺省视为 `private`。还记得吗, `private` 成员只能在其所处的 `class` 内部被访问。一般情况下我们总是把数据成员声明为 `private`。

2.4 Properties (属性)

译注: `property` 和 `attribute` (8.6 节) 在 C# 编程领域各有所指,而它们的中译词相近,往往令人混淆,本书大多保留不译或中英并陈,必要时将 `property` 译为“属性”,将 `attribute` 译为“特征属性”。

一个 `class` 发布给用户之后,其内部表述 (representation) 常常还会有所修改。例如在 `WordCount` 的第一个版本里, `m_trace` 声明为 `bool` 数据成员,也就是它能够被设为 `true` 或 `false`。然而对大型文本文件来说,用户会发现,生成的追踪文本 (trace text) 扑面而来——至少输出至控制台 (console) 时如此。于是用户提出要求,希望能够将“追踪输出”导向控制台或文件。要支持这种灵活性,需得改变 `m_trace` 的型别。`m_trace` 现在必须能够表示三种状态: `traceOff`, `toConsole` 和 `toFile`。

`m_trace` 若是被声明为 `public`，用户可以在他们的代码里直接任意地访问 `m_trace`。其结果是用户代码与 `class` 的实现之间出现紧耦合 (*tight coupling*) 状态——这是 `class` 设计者不易察觉的一个紧耦合。一旦 `class` 设计者改动了 `class` 的实现 (定义)，用户于是发现程序被破坏了。

所谓信息隐藏 (information hiding) 是指“用户不能访问 `class` 的实现细节”。这种机制使“用户代码”和“`class` 实现”之间保持松耦合 (*loose coupling*)，因而在“不破坏用户程序”的前提下提高了“设计者修改 `class` 实现细节”的能力。只要将 `class` 数据成员声明为 `private`，便可实施信息隐藏。

“信息隐藏”解决了紧耦合的难题，却又提出了另一道问题：如何让用户读写某个 `private` 数据成员？C# 的解决之道是在某个具名的 (named) `class` property 内提供 “get” 和 “set” 访问器 (accessors)，例如：

```
public class WordCount
{
    // private data member declaration
    private string m_file_output;

    // associated public property
    public string OutFile
    {
        get{ return m_file_output; } // Read access
        set
        { // Write access
            if ( value.Length != 0 ) // 备注: value 代表将来的 op= 右值
                m_file_output = value;
        }
    }
    // ...
}
```

通常 `property` 是 `class` 内的一个 `public` 成员，提供对此 `class` 的某个 `private` 数据成员的读取 (或涂写) 访问。欲定义一个 `property`，需指明访问级别、型别、以及 `property` 的名称。例如上述的 `OutFile` 便是型别为 `string` 的一个 `public` `property`。

如果我们想让 property 支持“读取” (read) 访问, 就为它提供一个 “get” 访问器 (也就是一个函数)。它必须返回与 property 型别相符的值。其程序代码应该置入一个语句块中。“get” 访问器无需指明返回型别或参数。最简单的情况是: get 访问器返回它所封装的那个数据成员。

如果我们想让 property 支持“涂写” (write) 访问, 就为它提供一个 “set” 访问器 (也是一个函数)。在其中, 标识符 value 的型别与 property 型别相同。运行期间 value 会被赋予“赋值操作符 (assignment operator)” 的右侧值。最简单的情况是, “set” 访问器把 value 赋值给它所封装的那个数据成员。

用户访问 property 的方式, 就像访问数据成员一样, 而不像是访问函数。例如以下片段:

```
string defaultFile = @"c:\text\wordCount.txt";

if ( theObj.OutFile == null )
    theObj.OutFile = defaultFile;
```

在 if 语句的条件式中出现的 OutFile, 将被编译器替换为 “get” 访问器。OutFile 第二次出现处 (作为赋值目标) 则被替换为 “set” 访问器, value 则被设置为 string object defaultFile。

我们可将 property 限制为只读 (read-only) 访问——只要不提供 “set” 访问器即可。

2.5 Indexers (索引器)

Indexer 用来为 class object 提供诸如 array 一般的索引功能。indexer 看上去有点像 property, 它和 property 一样提供了一对 get/set 访问器。不同的是, 这里应以关键字 this (而非其他某个标识符名称) 标明 indexer, 并且需要至少一个索引参数, 该参数可为任意型别。

举个例子, 假设用户要求以这样的下标 (subscript) 语法取回某个单词的出现次数:

```
int count = theObj[ "fiery" ];
```

我们如何支持上述操作? 下面的 indexer 定义式就可以搞定:


```

public class WordCount
{
    private Hashtable m_words;
    // 译注：通常支持 indexer 者，都内含一个支持 operator[] 的容器，如上

    // our indexer: it supports only read ...
    public int this[ string index ] // 译注：this 代表 indexer 自身
    {
        get
        {
            if ( index.Length == 0 )
                throw new ArgumentException(
                    "WordCount: Empty string as index" );

            if ( m_words == null )
                throw new Exception(
                    "WordCount: No associated file" );

            return (int) m_words[ index ];
        }
    }

    // ...
}

```

使用 indexer 时，只需把下标操作符 ([]) 直接运用到 class 的某个实体上即可。上述例子中我们并没有提供 "set" 访问器，因此如果你以下标操作符 ([]) 向一个 WordCount object 写入数据，会出现编译错误，例如：

```
theObj[ "fiery" ] = 1; // error: set not supported
```

以下的 indexer 同时支持读、写操作，它封装了一个 private array 数据成员：

```

public class Fibonacci
{
    public decimal this[ int index ]
    {
        get
        {
            check_index( index );
            return ms_elems[ index ];
        }
    }
}

```

```

    set
    {
        check_index( index );
        ms_elems[index] = value; // 译注: value 代表将来的 op= 右值
    }
}

```

```

private decimal [] ms_elems;
// 译注: 通常支持 indexer 者, 都内含一个支持 operator[] 的容器, 如上。
private void check_index( int index ) { ... }
}

```

以下例子展示 Matrix (矩阵) class 的二维 indexer。尽管此处的 indexer 以两个整数作为索引, 不过通常一个 indexer 的多个索引并非一定得属于相同型别:

```

public class Matrix
{
    // not shown: constructors, methods ...

```

```

    public int rows{ get{ return m_row; }}
    public int cols{ get{ return m_col; }}

```

```

    public double this[ int row, int col ]
    {

```

```

        get
        {
            check_bounds(row,col);
            return m_mat[row,col];
        }

```

```

        set
        {
            check_bounds(row,col);
            m_mat[row,col] = value;
        }
    }

```

```

private int m_row;
private int m_col;

```

```

private double [,] m_mat;
private void check_bounds( int r, int c ) { ... }
}

```


上述 indexer 的运用方式如下:

```
Matrix mat = new Matrix( 4, 4 );  
for ( int ix = 0; ix < mat.rows; ++ix )  
    for ( int iy = 0; iy < mat.cols; ++iy )  
        if ( mat[ix, iy] == 0 )  
            mat[ix, iy] = ix + iy;
```

2.6 成员初始化 (Member Initialization)

class 的所有数据成员都会自动初始化为其型别的缺省值。数值型别 (Numeric types) 如 int 和 double 的缺省初值为 0, bool 的缺省初值是 false, 所有 reference 型别的缺省初值是 null。缺省初始化动作 (default initialization) 是 operator new 被调用时自动执行起来的一部分动作。

如果缺省值适合作为成员的初值, 那么在初始化方面我们没什么别的事情要做。但如果要为各成员赋予缺省值以外的初值, 实际做法将取决于 class 设计者或 class 用户认为“这些成员该是什么”。

如果 class 设计者正是决定初值的人, 他可以在 class 成员声明式中明白指定初值, 例如:

```
class Login  
{  
    private string m_password = "ChangeMe";  
    private int m_max_dirs = 20;  
    private bool m_save_all = true;  
    private string m_login;  
  
    // ....  
}
```

当我们声明一个 Login 实体时, 仅有 m_login 保留其缺省值 null, 其余三个成员则被重新赋值为上述定义式中明确给出的值。这些重赋值 (reassignments) 动作将按成员的声明顺序发生。例如 m_password 会在 m_max_dirs 之前先初始化, m_save_all 则是最后才初始化。所谓“明确值”并不限于常量表达式, 例如,

```
class Login
{
    private ArrayList m_history = new ArrayList();
    private string [] m_lib_dirs = new string []
    {
        @"C:\Program Files\Microsoft.Net\FrameworkSDK\Lib\",
        @"C:\Platform SDK\Lib\",
        @"C:\MSSDK\DXF\LIB"
    };
    // ...
}
```

这一类“明确值”声明语法，允许 class 设计者指定赋予某一成员的确切值。这是“让每个 class 成员被初始化为其型别缺省值”之外的一种补充做法。至此，初始化动作尚未解决的另一个问题是，让用户在创建 class object 时有能力指定成员初值。这个功能将由 class 构造函数 (constructor) 提供，见下一节。

2.7 Class 的构造函数 (Constructor)

我们的 WordCount 程序要求用户提供文件名称。此外用户也可以选择性地开启若干选项，例如“生成追踪输出”及“计时诊断”。程序入口 Main() 必须处理命令行选项，而后创建一个 WordCount object，再把文件名称以及可能的选项传给它，然后以此 object 处理文件。

当然，有个解决办法是，按缺省值创建 WordCount object，随后依据命令行选项传人的数据重新设置 member 值。这个办法的问题在于需要两个步骤才能完成，而用户可能忽视至关重要的第二步。

class 构造函数 (constructor) 是这样一种机制：通过它，用户可以在构建 class object 时提供赋予数据成员的值。例如我们可能愿意这样创建一个 WordCount object:

```
WordCount theObj = new WordCount( file_name, spyOn, traceOn );
theObj.processFile();
```

这里的 file_name, spyOn 和 traceOn 都是在 Main() 处理命令行参数时就被设置好了。

上述调用动作对应的构造函数看起来如下所示:

```
public WordCount( string file_name, bool spy, traceFlags trace )
{
    m_file_name = file_name;
    m_spy = spy;
    m_trace = trace;

    if ( m_spy )
        m_times = new ArrayList();
}
```

构造函数是一种特殊的 class 成员函数,它与 class 同名,而且不能返回任何值,也不能声明返回型别 (return type) ——即便声明为 void 也不行。

用户通过 operator new 创建一个 WordCount object 时,其实细分为两个步骤。
第一步, new 会分配一些堆内存 (heap memory) 用以安置 object。第二步,构造函数会将该 object 初始化。构造函数由编译器自动调用,无需用户操心。

如果 class 只提供一个构造函数定义式,那么任何时候调用 operator new,都必须提供数量和型别都正确的引数 (arguments), 传给这个构造函数。对目前的 WordCount class 来说,当我们调用 operator new 时必须提供 3 个引数。更明确地说我们再也无法“不提供任何引数就创建 class object”了。因此,以下的调用将导致编译期错误:

```
WordCount theObj = new WordCount(); // error
```

这是因为我们只写出一个构造函数,而它希望我们提供三个引数给它。

身为 class 设计者,我们必须决定是否提供构造函数。如果我们决定提供,又该提供多少个呢? 也就是说,我们希望支持多少种“class object 构造方法”?

C# 允许我们定义多个构造函数,并为每个构造函数提供“独一无二”的参数列 (parameter list) ——也就是参数的个数不同或型别不同。当我们为相同名称的函数写下多份函数实体时,便是所谓“重载” (overloaded) 了这个函数。例如我们让 WordCount 的第二个构造函数仅要求一个文件名称:

```
public WordCount( string file_name )
    : this( file_name, false, traceFlags.turnOff ) {}
```

class 的某个构造函数可以调用同一个 class 的另一个构造函数, 就像上面这样。语法上三点要求:

1. 紧跟在构造函数标记式 (signature) 之后需有一个冒号 (:), 提醒编译器和读者此处将调用另一个构造函数。
2. 关键字 (keyword) `this`, 提醒编译器和读者, 此处调用的构造函数是这个 class (译注: 而非其 base class) 的一个成员。
3. 一些引数 (arguments), 用来传递给那个又被调用的构造函数。这些引数的型和数目将决定最终到底调用哪一个构造函数。

`this` 关键字所代表的那个构造函数会先被调用。本例中接受三个参数的那个构造函数会先被调用, 一旦该 (三参数) 函数运行完毕, 原先的 (单参数) 构造函数才会被运行。在这里, 由于单参数构造函数已无事可做, 所以我们写的是一个空的函数体。

以下的 `Point3D` class 是 *dispatch-to-another-instance constructor idiom* (分派至另一个构造函数之惯用手法) 的另一个示例, 其目的是允许用户在创建 `Point3D` object 时, 可以选用三个、两个或一个初值, 或甚至不给任何初值, 每个省略不写的坐标值均被赋予缺省值 0:

```
Point3D origin = new Point3D();           // Point3D(0,0,0)
Point3D x_offset = new Point3D(1.0);      // Point3D(1.0,0,0)
Point3D translate = new Point3D(1.0,1.0); // Point3D(1.0,1.0,0)
Point3D mumble = new Point3D(1.0,1.0,1.0);
```

为支持以上动作, `Point3D` 的“构造函数集”看起来像这个样子。注意, 函数的声明顺序无关紧要。

```
class Point3D
{
    public Point3D( double v1, double v2, double v3 )
        { x = v1; y = v2; z = v3; }

    public Point3D(double v1,double v2): this(v1,v2,0.0) {}
    public Point3D( double v1) : this( v1, 0.0, 0.0 ){}
    public Point3D() : this( 0.0, 0.0, 0.0 ){}

    // ...
}
```


我们也可以将构造函数声明为 `nonpublic` (亦即 `private` 或 `protected`)，`nonpublic` 构造函数对 `class` 用户来说是不可用的，然而 `class` 成员函数却可以调用它，使 `class` 能够为了内部使用目的而创建特定内容的 `object`。

C# 为 `class` 设计者提供了三种初始化方案，其中仅有一种需要引入 `class` 构造函数。`new` 表达式的副作用之一就是自动将每个数据成员初始化为其型别所对应的缺省值。我们从来不必明确地 (*explicitly*) 将数据成员设为缺省值——那是多余的。

如果有必要为某个数据成员设置不同于其缺省值的值，我们有两种选择。如果初始化动作是以用户输入为依据，我们需要借助构造函数来接受用户输入的信息，否则可以将初始化行为当做成员声明的一部分 (如 2.6 节所述)。

2.8 隐含的 (implicit) `this` Reference

到目前为止，我们的所有数据成员和成员函数都是所谓的 *instance members* (实体成员)。也就是说，在我们创建的每个 `class object` (每一个实际个体) 中，都存储有每一个 “*instance* 数据成员” 的复本。至于所谓 “*instance* 成员函数”，意味着必须通过某个 `class object` 才能将它调用。

译注：本节所谓 *instance* 意指从 `class` 声明和定义中做出的一个实际物体，有别于只是 “声明+定义” 的 `class`。在这个意义上，*instance members* (*non-static members*) 的对照是 *class members* (也就是 *static members*)。

当我们在成员函数中访问 “*instance* 数据成员” 时，后者的名称扮演占位符 (*placeholder*) 的角色。是的，成员函数并不为它所访问的每一个 “`class` 数据成员” 都维护一份复制品。当通过一个真实的 `class object` 调用这一成员函数时，函数中出现的数据成员名称将被绑定 (*bound*) 至 `class object` 的相应实体上，这一绑定工作乃是通过隐含的 *this reference* 才得以完成。

在 “*instance* 成员函数” 之内，*this reference* 代表 (指向) 一个 `class object`，而成员函数正是被该 `object` 调用的。在编译器内部，*this reference* 的作用是将 “使用数据成员的操作行为” 绑定到 `class object` 内的相应数据实体上。考虑如下代码：

```
private void countWords()
{
    m_sentences = new string[ m_text.Count ][];
    m_words     = new Hashtable();
    string str;
```

```
for( int ix = 0; ix < m_text.Count; ++ix ) { ... }  
}
```

编译器对于“instance 数据成员”的每一次“未受特殊限制的访问动作 (unqualified access)”都会加以扩展, 使 this reference 指向该成员, 例如:

```
this.m_sentences = new string[ this.m_text.Count ][];  
this.m_words      = new Hashtable();  
  
for( int ix = 0; ix < this.m_text.Count; ++ix ) { ... }
```

我们通过“在每个 instance 成员函数的参数列中增加额外的 class 参数”把 this reference 传入这些函数:

```
private void countWords( WordCount this ){ ... }
```

这意味着“成员函数的调用动作”必须被编译器重写, 以便反映其内部的 (真实的) 标记式 (signature)。例如我们写下:

```
theObj.processFile();
```

将被编译器转换为:

```
processFile( theObj );
```

如果对某个“instance 成员函数”进行“未受特殊资格限制的访问动作 (unqualified invocation)”, 例如:

```
public void processFile(){ countWords(); }
```

编译器首先会在函数调用动作之前添加 this reference:

```
public void processFile( WordCount this )  
{ this.countWords(); }
```

随后再重写, 以反应出每个函数的扩充参数:

```
public void processFile( WordCount this )  
{ countWords( this ); }
```

在程序的某些场合中, 访问 this reference 能解决某些棘手的难题。例如, 设想我们需要实现一个字符串双向链表 (doubly-linked list of strings)。我们设计一个 class StringNode, 包含 3 个数据成员:

上述调用动作对应的构造函数看起来如下所示：

```
public WordCount( string file_name, bool spy, traceFlags trace )
{
    m_file_name = file_name;
    m_spy = spy;
    m_trace = trace;

    if ( m_spy )
        m_times = new ArrayList();
}
```

构造函数是一种特殊的 class 成员函数，它与 class 同名，而且不能返回任何值，也不能声明返回型别（return type）——即便声明为 void 也不行。

用户通过 operator new 创建一个 WordCount object 时，其实细分为两个步骤。第一步，new 会分配一些堆内存（heap memory）用以安置 object，第二步，构造函数会将该 object 初始化，构造函数由编译器自动调用，无需用户操心。

如果 class 只提供一个构造函数定义式，那么任何时候调用 operator new，都必须提供数量和型别都正确的引数（arguments），传给这个构造函数。对目前的 WordCount class 来说，当我们调用 operator new 时必须提供 3 个引数。更明确地说我们再也无法“不提供任何引数就创建 class object”了。因此，以下的调用将导致编译期错误：

```
WordCount theObj = new WordCount(); // error
```

这是因为我们只写出一个构造函数，而它希望我们提供三个引数给它。

身为 class 设计者，我们必须决定是否提供构造函数。如果我们决定提供，又该提供多少个呢？也就是说，我们希望支持多少种“class object 构造方法”？

C# 允许我们定义多个构造函数，并为每个构造函数提供“独一无二”的参数列（parameter list）——也就是参数的个数不同或型别不同。当我们为相同名称的函数写下多份函数实体时，便是所谓“重载”（overloaded）了这个函数。例如我们让 WordCount 的第二个构造函数仅要求一个文件名称：

```

class StringNode
{
    private StringNode back_link;
    private StringNode front_link;
    private string      text;

    public StringNode( string str ){ text = str; }
    // ...
}

```

我们需要提供一个 Append() 操作，支持以下用法：

```

StringNode node = new StringNode( "a node" );
// ...
node.Append( new StringNode( "also a node" ) );

```

添加 (append) 一个 StringNode object 的动作分为三个步骤：(1) 新节点 (node) 的 front_link 设为既有节点的 front_link；(2) 既有节点的 front_link 设为新节点；(3) 新节点的 back_link 设为既有节点：

```

public void Append( StringNode new_node )
{
    new_node.front_link = front_link;    // (1)
    front_link = new_node;              // (2)
    new_node.back_link = ?????          // (3)
}
// 译注：以上动作虽名 Append，似乎更像 Insert

```

啊呀，要是没有 this reference，我们就没有办法直接引用代表既有节点的那个 object —— 我们正是通过该 object 调用 Append()。要完成任务，只需将 this 赋值给 back_link：

```

new_node.back_link = this;    // (3)

```

如果你喜欢，你也可以在访问 “instance 成员” 时直接加上 this reference 前缀。有些人主张增加 this 前缀会使代码更具可读性，主要是基于以下想法：这样做能清楚指明函数中出现的 “instance 成员”。但就我个人而言，函数中出现 this reference 会为我带来一种 “代码混乱” 的感觉⁸。

⁸ Visual Studio 代码生成向导 (code generation wizards) 就十分机械式地在代码中插入 this。

2.9 static (静态) 成员

Class 的成员并不一定都是“instance 成员”。有些成员提供服务或者包含信息，而这些服务和信息独立于具体的“class object 实体”。例如某个整数值是“instance 成员”，而“整数可以表示的最大值”却不应该是“instance 成员”。是的，这个最大值对所有整数 objects 而言都应该是一样的。

我们并不想让每个整数 object 都持有一份“整数所能表示的最大值”。但是，整数型别的用户可能会希望获取这份信息，以便和某个整数 object 作比较。

换言之，“整数的最大值”作为“整数的 class 成员”而存在，才有意义（译注：也就是说它属于整个 class type，而非属于特定某个 class object），如果把它视为这个 class 的“instance 成员”，没有任何意义。我们将这种所谓的“class 成员”声明为 static 或 const。

static 成员扮演着 class 成员（而非 instance 成员）的角色。例如，Today 和 Now 是 DateTime class 的 static 属性（properties）。Today 表示当前日期（时间部分设为 0）；Now 表示当前的日期和时间。成员函数 WriteLine() 和 ReadLine() 是 Console class 的 static 成员函数。MaxValue 是 Int32 class 的 static 数据成员。

在我们的 WordCount class 中，用来将一整行文本分割为一个个单词的“separators 字符数组”就适宜作为一个 static 成员。声明方法如下：

```
public class WordCount
{
    static public char [] ms_separators;
```

static 数据成员在整个程序中只存在唯一一份实体。它独立于个别的 class objects 而存在。在 class 成员函数中，访问 static 成员的语法，和访问 instance 成员的语法完全一致。例如：

```
private void countWords()
{
    // ...
    for( int ix = 0; ix < m_text.Count; ++ix )
    {
        str = ( string )m_text[ ix ];
        m_sentences[ ix ] = str.Split( ms_separators );
```

如果在 class 本体之外，我们可以通过 static 成员所隶属的 class 名称来直接访

问它（不必经由成员函数），例如：

```
static void Main()
{
    char bang = '!';
    int ix = 0;

    for( ; ix < WordCount.ms_separators.Length; ++ix )
        if ( WordCount.ms_separators[ ix ] == bang )
            break;

    if ( ix == WordCount.ms_separators.Length )
        throw new Exception( "Insufficient separators" );
}
```

我们不能通过 class 的一份 object 来访问 static 成员，这样做会触发错误，例如，

```
int len = theObj.ms_separators.Length; // error
```

一个 static 成员也会被初始化为其型别所对应的缺省值。在我们的例子中，由于数组是一种 reference 型别，所以 ms_separators 被设初值为 null。当然我们也可以声明式中明白地为它指明初值，像这样：

```
public class WordCount
{
    static public char [] ms_separators = new char []
    {
        ' ', '\n', '\t', '.', '\'', ',', ';', ':',
        '?', '!', '}', '(', '<', '>', '[', ']' };
}
```

另一个办法是在 static 构造函数中初始化 class 的 static 成员，例如：

```
public struct WordCount
{
    static public char [] ms_separators;
    static WordCount() // static constructor
    {
        ms_separators = new char []
        {
            ' ', '\n', '\t', '.', '\'', ',', ';', ':',
            '?', '!', '}', '(', '<', '>', '[', ']' };
    }

    // ...
}
```


每一个 class 至多只可定义一个 static 构造函数。其做法很简单，只需在构造函数名称之前加上 static 关键字。缺省情况下 static 构造函数的访问级别为 public；但如果你明白写出访问级别（哪怕是写 public）将导致错误。static 构造函数的参数列必须是空的，这就是为什么我们只能定义唯一一个 static 构造函数的原因——如果我们想要重载某个函数，每个重载函数必须有独一无二的标记式（signature）。

Static 构造函数仅在“此 class 的某个实体被创建”或“此 class 的某个 static 成员被取用”的条件下，才会被调用（而且只被调用一次）。程序中调用 static 构造函数的准确顺序是不可预知的（未有定义的）。

在 static 构造函数中，你只能够初始化 class 的 static 数据成员，或是调用 static 成员函数。如果尝试访问 instance 成员，会导致错误。然而 static 构造函数的功能并不只限于初始化 static 成员，它可以用来在“创建 class 的第一个 object，或第一次访问某个 class static 成员”之前运行任何动作。8.5 节有一个颇为有趣的 static 构造函数运用范例。

通常情况下我们把 static 数据成员声明为 private，然后通过 static property 提供读写访问。我们也可以将成员函数声明为 static。static 成员函数由于缺少 this reference，因此不能直接访问它所属的 class 的任何 instance 成员。

2.10 const 和 readonly 数据成员

const 数据成员的声明式必须包含初值，而且这个初值必须是个常量表达式（constant expression），也就是编译期就能完全评估的表达式。以下程序片段：

```
class Matrix {  
    private const int ms_default_row_size = 4,  
                    ms_default_col_size = 4;  
}
```

声明了两个 const int 成员，并初始化为 4。如果你尝试修改 const 成员，会产生编译期错误。

const 成员总是可以使用另一个 const 成员来初始化，前提是两者之间没有循环相依性（circular dependency）。例如以下三个 const 声明式能够通过编译，并把 x 设为 10，y 设为 8，z 设为 4：

```
class Illustrate  
{
```

```
// OK: no circular dependency

private const int x = y + z/2;
private const int y = z * 2;
private const int z = 4;
}
```

下面的例子将导致编译期错误，因为 x 和 z 的定义式之间有循环相依性：

```
class Illustrate
{
    // error: circular dependency

    private const int x = y + z/2;
    private const int y = z * 2;
    private const int z = x;
}
```

`const` 数据成员并非一种 *instance* 成员；`const` 成员仅存在唯一一份实体。`const` 成员本质上就是一种只读的 (*read-only*) *static* 成员：访问时必须使用 *class* 名称，而非 *object* 名称。

`const` 成员不可能隶属于 *reference* 型别。花数秒钟想想原因。是的，*reference* 型别需要调用 `operator new`，而后者只能在运行期 (*runtime*) 评估求值。我们必须拿能够在编译期评估的表达式来初始化 `const` 成员。以下声明式就是非法的，会产生编译期错误：

```
// illegal: cannot be evaluated at compile time
public const Matrix identity =
    new Matrix( 1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1 );
```

另一个关键字 `readonly` 使我们得以在确保“只读访问”的前提下，把 *object* 的初始化动作推迟到运行期来进行。如果尝试修改 `readonly` 成员，会产生编译期错误。如果要模拟 `const` 成员，我们可以将 `readonly` 成员声明为 *static*（译注：请注意 `readonly` 和 *static readonly* 之间的区别。`readonly` 成员是一种 *instance* 成员）。例如：

```
public static readonly Matrix identity =
    new Matrix( 1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1 );
```


唯一一个“可声明为 const”的 reference 型别是 string，也许你还记得，string 的初始化可以不借助 operator new，像这样：

```
class Illustrate {  
    // OK: string reference type is an exception  
    private const string default_login = "Guest";  
    private const string default_pswrd = "ChangeMe";  
}
```

const 成员和 readonly 成员之间最主要的区别是初始化时机的不同（编译期 vs. 运行期）。由于 const 成员的值在编译期就能知道，所以在其名称出现处可以进行常量折叠操作（constant-folded，亦即“取代”之意）。这对 readonly 成员来说是不可能的。

2.11 enum (枚举) value 型别

编写程序时，我们常需要为某个 object 定义一组与之相关的可选用的属性（特性）。例如，文件的开启状态可能是以下三种之一：input、output 和 append。要让这些状态值明了化，办法之一是给每个状态赋予一个独一无二的常量，因此你可以这样写：

```
public class FileMode  
{  
    public const int input = 1;  
    public const int output = 2;  
    public const int append = 3;  
    // ...  
}
```

然后这样使用它们：

```
bool open_file( string file_name, int open_mode);  
// ...  
open_file( "Phoenix_and_the_Crane", FileMode.append );
```

这样做的好处是，使用“便于记忆的常量型 object”远比“记住并搞清楚每个相应的字面数值”来得容易，缺点则是无法将输入值限制为 input、output 和 append 三者之一。例如编译器不会拒绝你将 1024 这个值传给 open_file()。因此，确保“传入值是个有效值”的担子就落在程序员身上。

所谓 `enum` 型别，定义了一组互有关联的具名 (named) 整数常量。带有 `enum` 型别的 `objects` 只能被赋值为那些具名常量的其中之一。定义 `enum` 型别时应写出关键字 `enum` 再加上一个标识符。置于花括弧中并以逗号隔开的那些具名的枚举元 (enumerators)，用来确定 `enum` 型别的每一个值。每个枚举元的名称必须独一无二，多个枚举元可以代表同一个值。

缺省情况下，第一个枚举元被赋值为 0。其后每个枚举元的值自动增加 1。如果要改写缺省规则，只需为枚举元赋予一个整数常量表达式，例如：

```
enum open_modes{
    input = 1, output, append,
    last_input_mode = input, last_output_mode = append ;}
```

在 `enum` 定义式内，枚举元可以不经任何资格修饰地使用。但是在 `enum` 定义式之外，枚举元必须以其所属之 `enum` 型别的名称加以修饰，例如 `open_modes.input`。嵌套的 `enum` 则必须以其所属的 `class` 名称再加一层外围修饰，例如：
`MyFileClass.open_modes.input`。

如果坚持要将“整数常量表达式”赋予 `enum object`，你必须将整数明确转为 `enum` 型别才可以：

```
void open_file( string file_name, open_modes om );

open_modes om;
string fname;
// ... set om and fname ...

open_file( fname, om );           // OK
open_file( fname, open_modes.append ); // OK
open_file( fname, 1 );           // error!
open_file( fname, (open_modes)1 ); // OK, but ...
```

但是这种做法会忽略针对此值的一切型别检查，因此并不值得推荐。

我们可以使用 `enum` 型别来表示 `WordCount class` 所可能存在的各种追踪 (trace) 模式，例如：

```
public class WordCount
{
    public enum traceFlags { turnOff, toConsole, toFile };
    private traceFlags m_trace;
```


根据命令行参数, 程序入口 `Main()` 将采用三个值中的某一个, 用以初始化

`m_trace`:

```
static public void Main( string [] args )
{
    WordCount.traceFlags traceOn =
        WordCount.traceFlags.turnOff;
    // ...

    foreach ( string option in args )
        switch ( option )
        {
            case "-t":
                traceOn = WordCount.traceFlags.toConsole;
                break;

            case "-tf":
                traceOn = WordCount.traceFlagsToFile;
                break;

            // ...
        }
}
```

很自然地, 枚举元 (Enumerators) 会被联想到适合做为 `switch` 语句的 `case` 标签。例如在 `WordCount` 的 `openFile()` 中, 当我们创建 `trace object` 时, 便是依据 `m_trace` 来确定究竟绑定到控制台或是某个文件:

```
if( m_trace != traceFlags.turnOff )
    switch ( m_trace )
    {
        case traceFlags.toConsole:
            cout = new TextWriterTraceListener(Console.Out);
            Trace.Listeners.Add( cout );
            break;

        case traceFlagsToFile:
            m_tracer = File.CreateText(m_diag_file);
            cout = new TextWriterTraceListener(m_tracer);
            Trace.Listeners.Add( cout );
            break;
    }
}
```

缺省情况下, 每个枚举元均以 `int` 型别来表现。我们也可以指定其他整数型别作为枚举元的载体。前提是我们指定的型别能够表现所有枚举元的值。例如:

```
public enum weekdays : byte
{
    sunday, monday, tuesday, wednesday,
    thursday, friday, saturday
};
```

我们可以用 increment 或 decrement (递增或递减) 操作符来遍历枚举元(s), 例如:

```
public static void translator( string [] foreign )
{
    weekdays wd = weekdays.sunday;

    for ( ; wd <= weekdays.saturday; ++wd )
        Console.WriteLine( wd + " : " + foreign[(int)wd] );
}
```

此处 foreign 代表某种外语中每周七天的名称字符串数组。translator() 将一周中的每天名称以两种语言打印出来 (一个是英语, 一个是以 foreign 数组表示的外语)。请注意, 数组的索引号必须由 wd enum object 显式转型为 int。这段代码编译并执行后, 会生成类似这样的输出:

```
sunday : dimanche
monday : lundi
tuesday : mardi
wednesday : mercredi
thursday : jeudi
friday : vendredi
saturday : samedi
```

.NET class framework 运用了大量 enum 型别来封装 class properties (属性) 所需的一些二择一或多择一的模式 (modes) 或属性 (特性, attributes), 例如 BorderStyle, TextBoxMode, FontSize.

2.12 delegate 型别

delegate 型别会创建出一种所谓的 function object (函数对象, 译注: 这在 C++ 是一种对 operator() 进行重载的 class, 也称为 functor)。delegate 型别看起来很像函数声明式, 但实际上它所定义的类型用来指向一个或多个“具有特定标记式 (signature) 和返回型别”的函数。delegate 型别有三个主要特征:

1. `delegate object` 可以同时代表（指向）多个成员函数。当程序调用一个指向多个成员函数的 `delegate` 时，这些函数会按它们被赋给这一 `delegate object` 的先后顺序而被逐一调用。我们很快就能看到具体怎么做。
2. 被 `delegate object` 代表（指向）的多个函数并不一定是同一个 `class` 所属的成员函数。一个 `delegate object` 代表的所有函数必须有着相同原型（`prototype`）和标记（`signature`）。然而这些函数既可以是 `static` 函数也可以是 `non-static` 函数，而且它们可以隶属单一 `class` 或隶属多个不同的 `classes`。
3. 声明一个 `delegate` 型别，就等于在内部创建了“.NET library framework 的 `Delegate` 或 `MulticastDelegate` 抽象基类”的一个新子类（`subtype`）实体，这两个抽象基类提供一套 `public` 成员函数，可支持“查询 `delegate object` 及其所指函数”。

`delegate` 型别的声明式通常由四部分组成：(1) 访问级别（`access level`）；(2) 关键字 `delegate`；(3) 这一 `delegate` 型别所要代表的函数的返回型别和标记式（`signature`）；(4) `delegate` 型别名称——置于函数返回型别和标记式之间。

举个例子，以下将 `Action` 声明为一个 `public delegate` 型别，用以代表“无参数，返回型别为 `void`”的函数：

```
public delegate void Action();
```

如果某个 `delegate` 型别被用来在任一时刻只代表单个函数，那么它可以指向“返回型别和标记式（`signature`）均不受限”的成员函数。但如果某个 `delegate` 型别被用来同时代表两个或多个函数，其返回型别就必须是 `void`。为此，上述的 `Action` 可以用来代表一个或多个函数。

让我再举个例子。考虑一下 `testHarness class` 的设计，它必须允许任何 `class` 注册单个或多个 `static/nonstatic` 成员函数，以便稍后加以执行。这时候 `delegate` 型别是实现的核心关键。

在这个例子中，不妨将 `delegate object` 声明为 `testHarness class` 的 `private static` 成员，例如：

```

public delegate void Action();           // 译注: 这是个 delegate type
public class testHarness
{
    static private Action theAction; // 译注: 这是个 delegate object
    static public Action Tester      // 译注: 这是个 property,
    {                               // 传回或设置 Action object.
        get{ return theAction; }
        set{ theAction = value; } // 译注: value 代表未来的 op=右值
    }

    // ...
}

```

C# 中的 delegate 型别是一种 reference 型别。因此其声明式 (例如):

```
Action theAction;
```

意思是说, theAction 是个 handle, 指向 Action delegate 型别所生成的某个 object. theAction 自身并不是个 delegate object. 缺省情况下这个 handle 被设为 null; 如果为它赋值之前就尝试使用它, 会产生编译期错误。例如以下语句将造成被 theAction 代表的那些个函数(s) 被调用:

```
theAction();
```

但除非在 theAction 定义式和上述语句之间曾经对 theAction 进行赋值动作, 否则上述语句将触发编译期错误消息。

如果你希望让 theAction 代表某个 class 成员函数, 你必须以 new 创建一份 Action delegate 型别实体. 如果代表的对象是个 static 成员函数, 你可以为 Action 构造函数指定一个这样的引数: class 名称 + dot (.) 操作符 + static 成员函数名称, 像这样:

```

theAction = new Action( Announce.announceDate );
// 译注: Announce.announceDate 出现在下一页

```

但如果你的代表对象是个 nonstatic 成员函数, 你可以为 Action 构造函数指定一个这样的引数: object 名称 + dot (.) 操作符 + nonstatic 成员函数名称, 像这样:

```

Announce an = new Announce();
theAction   = new Action( an.announceTime );
// 译注: announceTime 出现在下一页

```


Announce class 的 static 成员函数 announceDate() 的作用是对着标准输出设备以长格式 (long form) 打印当前日期, 如下:

```
Monday, February 26, 2001
```

nonstatic 成员函数 announceTime() 则是对着标准输出设备以短格式 (short form) 打印当前时间, 例如 00:58, 前两位数代表小时, 从午夜 00 开始, 后两位数代表分钟。这个 class 的定义用到了 .NET class framework 提供的 DateTime class (5.5.4 节将有更多细节)。

```
public class Announce
{
    public static void announceDate()
    {
        DateTime dt = DateTime.Now;
        Console.WriteLine( "Today's date is {0}",
                           dt.ToLongDateString() );
    }

    public void announceTime()
    {
        DateTime dt = DateTime.Now;
        Console.WriteLine( "The current time is {0}",
                           dt.ToShortTimeString() );
    }
}
```

正如稍早所见, 在 delegate object 之后运用 call 操作符 (()), 就能调用此 delegate object 所代表的成员函数:

```
testHarness.Tester();
```

上一行语句中, 首先调用 Tester property 的 get 函数, 返回 theAction delegate handle. 接下来 call 操作符 (()) 施行于此 handle 之上, 导致调用这个 delegate object 所代表的成员函数。但如果此时 theAction 并未代表任何 delegate object (译注: 也就是说没有代表任何成员函数), 会有异常 (exception) 被抛出来。如果要防止抛出异常, 可使用典型的“delegate 测试并运行”代码序列。在 testHarness class 之外, 这一代码序列长成这般模样:

```
if ( testHarness.Tester != null )
    testHarness.Tester();
```

在 class 之内，这一代码序列形式如下：

```
static public void run()
{
    if ( theAction != null )
        Action();    // 译注：恐为 theAction()
}
```

为了让 delegate 代表多个成员函数，我们主要运用 += 和 -= 操作符操控之。举个例子，假设我们定义了一个 testHashtable class，并决定在其 static 构造函数中将每个相关测试添加至 testHarness object（译注：p.88）：

```
public class testHashtable
{
    public void test0();
    public void test1();
    static testHashtable()
    {
        testHarness.Tester += new testHarness.Action( test0 );
        testHarness.Tester += new testHarness.Action( test1 );
        // 译注：上式有误，new 之后不应有 testHarness。
    }
    // ...
}
```

类似情况，我们再定义一个 testArrayList class，并决定在其 static 构造函数中将每个相关测试添加至 testHarness object（译注：p.88）。请注意这里用到的成员函数都是 static：

```
public class testArrayList
{
    static public void testCapacity();
    static public void testSearch();
    static public void testSort();
    static testArrayList()
    {
        testHarness.Tester += new testHarness.Action(testCapacity);
        testHarness.Tester += new testHarness.Action(testSearch);
        testHarness.Tester += new testHarness.Action(testSort);
        // 译注：上式有误，new 之后不应有 testHarness。
    }
    // ...
}
```


这些成员函数的调用顺序如何？是的，它们将依照加入 delegate 的先后顺序被依次调用。以本例而言，testHashtable 的 test0 总是在 test1 之前被调用，但由于我们通常无法确知“static 构造函数”被调用的确切时间（只知道它在 class 被使用之前一定会被调用），因此我们无法准确说出 testArrayList 的成员函数或 testHashtable 的成员函数何者先被加入 Tester。

现在，考虑以下局部块（local block）内的代码序列：

```
{
    Announce annc = new Announce();
    testHarness.Tester +=
        new testHarness.Action( annc.announceTime );
    // 译注：上式有误，new 之后不应有 testHarness。
}
```

如果为 delegate object 加入 nonstatic 成员函数，那么该函数的地址，以及“藉以调用该函数”的那个 class object（译注：上例中的 annc）的 handle 都会被存储起来。造成的结果是：该 class object 的引用计数会被累加 1。

当我们以 new 初始化 annc 时，位于 managed heap（受控堆）上的 object 引用计数（reference count）将被初始化为 1。一旦我们把 annc 传给 delegate object 的构造函数，Announce object 的引用计数就相应地增为 2。随着上述局部块（local block）的结束，annc 生命终止，其相应的引用计数又减回 1。

好消息是，我们不必担心 delegate object 所代表的成员函数所属的 object 会失效。直到 delegate object 不再代表那个成员函数，此 object 才可能会被垃圾回收。也不必担心 object 在我们不注意时突然消失了。坏消息是，此 object 会持续存在，直到 delegate object 不再引用该 object 的成员函数。

-- 操作符被用来移除 delegate 所代表的成员函数。例如以下这个局部块，首先设置 delegate object，然后运行之，然后再将 announceTime() 从该 delegate object 移除：

```
{
    Announce an = new Announce();
    Action act = new testHarness.Action( an.announceTime );
    // 译注：上式有误，new 之后不应有 testHarness。

    testHarness.Tester -= act;
}
```

```

    testHarness.run();           // 译注：定义式见 p.90
    testHarness.Tester -= act;
}

```

另一种实现方案是，先检查 `Tester` 是否已经代表一个或若干个成员函数。如果是，就把当前的 `delegation list` 保存起来，再将 `Tester` 重置为 `act`，然后调用 `run()`，然后再将 `Tester` 置换回原来的 `delegation list`。

如果想要查询 `delegate` 所代表的函数个数，可利用底层的 `Delegate class interface`，例如：

```

if ( testHarness.Tester != null &&
    testHarness.Tester.GetInvocationList().Length != 0 )
{
    Action oldAct = testHarness.Tester;

    testHarness.Tester = act;
    testHarness.run();
    testHarness.Tester = oldAct;
}
else { ... }

```

`GetInvocationList()` 会返回一个由 `Delegate class objects` 组成的数组，数组内的每个元素代表此 `delegate object` 当前代表的某个成员函数。`Length` 是底层的 `Array class` 的一个属性（property），C# 内建的数组型别（array type）也是以 `Array class` 实现出来的。

2.13 函数参数语义学（Function Parameter Semantics）

函数的参数可当作函数体内的占位符（placeholders）。每当外界调用此函数，其参数（**parameters**，又称形式参数，形参）就被绑定到实际传给该函数的引数（**arguments**，又称实际参数，实参）身上。缺省情况下，这一绑定动作以 *by value* 方式实现。但我们可以改变缺省方式，只要以关键字 `ref` 或 `out` 来修饰参数即可。这些以 `ref` 或 `out` 加以修饰的参数，都将以 *by reference* 方式绑定到引数身上。

每个参数最少由一个型别指示符（例如 `int`、`string`、`Matrix`）和一个名称组成。参数名称只在函数内可见（有效），所以相同的名称可以在函数外被重复使用，不至于引发冲突。如果有多个参数，应以逗号隔开。例如我们可以这样写：

```
f( int i, int j ){ ... }
```

但不能写成这样：


```
// error!  
f( int i, j ){ ... }
```

如果你的参数列是空的，表示该函数不接受任何参数。

让我们检视下面这个 static 成员函数，它接受两个参数：一个是 Matrix object，一个是 double 值。该函数的作用是将第一参数（代表一个矩阵）的每个矩阵元素乘以第二参数，并将结果存入另一个 Matrix object，最后返回新的 Matrix object（译注：简言之就是实现矩阵的数乘运算）：

```
public static  
Matrix multiplyByDouble( Matrix mat, double dval )  
{  
    Matrix result = new Matrix( mat.rows, mat.cols );  
  
    for ( int ix = 0; ix < mat.rows; ix++ )  
        for ( int iy = 0; iy < mat.cols; iy++ )  
            result[ ix, iy ] = mat[ ix, iy ] * dval;  
  
    return result;  
}
```

这个函数可以这样被调用：

```
Matrix mat = Matrix.multiplyByDouble( location, modval );
```

这里的 location 代表一个 Matrix object，modval 代表一个 double object，两者都必须在程序中事先定义好。

形式参数（上例的 mat 和 dval）和实际引数（上例的 location 和 modval）之间有什么关系？每个形式参数代表函数内的一个 local object，与函数体内定义的 objects 地位上没什么区别。因此我们可以说，上述的 multiplyByDouble() 定义了 5 个 local objects：2 个是 local Matrix objects (mat 和 result)，1 个是第二参数 dval，还有 2 个是整数 objects ix 和 iy（做为矩阵索引）。

每个 local object 在使用前都必须先初始化（至少要赋予某值），就 result, ix 和 iy 来说，初始化动作很明显。但参数呢？参数何时被初始化，如何被初始化？

2.13.1 传值 (Pass by Value)

参数 (parameter) 会在函数的每一个调用点 (call point) 处被初始化。缺省情况下, 参数通过所谓的 *pass by value* 机制完成初始化, 也就是说, 每个参数值将成为实际引数 (actual argument) 值的一份复本。例如一旦调用前述函数, `dval` 就被初始化为 `modval` 所持有值的一份复本。既然 `dval` 和 `modval` 代表各自的值, 因此函数中对 `dval` 所做的任何改动, 都不会影响 `modval`。这带来好的一面, 但也有坏的一面。

好的一面是, 我们不必提防实际引数不经意地被修改了。在函数中, 我们操作的是引数的一份复本 (而非引数自身)。对 `local object` 所做的任何修改, 都将随着函数的退出而消失。实际引数一点也不为所动。

如果对 `local object` 所做的改动, 有必要反映到调用此函数时的实际引数身上, 情况就不太妙了。这种情形下我们得舍弃 *pass-by-value* 机制。下一小节会展示做法。

如果形式参数 (formal parameter) 是一种 *reference* 型别, *pass-by-value* 机制就让人稍微有点犯糊涂。当我们以 *by value* 方式传入 *reference* 型别时, 一个独立的 `local` 实体会被创建出来, 就像传入 *value* 型别所做的那样。不同之处是, 引数和 `local` 实体都指向 (代表) `heap` 上的同一个 `object`。

还记得吗, *reference* 型别由 "handle/object" pair 组成。其中 `handle` 保存着它所指的 `object` 的地址。 `object` 本身存在于 `managed heap` (受控堆) 之中。当我们以 *by value* 方式传递 *reference* 型别时, 其 `handle` 部分被复制到 `local` 实体。这么一来, 原本的 `handle` 和新的 `local object` 两者就都指向位于 `heap` 上的同一个 `object`。前例之中, `localtion` 和 `mat` 两者都指向 `heap` 上的同一个 `Matrix object`。

这意味着通过 `local` 实体对 `heap object` 所做的改动, 将永久 (真正) 改变那个 `object`。然而对 *reference* 型别的 "handle 部分" 的改动却不是这样, 这样的改动只影响 `local` 实体, 一旦函数运行完毕, 也就被丢弃了。

想要弄清楚这一点, 请看一个实例。

函数 `byValue()` 以 *by value* 方式接受单一 `string` 参数, 并在函数内运用

`ToUpper()` 将字符串内的所有字符转为大写。由于 `string` 是不可变的 (immutable)，所以 `ToUpper()` 返回一个新的 `string` object (译注：而不是直接在原 `string` 上头修改)。而后，参数 `s` 被改指向新的 (被 `ToUpper()` 返回的) 那个 `string` object。此时，实际引数和 `local` object 已经分道扬镳，分别指向不同的 `string` objects：

```
static public void byValue( string s )
{
    Console.Write( "\nInside byValue: " );
    Console.WriteLine( "original parameter: \n\t" + s );

    // now refers to a different string object!
    s = s.ToUpper();

    Console.Write( "\nInside byValue: " );
    Console.WriteLine( "modified parameter: \n\t" + s );
}
```

`local` 实体指向新的 `string` object，其内字母已转为大写。但由于我们修改了身属 `value` 型别的参数 `handle`，所以上述改动不会反映到实际引数身上。从程序的输出就能看出来端倪：

```
string to be passed by value:
  A fine and private place
Inside byValue: original parameter:
  A fine and private place
Inside byValue: modified parameter:
  A FINE AND PRIVATE PLACE
back from call -- string:
  A fine and private place
```

正如你所见，`string` object 在调用 `byValue()` 前后，是同一个样子。这种情况下，我们第一反映是：哎呀，“字符串转换”搞砸了。然而从 `byValue()` 之内显示的 `string` object 却能清楚看出，转换功能运作良好。噢，问题不在于函数本身，而在于如何将参数传入和传出！下一节我将示范如何改变这种缺省的 `pass-by-value` 行为。

在此之前，我想扼要回顾一下作用于 `reference` 型别身上的浅拷贝 (shallow copy) 效果。考虑以下代码：

`ToUpper()` 将字符串内的所有字符转为大写。由于 `string` 是不可变的 (immutable)，所以 `ToUpper()` 返回一个新的 `string` object (译注：而不是直接在原 `string` 上头修改)。而后，参数 `s` 被改指向新的 (被 `ToUpper()` 返回的) 那个 `string` object。此时，实际引数和 `local` object 已经分道扬镳，分别指向不同的 `string` objects:

```
static public void byValue( string s )
{
    Console.Write( "\nInside byValue: " );
    Console.WriteLine( "original parameter: \n\t" + s );

    // now refers to a different string object!
    s = s.ToUpper();

    Console.Write( "\nInside byValue: " );
    Console.WriteLine( "modified parameter: \n\t" + s );
}
```

`local` 实体指向新的 `string` object，其内字母已转为大写。但由于我们修改了身属 `value` 型别的参数 `handle`，所以上述改动不会反映到实际引数身上。从程序的输出就能看出来端倪：

```
string to be passed by value:
    A fine and private place
Inside byValue: original parameter:
    A fine and private place
Inside byValue: modified parameter:
    A FINE AND PRIVATE PLACE
back from call -- string:
    A fine and private place
```

正如你所见，`string` object 在调用 `byValue()` 前后，是同一个样子。这种情况下，我们第一反映是：哎呀，“字符串转换”搞砸了。然而从 `byValue()` 之内显示的 `string` object 却能清楚看出，转换功能运作良好。噢，问题不在于函数本身，而在于如何将参数传入和传出！下一节我将示范如何改变这种缺省的 `pass-by-value` 行为。

在此之前，我想扼要回顾一下作用于 `reference` 型别身上的浅拷贝 (shallow copy) 效果。考虑以下代码：


```

public static
Matrix multiplyByDouble( Matrix mat, double dval )
{
    Matrix result = new matrix( mat.rows, mat.cols );
    for ( int ix = 0; ix < mat.rows; ix++ )
        for ( int iy = 0; iy < mat.cols; iy++ )
            result[ ix, iy ] = mat[ ix, iy ] * dval;

    return result;
}

```

为什么这段代码不直接把 `mat` 复制给 `result`, 而要在 `heap` 上创建一个新的 `Matrix` object? 毕竟, 把 `mat` 复制给 `result`, 能够简化乘法运算:

```

public static
Matrix multiplyByDouble( Matrix mat, double dval )
{
    Matrix result = mat;
    for ( int ix = 0; ix < mat.rows; ix++ )
        for ( int iy = 0; iy < mat.cols; iy++ )
            result[ ix, iy ] *= dval;

    return result;
}

```

如果我们以 `mat` 作为 `result` 的初值, 发生的是浅拷贝, `result` 将仅仅只是实际那个 `Matrix` object 的第二个 handle 罢了。此后“每个元素乘以 `dval`”的动作将直接修改原始 `Matrix` object, 这不符合我们的设想。如果要使修改动作作用于 `Matrix` object 的副本上, 我们必须采用 `new`, 模拟深拷贝 (deep copy)。

当传送或复制一个 *reference* 型别时, 你必须弄清楚, 到底是共用一份实体比较合适, 还是独立的实体更合适。

2.13.2 传址 (Pass by Reference): `ref` 参数

pass by value 的负面影响和它的正面影响其实是一体两面: 函数内针对“引数所对应之 `local` 实体”身上所做的改动, 都丢掉了 (不会反映到引数身上)。如果需要保留改动, 你必须通知编译器以 *byreference* 方式传递实际引数。只需要在参数声明之前加上关键字 `ref`, 即可办到这一点:

```
static public void byRef( ref string s )  
{ /* the body of the function is the same ... */ }
```

reference 参数就像是传递进来的实际引数的一个别名 (alias)⁹。reference 参数并不代表一个独立的 local 物体。这种“参数绑定”形式称为 *pass by reference*。函数中对 *reference* 参数的修改，将直接改动相应的引数。举个例子，如果 `byRef()` 像 `byValue()` 那样在函数内修改 `string`，我们会发现，函数内对参数的修改，会导致我们传给函数的那个实际引数也随之改动：

```
string to be passed by ref:  
A fine and private place
```

```
Inside byRef: original parameter:  
A fine and private place
```

```
Inside byRef: modified parameter:  
A FINE AND PRIVATE PLACE
```

```
back from call -- string:  
A FINE AND PRIVATE PLACE
```

在函数调用处，实际引数也需前缀以关键字 `ref`：

```
string str = "A text string";  
byRef( ref str );
```

要是调用函数时忘了对应写上关键字 `ref`，就会出现编译错误。尽管乍见之下这些似乎太过繁琐，但这使我们得以仅凭“关键字 `ref` 出现与否”来重载 (overload) 并决议 (resolve) 成员函数。(2.14 节对函数的重载有详细介绍)

2.13.3 传址 (Pass by Reference)：out 参数

无论我们将实际引数传给 *value* 参数或 *ref* 参数，编译器都必须确认，在函数调用之前，实际引数已先被赋予某值 (亦即已被初始化)。否则会触发编译错误。

⁹ 然而，这决不会引发对 *value* 型别的 “boxing” 动作！1.14.1 节曾经讨论过所谓的 boxing (装箱)。

然而有时候我们需要实现出这样的函数：在函数体内对参数赋予某值。这可以使函数在执行效果上返回多个值（译注：因为语言规定，函数的正式返回值只能有一个，只有出此策略）。这时候我们需以 *by reference* 方式传递引数，否则一旦函数运行完毕，函数中对参数的任何修改也就报销了。但我们并不想仅仅为了满足编译器，就将引数设为一个哑值（dummy value），毕竟调用这个函数时，该哑值（dummy value）会被覆填内容。

要达到这个目的，只需在参数前加上关键字 *out*，藉以提醒编译器，函数内会为此参数赋值。事实上编译器要求每个 *out* 参数在函数内的每个退离点（exit point）都要被赋值，不然就会触发编译错误。下面这个例子中，我把 *multiplyByDouble()* 重写一遍，让它接受一个 *out* 参数：

```
public static
void multiplyByDouble( out Matrix mat, double dval )
{
    for ( int ix = 0; ix < mat.rows; ix++ )
        for ( int iy = 0; iy < mat.cols; iy++ )
            mat[ ix, iy ] *= dval;
}
```

就像关键字 *ref* 一样，函数的标记式（signature）和调用处都必须指明关键字 *out*：

```
multiplyByDouble( out scaleMat, factor );
```

传给 *out* 参数的引数，既可以被初始化，也可以不初始化。不过在带有 *out* 参数的函数体内，此参数被假设为未初始化。函数之中明确为 *out* 参数赋值之前，此参数是不能被使用的。例如下面以 *out* 参数重新编写的 *byValue()* 就不合法：

```
static public void byValue( out string s )
{
    Console.WriteLine( "\nInside byValue: " );

    // error: s is treated as being uninitialized
    Console.WriteLine( "original parameter: \n\t" + s );

    // ... rest of the function
}
```

一个可能运用 out 参数的场合是：将单词加入一张查找表 (lookup table)。是的，也许我们想要“规整化” (normalize) 单词的大小写和后缀字 (例如 Fly, flies 和 fly 等单词在查找时统统应该被视为相同)，但又不修改原始的本文单词。

要解决这个问题，只需简单地以 by value 方式传入原始单词，并将第二个 string 作为 out 参数传入，用以接受修改后的单词。何不简单返回新的 string 呢？唔，或许我们想让函数的正式返回值用来指示“是否可以添加后缀”，或用来“标明单词语音”。

out 参数为“从函数返回某些结果”提供了另一种途径，使我们能够方便而有效地返回多个值。

2.14 函数重载 (Function Overloading)

两个或多个函数可以共用同一个名称——如果每个函数的参数列 (parameter list) 在参数型别或参数个数上独一无二的话。例如以下五个声明，代表了 message() 的五个重载实体：

```
class MessageHandler
{
    public void message(){ ... }
    public void message( char ch ){ ... }
    public void message( string msg ){ ... }
    public void message( string msg,int val ){ ... }
    public void message( string msg,int v1,int v2 ){ ... }
}
```

编译器如何知道该调用重载函数的哪一个实体呢？是的，编译器会将函数被调用时所获得的实际引数拿来和每个重载函数实体的参数列进行比对，选出最佳匹配者 (best match)。

函数的返回型别 (return type) 并不能够被拿来用以区分函数的重载实体。原因是，返回型别不能保证足够的脉络环境 (context) 供区分出各个重载实体。想象一下，如果多个 message() 函数重载实体仅仅倚赖返回型别来区分，那么以下调用无法提供足够的信息让编译器判断用户想要调用的究竟是上述哪一个实体：


```
// which one?  
message( '\t' );
```

将一组“用以执行类似任务，却有各自独立实现”的函数加以重载，对用户来说，可以简化这些函数的使用。要是没有加以重载，我们得为每个行为类似的函数准备一个独一无二的名称。

2.14.1 重载函数的决议 (Resolving)

编译器决议使用哪一个重载函数时，必须寻找最佳匹配 (best match)，也就是在“调用时传入的实际引数”和“重载函数声明的形式参数”之间挑选出最佳匹配。第一步是确认函数调用动作所对应的“重载函数集” (overloaded functions set)，其中所有函数被称为“候选函数” (candidate functions)。第二步是选出“可行函数” (viable functions)，挑选依据是是否吻合“调用时指定的引数个数和型别”。第三步则是从可行函数(s)中选出最佳匹配，也就是“最佳可行函数”。

为了说明，让我们考虑以下一组重载函数：

```
public static void f( int i1, int i2 ) { ... }  
public static void f( float f1, float f2 ) { ... }  
public static void f( string s ) { ... }
```

现在假设我以两个整数引数调用 `f()`：

```
f( 1024, 2048 );
```

“候选函数集”由上列三个函数实体组成 (译注：因为同名)。然而“可行函数集”只包含前两个函数实体。这两个可行候选函数都接受两个参数，并且调用端的两个整数引数可以传给其中任一函数。最后一个步骤是决定两个可行函数中哪一个为最佳匹配。两个整数引数完全匹配 (exactly match) 第一个函数实体中对应的形式参数。对第二个函数实体而言，要想让两个整数引数匹配两个 `float` 参数，每个引数都必须从 `int` 隐式转换 (implicitly converted) 为 `float`。是的，“完全匹配”肯定好过“需要某些转换”，所以 `f(int, int)` 入选为最佳匹配。

事情总是这样顺利么？不见得。也许第一步骤根本就没有找到任何候选函数，导致编译错误：函数名称未知。如果我们误将 `f()` 写为 `F()`，就会出现这种错误，此时这一调用就无法被正确决议 (resolved) 出来。类似情况，第二步骤里可能根

本没有“可行函数”（因为引数型别的不匹配或引数个数的不匹配）。如果我们以三个引数调用 `f()`，就不会有可行函数，那么也就导致一个编译错误。

2.14.2 寻求最佳匹配 (Best Match)

可行函数的形式参数的个数，和调用动作所传入的实际引数个数，必须完全相同。此外，如果引数型别不与对应的参数型别完全匹配，就必须与对应参数的型别之间存在隐式转换。对 `ref` 参数和 `out` 参数而言，引数型别必须完全匹配（不考虑转换）。

C# 之中，转换动作非“显”即“隐”（either explicit or implicit）。但唯有隐式转换，也就是编译器自动运行的那种转换，在重载函数的“调用决议”过程中才会被考虑。

C# 为内建的（built-in）数值型别定义了一套隐式转换标准，其准则是：数值型别只能隐式转换为“同等大小”或“更大”的型别，举个例子，`long` 可以隐式转换为 `float`、`double` 或 `decimal`，但不能隐式转换为 `int`。因此，以下调用：

```
int ival;
long lval;
// ...
f(ival, lval); // which one?
```

就只有唯一一个可行函数。由于 `long` 不能隐式转换为相对较小的 `int` 型别，也就不可能调用 `f(int, int)`，因此唯一的可行函数是 `f(float, float)`。

现在我修改对 `f()` 的调用动作，让它拥有两个可行函数：

```
int ival;
short sval;
// ...
f(ival, sval); // which one?
```

如何选出最佳可行函数？为选出这个函数，将引数转换为“可行函数之参数”的那些转换行为会被分等（ranked。译注：可参考 *C++ Primer*, p.457, 简体版 p.380）。某个转换动作为何好于另一个转换动作？语言规格书上描述的评价标准如下：

如果由 T1 至 T2 的某个隐式转换存在，而且由 T2 至 T1 的隐式转换不存在，那么“S 转为 T1”就是“较佳的”。

此处 S 代表实际引数的型别，T1 和 T2 分别代表两个可行函数的形式参数的型别。让我们看看能不能把这句话的意思搞得更清楚些。

前述调用的第二引数 sval 的型别为 short，与两个可行函数的参数型别都并非完全匹配（前者的形式参数型别为 int，后者的形式参数型别为 float），但都可以隐式转换。面对第一个可行函数，short 被晋升为 int，面对第二个可行函数，short 被晋升为 float。那么，哪一个比较好呢？

现在，运用语言规格书上的转换规则，我们将实际型别反映（映射）到 S、T1 和 T2。引数实际型别 short 映射为 S，T1 映射为 int，T2 映射为 float。

T1 至 T2 的隐式转换（也就是从 int 至 float 的隐式转换）存在吗？存在！那么 T2 至 T1 的隐式转换（也就是从 float 至 int 的隐式转换）存在么？不存在！因此“从 short 至 int 的转换”比“从 short 至 float 的转换”更好（等级更高）。

总的来说，这条规则意味着“最容易达到的转换，就是首选转换”。例如，从 long 到 float 的转换总是优于从 long 到 double 的转换——虽然它们都可行。这条转换规则也符合我们对“转换该当如何进行”的直观理解。

有时候并不存在最佳可行函数（best viable function），例如面对第一参数时，某个函数占优势，面对第二参数时却是另一个函数占优势。如果这样，这个调用动作就会被标示为模棱两可（歧义，ambiguous），并导致编译错误。例如：

```
public static void g( long l, float f ){ ... }  
public static void g( int i, double d ){ ... }  
  
g( 0, 0 ); // ambiguous
```

这个调用动作的“第一引数”和“第二函数的头一个参数”完全匹配，但却需要隐式转换才能和“第一函数的头一个参数”匹配。因此对第一参数而言，第二函数是最佳匹配。

至于第二个引数 0, 对两个函数的参数都需要做转换动作。然而, `int` 至 `float` 的标准转换 (standard conversion) 要比 `int` 至 `double` 的转换更好。所以对第二参数而言, 第一函数是最佳匹配。

以上结果打了个平手: 两个函数中的任一个都获得了一次最佳匹配奖。由于不存在最佳可行函数, 所以这一调用动作导致模棱两可 (歧义) 错误。要解决这个错误, 可将其中一个引数做显式转换 (explicitly convert), 例如:

```
// OK: public static void g(int i, double d)
g( 0, (double)0 );
```

2.15 可变长度之参数列

上一节那个 `message()` 函数真是问题多多, 它的参数代表“用户可能想要显示的一组值”。问题是用户感兴趣的值, 其型别和数目无穷无尽。即使只是提供各种变体 (variations) 的一个子集, 也会很快令我们疯掉。

一个更易于管理的解决办法是, 声明一个特殊的标记式 (signature), 允许传入任意数目的引数 (引数型别甚至也可以是任意的)。可变长度之参数列 (variable-length parameter) `params` 可以实现这个目标, 特征如下:

- 关键字 `params` 用以指明“参数个数不定”。
- 紧接着关键字 `params` 之后有个 `array`, 其型别是我们愿意接受之引数型别。这个 `array` 内含传入的实际引数。如果你甚至希望接受各种不同的引数型别, 可以将此 `array` 的型别声明为 `object`。

例如, 下面是修正后的 `message()` 函数集, 用来说明关键字 `params` 的用法:

```
class MessageHandler
{
    // our fixed-length parameter instances
    public void message() { ... }
    public void message( char ch ) { ... }
    public void message( string msg ) { ... }
```



```
// our variable-length parameter instances
public void message(string msg, params int[] args) {...}
public void message(string msg, params double[] args){...}
public void message(string msg, params object[] args){...}
// ... rest of class
}
```

前三个函数实体 (instances) 的参数数目是固定的：第一个函数没有参数，第二个函数有唯一参数，型别是 `char`，第三个函数也有唯一参数，型别是 `string`。下面是调用这些函数的示例：

```
MessageHandler mh = new MessageHandler( file );

mh.message();
mh.message( 'a' );
mh.message( "hello" );
```

以上每个调用动作都和 `message()` 的前三个重载版本中的某一个版本的参数列完全吻合。这些调用动作平淡无奇，但接下来的三个函数实体就变得有趣起来。

看看下面这些调用动作，全都匹配“声明了一个 `params int[]`”的那个 `message()` 重载实体：

```
int ival= 10, dval = 0;

// all match: message( string, params int[] )
mh.message( "mumble", 10, ival, dval, 1024 );
mh.message( "mumble", ival );
mh.message( "fib: ", 1,1,2,3,5,8,13,21,34,55 );
```

如果以上调用动作的引数由 `int` 改为 `double`，则匹配的是“声明了一个 `params double[]`”的那个 `message()` 重载实体。如果我们将多种型别混合运用，会发生什么？例如以下调用式混合运用了 `int` 和 `double` 型别：

```
mh.message( "mix types", 10, 3.14159 );
```

这一调用的最佳匹配函数是哪一个？此处，`params array` 的第二个值 3.14159 的型别是 `double`，而 `double` 并不能隐式转换为 `int`，因此带有 `params int[]` 的那个函数实体不是一个可行函数。于是带有 `params double[]` 的那个函数入选。至于第一个数值 10，则被隐式晋级 (implicitly promoted) 为 `double` 型别。

要是我们引入“无法被转换为 `int` 或 `double`”的参数型别，会怎样？例如以下调用动作：

```
mh.message( "weird types", false, mh, 3.14, 2m );
```

表面上看并没有适合这一调用的函数实体，但此调用却能正确运作。被调用的是那个带有 `params object[]` 的重载函数实体。是的，布尔字面常量 `false`、`MessageHandler` `object` `mh`、`double` 数值以及 `decimal` 数值，全都被隐式转换为 `object` 型别。

我们可以查询“参数 `array`”的长度（`Length`，是个 `property`），得知每次调用所传入的 `params` 参数实际个数。如果长度不为零，可以遍历这个 `array`，依次取用每个引数。我们还可以利用 *type reflection* 找出每个参数的实际型别（8.2 节对此有所讲解）。例如下面是一个可能的实现做法：

```
public void message( string msg, params object[] args )
{
    Console.WriteLine( msg );

    if ( args.Length != 0 )
        foreach ( object o in args )
            Console.WriteLine( "\t{0}", o.ToString() );
}
```

另一个办法是，通过 `array` 的索引，访问每一个参数：

```
public void message( string msg, params int[] args )
{
    Console.WriteLine( msg );

    for ( int ix = 0; ix < args.Length; ++ix )
        Console.Write( "{0} ", args[ix].ToString() );
    Console.WriteLine();
}
```

一个函数只能带有一个 `params array`，而且必须在参数列最末尾处声明，`params array` 只能是一维，而且不能以关键字 `ref` 或 `out` 加以修饰。

典型情况下，使用 `params array` 将会扩张“接受一个或多个明确参数”的函数集，就像 `message()` 的定义式那样。在先前出现的例子中，我们处理了三种情况：

无参数、单一 string 参数、单一 char 参数，所有其他调用动作所传递的引数都必须以 string 打头，其后跟着零个或多个不限型别的引数。

如果 params array 跟在一个或多个明确参数之后，就像我们先前的那三个 message() 函数实体那样，这些明确参数中的每一个都必须由用户提供实际引数与之对应。params array 则负责处理其余所有引数（如果有的话）。

如果想要接受零个或多个“特定型别”的引数，我们可以令 params array 成为函数的唯一参数。如果想要接受零个或多个“任意型别”的引数，我们可以将此 params array（这将是函数的唯一参数）声明为 object 型别：

```
// accepts zero or more arguments of type int
static void func( params int [] args )

// accepts zero or more arguments of type string
static void func( params string [] args )

// accepts zero or more arguments of any type
static void func( params object [] args )
```

面对关键字 params，重载决议工作（overloading resolution）又是如何开展的呢？考虑以下两个函数：

```
static void display( string msg )
static void display( string msg, int ix, params object [] args )
```

和以下调用动作：

```
display( "message", 42, arg1, arg2 );
```

display() 的第一个实体并非可行函数（viable function），因为它只接受单一参数。第二个函数实体倒是有可能匹配。

编译器对标记式（signature）的评估步骤如下：首先在不考虑 params array 的情况下尝试明确（explicit）参数的匹配情况。函数的明确参数（此处的 string 和 int）必须吻合——完全吻合或通过隐式转换均可。如果完成第一阶段匹配，剩余引数（如果有的话）再被编译器打包为一个 array，传给匹配的那个函数。本例之中，arg1 和 arg2 分别成为打包所得的 array 的第一元素和第二元素。

2.16 操作符重载 (Operator Overloading)

操作符重载使我们能够为业已存在的操作符提供“与 class 相应”的实体，例如加法、乘法、相等测试等等。举个例子，对 `Matrix` class (矩阵类) 而言，我们不必提供一个具名函数，如 `multiplyByDouble()`，我们可以重载乘法操作符 (multiply operator) 来执行相同任务。也就是说用户不必写出：

```
Matrix newMat = Matrix.multiplyByDouble( mat, dval );
```

可以更直观地这么写：

```
Matrix newMat = mat * dval;
```

以下是实现乘法操作符 (multiply operator) 的一种可能做法：

```
public class Matrix
{
    public static Matrix operator*(Matrix mat, double dval)
    {
        Matrix result = new Matrix( mat.rows, mat.cols );

        for ( int ix = 0; ix < mat.rows; ix++ )
            for ( int iy = 0; iy < mat.cols; iy++ )
                result[ix,iy] = mat[ix,iy] * dval;

        return result;
    }
    // ... rest of the Matrix class
}
```

所有操作符函数都必须声明为 `public` 和 `static`。紧跟在关键字 `operator` 之后的是待被重载的操作符 (像是 `+`, `-`, `*`, `/` 等等)。这个操作符至少必须有一个参数是 `class` 实体。此处我们重载的是乘法操作符，接受一个 `Matrix` object 和一个 `double` 值，并返回一个新的 `Matrix` object。

重载后的操作符可被直接运用到 `class` object 身上，其运用形式看起来和内建的操作符没什么两样。当编译器遇见一个表达式，例如 `mat * dval` 时，它会以操作数的型别为依据，加以决议，看看到底应该调用乘法操作符的哪一份函数实体。此处编译器调用的是我们重载后的那个 `Matrix` 操作符实体。

如果我们写出以下语句，会发生什么？

```
mat *= dval; // 译注：这就是下面所说的 compound assignment operator
```

既然我们没有明确提供乘法运算的“复式赋值操作符”（compound assignment operator）的重载实体，这个语句会导致编译错误吗？不，不会，一旦我们提供某个操作符的重载实体，而该操作符有一个对应的“复式赋值操作符”（例如加法、减法、乘法等等），那么对此“复式操作符”的支持会自动实现。事实上如果你为“复式赋值操作符”明白提供了重载实体，反而会导致编译错误。

如果我们交换两个操作数，又会如何？像这样：

```
Matrix newMat = dval * mat;
```

如果这么做，编译器会将这一表达式视为非法。因为它不知道如何将 double 乘以 Matrix；它只懂得如何将 Matrix 乘以 double。我们必须明确提供第二个乘法操作符函数实体，才能让上述语句有效运作起来。欲实现此处的第二个操作符，只需在其函数体内调用第一个操作符即可：

```
public static Matrix  
operator*( double dval, Matrix m ){ return m*dval; }
```

如果我们将 Matrix object 乘以 int 或 float 会怎样？既然我们的操作符期盼的是 double 操作数，这样做会导致错误么？

不，不会导致错误。如果单独一个隐式转换就能匹配形式参数的型别，那就没有问题。

“被重载之操作符”的参数不可以是 ref 或 out，此外我们也不能改变操作符原始定义的引数个数。例如我们无法定义出“只接受一个操作数”的除法操作符。同样道理，我们也不能够改变操作符的优先级（precedence）。例如乘法操作符永远必须比加法操作符有较高的优先级。（请见 1.18.4 节对操作符优先级的讨论）

C# 语言要求，“被重载之操作符”至少需有一个参数隶属于该操作符所属的 class 型别。这个约束条款可以防止我们重复定义（redefining）业已存在的操作符（例如 int 的加法操作符）。

下面是八个可被重载的一元操作符（unary operators）：

```
// the overloadable unary operators
```

`+, -, !, ~, ++, --, true, false`

这些一元操作符的唯一参数的型别必须等于“该操作符所隶属之 class”。有三个一元操作符不允许被重载: *selection* 操作符 (`.`), *call* 操作符 (`()`), *new* 操作符。

如果你定义了 `true` 操作符, 必须同时也定义 `false` 操作符, 二者都必须返回 `bool` 值。例如我们的 `StringNode` class 可以自行定义一个 `true` 操作符, 表示其 `string` 成员不是 `null`, 并定义一个 `false` 操作符, 表示其 `string` 成员是 `null`。

```
class StringNode
{
    private string text;
    public string Text{ get{ return text; } };

    private StringNode front_link;
    public StringNode Next{ get{ return front_link; } };

    public static bool
        operator true(StringNode sn){return sn.text != null;}

    public static bool
        operator false(StringNode sn){return sn.text == null;}
}
```

接下来就可以这样使用 `StringNode` object:

```
static public void display( StringNode sn )
{
    while ( sn != null )
    {
        // invokes: bool operator true( StringNode )
        if ( sn ) // 译注: 由于 if 条件句要求的是 true 或 false, 所以编译器
                // 会调用 sn (一个 string object) 的 true 和 false 操作符
            Console.WriteLine( sn.Text );
        sn = sn.Next;
    }
}
```

increment 操作符 (`++`) 和 *decrement* 操作符 (`--`) 必须返回其所隶属之 class 的一个 object。C# 语言无法区分这两个操作符的前缀 (`prefix`) 和后缀 (`postfix`) 形式 (译注: C++ 可以)。

以下的二元操作符 (binary operators) 都可以被重载:

```
// the overloadable binary operators
+, -, *, /, %, &, |, ^, <, >, <<, >>, ==, !=, <=, >=
```

这些操作符的两个参数之中, 至少有一个其型别必须是“该操作符所隶属之 class”。

下面是不允许被重载的二元操作符:

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

以下操作符必须成对重载:

```
(==, !=); (<, >); (<=, >=)
```

如果你重载了 *equality* 操作符 (==) 却没有重载 *inequality* 操作符 (!=), 编辑器会将后者的缺席视为一种错误。

2.17 转换式操作符 (Conversion Operators)

C# 提供一种机制, 让每个 class 都可以定义一组“可应用于其 object 身上”的隐式 (implicit) 转换或显式 (explicit) 转换。这样做有何意义?

假设我们设计了一个 *BitVector* class。为了便利用户, 我们决定将此 class 内部的 0,1 bits 序列转为以 *unsigned long* 或 *string* 来表示。同样道理, 我们可能需要在需要 *BitVector* 的地方以 *string* 或 *unsigned long* 表示 bit 序列, 并让程序知道如何将这些表示法转换为 *BitVector*。

一种转换方向是, 提供一个或多个转换算法 (conversion algorithms), 将此 class 的一个 object 转换为另一个型别的 object。通常我们会将这些转换式操作符指定为 *implicit*, 因为它们总是应该成功。这意味着编译器会自动运行这些操作符以完成转换动作, 因此这些操作符得以参予“重载函数的决议进程” (overloaded functions resolution)。

举例来说, 我们的 *BitVector* 转换为 *string* 或 *ulong* 是一种良好的行为, 因此我们可能这样用它:

```
// our overloaded methods
public static void display( string s ){ ... }
public static void display( HashTable ht ){ ... }
```

```
public static void Main()
{
    BitVector bv = new BitVector( 32 );

    // implicitly convert bv into a string
    display( bv );

    // implicitly convert bv into a ulong
    ulong ul = bv;
}
```

欲支持上述两个 `BitVector` 隐式转换动作，我们可以这样声明转换式操作符：

```
public class BitVector
{
    static public implicit
        operator string( BitVector bv ){ ... }

    static public implicit
        operator ulong( BitVector bv ){ ... }

    // ... rest of BitVector class definition
}
```

就像“重载操作符”一样，“转换式操作符”必须为 `static` 和 `public`。此外还必须指定关键字 `implicit` 或 `explicit`。紧跟在关键字 `operator` 之后的那个型别，就是转换式的返回型别，其唯一参数所代表的，是转换动作的原始型别 (source type)。此参数既不能是 `ref` 参数也不能是 `out` 参数。转换动作的“返回型别”和“参数”二者之中必须有一个是“这一转换式操作符所隶属的那个 `class`”。就拿上例来说，两个操作符都对 `BitVector` object 进行转换，第一版本将 `BitVector` 转换为 `string`，第二版本将 `BitVector` 转换为 `ulong`。

另一种转换方向是，提供一个或多个转换算法，将其他型别的 object 转换过来。通常此类转换式会被指定为 `explicit`，这意味着如果需要此种转换，用户必须有显式的转型动作。我们不把两个转换方向都弄成 `implicit`，有以下两点理由。

第一个理由是，如果我们将两个转换方向都指定为 `implicit`，编译器可能会视某些情况为模棱两可 (歧义, *ambiguous*)。例如我们这样写：

2.18 Class 的析构函数 (Destructor)

C# 支持非确定性的析构函数 (nondeterministic destructor method)，但并不鼓励使用。析构函数的非确定性表现在两方面。第一，我们无法预知析构函数何时被调用，甚至无法预知它是否会被调用。第二，我们无法预知析构函数被调用的顺序。从性能角度看，“带有析构函数”的 classes 在与垃圾回收器 (garbage collector) 打交道时会背负沉重得多的额外开销。

析构函数 (destructor)，或说某种起终结作用的例程 (finalization routine)，在以下场合还是必需的：class object 在其生命期间获取了“非受控资源” (unmanaged resources)，而我們希望在用完该 object 之后就释放那些资源。资源可能包括 window handles 或 file handles，或是数据库连接 (database connections)。习惯上我们把资源归还动作安排在 Dispose() 之中，这个成员函数必须由用户手工调用。

4.8 节讨论 IDisposable 接口和“自动调用 Dispose() 的特殊 using 语句语法”时，我们再回到这个问题来。

2.19 struct value 型别

struct 机制使我们得以在应用程序中引入新的 value 型别 (译注：相对于 reference 型别)。struct 的声明式看起来简直和 class 一模一样 (除了关键字 struct 之外)，例如：

```
public struct matrix
{
    private double[,] m_mat;
    private int      m_row;
    private int      m_col;

    // ...
}
```

Value 型别总是直接在 object 内存存储数据。上述的 matrix object 就直接持有 m_row 和 m_col 整数值，并持有 m_mat，那是身为 reference 型别的一个二维 array handle。

当我们以某个 struct object 初始化另一个 struct object，或是将一个 struct object 赋值给另一个 struct object 时，发生的是深拷贝 (deep copy)，意味着两个 objects 持有相同的值，但仍旧保持独立：这一点和 reference 型别的“浅拷贝语

义” (shallow-copy semantics) 不同。这不仅简化了加法运算，还提高了速度，例如，也许我们只打算将某个 `matrix` object 的元素“注入”另一个之中：

```
// 译注：以下运算会影响引数 m1 的值。或许作者原意是想实现 operator+=。
public static matrix operator+( matrix m1, matrix m2 )
{
    check_both_rows_cols( m1, m2 );
    matrix mat = m1; // 译注：将发生深拷贝 (deep copy)

    for ( int ix = 0; ix < m1.rows; ix++ )
        for ( int ij = 0; ij < m1.cols; ij++ )
            mat[ ix, ij ] += m2[ ix, ij ];

    return mat;
}
```

`struct` object 并非在 `managed heap` (受控堆) 中分配内存，因此不受垃圾回收器的支配。当我们像这样创建一个新的 `struct` object 时：

```
public void func()
{
    matrix mat = new matrix();
    // ...
}
```

并没有真正调用 `new` 操作符。`matrix` object 直接在函数内获得分配；它在函数开始运行时现身，并在函数运行完毕后悄然而去。

编译器会自动为每个 `value` 型别提供 `default` (缺省) 构造函数，亦即无需引数的构造函数，这个 `default` 构造函数的功能是将“这一 `value type` object 内”存储的数据成员的内容设置为零。在 `struct` 定义式中明白提供 `default` 构造函数，将是一种错误行为。

在 `value` 型别上调用 `new` 操作符，会导致仅仅单纯地执行相关构造函数，如果调用 `new` 操作符时没有传入引数，则编译器调用的是 `default` 构造函数。如果提供了引数，编译器会查找一个得以匹配这些引数的构造函数。例如以下语句要求我们必须为 `matrix` 提供“接受两个引数”的构造函数：

```
matrix mat = new mat( 4, 4 );
```



```
public struct matrix
{
    public matrix( int row, int col )
    {
        m_row = ( row <= 0 ) ? 1 : row;
        m_col = ( col <= 0 ) ? 1 : col;
        m_mat = new double[ m_row, m_col ];
    }
}
```

面对 struct, 除了“不得为它引入 default 构造函数”外, 另还有两个限制: (1) 其数据成员的声明式不得包含初始化动作 (initializer, 见 2.6 节), (2) 不得为 struct 提供析构函数。

某些情况下这些限制会带来问题。例如在计算机图形学领域, 我们常常定义一个特殊化的 4×4 矩阵, 作为几何变换之用, 包括旋转 (rotation) 和缩放 (scaling)。由于我们既不能够定义 default 构造函数, 又不能够定义 explicit 成员初始式 (member initializer), 所以我们缺乏直接了当的手法 (像下面那样) 将一个 4×4 矩阵定义为 struct:

```
public struct Matrix44
{
    private double[,] mat; // 问题在于 ...
```

问题在于, 代表这一矩阵的 array 是一个 reference 型别, 其声明式中并未说明两个维度的大小。在 class 定义式中, 我们可以加上维度大小, 像下面这样:

```
public class Matrix44
{
    private double[,] mat = new double[ 4, 4 ];
```

但是在 struct 内却不能这么干。同样道理, 在 class 定义式中我们可以在 default 构造函数内初始化 mat, 但我们却不能够为 struct 提供 default 构造函数。

环绕着这个限制, 有不少小窍门可以解决问题, 但是求助于小窍门毕竟总是令人稍感失望。本书示例程序中, 我提供了一个名为 structFixMatrix 的简单例子, 展示如何在这些限制之下编写程序。(译注: 我没有找到这个例子, 最接近的例子是位于 chapter2\MatrixVector 目录下的 Matrix.cs)

为什么要舍弃 class 不用而定义一个 struct? 主要是为了提高运行期性能。举个例子, 如果 `SmallInt` 是个 struct, 以下声明式将创建出“含有 1000 个 `SmallInt` object”的 array:

```
SmallInt [] vertices = new SmallInt[ 1000 ];
```

但如果 `SmallInt` 是个 class, 上述声明式将产生“内含 1000 个 null handles”的 array, 并额外调用 1000 次 `operator new`, 分别为这 1000 个 objects 在受控堆 (managed heap) 上分配内存空间。

请记住, struct 的成员如果是 *reference* 型别, 则此成员依旧遵循“浅拷贝”语义。换言之, 如果我们的 `Matrix44` struct 包含一个隶属于 *reference* 型别的 array 成员, 那么缺省情况下将一个 `Matrix44` object 复制给另一个 `Matrix44` object 时, 会导致两个 objects 内的 array 成员指向同一个 heap array object。

我们何时才会将一个抽象想法 (abstraction) 选择以 struct 表现? 答案是: 这个抽象体应该很小, 否则“深拷贝”适得其反, 特别是在以 *by value* 方式将 object 传入函数时。此外这个抽象体应该被大量使用, 否则或许我们可以忽略性能方面的好处。在 .NET Framework 下, 预定义的算术型别 (`int`, `double` 等等) 都是 *value* 型别而非 *reference* 型别。

3

面向对象程序设计

Object-Oriented Programming

Class 的主要用途在于引入新型别，以便能够更直接地在我们的应用程序中表现我们想表现的物体（entity）。例如图书馆借阅系统，如果我们利用 Book（书）、Borrower（借阅人）、DueDate（到期日）这几种 classes 来进行程序设计，往往会比使用基本的字符型别、算术型别、布尔（Boolean）型别轻松得多。

当我们的应用程序布满许多 classes，而且其间彼此有着“是一种（is-a-kind-of）”的关系时，如果无法让它们产生某种关连（译注：彼此无关连的 classes 也就是所谓的 **object-based** 编程模型），反而会拖累我们的程序设计工作。举个例子，设想随着时间的流逝，我们的图书馆借阅系统除了支持原本的 Book class 之外，还增加了对 RentalBook class、AudioBook class 以及 InteractiveBook class 的支持。每个 class 可能会想要共享数据成员和成员函数来管理数据，每个 class 也可能需要增加额外的数据成员，用以表示其自身状态，尽管每个 class 都享有相同的 interface（接口），但每个 class 也可能（或可能不）具备特殊借还程序，以及逾期不还的罚款计算方式。

前一章介绍的 “object-based” class 机制，无法轻易模塑出上述四个具有 “are-a-kind-of（隶属某种）” 性质的 Book classes 之间的共同处和差异处。为什么？因为该机制没有支持任何方法得以描述 “特定 classes 之间的关系”。要想获得这种支持，我们需要 “object-oriented”（面向对象）编程模型。

3.1 面向对象编程概念

面向对象程序设计（Object-Oriented Programming）的两个最主要特征是：继承（inheritance）和多态（polymorphism）：

1. 继承机制（inheritance）使我们得以将一群彼此相关的 classes 组织起来，并让我们得以分享其间的共通数据和共通操作行为。想想 1.17 节介绍的 exception

classes 族系。

2. 多态机制 (**polymorphism**) 让我们在这些 classes 身上进行编程时, 可以如同操控单一 class (而非相互独立的数个 class) 似的方便, 并赋予我们更多弹性来加入或移除任何特定 classes。

继承机制定义了父子 (parent/child) 关系, 父类 (parent) 定义出“所有子类 (children) 共通的公开接口 (public interface) 和私有实现 (private implementation)”。每个子类都可以增加或覆写 (override) 继承而来的东西, 以便实现它自身独特的行为。

例如, 子类 `AudioBook` (有声书) 除了从父类 `Book` 继承“作者”和“标题”之外, 还增加“演讲者”以及它所使用的“卡带盘数” (或 CD 张数)。除此之外, 它也改写了从父类继承而来的 `check_out()` 成员函数。

在 C# 中, 父类又被称为 **base class** (基类), 子类又被称为 **derived class** (派生类)。父类 (即 **base class**) 和其子类之间的关系, 形成所谓的继承阶层体系 (**inheritance hierarchy**, 简称继承体系)。例如在设计评论会议中, 我们可能会这么说: “我们希望实现一个 `AudioBook` derived class, 它覆写 (override) **base class** `Book` 的 `check_out()` 成员函数, 并沿用继承而来的 `Book` class 的数据成员和属性 (**properties**), 亦沿用管理架位、作者、标题等信息的成员函数”。

图 3.1 绘出图书借阅管理系统中用到的各种馆藏素材的 class 继承体系。此继承体系中最根本的 class 乃是一个 **abstract base class** (抽象基类): `LibMat`。`LibMat` 定义了所有馆藏素材的共通操作行为, 包括 `check_in()`、`check_out()`、`shelf_location()`、`fine()`、`due_date()` 等等。然而 `LibMat` 并不代表图书借还管理系统中实际存在的任何一份馆藏, 仅仅是为了设计上的需要而存在。不过, 这一额外添加物十分重要而关键, 我们称之为 **abstract base class** (抽象基类)。

在一个面向对象程序中, 我们利用 **abstract base class** 来间接操控应用程序里的 **class objects**, 而非直接操控应用程序中的 **derived-class objects**。间接操控使我们得以在“不更动既有程序”的情况下添加或移除 **derived class**。例如下面是我们的 `Library` class 的成员函数 `loan_check_in()` 的一种可能手法:


```
class Library
{
    public void loan_check_in( LibMat mat )
    {
        // mat refers to a derived-class object,
        // such as Book, RentalBook, Magazines, etc. ...
        mat.check_in();

        if ( mat.is_late() )
            mat.assess_fine();

        if ( mat.waiting_list() )
            mat.notify_available();
    }

    // ... rest of the Library class definition
}
```

我们的应用程序中并不存在具体的 (concrete) LibMat objects, 我们所能够拥有的只是 Book、RentalBook、AudioBook 等等 class objects, 以及读者能够借阅的其他实际馆藏素材。此函数如何运行呢? 例如, 通过 mat 调用 check_in() 时, 究竟发生什么事? 这个函数如果希望具备合理意义, 那么每当运行 loan_check_in() 时, mat 必须以某种方式指向 (代表) 应用程序中某个实际的 class objects。此外, 函数内调用的成员函数 check_in() 也势必以某种方式被决议 (resolved) 为 “mat 所代表的那个实际 class object” 所拥有的 check_in() 函数实体。这便是所有发生的事情。问题是, 它实际上究竟如何运行?

本节早先提到过, 面向对象程序设计的第二个独特性是多态 (polymorphism), 也就是 “base-class object 有能力以十分透通的 (transparently) 方式指向 (代表) 任何一个 derived-class objects”。例如在 loan_check_in() 函数中, mat 总是指向一个 “从 LibMat 继承下来的某个 class” 的 object。到底是哪一个呢? 只有在运行这段程序时才能确定, 而且很可能每次调用 loan_check_in() 时情况都不相同。多态之所以能够成功运作, 关键机制在于动态绑定 (dynamic binding)。在 “非面向对象” 的程序中, 当我们写下:

```
mat.check_in();
```

在这里, 编译期间便依据 mat 的 class 种类, 决定运行哪一个 check_in() 函数实体。由于调用的函数是在程序运行前就获得决议 (resolved), 所以称作静态绑定 (static binding)。

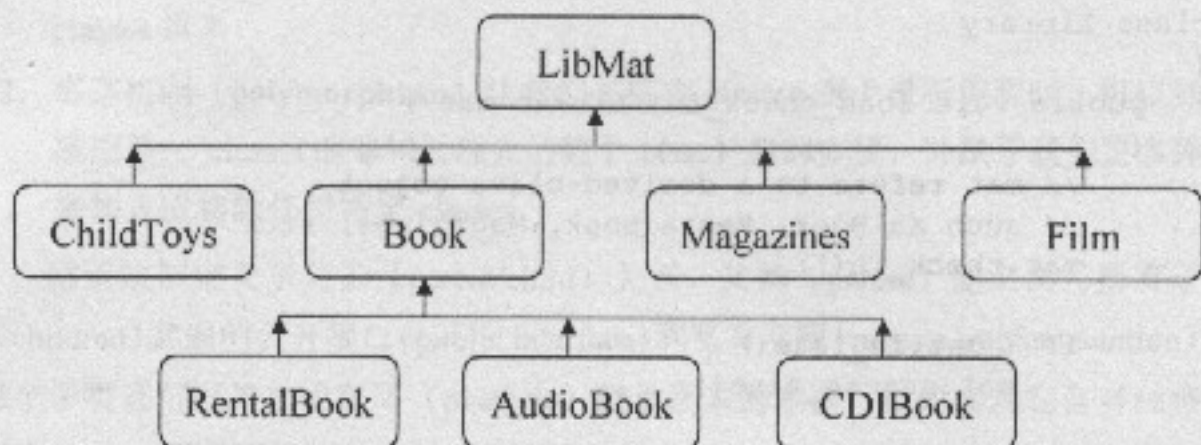


图 3.1 图书馆借还系统中的“馆藏素材种类阶层体系” (Class Hierarchy)

沿续上页讨论, 在面向对象程序设计中, 编译器不知道该调用哪个 `check_in()` 实体, 所以函数调用的决议动作必须推迟至运行期才得以完成。决议动作是根据每次调用 `loan_check_in()` 时 `mat` 实际代表的 `derived-class object` 来决定的。这就是我们所谓的动态绑定 (**dynamic binding**), 这是面向对象程序设计的一个基本性质。

继承 (**inheritance**) 使我们得以定义“共用一个公共接口 (**interface**)”的一整群 `classes`, 就像我们的图书馆馆藏素材那样, 多态 (**Polymorphism**) 让我们得以运用“与型别无关的方式”来操控这些 `classes objects`。通常我们会通过抽象基类 (**abstract base class**) 的一个实体来对公共接口进行编程, 至于其所运行的实际操作, 只有等到运行期, 视该实体所指的 `object` 的真实型别, 才能确定。

如果图书馆决定不再出借交互式 (“**interactive**”) 书籍, 我们只需从继承体系中移除 `InteractiveBook class` 即可, `loan_check_in()` 的实现不需任何改变。类似情况, 如果图书馆决定对某些 “**audio**” 书籍收取租金, 我们只需实现一个派生的 `AudioRentalBook class`, `loan_check_in()` 仍然不需改变, 如果图书馆决定出借膝上型计算机或电视游戏机的设备和卡带, 面对这种变动, 我们的继承体系依然能够应付¹⁰。

¹⁰ 注意, `struct` 不是 `object-oriented programming` 的一部份。它既不能成为继承的对象, 也不能参与动态绑定。(译注: 这一点和 C++ 不同)

3.2 实现一个“多态查询语言” (Polymorphic Query Language)

本章剩余篇幅，我要带领各位实现一个“查询语言 classes 继承体系” (query language class hierarchy)，以此介绍 C# 语言中为了支持面向对象编程而提供的各种构件 (constructs) 及编程惯用技法 (programming idioms)。什么是我所谓的“查询语言”？这是普通的文字查询应用的一部分，允许用户在一个文本文件中搜索某个 (或数个) 单词的出现点。举个例子，以下是我们要查询的文本文件：

```
Alice Emma has long flowing red hair. Her Daddy says
when the wind blows through her hair, it looks almost alive,
like a fiery bird in flight. A beautiful fiery bird, he tells her,
magical but untamed. "Daddy, shush, there is no such creature,"
she tells him, at the same time wanting him to tell her more.
Shyly, she asks, "I mean, Daddy, is there?"
```

以下是文字查询系统的运行示范：

```
TextQuery: begin()

Reading c:\QueryManager\alice_emma.txt -- please wait!
Reading c:\QueryManager\alice_emma.txt -- OK, done!

The text file alice_emma.txt
    contains 6 lines in 357 bytes.

entering text into dictionary -- please wait!
entering text into dictionary -- OK, done!

    Time to read text file: 0.004 seconds
    Time to enter text into dictionary: 0.009 seconds

Please enter a query or 'q' to Quit: alice
alice
    1 line(s) match: [ 1 ]

[1]: Alice Emma has long flowing red hair. Her Daddy says
```

```

Please enter a query or 'q' to Quit: alice || daddy
alice
    1 line(s) match: [ 1 ]
daddy
    3 line(s) match: [ 1 4 6 ]
alice || daddy
    3 line(s) match: [ 1 4 6 ]

[1]: Alice Emma has long flowing red hair. Her Daddy says
[4]: magical but untamed. "Daddy, shush, there is no such creature,"
[6]: Shyly, she asks, "I mean, Daddy, is there?"

Please enter a query or 'q' to Quit: q

OK, bye!
TextQuery: end()

```

我们的查询系统支持设施由以下元素构成:

- *name query*, 查询单一单词, 例如查询 "Alice" 或 "untamed". 所有包含该单词的每一文本行都会被显示出来. 行号置于方括弧中, 并按递增顺序排列.
- *NOT query*, 以操作符 ! 代表. 举个例子, 如果查询 !daddy, 所有不含单词 daddy 的文本行将会被显示出来:

```

Please enter a query or 'q' to Quit: !daddy
daddy
    3 line(s) match: [ 1 4 6 ]
!daddy
    3 line(s) match: [ 2 3 5 ]

[2]: when the wind blows through her hair, it looks almost alive,
[3]: like a fiery bird in flight. A beautiful fiery bird, he tells her,
[5]: she tells him, at the same time wanting him to tell her more.

```


- OR query, 以操作符 `||` 表示。例如查询 `alice || daddy`, 会显示所有“包含这两个单词中的任意一个”的文本行。
- AND query, 以操作符 `&&` 表示。例如查询 `alice && daddy`, 会显示所有“同时包含这两个单词”的文本行。

Please enter a query or 'q' to Quit: `alice && daddy`

`alice`

1 line(s) match: [1]

`daddy`

3 line(s) match: [1 4 6]

`alice && daddy`

1 line(s) match: [1]

[1]: Alice Emma has long flowing red hair. Her Daddy says

这些元素可以合在一起使用, 像这样:

`fiery && bird || shyly`

评估 (evaluation) 顺序由左而右, 每个操作符享有相同的优先级。例如以下查询动作将找出所有“同时包含 “fiery” 和 “bird” 的文本行, 或“仅包含 “shyly” 的文本行:

Please enter a query or 'q' to Quit: `fiery && bird || shyly`

`fiery`

1 line(s) match: [3]

`bird`

1 line(s) match: [3]

`fiery && bird`

1 line(s) match: [3]

`shyly`

1 line(s) match: [6]

`fiery && bird || shyly`

2 line(s) match: [3 6]

[3]: like a fiery bird in flight. A beautiful fiery bird, he tells her,

[6]: Shyly, she asks, "I mean, Daddy, is there?"

我们的查询设施也支持圆括弧，这样便可以对查询动作进行“分组”（subgrouping）。例如以下查询：

```
fiery && ( bird || shyly )
```

会找出所有“同时包含 “fiery” 和 “bird””或“同时包含 “fiery” 和 “shyly””的文本行：

```
Please enter a query or 'q' to Quit: fiery && ( bird || shyly )
fiery
  1 line(s) match: [ 3 ]

( bird
  1 line(s) match: [ 3 ]

shyly
  1 line(s) match: [ 6 ]

( bird || shyly
  2 line(s) match: [ 3 6 ]

fiery && ( bird || shyly )
  1 line(s) match: [ 3 ]

[3]: like a fiery bird in flight. A beautiful fiery bird, he tells her,
```

以上就是我们的 class 继承体系所要表现的功能。现在的问题是，如何实现？

3.3 设计一个 Class 继承体系

我们在 class 设计方面的第一个想法是，将每一个查询操作以单一而独立的 class 表示出来：

```
NameQuery    // Alice
NotQuery     // ! Alice
OrQuery      // Alice || fiery
AndQuery     // Alice && Daddy
```

乍见之下，这个设计似乎是适当的。每个 class 提供一个 eval() 成员函数，用来完成它所代表的查询动作，并提供一个 display_solution() 成员函数，用来显示查询成果。所谓成果，是一个以递增顺序排列，而且内容不重复的“行号集”（line numbers set）。

NameQuery class 的 eval() 成员函数仅仅单纯地返回包含 "name" (查询对象) 的行号。OrQuery 的 eval() 成员函数则必须建立起 "包含两个操作数中的任意一个" 的文本行的行号并集。AndQuery class 和 NotQuery class 也都必须实现出特定功能的 eval()。

如果用户键入以下查询:

```
untamed || fiery
```

我们将创建两个 NameQuery class objects, 分别表示字符串 "untamed" 和 "fiery"。然后创建一个 OrQuery class object, 将上述两个 NameQuery objects 当作它的操作数。然后调用 OrQuery 的 eval() 成员函数, 获得一份运算成果。最后再调用 print() 显示成果。事实上, 这样看起来实在没必要引入什么面向对象程序设计!

但是当处理类似以下的 "组合式查询" 时, 上述 "四个独立 classes" 的设计就出现问题了:

```
Alice || Emma && Weeks
```

这个查询由两个子查询组成: 一个是 OrQuery object (包含两个 NameQuery 操作数, 分别对应字符串 "Alice" 和 "Emma"), 另一个是 AndQuery object, 其右操作数是一个 NameQuery object (对应字符串 "Weeks")。到目前为止一切顺利: 每个操作数都是 NameQuery object, 这很容易表现。

然而, AndQuery 的左操作数让我们的简单设计彻底崩溃:

```
AndQuery
// here is the problem!
OrQuery
    NameQuery ("Alice")
    NameQuery ("Emma")
NameQuery ("Weeks")
```

AndQuery 的左操作数是个 OrQuery object。更一般地, 我们意识到, 一个 query object 的操作数除了可能是 NameQuery 之外, 还可能是任何其他 query type。在我们不知道操作数型别的情况下, 我们如何表示它呢? 或者, 如果我们知道操作数可能是若干不同型别中的一个, 我们该如何表示它呢? 这个问题在于两方面:

1. 我们必须能够在 OrQuery、AndQuery、NotQuery classes 内声明操作数的型别，该型别必须能够持有 (hold) 四个不同的 query class types 中的任何一个。
2. 不管上述第 1 项如何解决，我们必须能够在运行期针对每个操作数调用其 class 专属的 eval() 成员函数。

这才是需要面向对象程序设计的地方。只要将我们的 classes 重新模型为 query 继承体系，就能解决以上两个问题。

通过继承，我们可以为先前四个独立的 query classes 定义彼此之间的关系。我们引入一个抽象的 Query base class (基类)，其他 classes 都从中派生而来。这便立刻解决了我们的第一个问题：base class object 能够十分透通地指向 (代表) 其任何 derived-classes object。

“继承”赋予了 base class 和其 derived classes 之间一种特殊的 type/subtype 关系。当我们把 base-class object 以其 derived class object 作为初值，进行初始化，或是将其 derived class object 赋值给 base-class object 时，会自动完成隐式转换。在我们的 Query 继承体系中，Query object 可以完全透通地指向 (代表) OrQuery、AndQuery、NotQuery、NameQuery 等等 class object。

当你要指明某个 class 从另一个 class 继承而来，应该这样写：

```
class NotQuery : Query
```

NotQuery class 名称之后的冒号 (:) 表示，它是从 Query class 继承而来。NotQuery 是派生类 (derived class)；Query 是其直接 (上一层) 基类 (base class)。base class 的两个主要约束条件是：

1. base class 必须事先定义好。
2. base class 必须至少和 derived class 一样可访问 (accessible)。

在 Query 继承体系被创建出来之前，我们唯一能够定义“可接受任何 query class”的函数的办法是：明白地 (显式) 将函数重载，使之能够接受不同种类的实体：

```
class testQuery
{
    public static void eval_print( OrQuery q )    // (1)
    { q.eval(); q.print_solution(); }
```



```

public static void eval_print( AndQuery q )    // (2)
{ q.eval(); q.print_solution(); }

public static void eval_print( NotQuery q )    // (3)
{ q.eval(); q.print_solution(); }

public static void eval_print( NameQuery q )  // (4)
{ q.eval(); q.print_solution(); }

// ...
}

```

然而一旦我们引入 Query 这个抽象基类, 并使所有 query types 都派生自它, 那就创建了一个继承体系。于是我们的四个函数就可精简为一个:

```

class testQuery
{
    public static void eval_print( Query q )
    { q.eval(); q.print_solution(); }

    public static void Main()
    {
        NameQuery nq1 = new NameQuery( "Mason" );
        NameQuery nq2 = new NameQuery( "Dixon" );
        AndQuery  aq  = new AndQuery( nq1, nq2 );
        OrQuery   oq  = new OrQuery( nq1, nq2 );
        NotQuery  nq   = new NotQuery( nq1, nq2 );

        // OK: automatic conversion from a
        //      derived class to its base class

        eval_print( nq1 ); eval_print( aq );
        eval_print( nq );  eval_print( oq );

        // error: string is not derived from Query;
        //      no conversion
        eval_print( "MasonDixon" );
    }
}

```

derived class 可以在必要时隐式转换为 base class, 就像我们调用 eval_print() 时所发生的那样。然而如果尝试将一个 string object 赋值给 Query object, 不会成功, 因为 Query 和 string 毫无关系。

共同 base class 下的两个 derived-class 实体, 其中一个被赋值给另一个, 你认为会发生什么呢? 例如将 OrQuery object 赋值给 AndQuery object. 答案是: 这种赋值行为还是会失败, 因为两者之间并无上下 (父子) 关系.

继承机制得以解决“以透通方式代表一群型别”的问题, 但是继承并没有解决“以透明方式对这些型别加以编程”的问题, 这个问题需要所谓的动态绑定 (dynamic binding). 有了动态绑定, 当我们通过 Query object 调用 eval() 时, 也就是当我们执行 q.eval() 时, 真正调用的是 q 所指的那个实际 derived-class object 相应的那一份 eval() 函数实体, 而不是 Query class (基类) 对应的那一份 eval() 函数实体.

缺省情况下, 成员函数是通过静态绑定 (static binding) 来完成决议的, 也就是说, 在编译期依照调用者 (某个 object) 的型别来决定到底调用哪一个成员函数. 换言之, 缺省情况下, q.eval() 调用的是 q 的 class type (Query) 所拥有的那一份 eval().

如果要想某个成员函数通过动态绑定来进行决议, 也就是在运行期依照调用者 (某个 object) 的实际型别来决议, 我们必须将函数明白标上关键字 virtual, 否则函数会被视为 non-virtual. 在 C# 语言中, 面向对象设计的重要内容之一, 就是决定是否要对某些 base-class 的成员函数冠以 virtual. 请注意, static 成员函数并不支持动态绑定, 原因是你不能够通过 class object 调用 static 成员函数. (译注: 只能通过 class 名称来调用, 参见 2.9 节)

设计一个继承体系, 例如我们的 Query 继承体系, 需要两个主要设计步骤: (1) 提取 (factoring) 一组共用的操作集 (operations set), 放入抽象基类的接口 (interface) 中; (2) 确定此接口中的成员函数哪一些需要动态绑定 (也许全部都需要也不一定). 不过, 在设计并实现 Query 继承体系之前, 我们需要停下脚步, 回顾一下 C# 的 unified type system (统一型别系统).

3.4 关于 Object

前一节我们引入抽象基类 Query (其他 query types 都继承自此), 解决了令人困惑的“存储与传递 (storage and transmittal)”问题 (译注: 意指 p.126 顶部提出的两个问题).

举个例子，当我们把 `NotQuery` 的操作数声明为一个 `Query object` 时，此操作数就能代表所有 `derived query types` ——不仅是目前存在的，也包括将来引入的。这同时也解决了传递（transmittal）问题。每个函数参数如果声明为 `Query` 型别，就可以将任何 `derived query type object` 传递给它。

本节之中我们将关注一个类似问题，但并非独立 `classes` 之间的联系，而是独立于 `classes` 体系之间的联系。截至目前，本章已定义出两个 `classes` 体系：`LibMat` 体系和 `Query` 体系。第一章也让我们了解到 `System` 命名空间所提供的的一个 `Exception` 体系的一部分。每个 `classes` 体系都代表独立的一群型别。

现在，假设我们需要写一个函数，其参数可接受以上三个独立 `classes` 体系中的任何一个型别。当然，解决办法之一是定义一组（三个）重载函数：

```
class TheClassOfAllTypes
{
    public string ToString( LibMat m ){...}
    public string ToString( Query q ){...}
    public string ToString( Exception e ){...}

    // ...
}
```

这三个重载的成员函数可接受定义于 `LibMat`、`Query`、`Exception` 体系中的每一个 `class`（无论是目前存在的，还是将来加入的）。一旦引入一个新的 `classes` 体系或一个独立 `class`，并且也希望调用我们的 `ToString()` 成员函数时，问题就来了。哎呀，每当需要支持一个新的 `class` 或一群新的 `classes` 体系，我们就得回头针对它（们）添加一个新的重载函数版本。毫无疑问我们需要一个更好的解法。

这又是一个“传递与存储问题（transport and storage problem）”，不过这次牵涉的是独立的型别阶层体系，如 `Query.Query` 的各个子类均表示“一种（a kind of）”特殊的查询。其共同特征是查询（`query object`）行为的一般集合。

除了都代表程序的型别系统（`type system`）中的 `class` 型别，`LibMat classes` 体系和 `Query classes` 体系的操作行为并没有任何共通之处。它们之间唯一的“共同特征”是，程序员将“操作行为的一般集合”应用于运行期系统中的型别。

这个“传递和存储问题”的解法和先前类似：引入一个抽象基类（`abstract base class`），并让程序中的所有型别都派生自它。这个 `base class` 应该怎么称呼？它应

该提供哪些 public 操作（如果有的话）？

我把它称为 `TheClassOfAllTypes`，因为它代表的是程序所用到的所有型别的共通性。在 .NET 中，这个 class 被命名为 `Object`，定义于 `System` 命名空间中，是 .NET 型别体系的最根源（root），其他所有型别（例如 `Exception`, `string`, `int`, `LibMat`, `Query` 等等）都派生自 `Object`。C# 预定义的 `object` 型别则是 `Object` class 的一个别名。

由于所有型别都直接或间接派生自 `object`，所以型别为 `object` 的物体（entity）可以接受其他任意型别的 `object` 为初值，或说我们可以将其他任意型别的 `object` 赋值给型别为 `object` 的 `object`，这里的“其他任意型别”甚至包括整数常量和字符串之类的字面值（literal values）。这就解决了设计 `ToString()` 时遭遇的困难：如何定义一个函数，使能接受“所有当前型别和将来可能的型别”？是的，只需将参数型别声明为 `object` 即可：

```
class TheClassOfAllTypes
{
    static public string ToString( object o ) { . . . }
    // ... what else?
}
```

一个有趣的问题是，其他所有 classes 的 base class 应该提供什么样的成员函数呢？正如你所猜测的，`Object` class 提供的公开操作函数之一是 `ToString()`，之二是 `Equals()`，之三是 `GetType()`。系统中的每个型别都继承了这三个操作函数。我们可以在任何 objects 或任何字面值上调用 `ToString()`、`Equals()`、`GetType()`。当然，另一个有趣的问题是：这些函数真正做了什么？

“为所有可能的型别编写操作函数”的难处之一，就是它无法为任何特定 class 进行太多专属动作。这一点对于 `ToString()` 和 `Equals()` 尤其如此。是啊，我们怎么能够轻易实现出对任何可能型别都有意义的 *equality* 操作呢？判断两个 `int` 是否相等，和判断两个 `JulianCalendar` object 是否相等，根本是不同的两回事，需要不同的算法。

解决之道有两个方向。`Object` class 中的缺省实现，提供了所谓的“最小公分母（least-common-denominator）”解答方案。对于 `Equals()`，缺省实现是所谓的“reference equality”，也就是只有在参与比较的双方是同一个 `object` 时才返回 `true`。

对于 `ToString()`，缺省实现则是打印出型别的完整名称。这是我们所定义的每一个新 `classes` 都会经由继承取得的功能。

另一个解决方向是，将这两个成员函数声明为虚函数 (virtual functions)，这便使得每一个 `derived class` 有权选择覆写 (override) 缺省实现。例如，所有算术相关 `classes` 都覆写了 `ToString()`，用以打印 `object` 持有的数值，此外还覆写了 `Equals()`，用以实现“数值相等检验”——如果两个 `objects` 持有相同值，`Equals()` 就返回 `true`。

`Object class` 提供的第三个函数是个非虚函数 `GetType()`。此函数返回一个 `Type object`，其中封装了 `GetType()` 调用者 (某个 `object`) 的实际型别的所有信息，包括对该型别的属性 (properties)、成员函数、构造函数等描述。`Type class` 包含一组“判断属性” (predicate properties)：`IsClass`、`IsArray`、`IsPublic` 等等，用以回答型别相关问题。`GetType()` 是一个非虚函数，因为它提供的是“与型别无关”的操作，任何 `derived class` 都没有必要覆写之。`Type class` 是通向运行期型别反射 (runtime type reflection, 8.2 节主题) 的大门。

无论何时，只要你定义一个 `class` 而未曾明确为它指出 `base class`，这个 `class` 就暗中继承了 `Object class`。回头看看我们的 `Query` 继承体系，我们可能写出这样的声明：

```
class Query { ... }
class NameQuery : Query { ... }
class NotQuery : Query { ... }
class OrQuery : Query { ... }
class AndQuery : Query { ... }
```

从声明式来看，`Query` 似乎是个独立的 `class`。然而其实它在内部被扩充为“以 `Object` 为其 `base class`”：

```
// internally augmented to derive from System.Object  
class Query : System.Object { . . . }
```

Query 是每个 derived query type 的直接（上一层）base class，Object 则是每一个 derived query type 的间接 base class。这 5 个 classes 都继承了定义于 Object 中的所有 public 成员函数。

3.5 设计一个抽象基类 (Abstract Base Class)

从设计上说，独立的 class（例如 Matrix 和 Image）通常会有一个供应者和多个使用者。供应者设计并往往实现 class，使用者则采用供应者完成的 public 接口。这种分离模式所反映的便是：将 class 分割为 private（私有的）实现部分和 public（公开的）一组成员函数和属性（properties）。

一旦有了继承机制，或许出现多个 class 供应者，其中之一提供 base class 的实现（可能也会实现某些 derived classes），其他人则在整个继承体系的生命期间提供 derived classes。这其实也是一种实现活动。derived classes 的供应者常常（但不总是）需要访问 base class 的部分实现。为了允许这样的访问，并且防止对实现的一般性访问，C# 提供了另一种访问级别 protected。Class 的 protected 段落中的数据成员和成员函数都可被 derived class 访问，但都不能被一般程序访问。

评估一个 class 成员“是否应为 public”的标准并不因这个 class 是否作为 base class 而有所改变。问题是，一个 nonpublic 成员究竟应该声明为 protected 还是声明为 private 呢？凡是 base class 声明为 private 的成员，在其 derived classes 中是不可访问的。

如果我们觉得，为能够有效实现出 derived class，base class 应该提供必要的操作和数据，而这些操作和数据又不想对 class 继承体系的一般用户开放，那么就请把相应的 base class 成员设为 protected。例如 Query base class 的成员函数 display_partial_solution() 为每个 derived class 提供了公共服务，但并不想被 Query class 阶层体系的用户直接调用，如果我们将它声明为 public，就无法阻止

一般用户访问它。如果把它声明为 `private`，则不仅阻止了一般用户，也阻止了 `derived class` 访问它。`protected` 访问级别恰好合乎需要，它同时满足 `class` 用户的访问要求，也满足“想通过继承来扩展 `class`”的实现者的访问要求¹¹。

设计 `base class` 时应该考虑哪些问题呢？我们必须确定代表“整个 `classes` 体系”的一些 `public` 接口的操作和属性 (`properties`)，并确定其中哪些操作与型别相依 (`type-dependent`)，并令这些操作成为 `class` 体系中的虚函数。

没有什么魔法公式 (`magic formula`) 可以回答这些问题。即便回答了，也没有什么魔法公式来保证答案的正确性和完备性。面向对象设计 (`object-oriented design`) 的过程是反复迭代的，不断需要对改进中的 `class` 体系做些添加和修改。在本章的剩余部分，我们将经历 `Query` 体系的迭代演化过程。

3.6 声明一个抽象基类 (Abstract Base Class)

`Query` 表示一个抽象基类 (`base class abstraction`)，应用程序中操控的所有实际的 `query` 种类都从它继承而来。用户永远也不必创建实际的 `Query object`。事实上我们的应用程序中的 `Query object` 总是指向某个派生的 `query type` 实体。为了强调这种用法，我们可以在 `Query class` 定义式前加上关键字 `abstract`：

```
abstract public class Query { ... }
```

关键字 `abstract` 告诉编译器和程序读者：创建 `Query class` 实体是不可以的。以下语句尝试创建一个 `Query object`，便会触发一个编译错误：

```
// error: Query is abstract
Query q = new Query();
```

¹¹ `derived class` 究竟是“直接访问其 `base-class` 成员”抑或“通过 `base-class property` 来访问数据”？这个问题有些争议，一方面，“直接访问”导致 `base class` 与 `derived class` 紧密耦合 (`tight coupling`)，暗暗妨碍了 `base class` 的发展演化，或至少使得 `base class` 的发展演化十分困难。另一方面，`derived class` 实现者声称，直接访问成员，是“让程序达到可接受之性能”的必要措施。

确切地说, Query object 只能用来指向 (代表) derived-class query types object:

```
// OK: Query addresses a derived-class object
Query q = new AndQuery();
```

Query class 阶层体系的两个主要操作分别是“执行查询动作”以及“显示符合要求的文本行 (如果有的话)”。这两个成员函数分别取名为 eval() 和 display_solution()。

每个 derived-class query type 都有特殊的 eval() 实现, 毫无疑问 eval() 应声明为虚函数。base class 中应该怎么处理呢? 我们有两个选择: 提供缺省实现或只是提供一个抽象占位符。

如果成员函数具备饶富意义的缺省实现, 我们就提供成员函数的定义式, 并使用关键字 virtual:

```
abstract public class Query
{
    virtual public void eval()
    { /* default implementation */ }
}
```

关键字 virtual 表示, 这个成员函数“与型别相依”, 并指出这个成员函数的定义式将成为那些“没有提供相应函数实体”的 derived classes 的一份缺省实现。

derived class 可以自行实现 (或称覆写) 继承而来的 base class 的成员函数。然而我们并不要求 derived class 一定得覆写虚函数, 因为 base class 成员函数的实现也可能适用于 derived class。这种情况下 derived class 不必提供一份自己的函数实现, 只需复用 (reuse, 或说继承) base class 的实现即可。例如当我们定义一个 class, 可以选择覆写 Object base class 的 ToString() 虚函数, 或复用 Object class 的缺省实现 (译注: 也就是返回 class 全名)。

然而如果 base-class 的某个成员函数不具备带有意义的缺省实现, 我们就应该把这样的成员函数声明为 abstract, 并且不为它写出函数体:


```
abstract public class Query
{
    // no function body is provided ...
    abstract public void eval();
}
```

一个 `abstract` 函数被视为一个 `virtual` 函数, 其声明式表示, 这个函数对 `base class` 而言不存在富有意义的实现, `abstract` 函数就像一个占位符, 只引入接口: 成员函数的名称、访问级别、标记式 (signature)、返回型别, 具体实现则由每个 `derived-class` 提供。

面对一个 `abstract` 函数, `derived-class` 没有可继承的缺省实现。因此, 如果 `derived class` 没有为它提供一个定义式, 那么 `derived class` 就依然是个抽象类, 不能直接创建实体。

为什么需要引入 `abstract derived class` 呢? 是的, 它提供了一种细分 (subpartitioning) `class` 体系的方法。例如, 我们可以修改我们的 `class` 体系, 让 `AndQuery class` 和 `OrQuery class` 从 `abstract BinaryQuery class` 派生而来, 这让我们得以将二元查询 (binary queries) 的所有共同成员 (包括数据和操作函数) 提取出来, 放入单一的共享 (shared) `class` 中:

```
abstract public class BinaryQuery : Query
{
    // inherits the abstract eval() method
}

public class AndQuery : BinaryQuery
{
    // OK: an implementation of eval() here
}
```

我们称 `Query class` 和 `BinaryQuery class` 都是阶层体系中的抽象类 (abstract class), 而 `AndQuery` 和 `OrQuery` 则是应用域中的具象类 (concrete class, 具体类)。

引入或继承抽象成员函数 (或抽象 `property/indexer`) 的任何 `classes`, 声明时都必须冠以关键字 `abstract`。如果忘了指定关键字 `abstract`, 会触发编译错误。

引入 `abstract BinaryQuery class` 是更好的设计吗？在了解如何使用 `AndQuery` 和 `OrQuery classes` 之前，我们不能下此结论。如果有这么一种情况：我们将这两个查询作为一个整体（二元查询）来操控，或如果二元查询有一组独一无二的共用操作，而其行为对其他 `query types`（例如 `NotQuery`）并不适用，那么引入 `abstract BinaryQuery class` 就是合理的。这种对 `classes` 体系的重构（`refactoring`）行为在整个 `classes` 体系的生命发展期间会经常发生。

属性（`property`）和索引器（`indexer`）也可以声明为 `abstract` 或 `virtual`。让我们看一个实例。

与 `Query object` 相关的一个 `property` 是：本程序的 `solution set`（解集），也就是满足用户查询条件的一些文本行号，以整数 `array` 表示。`solution set` 存在与否，取决于 `query object` 是否调用了 `eval()`。如果 `solution set` 不存在，这个 `property` 必须返回 `null`，否则就返回一个 `handle` 指向 `solution set`。我们允许用户读取 `solution set`，但不能让用户修改它，因此我们只写出 `get` 访问器：

```
abstract public class Query
{
    virtual public int [] Solution
    {
        get
        {
            return null;
        }
    }
}
```

关键字 `virtual` 或 `abstract` 是作用于 `property`（或 `indexer`）整体之上，不是作用于单个访问器身上。如果 `property` 声明为 `abstract`，那么访问器就不要提供任何代码：

```
abstract public class Query
{
    abstract public int [] Solution
    {
        get;
    }
}
```


indexer (索引器) 的情况和 property (属性) 差不多, 例如:

```
abstract public class NumericSequence
{
    abstract public int this[ int index ]
    {
        get;
        set;
    }
}
```

3.7 抽象基类 (Abstract Base Class) 的 static 成员

被用户查询的那个文本文件呢? 文本文件只有一份实体 (如果程序支持“多文本文件”, 那么每一个文本文件只有一份实体), 所以我们将文本文件以 static 成员表示, 并附带数个起支持作用的 static 成员和一个 indexer (索引器):

//译注: 原书此处有多处语法错误, 已修正如下

```
abstract public class Query
{
    static private string[] ms_textfiles;
    const private int ms_maxFiles = 24;

    static protected void check_index( int ix ) { ... }
    static public void add_file( string name ) { ... }

    static public int MaxFiles { // 译注: 这是 property
        get{ return ms_maxFiles; }
    }

    public string this[ int ix ] // 译注: 这是 indexer
    {
        get{ check_index( ix ); return ms_textfiles[ix]; }
        set{ check_index( ix ); ms_textfiles[ix] = value; }
    }
}
```

static method、static indexer 或 static property 都不能声明为 virtual 或 abstract (译注: 事实上不存在所谓 static indexer)。构造/析构函数也不能声明为 virtual。

实际上到底存在多少份 static ms_textfiles (或 static ms_maxFiles) 实体? 不论 base class 派生出多少 classes, 这实体仅有一份。甚至在我们提供四个 Query derived-class 之后, base class 的每个 static 或 const 数据成员还是仅存在一份实体。

我将两个成员声明为 `private`。要知道, `base class` 的 `private` 成员不能在 `derived class` 中被直接访问, 这意味着每个 `derived-class` 都必须通过 `public static property` 或 `public index` 才能读取 (或涂写) 这个 `private` 数据成员。

`private base-class` 成员的好处是, `base class` 与后来的 `derived classes` 之间呈现一种松耦合 (`loose coupling`) 关系。是的, 如果允许 `derived classes` 直接访问 `base class` 的成员, 那么对此成员的任何改动, 都可能破坏那些直接访问该成员的 `derived classes`。

上述做法的潜在缺点是, 为 `derived classes` 的性能 (效率) 带来不易被接受的额外开销, 因为 `derived class` 没有能力直接访问关键性的 `base-class` 成员。解决办法是将成员声明为 `protected`, 那么就既允许 `derived class` 直接访问成员, 又能阻止一般用户访问此成员。

3.8 混合型抽象基类 (Hybrid Abstract Base Class)

译注: 以下请注意, `instance` 成员就是 `non-static` 成员, 相对于 `static` 成员。

每个 `derived-class` `query type` 的 `solution set` 数据成员都一样: 一个整数 `array`, 指出符合查询条件的文本行号码。在最初的设计中, `Query class` 定义了一个 `abstract property` 来封装 `solution set`:

```
abstract public class Query
{
    abstract public int [] Solution ( get; )
    // ...
}
```

每个 `derived class` 都定义有实际的 `solution set` 数据成员, 并为 `Solution property` 的 `get` 访问器提供一份实现, 例如:

```
public class NotQuery : Query
{
    private int [] solution_set;
    virtual public int [] Solution
    {
        get { return solution_set; }
    }
    // ...
}
```


考虑以下代码片段，其中 `parseQuery()` 是个工具函数，用来将用户的查询字符串转换为对应的 `class` 表述。它返回一个指向实际 `derived-class query type` 的抽象 `Query` 节点：

```
Query theQuery = parseQuery( userQueryString );
theQuery.eval();
int [] solution = theQuery.Solution;
```

以上代码中，`eval()` 和 `Solution` 都是在运行期通过虚函数机制被调用。毫无疑问，`eval()` 和实际的 `query type` 相依，因此虚拟机制 (`virtual mechanism`) 是唯一可行的解决方案。但是将 `Solution` 声明为 `abstract`，是否必要呢？

每个 `derived query types` 实现 `Solution` 的方法都一样，其 “instance 成员” `solution_set` 的型别也都一样。既然如此，另一种设计方案便是，将这些共用的 “instance 成员” 和 `property` 抽取出来，整合到 `abstract Query base class` 中：

```
abstract public class Query
{
    abstract public void eval();
    protected int [] solution_set;
    public int [] Solution
        { get { return solution_set; } }

    // ...
}
```

这样的重构 (*refactoring*) 行为并不会破坏原程序的兼容性。以下的调用动作：

```
int [] solution = theQuery.Solution;
```

会正确返回 `derived query object` (由 `parseQuery()` 返回) 相应的 `solution_set`。不同之处在于，这一调用动作的决议是在编译期静态进行的。

在 `base class` 中引入 “instance 成员”，牵连到许多设计上的问题。比如说 `Query class` 因此需要一个构造函数。但奇怪的是这个 `class` 依旧是 `abstract base class`，你不能为它创建任何 `object`。下面数小节将谈谈这方面的问题。

3.8.1 单一继承下的对象模型 (Object Model)

Derived class 继承了 base class 的所有成员, 也继承了 base class 的 base class 的所有成员。记住: 所有 classes 都是从 Object 派生而来。实际上每个 derived-class object 包含由 base class 的所有“instance 成员”构成的 base class subobject (子对象)。例如一个 NotQuery object 内含一个 Query subobject, 而一个 Query subobject 又内含一个 Object subobject。你可以把不同的 subobjects 想象为“一块叠着一块”的一堆乐高积木。每一堆积木的最底部 (base) 就是 Object subobject。

尽管“把每个 base class 想象为 derived-class object 内独立的 subobject”可能会让人感到困惑, 但这可以为单一继承 (single inheritance) 提供有效模式:

- 就初始化而言, 运行 derived-class 的构造函数之前, 相应的 base-class 构造函数会自动施行于 subobject 之上。例如创建一个 NotQuery object 时, 首先会调用相应的 Object 构造函数, 随后是 Query 构造函数, 最后才是 NotQuery 构造函数。(3.9 节会介绍如何将引数传递给上一层 base-class 构造函数)
- 就型别转换而言, 由于 derived-class object 内含 base class 的完整实体, 所以型别转换并不需要任何运行期支持。这种型别转换将会丢失对于 derived-class 的非虚接口 (non-virtual interface) 的了解, 例如:

```
NotQuery nq = new NotQuery( ... );
Query oper = nq.Operand; // OK

// still a NotQuery object,
// but now only the Query interface can be invoked
Query q = nq;

// OK: eval() is part of the Query interface,
//      and it is virtual: it invokes NotQuery.eval()
q.eval();

// OK: Solution is part of the Query interface,
//      but it is nonvirtual: it invokes Query.Solution
int [] solution = q.Solution;
```



```
// error: Operand is part of the NotQuery interface;
//      it cannot be invoked through Query
Query op = q.Operand;

// still a NotQuery object,
// but now only the Object interface can be invoked
Object o = q;

// error: eval() is part of the Query interface;
//      it cannot be invoked through Object
o.eval();
```

现在,我要对先前说过的话作一些修正.我曾经声称 `derived class` 可以直接访问从 `base class` 继承而来的成员,但这其实只是部分成立:尽管 `derived class` 继承了其 `base class` 的所有成员,但 `derived class` 不能访问继承而来的 `private` 成员.事实上 C# 不允许函数被同时声明为 `virtual` 和 `private`.

如果 `base-class` 的 `private` 成员不能被 `derived class` 访问,为何语言要自找麻烦将这些成员继承下来,尤其是这样的继承占据了每一个 `derived-class object` 的空间?原因是为了维持 `base-class subobject` 的完整性.虽然我们不能直接访问 `private` 成员,却常常需要间接访问它(通过 `base class` 的 `property`、`indexer`、`constructor`、`method` 等各种途径),因此它应该在那儿.

3.8.2 混合型抽象类 (Hybrid Abstract Class) 有何特别?

一旦在 `abstract base class` 中引入 `instance` 成员,就必须让用户控制这些成员的初始化行为.也就是说有必要引入构造函数,而且不限一个.例如:

```
abstract public class Query
{
    protected Query( int [] aSolution ) { ... }

    protected int [] solution_set;
    public int [] Solution
    { get { return solution_set; } }
    // ...
}
```

请注意它的构造函数声明为 `protected` 而非 `public`。是的, `Query object` 只打算作为某个 `derived query types` 的 `subobject`, 因此如果其构造函数声明为 `protected`, 那么就只有 `derived query types` 可以创建 `Query` 实体。很好。

藉由将共用的 `instance` 数据抽出重整 (*refactoring*) 到 `base class` 中, 我们消除了支持“访问这些数据成员的 `property`”的抽象特性。这主要有两个好处:

1. `Solution property` 运行起来要快得多。程序不必等到运行期才进行决议 (*resolution*), 才决定调用哪一份 `Solution` 实体。程序可以在编译期就决议出应该被调用的实体。结构如此简单的 `get` 访问器 (*accessor*) 几乎可以确定将在每个调用点被就地展开 (*be expanded inline*), 这就完全去除了调用函数时的额外开销。
2. `derived-class` 的实现得以简化。在纯抽象设计中, 每个 `derived class` 除了必须为自己提供独特的算法之外, 还要提供共享的基本设施, 用以存储和返回该型别。在混合型设计中, `derived class` 的设计者继承了这些基本支持设施。每个 `derived type` 的设计因此得以简化。

提高性能的意义大吗? 这取决于调用 `method` 或 `property` 的频繁度。如果应用程序的关键部分高度频繁地调用 `Solution`, 上述的设计改变就十分有意义。否则就可有可无。

同样道理, 简化 `derived-class` 的实现, 未必真的有意义。如果 `abstract base class` 的设计者同时也提供 `derived-class query types`, 而且 `derived classes` 集合并不预期需要经常扩大, 那么混合型设计就没多大意义。但如果 `classes` 阶层体系的主要活动就是为了一再引入新的 `query class types`, 并且此项工作已被委托给熟悉图书管理 (而非编程) 的人士去做, 那么重构之后的设计显然更好。

规则到底如何? 不, 这里没有规则, 仅仅只是一种设计上的可选方案罢了。`Abstract base class` 能够合理地定义 `Instance` 数据成员和 `nonvirtual` 成员函数。这便是所谓的实现继承 (*implementation inheritance*)。如果你在 `COM` (`Component Object Model`) 编程模型中长大, 这种“实现上的重构” (*implementation refactoring*) 不

会是你熟悉的选择。然而在 C# 和 .NET 之下，确实有这种做法。甚至 Object 都提供了 `nonvirtual GetType()`。

3.9 定义一个派生类 (Derived Class)

一般说来，`derived class` 只需为某些“不同于 `base class`”的行为或“对 `base class` 进行扩充”的行为编写程序。例如 `NotQuery` 必须提供一个 `eval()` 定义式用以实现 `Not` 语义。它也必须引入一个 `instance` 成员来存储表示否定语义的 `query` 操作数。如果合适的话，还需引入一个 `property`，对此操作数实施 `get` 和 `set` 访问。`AndQuery` 和 `OrQuery` classes 都得支持左操作数和右操作数。`NameQuery` 得拥有 `string` 成员。这些 classes 都必须提供 `abstract virtual eval()` 的定义。

`Derived class` 如果引入 `abstract` 成员，或是没有为继承而来的 `abstract` 成员提供实现，就被认为也是个 `abstract`。这两种情况下你都必须将它明白标示为 `abstract class`。以下样例展示 `BinaryQuery` `derived class`。由于它没有提供 `abstract eval()` 的实现，因此它也是一个 `abstract class`：

```
abstract public class BinaryQuery : Query
{
    protected BinaryQuery( Query leftOp, Query rightOp,
                           int [] solution )
        : base( solution )
        ( m_lop = leftOp; m_rop = rightOp; )

    protected Query m_lop, m_rop;
    public Query LeftOp
    { get{ return m_lop; } set{ m_lop = value; } }

    public Query RightOp
    { get{ return m_rop; } set{ m_rop = value; } }
}
```

`BinaryQuery` 构造函数在冒号之后用上了关键字 `base`，表示将以 `solution` 为引数，调用 `Query` `base-class` 构造函数。`base-class` 构造函数在 `derived-class` 构造函数体运行之前就被调用。这样的调用顺序保证继承而来的 `base-class` 成员的初始化动作一定发生在 `derived-class` 构造函数体之前——为的是 `derived-class` 构造函数体有可能用到 `base-class` 成员。

C# 不允许构造函数的“调用清单 (invocation list)”中同时包含 `base` 和 `this` 两个部件 (constructs)。如果两个都要，我们必须将它们分入各自的构造函数组，就像以下的 `Point3D` class 所示：

```
public class Point3D : Point2D
{
    // cannot have base() and this() in one constructor
    public Point3D( float x, float y, float z )
        : base( x, y ){ m_z = z; }

    public Point3D() : this( 0.0F, 0.0F, 0.0F ){}

    public Point3D( float x ) : this( x, 0.0F, 0.0F ){}
    public Point3D( float x, float y )
        : this( x, y, 0.0F ){}

    // ...
}
```

那么 `AndQuery` derived class 呢？它从它的直接（上一层）`abstract BinaryQuery` base class 继承了两个操作数，从更上层的 `Query` base class 继承了 `solution set` 的支持设施。看来需要做的也就只有实现继承而来的 `abstract eval()` 吧。唔，差不多。

另外还有两个需要考虑的事项。第一个要注意的是“我们从 base class 继承了所有东西”这一规则。实际上 `derived class` 并没有继承其 `base class` 的构造函数(s)。因此即使 `derived-class` 构造函数所要做的只不过是“接受一组参数并传递给 base class 的构造函数”，也得明确（显式）实施之。也就是说，它与 `base-class` 构造函数(s)完全没有继承关系。

第二个要注意的是，`derived class` 必须为它所允许的形形色色的初始化动作定义一些构造函数。例如 `AndQuery` class 允许以如下两种方式创建实体：(1) 传入两个型别为 `Query` 的操作数；(2) 传入两个操作数加上一个整数 `array`：

```
public class AndQuery : BinaryQuery
{
    public AndQuery( Query leftOp, Query rightOp, // 译注：上述形式(2)
        int [] solution )
        : base( leftOp, rightOp, solution ){}

    public AndQuery( Query leftOp, Query rightOp ) // 译注：上述形式(1)
        : this( leftOp, rightOp, null ){}
}
```



```
public virtual eval() { ... }  
// ... anything more ???  
}
```

尝试以其他引数组合来创建 `AndQuery` object, 都会导致编译错误。即便不以任何引数创建 `AndQuery` object 也不行:

```
// OK: invokes the associated constructor  
Query q1 = new AndQuery ( new NameQuery( "Civil" ),  
                           new NameQuery( "War" ) );  
  
Query q2 = new AndQuery ( q1, q1, q1.Solution );  
  
// error: no constructor provided to support this  
Query q3 = new AndQuery();
```

无引数的 (no-argument) 构造函数受到编译器的特殊对待。如果你没有为 class 提供任何构造函数, 编译器会自动为它提供一个无引数构造函数。但如果我们定义了一个或多个构造函数, 编译器就不再自动提供无引数构造函数。

3.10 覆写继承而来的虚接口 (Virtual Interface)

Derived class 可以继承或覆写 (override) 其 base class 的虚接口 (Virtual Interface)。如果想要继承某个成员, derived class 不需做任何事, 因为“继承”是一种缺省行为。但如果想要覆写继承而来的 abstract 或 virtual 成员, 我们必须在定义式前标上关键字 `override`, 否则编译器会将这一实体视为独立定义, 只不过是重用 (reuse) 了继承而得的成员名称罢了。如果你这么做, 无论有意或无意, 编译器都会给你警告。如果不想看到这个警告 (也就是说你确实想重用这个成员名称), 可以使用关键字 `new` 把它关掉, 3.12 节对此有些介绍。

Base class 与 derived class 的虚函数实体必须具备相同的访问级别、相同的标记式 (signature), 以及相同的返回型别¹²。举个例子, 以下不是合法的覆写行为,

¹² C# 不支持 covariance return type (协变式返回型别)。也就是说, 如果 base class 的虚函数返回 base class object, 那么 derived class 中的对应函数实体也必须这么做。Derived class 的对应函数实体不能声明传回一个 derived-class object。 (译注: C++ 改变了这条规则, 可参考 *C++ Primer*, p924, 或 *Modern C++ Design*, p212)

由于 derived class 实体有一个 ref string 参数，因而形成与 base class 不同的调用语法：

```
public class Base
{
    public virtual void display( string msg ) { ... }
}

public class Derived : Base
{
    // error: signatures differ by ref keyword
    public override void display( ref string msg ){ ... }
}
```

上述 derived class 的那一份 display() 实体会导致编译错误，因为关键字 override 会要求编译器寻找吻合的继承实体，而编译器却找不到这样的实体。

覆写 property 时，我们必须指明相同的 property 型别，访问器 (accessors) 也不能有任何差异。索引器 (indexer) 也是如此——索引的型别和数目都不能有所出入。

3.11 覆写 Object 的虚函数 (Virtual Methods)

设计 derived class 时，需要考虑的第二个问题是，每一个 class 从 Object 暗中 (implicitly) 继承而来的三个虚函数 (译注：GetHashCode(), ToString(), Equals())。还记得吗，缺省情况下 ToString() 打印出 class 全名。通常如果我们能够利用它显示 class 的内部状态，可能更有用。例如下面这个 OrQuery class 的 ToString() 实现，其中的 m_lparen 和 m_rparen 成员用来追踪 query 语句中出现的括弧：

```
override public string ToString()
{
    StringBuilder sb = new StringBuilder( 8 );

    if ( m_lparen != 0 )
        gen_lparen( ref sb, m_lparen );
```



```
sb.Append( m_lop.ToString() );  
sb.Append( " || " );  
sb.Append( m_rop.ToString() );  
  
if ( m_rparen != 0 )  
    gen_rparen( ref sb, m_rparen );  
  
return sb.ToString();  
}
```

虽然 ToString() 并没有限制我们如何使用它，但它主要用来辅助调试 (debugging)。无论如何，许多用户很不愿意为了非调试目的而调用 ToString()。基于这个理由，你不如考虑提供 Display() 或 Print() 函数。

为什么我使用 StringBuilder class object 而不直接生成一个 string 呢？这纯粹是为了效率，string object 是不可变的 (immutable)，每次改动 string，其实都会导致产生新的 string object。举个例子，如果要直接生成一个 string object 用以表现以下的 OrQuery:

```
(( alice && emma ) || weeks )
```

会导致产生 9 个临时性的 string objects——每插入一个新元素就产生一个新的 string object。StringBuilder 则是可变的 (mutable)，这使我们得以生成一个 string 而不产生多个临时实体。完成对字符串的修改后，我再以 StringBuilder 的 ToString() 从 StringBuilder object 中取出字符串来。

缺省情况下，Equals() 实现出 “reference 相等” 语义，也就是只有在 “两个被比较的 objects 实际上是同一个 object” 时才返回 true。一般说来当我们实现一个 class 时，我们会覆写 Equals() 以表现出 “value 相等语义”，也就是当 “两个独立的 class objects 包含相同内容” 时就返回 true。

3.12 成员访问：new 修饰符和 base 修饰符

从 base class 继承而来的成员看来就好像是 derived class 的成员。这就是我们能够直接使用它而无需资格修饰 (qualification) 的原因。然而它们毕竟不是 derived class 的成员。这些成员与它们所在的 base class 的声明空间 (declaration space) 有着密切关联。

当 derived class 之中出现一个名称时, 编译器会先搜索 derived class 的声明空间, 企图找出其定义。如果没有找到, 再搜索 base class 的声明空间。这就是为什么 derived class 可以毫无障碍地复用(reuse)其 base class 的成员名称的原因。derived class 复用的那一份实体其实是进入了 derived class 声明空间之中。在 derived class 内, 对此名称的所有未经资格修饰(unqualified)的运用场合, 都会被决议为 derived class 的实体。这时候我们称 derived instance 遮掩(hide)了继承而来的 base class 实体。例如:

```
abstract public class Query
{
    public void func1( int i ) { ... }
    public virtual void func2( int ival )
        { func1( ival ); }
    // ...
}

public class OrQuery : Query
{
    // necessary because this definition
    // hides the inherited base-class member
    new public void func1( int i ) { ... }

    public override void func2( int ival )
    {
        // OK: we want the hidden base-class member
        //   invoked here
        base.func1( ival );    // 译注: qualified reference

        // unqualified reference invokes the
        // OrQuery instance ...
        func1( ival );
    }
    // ...
}
```

至此, 但愿你能明白是怎么回事儿。Base class 和 derived class 之中分别定义了一份 func1() 函数实体, 它们不是虚函数, 它们具有相同的标记式(signature), 所以 derived class 那份实体“遮掩”了 base class 那份实体。为了告诉编译器和阅读这份程序代码的人, 告诉他们这样的“重新命名”是故意的, 我们以关键字 new

修饰 derived-class 实体。在这种情况下, 将成员的定义式施以关键字 new, 表示它将遮掩继承自 base class 的同名实体的“无饰词访问”(unqualified access)。如果不写 new, 编译器会发出警告, 因为它不能确定我们是否意识到这里遮掩了 base class 的成员。

在这种情况下, 如果我们在 derived class 中访问 base class 成员, 必须以关键字 base 修饰其名称:

```
base.func1( ival );
```

这一行将调用 base class 所定义的成员函数 func1(), 本例中调用的是 Query class 定义的 func1()。如果 Query 没有定义 func1(), 此句会被编译器视为错误。

在 derived class 之外, 我们不能使用关键字 base。没有任何语法支持“通过 derived class object 来访问被隐藏的 base class 的成员”。如果绝对需要这么做, 我们可以求助显式转型 (explicit cast):

```
static public void Main()
{
    OrQuery or = new OrQuery();

    or.func1( 1024 );           // OrQuery.func1( int )
    ((Query)or).func1(1024);    // Query.func1( int )

    // ...
}
```

由于 func1() 不是虚函数, 因此通过 Query object 调用 func1(), 总是被编译器决议为“Query 只能定义”的那个成员, 即便 Query object 实际指向型别为 OrQuery 的 object, 也是如此。例如:

```
Query q = new OrQuery();
q.func1( 1024 );           // nonvirtual: Query.func1()
q.func2( 1024 );           // virtual: OrQuery.func2();
```

一般说来, 对于那些“非虚函数复用了相同名称”的 class 体系设计, 你应该怀疑其价值。用户会因而搞不清楚到底调用了哪一份函数实体, 真的。

如果某个 derived class 内有 virtual 函数为 (继承而来的) abstract 函数提供了定义 (即 virtual 函数覆盖 (override) 了 abstract 函数), 那么这个 virtual 函数无需指定关键字 new, 因为此处并没有“遮掩”情况发生。是的, 虚拟机制会在运行期依据“调用函数时所凭借的那个 object 的实际型别”来调用相应的函数。

3.12.1 可达性 (Accessibility) 与可见性 (Visibility)

请考虑以下 class 体系, 其中 base class 与 derived class 都定义了一些成员:

```
class aBase { public string s; }  
class aDerived : aBase { new private string s; }
```

在 base class 中用到的所有 s, 都代表其 public string 成员。同样道理, 在 derived class 中引用的所有未经资格修饰的 s 都代表其 private string 成员。我们可以在 derived class 中运用关键字 base 引用其 base-class 的成员:

```
base.s; // refers to aBase.s
```

上例的关键字 new 用来提醒编译器 (以及程序代码的读者): derived class 的设计者打算在 derived class 的声明空间中隐藏 base class 的成员 s。

现在我们再派生出另一个层级:

```
class aMostDerived : aDerived  
{  
    // OK: which member 's' is assigned?  
    public void foo( string str ) { s = str; }  
}
```

这个新的 class 继承了 s 的两份成员实体。如果在 aMostDerived class 中未经资格修饰地使用 s, 会被编译器决议为哪一份实体呢? 是隶属 aBase 还是隶属 aDerived 的那一份?

在 C++ 里, 对于“所用成员”的决议动作是在考虑其可达性 (accessibility, 可访问性) 之前进行的。C++ 首先检查此成员函数的最直接生存空间 (immediate scope), 然后考虑这一成员函数所属 class 的生存空间, 最后是外围的 base class 生存空间。因此在 C++ 中, aDerived class 的那一份 s 实体会被选中, 而后会产生错误, 因为 aMostDerived class 里不能访问继承而来的 private 成员 (而 s 正

是如此)。这样做的目的是当一个成员的访问级别有所改变时, 不会影响程序的意图。

在 C# 中, 对于所引用的名称的决议行为, 是在“考虑了可达性”之后才发生的。也就是说, 虽然 aDerived 中的 s 的实体是直接(上层) derived-class 的那份实体, 但由于它是 private (因而无法被访问), 所以它不被编译器视为名称决议的候选者。结果, 选中的是 aBase 中的那一份实体。如果 classes 设计者稍后将 aDerived 的那一份 s 实体重新声明为 protected, 那么就会改而选中 aDerived 内的那份实体。

3.12.2 将“对基类(Base Class)的访问”封装起来

考虑以下简化了的 Point/Point3D class 体系, 坐标名称 (x_、y_、z_) 横跨 base class 和 derived class:

```
class Point
{
    protected float x_, y_;
    public Point( float x, float y ){ x_ = x; y_ = y; }

    public virtual void display()
        { Console.Write( "{0}, {1}", x_, y_ ); }

    // ...
}

class Point3D : Point
{
    protected float z_;
    public Point3D( float x, float y, float z )...
    public override void display(){ ... }
}
```

首先, 我们该如何实现 Point3D 构造函数呢? 基本上有两种选择:

1. 以关键字 base 将坐标 x、y 传入 Point 的构造函数中:

```
public Point3D( float x, float y, float z )
    : base( x, y ) { z_ = z; }
```

2. 在 derived class 的构造函数中直接初始化这三个数据成员:

```
public Point3D( float x, float y, float z )  
    { x_ = x; y_ = y; z_ = z; }
```

两种实现都能将数据成员正确地初始化。然而我们总是应该更喜欢第一个方案，也就是让 base class 的构造函数负责初始化 base class 自己的数据成员。

这么做的主要理由是，维持 base class 和 derived class 之间的松耦合 (loose coupling) 关系。这么做的话，如果 base class 的实现有所改变，derived class 不会因而诡异地变得无法编译。举个例子，图形程序员常常争论的焦点之一就是，将坐标成员 (coordinate members) “独立存储” 还是 “将它们存入数组” 比较好。我真的见过有些实现方案在上述两种做法中改来改去——随着负责维护那些 classes 的程序员而异动而变迁。

接下来，我们该如何实现 Point3D 的那份 display() 实体呢？我们所面临的选择十分相似：(1) 直接打印全部三个成员；(2) 调用 base class 的 display() 函数来处理 base class 的成员，再打印出 derived class 自己的成员，也就是坐标 z。

首选方案是将责任局部化 (缩小责任范围)：谁的成员，谁负责显示。derived class 的实体只要关心如何显示它自己的成员就好：

```
class Point3D : Point  
{  
    public override void display()  
    {  
        base.display();  
        Console.Write( ", {0}", z_ );  
    }  
    // ...  
}
```

在这里我们必须以关键字 base 调用 Point 的 display() 实体。未经资格修饰而调用的 display() 将被编译器决议为 Point3D 自己的那份实体，导致无穷递归。如果有了关键字 base 指明到底调用哪一份 display()，调用行为的决议动作就会由编译器完成 (而不再凭借虚拟机制)。

3.13 将 Class 密封起来

如果要明确地防止从某个 class 派生新的 class，可以指定关键字 `sealed`：

```
sealed public class BinaryTree { ... }
```

如果你尝试继承一个 `sealed class`，会触发编译错误：

```
// error: BinaryTree is a sealed class  
public class RedBlackTree : BinaryTree { ... }
```

关键字 `sealed` 不能用于 `struct`，因为后者已被隐式声明为 `sealed`；亦不能用于 `abstract class`，因为后者要求其继承下去的 `classes` 必须提供具体实现。

我们为什么需要将 `classes` 密封起来？当然有其哲学性的一面，主要是基于我们对于这个 `class` 的运用上的认识。

不幸的是，人类感知的敏锐度有限。例如下一章我需要扩展 `BitArray` 群集类（`collection class`），而其实现者却已经将它声明为 `sealed`，阻止被任何人继承。这真是个问题。哲学自身是个拙劣的战略家！

将 `class` 密封起来，应该有更好的理由才是。下面是个好理由：可以提高我们的 `class` 的性能。

通常程序中如果调用虚函数或虚访问器（`virtual accessor`），这些动作必须直到运行期才能获得决议，因为“调用虚函数时所凭借的 `object`”自身可能并不指向其所隶属之 `class`，而是指向派生下来的某个未知的 `class`。如果我们密封了 `class`，便可以消除“调用行为的不确定因素”。

通过 `sealed class object` 而调用的虚函数可以在编译期被静态决议。这时候不仅不需要虚拟机制，还有机会做内联展开（`inline expansion`，一种编译器优化行为），可以显著提高被频繁运用（像是 `string` 和 `collection classes`）时的性能。

当某个抽象性（`abstraction`）需要最理想的性能（效率），但又不适合作为 `value` 型别时，那么 `sealed class` 是可供选择的另一种设计。`Sealed class` 仍旧在受控堆（`managed heap`）内创建 `object`，但消除了虚接口（`virtual interface`）。

3.14 Exception 继承体系

异常 (exception) 必须表现为一个 `Exception class object`, 或是一个从 `Exception` 派生的 `class object`. .NET 框架提供了许多预定义的异常类, 此外, 我们也可以派生出自己的异常类.

让我们考虑成员函数 `find()`. 它接受两个参数: 一个 `string array` 和一个 `string`. 只要其中之一是 `null`, `find()` 就抛出预定义的 `ArgumentNullException`:

```
public static bool find( string [] table, string item )
{
    if ( table == null || item == null )
    {
        Exception e = new ArgumentNullException();
        if ( table == null )
        {
            e.Source = "Argument One";
            if ( item == null )
                e.Source += " and Argument Two";
        }
        else e.Source = "Argument Two";
        throw e;
    }

    // ...
    return false;
}
```

`Exception class` 支持一大堆有用属性 (properties), 包含以下这些:

- `TargetSite`, 拥有“只读” (read only) 属性, 持有异常抛出者 (某个成员函数) 的名称. 在上述代码实例中它被设为 `Boolean find(System.String[], System.String)`.
- `Source`, 缺省值为“异常抛出者” (某个装配件, assembly) 的名称. 我们可以亲自动手, 明确地为它赋值. 上例之中它被设为 *Argument One* 或 *Argument Two* 或 *Argument One and Argument Two*.
- `Message`, 各个异常类的缺省消息 (message). 上述代码实例中所使用的 `ArgumentNullException class` 将 `Message` 设为缺省值 *Value cannot be null*. 我们只能在构造函数中改变这个缺省消息, 因为它拥有只读 (read only) 属性.

- `StackTrace`, 拥有只读 (read only) 属性, 负责跟踪抛出异常当时的调用栈 (call stack), 其中包含路径、文件以及代码行号等信息, 例如:

```
at EntryPoint.find(String[] table, String item)
in c:\c#programs\exceptions\class1.cs:line 17

at EntryPoint.Main()
in c:\c#programs\exceptions\class1.cs:line 29
```

- `InnerException`, 拥有只读 (read only) 属性, 其中持有内部异常 (如果有的话) 的 reference。

你可能不熟悉所谓“内部异常” (inner exception)。假设 `find()` 被名为 `queryText()` 的成员函数调用, 像这样:

```
public static void queryText()
{
    try
    {
        if ( find( get_text_array(), get_item() ))
            // ...
    }
    catch( ArgumentNullException ane )
    { /* what should we do here? */ }
}
```

于是 `queryText()` 会发出 `ArgumentNullException`, 指示 `get_text_array()` 或 `get_item()` (以下称此二者为“取用例程 (retrieval routines)”) 运行失败。我们可以检查所捕获的异常的 `Source` 属性, 确定到底是哪个函数运行失败。

然而, `queryText()` 调用端可能更加关心两个“取用例程”的运行失败问题, 而非 `find()` 的 `null` 引数问题。因此我们应该在 `queryText()` 内抛出一个意思更加明白的异常, 同时也返回原始异常。这就得借助于内部异常。

换言之, 内部异常是“只被 `catch` 子句进行部分处理”的异常。开发者并非重新抛出 (rethrowing) 相同的异常, 而是创建一个新的、提供更多信息的异常。创建新的异常对象 (exception object) 时, 应该将原始异常传给新异常的构造函数, 作为新异常的 `InnerException` 属性 (property), 使接下来的异常处理器能够同时访问二者 (新异常和原始异常)。例如:

```
catch( ArgumentNullException ane )
{
    string msg;

    switch ( ane.Source )
    {
        case "Argument One":
            msg = "get_text_array() failed";
            break;

        case "Argument Two":
            msg = "get_item() failed";
            break;

        default:
            msg = "both get_text_array() " +
                  "and get_item() failed";
            break;
    }

    throw new InvalidProgramException( msg, ane );
}
```

Exception class 阶层体系分裂为两个主要子树 (primary subtrees)。所有运行环境 (runtime environment) 抛出的异常都派生自 `SystemException`，后者又派生自 `Exception` class。诸如 `ArgumentException`、`ArithmeticException`、`FormatException` 等异常都派生自 `SystemException`，这些 classes 又派生出更细化 (更具特殊目的) 的 `ArgumentNullException` 或 `DivideByZeroException`。

如果我们想要引入自己的异常类，建议从 `ApplicationException` 派生它们。`ApplicationException` 直接派生自 `Exception`。同时我也建议，为这些异常提供缺省构造函数，将所有缺省属性 (default properties) 初始化，例如：

```
class TextQueryException : ApplicationException
{
    public TextQueryException()
        : this( "A TextQueryException has been thrown", null )
    {}
}
```



```
public TextQueryException( string message )
    : this( message, null ){}

public TextQueryException( string msg, Exception innerE )
    : base( msg, innerE ){}

// application-specific members go here ...
}
```

接下来我们可能还要提供更特殊化的子异常 (subclass exceptions)，像这样：

```
class ProhibitedQueryException : TextQueryException
{ ... }
```

编写一组 catch 子句时，我们总是应该把 derived class 排在其 base class 之前。例如我们总要试着先捕捉 TextQueryException，再试着捕捉其直接 (上一层) 基类 ApplicationException，然后才是基类的基类 Exception，依此类推。

这样做的理由是，所有异常的决议行为都是基于“最先匹配 (first match)”而不是基于“最佳匹配 (best match)”。因此如果相应于 base class 的 catch 子句出现在相应于 derived class 的 catch 子句之前，那么匹配的就是 base class 实体，余下的 catch 子句也就不再获得考虑了 (译注：这就不妙)。以下的 catch 子句排列顺序将优先处理继承体系中的最下层 (most-derived) class 的实体：

```
try {
    // ...
}
catch( ProhibitedQueryException pqe )
{ ... }

catch( TextQueryException tqe )
{ ... }

catch( ApplicationException ae )
{ ... }

catch( SystemException se )
{ ... }

catch( Exception e )
{ ... }
```

重载函数 (overloaded functions) 的匹配采取最佳匹配 (而非最先匹配) 决议算法。这就是为什么“重载函数的匹配动作与函数的声明顺序无关”，但异常 (exceptions) 的匹配不是这样。在重载函数的匹配过程中，两个 derived-class objects 的匹配”优于“一个 derived-class object 加上一个 base class object 的匹配” (译注：这句话的意思是如果有两个函数 `void foo(aDerived, aDerived)` 和 `void foo(aDerived, aBase)`，那么编译器会优先匹配前者)。但是在异常 (exceptions) 的匹配中，上述二者会被平等视之。因此，我们必须在 catch 子句中将 base class 的异常置于其派生实体之后。

4

接口继承

Interface Inheritance

Interface (接口) 详细说明了一组抽象的 methods 和 properties. 和 abstract base class 一样, interface 依靠 derived class 将它塑造为具体实物 (而非抽象接口), 和 abstract base class 不同的是, interface 不能提供缺省实现或是定义任何“状态” (state, 例如定义实体 (instance) 数据成员或常量)。

Abstract base class 为一整族相关型别提供了公共的 interface. 例如在计算机图形学中, Light 可能作为一族发光体的 abstract base class. 用户可能在一组场景中以“与实际型别无关”的方式来操纵这些发光体, 包括开启或熄灭、调换位置、改变颜色和亮度, 如此等等. 整个 Light 体系包含 abstract base class、数个 derived classes (也许代表聚光灯、定向灯、点光源 (如太阳) 等等). 我们可以利用这些预先建立好的 classes, 或从中再派生出新的 classes. 例如在迪斯尼电影《恐龙》一片中, 我们引入了一个新的 class, 代表“有挡光板的照明灯”。

换句话说, interface (接口) 为“在其他方面互不相干”的型别提供了一些公共服务和特征 (characteristic). 例如打算被用于 foreach 语句的型别, 都必须实现出 IEnumerable 接口¹³. ArrayList、BindingManager、DataView、BitArray classes 都派生自 IEnumerable 接口, 但除此之外它们鲜有共同处。

¹³ 遵循习惯, .NET 的所有 interfaces 都以大写字母 'I' 打头, 后面跟着一个便于记忆的标识符 (首字母大写). 这是从 COM 借用的习惯. COM 提供了一些像这样的接口: IUnknown、IDirectDraw.

在 C# 之中, class 只支持单一继承, 但 interface 却支持多重继承。例如 class `System.String` 是从 `System` 命名空间提供的 4 个 interfaces 派生而来:

```
public sealed class String:
    IComparable, ICloneable, IConvertible, IEnumerable { ... }
```

C# 程序员可以以三种不同的方式来使用 interface: (1) 使用“根据 `System` 所提供或用户自定义之 interfaces”而实现出来的具体 classes; (2) 为“`System` 提供或用户自定义”之 interfaces 实现具体的 class; (3) 提供新的 interfaces。本章将分别谈论以上三个方面。

4.1 实现 System Interface: IComparable

让我们以“为 `System` 命名空间中预定义之 interface 实现出自己的 class”作为旅途的开始。背景动机如下: 我们需要依照字符串长度, 由小到大, 对内含 `String` 的 `ArrayList` 排序。很好, 听起来够简单的, 但怎么做呢? 观察 `ArrayList` 的 public 成员函数, 我们便能发现, 它提供一组重载的 `Sort()` 成员函数:

```
System.Collections.ArrayList
Sort: Overloaded (已重载)
Sorts the elements in the ArrayList, or a portion of it.
(将 ArrayList 中的所有元素或部分元素排序)
```

现在让我们读一读这些重载函数的文档, 看看哪一个合用。以下是不带参数的 `Sort()`:

```
public virtual void Sort()

Sorts the elements in the entire ArrayList using
the IComparable implementation of each element.
(以元素所实现出来的 IComparable 为排序准则, 对整个 ArrayList 内的元素排序)
```

从名字就能知道 `IComparable` 是一个 interface。我们的 `ArrayList` 持有的元素是 `String`, 而 `String` 已实现出 `IComparable`。 `ArrayList` 提供的无参数 `Sort()` 将以“`String` 所具备之 `IComparable` 实现”来决定两个 `String` objects 的顺序。不过, 那是单词的字典顺序, 不是以长度为评断的顺序。因此, 这份 `Sort()` 函数实体对我们没有用。

好，也许下面这份重载实体有用：

```
public virtual void Sort( IComparer ic )
```

Sorts the elements in the entire ArrayList using the special comparer ic.
(以特定的比较器 'ic' 为排序准则，对整个 ArrayList 的所有元素排序)

这份实体为我们提供了一个改变 String class 缺省排序算法的途径。我们将为 IComparer 创建一份特殊实现——按长度对字符串排序。如果我们定义某个 class 并打算将此 class 装入容器 ArrayList 内进行排序，我们就得在实现该 class 时含入 IComparable。

我们该怎么办？IComparer 成员函数有哪些？它们是干什么的？如果想让我们的实现与此 interface 的其他实现能够无缝接合 (transparently meld)，那么我们不仅要为每个 interface 成员函数提供定义，还必须为每个成员函数实现出文档所规定的行为。

此外，在相同的（非正常）情况下，我们应该抛出相同的异常。编译器不能强迫我们这么做。识别并抛出异常是 interface 实现品的责任。

文档显示只一个成员函数需要我们实现，嗯，做起来也没什么困难：

```
int Compare( object x, object y );
```

Compares two objects and returns a value indicating whether one is less than (negative number), equal to (zero), or greater than (positive number) the other.
(比较两个 objects 并返回一值：如果第一参数小于、等于或大于第二参数，就分别返回负数、零或正数)

这个成员函数能抛出哪些异常呢？我们再次查看文档。正如所见，只能抛一个异常，这看来也是可行的：

```
ArgumentException
```

Neither x nor y implements the IComparable interface.
-or- x and y are of different types and neither one can handle comparisons with the other.
(x, y 都没有实现 IComparable。或者，x 和 y 型别不同，无法相互比较)

这意味着我们的 `IEnumerator` 实现品只须一个成员函数：`Compare()`。如果引数不合法，它应该抛出 `ArgumentException` 异常；对我们而言，引数不合法仅仅意味着两个引数不全是 `string`。我们将 class 命名为 `StringLengthComparer`：

```
public sealed class StringLengthComparer : IEnumerator
{
    // first obligation: the Compare() method
    public int Compare( object x, object y ){...}
}
```

即使我们的实现品只对明确的两个 `string` 参数感兴趣，我们的 `Compare()` 实体也必须将两个参数的型别定义为泛化的 `object`。这么做是因为 `Compare()` 暗自是个虚函数（其实所有 `interface` 成员函数都如此），因此覆写的函数实体的标记式（signature）必须和继承而来的成员函数的标记式（signature）完全吻合。

这一点不够方便。这意味着我们得在 `Compare()` 中检验其引数型别是否为 `string`。要是我们能明确地将其型别声明为 `string`，编译器就能自动实施型别检测。我将在 4.4 节解释如何绕开它——至少从部分意义上来说如此。

剩余要做的唯一一件事就是实现 `Compare()`。实现过程可分解为两大步骤。首先必须确认两个引数的合法性，而后比较其长度。

我们有两个“型别为 `object`”的参数，而且必须确认它们其实都是型别 `string` 的实体。策略之一是利用 `is` 操作符：

```
if ( !( ( x is string ) ) || !( ( y is string ) ) )
    throw new ArgumentException( "some dire message" );

// OK: arguments are valid;
// let's cast them to string objects
string xs = (string) x;
string ys = (string) y;
```

另一个策略是以 `as` 操作符进行向下转型（downcasting）。如果 `xs` 或 `ys` 之一被设为 `null`，我们就抛出异常：


```
string xs = x as string;
string ys = y as string;

if ( xs == null || ys == null )
    throw new ArgumentException( "some dire message" );
```

以上是唯一棘手的部分。以下是长度比较：

```
int ret_val = 1;

if ( xs.Length < ys.Length )
    ret_val = -1;
else
if ( xs.Length == ys.Length )
    ret_val = 0;

return ret_val;
```

这样看来，我们已经实现出 interface 的第一个具体实体（concrete instance）。

下面调用 Sort()：

```
ArrayList stringList = new ArrayList();
// fill it up
stringList.Sort( new StringLengthComparer() );
```

4.2 访问业已存在的 Interface

本节中我将实现一个泛型二叉树（generic binary tree），其内可持有任何型别的节点。正如你现在知道的，在 C# 中的实现办法就是将节点型别声明为 object。此外，一个节点值（node value）只安插到树中一次；如果节点值再次出现，我们就将它的出现次数累加 1。这个例子使我们有机会仔细体会 IComparable。我把节点称为 TreeNode：

```
public sealed class TreeNode
{
    private int      m_occurs;
    private object    m_nval;
    private TreeNode m_lchild, m_rchild;

    // ...
}
```

我将为这棵树实现出若干“政策”(policies), 为它提供“运用 .NET Framework 所定义的一些 interfaces”的机会。例如我们想在插入第一个元素之后将树的型别固定下来。我的这棵树名为 `BinaryTree`:

```
public class BinaryTree
{
    public delegate void Action( ref TreeNode node);
    private Type      m_elemType;
    private TreeNode  m_root;
    private Action    m_nodeAction;
    // ...
}
```

当用户创建一个 `BinaryTree` object 时, 我想让它能够持有任何型别的元素。只要将元素型别指定为 object 便可以做到这一点。然而一旦用户插入第一个值, 我们便希望从此所有后继插入的元素, 型别皆与第一元素相同。也就是说, 被插入的第一个元素锁定了树的型别。

以下是这棵树的必要行为: 当我们创建一棵新树, 它能持有任何型别的 objects: 一旦插入第一个元素, 这棵树就只能存储“与第一元素的型别相同”的其他元素。例如, 当我们写下:

```
BinaryTree bt = new BinaryTree();
```

`bt` 可以存储任何型别的元素。然而, 一旦我们写下:

```
bt.insert( "Piglet" );
```

`bt` 就变得只能持有型别为 `string` 的元素。接下来插入其他 `string` 元素都没有问题:

```
bt.insert( "Eeyore" );
bt.insert( "Roo" );
```

每个元素都将依照 `Comparable` 所保证的顺序插入树中。

现在我们希望阻止用户插入第一个元素后, 又插入与第一元素型别不同的 object。例如用户写下:

```
bt.insert( 1024 );
```


我希望将“目前插入之元素，其型别与先前确认的型别不符”这一错误消息标示出来。要做到这一点，抛出异常即可。

除了要求只接受“实现出 `Comparable`”的型别之外，对于树节点，我们再没有更多约束了，为什么？因为每个元素都是通过 `object` 参数传进来的，所有编译期型别信息都已丢失。`Comparable` 提供与型别无关的大小比较服务，例如：

```
private Comparable confirm_comparable( object elem )
{
    Comparable ic = elem as Comparable;
    if ( ic == null ){
        string msg = "Element type must support Comparable -- "
            + elem.GetType().Name
            + " does not currently do so!";
        throw new ArgumentException( msg );
    }
    return ic;
}
```

我们可能这样调用上述的 `confirm_comparable()`：

```
public void insert( object elem )
{
    // if this is the first element
    if ( m_root == null ) {
        confirm_comparable( elem );
        m_elemType = elem.GetType();
        m_root = new TreeNode( elem );
    }
    else
    {
        confirm_type( elem );
        m_root.insert_value( elem );
    }
}
```

上述 `insert()` 只在插入第一个 `object` 时检测元素型别是否吻合约束条件。之后，它只是确认新的 `objects` 与最初 `object` 的型别是否相同。如果型别相同，就将元素插入树中，如下所示：

```
public void insert_value( object val )
{
    // assumption is that BinaryTree has confirmed this ...
    IComparable ic = val as IComparable;

    // OK: zero means the two objects are equal
    if ( ic.CompareTo( m_nval ) == 0 ){
        m_occurs++;
        return;
    }

    // OK: less than; insert within left subtree
    if ( ic.CompareTo( m_nval ) < 0 ){
        if ( m_lchild == null )
            m_lchild = new TreeNode( val );
        else m_lchild.insert_value( val );
    }
    else ( // insert within right subtree
        if ( m_rchild == null )
            m_rchild = new TreeNode( val );
        else m_rchild.insert_value( val );
    )
}
```

当我们以实际型别操纵一个 `object` 时，我们几乎可以忽略该型别实现了哪些 `interfaces`。然而，当我们操纵型别为 `object` 的物体时，“找出 `interface`”和“使用 `interface`”就变得重要起来，因为所有编译期型别信息都遗失了。当然我们也可以在运行期间查询 `object` 的真实型别——8.2 节将介绍所谓的型别反射（**type reflection**）

4.3 定义一个 Interface

在 4.1 节中，我们为业已存在的 `interface` 实现了一个新实体。4.2 节比较简单，我们只不过是找出并使用某个型别的相关 `interfaces`。本节我将引入一个 `interface` 定义式，它支持“基于某个独特算法，生成并显示一个数列”。下面是我打算支持的操作集（两个 `methods`、一个 `property`、一个 `indexer`）：

- `Generate_sequence()`，生成元素的特定序列。
- `Display()`，输出所有元素。
- `Length`，返回元素数目。
- `Indexer`，让用户得以访问特定元素。

`Interface` 定义式都以关键字 `interface` 打头，后面紧跟一个名称。请记住，`.NET interface` 的名称习惯以大写字母 'I' 打头，后面紧跟首字母大写的名称。编译器并不强迫你遵守这个习惯。我把我的这个 `interface` 称为 `INumericSequence`：

```
public interface INumericSequence {}
```

如你所见，是的，空的 `interface` 是合法的，所以上一行表现的是一个完整的 `interface` 定义式。

“`Interface` 允许定义”的成员是“`class` 允许定义”的成员的子集。`Interface` 只能声明 `methods`、`properties`、`indexers`、`event` 作为其成员，例如：

```
public interface INumericSequence
{
    // a method
    bool Generate_sequence( int position );

    // a set of overloaded methods
    void Display();
    void Display( int first );
    void Display( int first, int last );

    // a property
    int Length { get; }

    // a one-dimensional indexer
    int this[ int position ] { get; }
}
```

`Interface` 的所有成员都暗自 (`implicitly`) 成为 `abstract`。我们不能为它提供缺省实现，无论这份实现是多么平淡无奇。`Interface` 的成员也都暗自 (`implicitly`) 成为 `public`。我们不能以关键字 `abstract` 或 `public` 来修改 (修饰) 其中某个成员。

Interface 之内不能定义自己的数据成员，也不能定义构造函数或析构函数——由于缺乏数据成员，所以这两个特殊函数其实也就不需要了。此外 interface 不能声明 static 成员。由于 const 成员暗中也是 static，所以 interface 也不允许声明 const 成员。操作符和 static 构造函数也不在允许之列（毫不令人意外^②）。

Interface 可以继承自另一个 interface，甚至继承多个 interfaces：

```
public interface INumericSequence : ICollection { ... }  
public interface INumericSequence  
    : ICollection, ICloneable { ... }
```

但 interface 不能显式继承自 class 或 struct。所有 interfaces 都隐式继承最根源的 Object class。

4.3.1 实现我们自己的 Interface：概念验证

Interface 一旦被定义，我们如何检测其正确性呢？通常在定义 interface 之后，我们至少会提供一个实现品，作为概念验证（proof of concept）或健全检测（sanity check）之用，以便确认 interface 的正确性。现在我提供一份实现品 Fibonacci，用以测试我们的 interface。

Fibonacci 数列的头两个元素是 1。将前两个元素相加就得到下一个元素。例如前 8 个 Fibonacci 元素为 1, 1, 2, 3, 5, 8, 13, 21。

现在我们应该从哪儿开始着手？

interface 的实现得靠 class 或 struct 的继承才得以完成。struct 不适合本例，因为我不希望在应用程序中的那些“对运行时间十分敏感”的地点创建并操控一大堆 Fibonacci objects。因此，我将 Fibonacci 声明为 class。由于我不希望它被继承，所以将它声明为 sealed：

```
public sealed class Fibonacci : INumericSequence { }
```

然而上述空定义是非法的。从 interface 继承下来的 class 必须为每一个 interface 成员提供实现。即便遗漏一个成员没有实现，也将导致编译错误。除非实现 interface 的那个 class 本身也是个 abstract class——如果真是这样，这个 class 可以将 interface 的成员函数声明为 abstract。我将在 4.5 节详细解说这些概念和做法。

实现 interface 成员之前,我们必须先确定支持数列抽象性(sequence abstraction)所必需的状态(state)信息。此处我们需要一个 array 来持有序列元素,还需要另外两个成员分别持有 array 的大小和容量。由于序列元素值固定不可变,我们只需要一个 array 就好。这个 array 和另两个成员都将声明为 static。

Interface 的实现有两个要素:(1) 为支持抽象性(abstraction)而必备的底层基础设施(如果有的话);(2) 为 interface 的每个成员(含所有继承而来的 base interfaces 的成员)提供定义。以下是 Fibonacci class 的设计骨架:

```
public sealed class Fibonacci : INumericSequence
{
    // infrastructure to support sequence abstraction
    private static int [] m_elements;
    private static short m_count;
    private static short m_capacity = 128;

    // Fibonacci-specific methods:
    // all for infrastructure supports
    static Fibonacci(){ ... }
    private void check_pos( int pos ) { ... }
    private void grow_capacity() { ... }

    // INumericSequence inherited members
    public bool Generate_sequence( int pos ) { ... }

    public void Display(){ ... }
    public void Display( int first ) { ... }
    public void Display( int first, int last ) { ... }

    public int Length { get{ return m_count; } }
    public int this[ int position ] { ... }
}
```

其中的 indexer (索引器) 必须先行验证所取位置的有效性。我把这样的验证工作专门放入一个 private 成员函数 check_pos() 内。如果位置不合法, indexer 就抛出一个异常, 否则就返回“位置减 1”处的元素:

```
public int this[ int position ]
{
    get
    {
        check_pos( position );
        return m_elements[ position-1 ];
    }
}
```

由于 C# 的 array 索引从 0 开始, 也就是说第一个元素的索引(下标)是 0, 所以我们必须修正(调整)用户指定的位置。为什么不要求用户自己修正呢? 这样的决定是否正确取决于我们对用户的假设。如果用户是软件研发人员, 要求索引(下标)从 0 开始也许说得过去。然而不曾当过程序员的人, 常常觉得“第一个元素的位置是 0”的想法很不自然。现在, 通过将修正行为封装到 class 中, 我们因而挑起了所有担子, 全权负责, 让代码用起来更安全, 让用户更感觉舒适。

接下来, 练习使用我们的实现品。首先直接产生一个 Fibonacci class object; 然后间接地将它视为一个一般性的 (generic) INumericSequence object。

```
public static void Main()
{
    // just some magic numbers -- used as positions
    const int pos1 = 8, pos2 = 47;

    // let's directly use interface through class object
    Fibonacci fib = new Fibonacci();

    // invokes indexer;
    // indexer invokes Generate_sequence( pos1 );
    int elem = fib[ pos1 ];
    int length = fib.Length;

    string msg = "The length of the INumericSequence is ";
    Console.WriteLine( msg + length.ToString() );
    Console.WriteLine(
        "Element (0) of the Fibonacci Sequence is {1}",
        pos1, elem );
}
```



```
fib.Display();

// OK: let's now use interface generically
INumericSequence ins = fib;

elem = ins[ pos2 ];
length = ins.Length;

Console.WriteLine( msg + length.ToString() );
Console.WriteLine(
    "Element {1} of the Fibonacci Sequence is {0}",
    elem, pos2 );

ins.Display( 44, 47 );
}
```

正如你所看到，我们的实现品稍有瑕疵，这也正是为什么需要测试的原因。程序的输出证明确实存在着问题：

```
The length of the INumericSequence is 8
Element 8 of the Fibonacci Sequence is 21
Elements 1 to 8 of the Fibonacci Sequence:
1 1 2 3 5 8 13 21

The length of the INumericSequence is 47
Element 47 of the Fibonacci Sequence is -1323752223
Elements 44 to 47 of the Fibonacci Sequence:
701408733 1134903170 1836311903 -1323752223
```

怎么回事？程序竟然声称 Fibonacci 数列的第 47 号元素是 -1323752223。这显然不正确。如果我们看看第 44 号元素至第 47 号元素的显示结果，也就很清楚究竟发生什么事了：

```
Elements 44 to 47 of the Fibonacci Sequence:
701408733 1134903170 1836311903 -1323752223
```

是的，我们将负责持有元素的 `m_elements` 的型别声明为 `int`，不幸的是第 47 号 Fibonacci 元素太大，以至无法以 `int` 表示。其值溢出（overflowed），影响了符号位（sign bit），所以数值被不正确地显示为负数。如果要更良好地处理大型 Fibonacci 元素，我们必须重新定义 `m_elements`，使它能够持有大型数值。

哪一种型别才最适合呢？除了若干整数型别 (integral types)，C# 还提供有两种浮点数型别 (单精度的 float 和双精度的 double) 以及 28-digit 精度的 decimal 型别。我将 m_elements 依次重新声明为 uint、ulong、decimal、double，然后在第 50 个位置处发现，只有 ulong、decimal、double 屹立不动：

```
Fibonacci Element #50 : (int)      : -298632863 (译注：结果错误)
Fibonacci Element #50 : (uint)     : 3996334433 (译注：结果错误)
Fibonacci Element #50 : (ulong)    : 12586269025
Fibonacci Element #50 : (decimal)  : 12586269025
Fibonacci Element #50 : (double)   : 12586269025
```

到了第 100 位置处，只剩下 decimal 和 double 能够正确表示其值，然而其中的 double 开始丢失精度 (四舍五入)，这让人难以接受。只有 decimal 没有舍入，这表示 Fibonacci 序列元素的最佳选择是 decimal 型别：

```
Fibonacci Element #100 : (ulong)   : 3736710778780434371
Fibonacci Element #100 : (decimal) : 354224848179261915075
Fibonacci Element #100 : (double)  : 3.54224848179262220
```

尽管如此，这还不是完整的解答。在 139 位置处，decimal 型别终于达到了其 28-digit 精度上限：

```
Fibonacci Element #139 : (decimal) : 50095301248058391139327916261
Fibonacci Element #139 : (double)  : 5.009530124805840628
```

如果尝试计算第 140 个位置，decimal 会溢出 (overflow) 并抛出以下异常：

```
System.OverflowException:
Value is either too large or too small for a Decimal.
at System.Decimal.Add(System.Decimal, System.Decimal)
at System.Decimal.op_Addition(System.Decimal, System.Decimal)
at Project1.Fibonacci.version_decimal(Int32)
at Project1.MainObj.Main()
```

看来，如果使用 C# 预定义的算术型别，我们只能存储并显示前 139 个 Fibonacci 元素。除非我们决定实现出一个自定义的数值型别，用以支持巨大整数，否则我们必须定义出“用户所能申请的元素位置”上限，这个上限对所有 Fibonacci 数列都成立，因此我们在 INumericSequence 中加上这一限制的定义，如下：


```
public interface INumericSequence
{
    // the two new properties
    int Length { get; }
    int MaxPosition { get; }

    // ... rest the same ...
}

public sealed class Fibonacci : INumericSequence
{
    // infrastructure to support sequence abstraction
    private static int [] m_elements = new int[ m_maxpos ];
    private static short m_count;
    private const short m_maxpos = 128;

    public int MaxPosition { get { return m_maxpos; } }

    // ... rest the same, except no longer need to grow array
}
```

这意味着尽管合法的 Fibonacci 元素位置有无限多个，但我们无法轻松地全部予以支持。事实上我们无法全部支持。此外，用户也许会指定一个非法位置——任何小于或等于 0 的位置都是不合法的。我们该如何处理这两种无效状态？C# 和 .NET 程序设计的习惯是，通过异常的引发来报告程序中的不正常情况。

但是抛出哪个异常呢？遇到“无效位置（小于或等于 0）”和“不被支持的位置（大于 MaxPosition）”时，是否应该抛出不同的异常？答案并无唯一性。谁来决定呢？如果每个 interface 的实现可以自行决定抛出什么异常（以及是否抛出异常），那就几乎不可能安全地以“一般化方式”对 interface 进行编程。举个例子，如果要写出以下代码：

```
INumericSequence ins = o as INumericSequence;
if ( ins != null )
    elem = ins[ pos2 ];
```

我们应当确知：每个 `indexer` 实现品都以完全相同的方式回报“无效位置”。

这意味着 `interface` 定义式必须决定“在什么条件下抛出异常”以及“抛出哪一种异常”，不幸的是我们只能通过文档加以说明。C# 无法直接将成员和其所能抛出的异常关联起来。（译注：意指无法像 C++ 那样以所谓的异常规格（exception specifications）指出成员能够抛出哪些异常，可参考 *C++ Primer*, p.564, 简体版 p.463）

4.3.2 将我们的 Interface 整合进入 System Framework

`INumericSequence` 是个良好的 `interface` 吗？小范围来看，似乎还不错，但是在 .NET 环境中这个 `interface` 可能令人失望，例如用户也许会枚举（enumerate）其中的元素——或许采用直接枚举方式，或许使用 `foreach` 循环，我们该如何支持这种功能？

我们得让每个 `INumericSequence` 实现品同时也实现出 `IEnumerable`。是的，让我们的 `interface` 继承自 `IEnumerable`，这才能确保我们的 `interface` 的每个实现品同时也实现出 `IEnumerable`：

```
public interface INumericSequence : IEnumerable
{ ... }
```

我们不必在 `INumericSequence` 定义式中列出 `base-interface` `IEnumerable` 的任何成员。但是当用户企图实现 `INumericSequence` 时，他不仅需要提供 `INumericSequence` 的所有成员的定义，还必须提供 `IEnumerable` 所有成员的定义，否则会编译失败，编译器会告诉你缺少了哪些 `interface` 成员。

总共有哪些成员呢？我们该如何实现它们呢？这一次我再次求助于文档，发现 `IEnumerable` 只有一个成员：

```
IEnumerator GetEnumerator();
```

这个函数返回一个 `IEnumerator` object。 `IEnumerator` 也是个 `interface`，定义于 `System.Collections` 命名空间中，它将“如何遍历群集（collection）”的必要知识封装了起来。

通常，当我们实现 `IEnumerable` 时，必须同时实现出相应的 `IEnumerator`。不过，`IEnumerator` 的实体（支持数列的遍历动作）是以独立 `class` 的形式实现出

来的（而不是作为 `INumericSequence` 的 base interface）：

```
class NSEnumerator : IEnumerator { ... }
```

这样做，枚举器（enumerator）就只需实现一次，`INumericSequence` 的多份实现品都可以复用（reuse）它。以下便是 `GetEnumerator()` 的实现：

```
public sealed class Fibonacci : INumericSequence
{
    private static int [] m_elements = new int[ m_maxpos ];
    private static short m_count;

    public IEnumerator GetEnumerator()
    { return new NSEnumerator( m_elements, m_count ); }

    // ...
}
```

我们如何实现出 `NSEnumerator` 的构造函数？再次求助 `IEnumerator` 文档，它说：

`IEnumerator` 是所有 enumerator（枚举器）的 base interface。当我们对 enumerator 具体实现出一个实体时，它会取群集（collections）当前状态下的一份瞬间快照（snapshot）。

这就是为什么我们要将 `m_elements` array 及其长度传给 `NSEnumerator` 的原因，因为那相当于我们的瞬间快照（snapshot）。

`IEnumerator` 有三个成员：一个是 property `Current`，返回当前元素；两个是成员函数 `MoveNext()` 和 `Reset()`。前者将 `Current` 推进一个位置，取得群集内的下一元素；后者将 `Current` 设为初始位置。一如文档所述，`IEnumerator` 的初始位置有着特殊语义：

enumerator（枚举器）最初被置于群集（collection）的第一个元素之前。使用它之前，必须先将它推进一位。

其行为稍微有些不太直观，初学者很容易弄错。但无论如何，用户相当快就可以学会。调用 `MoveNext()` 之前如果先取用 `Current`，会导致运行期异常：

```
public void iterate( ArrayList al )
{
    IEnumerator it = al.GetEnumerator();

    // oops: this access of Current before MoveNext()
    // generates an exception
    while( it.Current != null )
    {
        Console.WriteLine( it.Current.ToString() );
        it.MoveNext();
    }
}
```

`IEnumerator` 的正确用法是,在第一次取用 `Current` 之前先调用 `MoveNext()`。如果下一个元素能被取用, `MoveNext()` 就返回 `true`。以下是正确的实现:

```
public void iterate( ArrayList al )
{
    IEnumerator it = al.GetEnumerator();

    while( it.MoveNext() )
        Console.WriteLine( it.Current.ToString() );
}
```

这里存在的基本观点是,无论何时,当你准备实现别人定义的 `interface` 时,你必须确信自己理解并提供“如 `interface` 描述其成员那般”的精确语义。否则,一旦通过 `interface object` 来使用我们的实现品时,其行为将难以预期。编译器无法在这一层级上强迫程序员贯彻一致性。

下一个样例展示 `IEnumerator` 的部分实现(但仍然尚未实现其他三个 `interface` 成员)。这仅仅代表着支持“遍历群集(collection)”所必须的 `interface`:

```
class IEnumerator : IEnumerator
{
    decimal [] m_elems;
    int m_count;
    int m_curr;
```



```
public IEnumerator( decimal [] array, int count )
{
    // these are exceptions defined within System
    if ( array == null )
        throw new ArgumentNullException( "null array" );

    if ( count <= 0 )
        throw new ArgumentOutOfRangeException(
            count.ToString() );

    m_elems = array;
    m_count = count;
    m_curr = -1; // required semantics!
}

// ...
}
```

其中 Current 是个 property, 返回一个 object, 代表群集中的当前元素。如果 Current 被置于群集第一个元素之前, 或最后一个元素之后, 应该会出现异常 `InvalidOperationException`:

```
public object Current
{
    get{
        if ( m_curr == -1 || m_curr >= m_count )
            throw new
                InvalidOperationException( ToString() );

        return m_elems[ m_curr-1 ]; //译注: 恐为 m_elems[ m_curr ]
    }
}
```

请注意, Current 并没有将 `m_curr` 前移一位以便指向下一个元素。为什么不前移一位呢? 因为文档明确告诉我们, 不要那么做:

Current 并不会移动 enumerator 的位置。在调用 `MoveNext()` 或 `Reset()` 之前, 连续调用 Current, 会返回同一个 object。

如果我们将 `m_curr` 初始化为序列中的第一个元素, 或是如果我们在 Current 中推进一个位置, 那么用户遍历数列的时候, 就会不知不觉地漏掉一些元素。

为了让 `Current` 得以表示任何型别的元素, `IEnumerable` (译注: 恐为 `IEnumerator`) 将元素型别定义为 `object`. 这稍稍令人失望. 例如在 `NSEnumerator` 中我们已经知道, 元素型别是 `decimal`. `Current` 把它当做 `object` 返回, 意味着将会引发隐式装箱 (`implicitly boxed`) 动作, 用户必须以显式转型来拆箱 (`unbox`). 下一节我会告诉你如何设计才能绕开这个问题.

调用 `MoveNext()` 时, 如果枚举器 (`enumerator`) 前进至下一有效元素, 就返回 `true`. 如果枚举器越过了最后一个有效元素, 就返回 `false`:

```
public bool MoveNext()
{ return m_count > ++m_curr; } // 译注: 原书有误, 已改正
```

`Reset()` 会将枚举器 (`enumerator`) 设至初始位置. 记住, 文档规定, 初始位置必须设为群集的“第一个元素的前一个位置”:

```
public void Reset() { m_curr = -1; }
```

这个 `interface` 的实现代码十分直接了当. 其间的挑战只在于如何提供与文档一致的 `interface` 语义.

4.4 Interface 成员的显式实现 (Explicit Implementation)

如果你手上有型别为 `object` 的 `object`, 如何查明它是否支持某个特定 `interface` 呢? 一个最简单的方法就是利用 `is` 操作符来询问:

```
public static void iterate( object o )
{
    // true if o implements IEnumerable
    if ( o is IEnumerable ) // 译注: 询问
    {
        IEnumerable ie = (IEnumerable) o; // 译注: 转型
        IEnumerator iter = ie.GetEnumerator(); // 译注: 取得
        while ( iter.MoveNext() ) // 译注: 调用
            Console.WriteLine( "{0} ",
                                iter.Current.ToString() ); // 译注: 调用
    }
}
```


上述的 `iterate()` 是 C# 泛型函数 (generic function) 的一个样例。我们完全不知道参数 `o` 所指向的实际型别，因此我们问问它是否支持 `IEnumerable`。如果它不支持，函数就直接结束。如果它支持，我们就通过抽象的 `IEnumerator` object 访问之，并逐一移向下一元素。我们对枚举器 (enumerator) 的型别和群集的元素型别一点也不了解，但没有关系。这个函数可以接受任何型别的引数。

泛型编程 (generic programming) 提供了一种近乎神奇的灵活性。我们可以调用 `iterate()` 并赋予它一个 `Fibonacci` object:

```
Fibonacci fib = new Fibonacci();  
// ...  
iterate( fib );
```

或是赋予它一个 `ArrayList` object、一个内建 `array`，等等，每一种情况都能良好运作。

然而，如果我们想要直接运用 `IEnumerator` object 来遍历 `Fibonacci` object，泛型对我们就反而有点儿碍手碍脚了。即使我们知道实际返回型别，我们还是得泛化地对待返回值，像这样：

```
IEnumerator nse = (IEnumerator) fib.GetEnumerator();  
while ( nse.MoveNext() )  
{  
    decimal el = (decimal) nse.Current; // downcast  
    // ...  
}
```

为一个 `interface` 成员提供多份实现实体，有两条途径：当我们通过 `interface` 以泛型方式进行编程时，采用的是其中一条途径；当我们对 `interface` 的显式实体 (explicit instance，例如以下的 `IEnumerator`) 编程时，采用的是另一条途径。通过“`interface` 成员的显式实现”，我们可以做到，例如：

```
class IEnumerator : IEnumerator  
{  
    private void checkIntegrity(){  
        if ( m_curr == -1 || m_curr >= m_count )  
            throw new InvalidOperationException( ToString() );  
    }  
}
```

```

// invoked through an IEnumerator object
public decimal Current
{ get { checkIntegrity(); return m_elems[ m_curr-1 ]; } }
// 译注: 以上称为 m_elems[ m_curr ]

// the explicit interface member,
// invoked only through a generic IEnumerator object
object IEnumerator.Current
{ get { checkIntegrity(); return m_elems[ m_curr-1 ]; } }
// 译注: 以上称为 m_elems[ m_curr ]

// ...
}

```

确认“interface 成员显式实现”的方法是，在成员名称之前缀以这一成员所属的 interface 名称，两个名称之间以生存空间操作符（scope operator，`.`）分隔：

```

object IEnumerator.Current { ... } // explicit member
public decimal Current { ... }

```

显式成员（explicit member）的访问级别暗中（implicit）是 public。C# 不允许程序员明白指定其访问级别（即使是指明为 public 也不行）。如果声明某个 interface 的成员为显式成员（explicit member），而这个 interface 却没有包含该成员，将会导致错误。

只要通过 IEnumerator object 来取用 Current（就像 iterator() 那样），就会调用 interface 显式实体（explicit instance）：

```

IEnumerator nse = (IEnumerator) fib.GetEnumerator();

// downcast from object is no longer necessary
decimal el = nse.Current;

```

如果我们想除去 GetEnumerator() 的向下转型动作，可以提供一个返回 IEnumerator 的显式实体，以及一个返回 IEnumerator 的非显式实体。

4.5 继承得来的 Interface 成员

给你一道难题。假设有以下三层继承体系：

```

interface IControl
{
    void Paint();
}

```



```
class Control: IControl
{
    public void Paint() { ... }
}

class TextBox: Control
{
    new public void Paint() { ... }
}
```

以下代码调用的是哪一份 `Paint()` 实体?

```
IControl it = new TextBox();

// which instance of Paint() is invoked?
it.Paint();
```

大多数人的第一反应是 `TextBox.Paint()`，毕竟 `it` 指向一个 `TextBox` object。请注意 `IControl` 中的 `Paint()` 实体被当做抽象成员函数。你知道，成员函数的调用的决议动作，必须在运行期间依据 `it` 所指的 object 的实际型别来进行；此处 `it` 所指的是 `TextBox` object，所以调用的必定是 `TextBox.Paint()`。

当然，整个讨论是合理的，但结果错误。事实上成员函数 `Paint()` 的调用决议是静态完成的。因此，调用的是 `Control`（而不是 `TextBox`）的 `Paint()` 实体。先前遗漏的线索是：`TextBox.Paint()` 定义式中使用的关键字 `new`。

还记得第 3 章说过的吗？关键字 `new` 用来指明 `derived` 成员隐藏（遮掩）了继承得来的 `base class` 成员名称。我们的误解来自一个假设：`Control` 提供的 `Paint()` 被自动视为 `virtual`。

某个 `interface` 的实现品如果想让其成员被视为 `virtual`，就必须明白地以关键字 `virtual` 标明，就像我们为 `abstract base class` 的 `derived class` 所做的那样。例如：

```
class Control: IControl
{
    virtual public void Paint() { ... }
}
```

要让 derived class `TextBox` 中的 `Paint()` 函数实体覆写 (override) 继承而得的 `Control.Paint()` 函数实体, 我们必须指定关键字 `override`:

```
class TextBox : Control {  
    override public void Paint() { ... }  
}
```

现在, 当我们执行如下代码时:

```
IControl it = new TextBox();  
it.Paint();
```

调用的将是 `TextBox` 内的那一份 `Paint()` 实体!

如果 `Control` 与 `TextBox` classes 的实现者都十分强调 (重视) 相应的 `Paint()` 实体(s)性能, 以至于并不以 `virtual` 方式实现, 又该如何? 好吧, 我们也能支持这种可能:

```
class Control: IControl  
{  
    public void Paint() { ... } // nonvirtual  
}
```

```
class TextBox: Control, IControl  
{  
    public new void Paint() { ... } // nonvirtual as well!  
}
```

在这里, 我们明白地把 `IControl` 加到 `TextBox` 的继承清单中。 `TextBox` 在某种意义上像跳背游戏似地跳跨过 `Control` 提供的实现品。现在如果我们运行原来的代码:

```
IControl it = new TextBox();  
it.Paint();
```

还是调用 `TextBox` 的 `Paint()` 实体, 不过这次是静态调用, 而非通过运行期虚拟机制。

如果明确地为成员标上关键字 `abstract`, 也可以推迟 “interface 成员的实现”。例如 `abstract class NumericSequence` 推迟了 “与型别相依” 的数列生成动作的实现, 但它提供了公用基础设施。例如:


```

public abstract class NumericSequence : INumericSequence
{
    // type-dependent method remains abstract
    public abstract bool Generate_sequence( int pos );

    // concrete infrastructure of shared state
    private const int m_maxpos = 128;
    public int MaxPosition { get { return m_maxpos; }}

    // ...
}

```

具体的数列抽象性 (sequence abstractions) 乃是从 abstract base class (而非 interface) 派生而来：

```

public class Fibonacci : NumericSequence
{
    virtual public bool Generate_sequence( int pos ){ ... }
    // ...
}

public class Pell : NumericSequence
{
    virtual public bool Generate_sequence( int pos ){ ... }
    // ...
}

```

4.6 重载？掩盖？抑或模棱两可？

Overloaded, Hidden, or Ambiguous?

考虑以下的 class 继承体系：

```

class Base
{
    public void f( int ival ){ ... }
}

class Derived : Base
{
    public void f( ref int ix ){ ... }
}

```

Derived class 中的 f() 定义式需要 new 指示符 (specifier) 吗？或者说，在此程序中，以下哪些调用动作将导致编译错误（如果有的话）？

```

public static void main()
{
    Derived d = new Derived();
    int ival = 1024;

    d.f( ref ival ); // OK?
    d.f( ival ); // OK?
}

```

答案是：(1) 不！并不需要 new 指示符；(2) 这些调用行为都不会导致编译错误。也就是说，Derived class 中的 f() 定义式并没有隐藏（掩盖）其 base class 中的对应实体。在 Derived class 中，两份实体都可见，因此都能被调用。

为什么？还记得吗，函数如果共用同一名称（但各有独特标记式, signature），便被视为“重载”。如果 class 阶层体系或 interface 阶层体系中的两个成员函数共用同一个名称，那么只有在二者的标记式（signatures）相同的情况下，derived 所带的实体才会被编译器视为遮掩了 base class 所带的实体。

在我们的例子里，两个标记式（signatures）并不等价。是的，两个成员函数可以因为“ref 参数的出现”而完成重载。事实上我们甚至还能引入“仅以 out 参数加以区别”的第三个重载实体。

Interface 的多重继承性质导致我们有可能从多个 interfaces 继承得到相同的成员名称，这可能导致名称的模棱两可（歧义, ambiguity）。

考虑以下继承体系，其中的 aMed class 内可见到两份 doSomething() 实体。

```

interface a {
    void doSomething(object o);
}

interface b {
    void doSomething(object o);
}

class aMed : a, b {
    // oops! which doSomething?
    public void doit(){ doSomething( myObj ); }
}

```


只要为两个继承得来的 interface 实体提供“显式声明”，我们就可以完全解决 aMed class 中潜在的命名歧义性 (ambiguity)。此外我还提供一份合成 (综合) 实体，当程序直接操作 aMed class object 时，就会用到这份实体。例如：

```
class aMed : a,b
{
    public void doSomething( string s ){ ... }

    public void a.doSomething( object o ) { ... }
    public void b.doSomething( object o ) { ... }

    // ...
}

class EntryPoint
{
    public static void Main()
    {
        aMed am = new aMed();
        am.doSomething( "OK" ); // aMed.doSomething

        a aaa = am as a;
        aaa.doSomething( am ); // aMed.a.doSomething

        b bbb = am as b;
        bbb.doSomething( am ); // aMed.b.doSomething
    }
}
```

4.7 掌握 Copy (拷贝) 语义: ICloneable

当我们把一个 reference 型别拷贝给另一个 reference 型别时，如下所示，发生的是浅拷贝 (shallow copy)：

```
Matrix mat = new Matrix( 4, 4 );
// ...
Matrix mat2 = mat;
```

也就是说，mat 和 mat2 现在都指向 (代表) 位于受控堆 (managed heap) 的同一个 Matrix object。当我们希望通过第一个 handle 来修改 object，而仍旧要求第二个 handle 所指 object 保持原始状态时，问题出现了。就像下面这样：

```
// changes are seen through mat2 as well
mat[0,0]=mat[1,1]=mat[2,2]=mat[3,3]=1;
```

如果我们是 `class` 的用户，我们将无法修改 `class` 的实现以提供深拷贝（deep copy）语义。因此，我们需要明白地（自己动手地）实现深拷贝。首先为 `reference` 型别分配一份新实体，然后逐一拷贝每个值：

```
public class DeepCopy
{
    public static Matrix copyMatrix( Matrix m )
    {
        Matrix mat = new Matrix( m.Rows, m.Cols );

        for ( int ix = 0; ix < m.Rows; ++ix )
            for ( int iy = 0; iy < m.Cols; ++iy )
                mat[ix, iy] = m[ix, iy];

        return mat;
    }
}

Matrix mat = new Matrix( 4, 4 );
Matrix mat2 = DeepCopy.copyMatrix( mat );
```

这样得到的，是这个 `object` 的独立复制品。换句话说，我们将它 `clone`（克隆）了。

缺省情况下，`reference` 型别的拷贝是“浅拷贝”。作为 `class` 的设计者，我们需要思考，是否有必要提供“深拷贝”语义。通过 `ICloneable` 可以达成此点。`ICloneable` 声明了一个 `Clone()` 函数，它负责返回一个深拷贝实体（型别是 `object`）。例如：

```
class matrix : ICloneable
{
    public matrix( int row, int col )
    {
        m_row = ( row <= 0 ) ? 1 : row;
        m_col = ( col <= 0 ) ? 1 : col;

        m_mat = new double[ m_row, m_col ];
    }
}
```



```
public object Clone()
{
    matrix mat = new matrix(m_row, m_col);
    for ( int ix = 0; ix < m_row; ++ix )
        for ( int iy = 0; iy < m_col; ++iy )
            mat.m_mat[ ix, iy ] = m_mat[ ix, iy ];
    return mat;
}
```

现在用户可以选择使用“浅拷贝”或“深拷贝”，知道什么时候该使用哪一种拷贝，才是真正要点。考虑以下重载后的加法操作符：

```
public static matrix operator+( matrix m1, matrix m2 )
{
    check_both_rows_cols( m1, m2 );

    // not: matrix mat = m1;           // 译注：式(1)
    matrix mat = (matrix) m1.Clone(); // 译注：式(2)

    for ( int ix = 0 ; ix < m1.rows; ix++ )
        for ( int ij = 0; ij < m1.cols; ij++ )
            mat[ ix, ij ] += m2[ ix, ij ];

    return mat;
}
```

这儿有四份 matrix 复制品，其中只有一个需要深拷贝。

是的，“向函数传入两个 matrix objects”和“从函数返回一个代表运算结果的 matrix object”，正确方法是采用缺省的浅拷贝机制。此时深拷贝并非必需品，而且如果那么做也会导致效率不彰。

但是在上述加法操作符内如果使用浅拷贝（译注，如式(1)），将是个严重错误。因为如果采用浅拷贝，以 m1 初始化 local object mat，意味着稍后每一个针对 mat 的赋值动作也同时修改了 m1。这时候我们需要以深拷贝来保证两个 objects 彼此独立。

4.8 掌握 Finalize (终结) 语义: IDisposable

垃圾回收 (Garbage collection) 解决了一个重大问题：实现了“在 heap 之上分配和释放 objects”的自动管理。不过这个解决方案也引入了一个关于所谓“确定性终结” (deterministic finalization) 的小问题。意思是我们无法预知一个 object 何时

(或是否) 被垃圾回收器 (garbage collector) 终结掉 (finalized)。

当一个 class object 获得非受控资源 (unmanaged resources)，例如窗体 handle 或文件 handle，或数据库连接等等我们不希望占用太久、不需要时就应该释放的资源时，就会伴随发生终结问题。理想情况下我们希望资源在“持有该资源之 object 最后一次被使用”后就自动释放掉。不幸的是我们无法实现这样的自动操作——至少在时间控制上无法做得很好，原因是 .NET 并没有所谓的“确定性终结” (deterministic finalization)。

垃圾回收器 (garbage collector) 知道 managed heap (受控堆) 上有哪些内存可用。如果它检测到可用内存量低于某特定值，就会执行回收动作。在此之前，unreferenced (不再被引用的) objects 将继续存在于内存之中。

垃圾回收器对于外部资源 (诸如数据库连接等等) 一无所知。如果可用的数据库连接 (通道) 数已经降得太低，并无等价机制自动执行回收动作。

举个例子，一个维持着数据库连接的 unreferenced object 会一直保持这个连接，直到垃圾回收器终结 (finalize) 掉它，或是直到用户明白地释放这一资源。这就是 IDisposable 要应付的问题。

凡希望获得“非受控资源” (unmanaged resources) 的 class，都应该实现 IDisposable，其中声明了一个成员：Dispose()。object 持有的资源，以及 object 所含之其他 objects 所持有的资源，在 Dispose() 实现中都应该被释放掉。

如何调用 Dispose()？绝大多数情况下用户必须在最后一次使用 object 之后，亲手调用 Dispose()。万一用户忘记调用 Dispose()，我们也可以提供一个析构函数，在其中调用 Dispose()。如果用户手工调用 Dispose()，我们便通过“调用 GC (垃圾回收器) 的成员函数 SuppressFinalize()”将析构函数“禁用” (disable)。例如：

```
class ResourceWrapper : IDisposable
{
```

```
    // ...
```



```

public void Dispose()
{
    Dispose( true );

    // take us off of the finalization queue
    GC.SuppressFinalize( this );
}

protected virtual void Dispose( bool disposing )
{
    if ( disposing ) {
        // dispose of managed resources ...
    }

    // dispose of unmanaged resources ...
}

// just in case Dispose() is not explicitly invoked
~ResourceWrapper() { Dispose(false); }
}

```

我们也可以选择维护一个 `IsDisposed` 成员, 用以指明 `object` 是否已经调用过 `Dispose()` (译注: `IsDisposed` 建议为 `bool` 型别, 且必须是 *Instance* 成员). 这样便能避免多次释放同一份资源, 也能捕捉 (trap) “将已调用 `Dispose()` 的 `object` 拿来使用” 的错误情况——这时候最好是抛出 `ObjectDisposedException` 异常.

当我们编写 .NET 程序时, 要随时意识到可能抛出的异常. 这一点很重要. 例如以下代码就不是 “异常安全” (exception safe) 代码:

```

foo()
{
    FileStream fin = new
        FileStream(@"c:\fictions\alice.txt", FileMode.Open);
    StreamReader ifile = new StreamReader( fin );
    while (( str = ifile.ReadLine()) != null )
    {
        // ...
    }

    ifile.Close();
}

```

如果 `ReadLine()` 抛出异常, 就不会调用 `Close()` 成员函数, 因而导致 file handle 一直开启着, 直到 `ifile` 被收回。这里的 `Close()` 将“调用 `Dispose()`”的动作包装了起来。

一个安全但更复杂的实现办法是, 引入 `finally` 子句, 确保无论是否发生异常, `Close()` 或 `Dispose()` 能被调用:

```
foo()
{
    FileStream fin = new
        FileStream(@"c:\fictions\alice.txt", FileMode.Open);
    StreamReader ifile = new StreamReader( fin );
    try {
        while ( ( str = ifile.ReadLine() ) != null )
            // ...
    }
    finally {
        ifile.Close();
    }
}
```

下面是一种特殊的 `using` 语句写法, 可以为“确保调用 object 的 `Dispose()`”提供速记符号 (简写法):

```
foo()
{
    // equivalent to the earlier try/finally block
    using ( File f = new File( "c:\tmp" ) )
        { byte[] b = f.Read(); }
}
```

这里的 `using` 语句在内部被 (编译器) 扩展为 `try/finally` 区块, 内容与先前例中明白写出的代码一致。

4.9 BitVector: 以组合 (composition) 进行扩充

`System.Collections` 命名空间提供了 `BitArray` class。这个 class 控制 (管理) 一个布尔型的 object array, 就像 bit vector (位向量) 一般。元素值如果是 `true`, 表示该 bit 处于开启状态; 元素值如果是 `false`, 表示该 bit 处于关闭状态。`Count` 返回 array 的元素个数, `Count` 是 `ICollection` 的一个属性 (property), 而 `BitArray` 正是派生自这个 interface。(译注: 后文可以看到, `BitArray` 一共继承了 3 个 interfaces)

我们可以通过 array 的索引 (下标) 或 “显式设置元素” 来开启或关闭一个 bit:

```
bat1[ 2 ] = bat2[ 4 ] = true;  
bat1.Set(ix, bat[ ix ] );
```

函数 SetAll() 可以将 array 中的所有 bits 一起设为 true 或 false, 元素值可通过索引 (下标) 或以 Get() 成员函数读取出来:

```
// both return the value of the first element  
bool b1 = bat1[ 0 ];  
bool b2 = bat1.Get( 0 );
```

如果想要反转 BitArray object 中的元素值, 可使用 Not() 成员函数:

```
bat1.Not(); // bits are now reversed
```

BitArray 有 6 个构造函数。以下代码阐述它们的用法。

```
const int elemSize = 6;  
const bool elemValue = true;  
  
// create a BitArray of size elemSize;  
// each element is initialized to false  
BitArray bits0 = new BitArray( elemSize );  
  
// create a BitArray of size elemSize;  
// each element is initialized to elemValue  
BitArray bits1 = new BitArray( elemSize, elemValue );  
  
// create a BitArray copied from another BitArray  
BitArray bits2 = new BitArray( bits1 );  
  
// create a BitArray with the number and value  
// of elements from a bool array  
bool [] bvals = { false, true, false, true, false, true };  
BitArray bits3 = new BitArray( bvals );
```

```
// create a BitArray initialized to an array of bytes
// in which each byte represents 8 consecutive bits
byte [] byteValues = { 255, 0 };

// evaluates to 1111111100000000
BitArray bits5 = new BitArray( byteValues );

// create a BitArray initialized to an array of ints
// in which each int represents 32 consecutive bits
int [] intValues = { -7 };

// evaluates to 10011111111111111111111111111111
BitArray bits6 = new BitArray( intValues );
```

此外我们也可以在两个 BitArray objects 之间以 And() 成员函数执行 bitwise AND (按位与) 操作:

```
BitArray andResult = bat1.And( bat2 );
```

如果 bat1 和 bat2 同一位置上的两个元素都是 true, 那么 andResult 对应位置上的元素为 true, 否则为 false.

此外, BitArray 也以 Or() 成员函数支持 bitwise OR (按位或) 操作, 也就是说, 如果 bat1、bat2 同一位置上的两个元素当中有一个是 true, 其结果元素就是 true. 另还有以 Xor() 成员函数支持 bitwise XOR (按位异或) 操作, 也就是说, 如果两个操作数有一个为 true, 另一个为 false, 则结果元素为 true; 如果两个操作数有相同的布尔值, 则结果元素为 false.

BitArray class 有点让人失望的地方是, 没有提供“显示内部持有之所有 bits”的功能. 继承而来的 ToString() 只是打印出 class 型别而已:

```
System.Collections.BitArray
```

不能像下面这样以字符串方式显示所有元素:

```
00000101
```

.NET 文档中有一个显示 BitArray object 内容的样例: 用户自行遍历每一个元素, 并打印出字面值 (true 或 false):

```
False False False False False True False True
```

此法对拥有许多元素的 BitArray object 而言并不适用.

另一个小毛病是, BitArray 不支持将容器内的 bits 转换为一个数值。举个例子, 如果给定一个 BitArray object, 其内有 8 个元素分别为 00000101, 那么“取回一个不带正负号的数值”也许很有用, 本例应该取回 5, 但实际上办不到。

当年在迪斯尼梦工场动画工作室 (DreamWorks Animation) 工作时, 我是动画测试系统 *ToonShooter* 的技术总监。动画绘制人员使用这套系统, 通过一台以 GUI 驱动的数字式摄像机, 把他们的绘画录入电脑。在动画场景 (scene) 中运动的图画序列是按图层 (level) 录入的, 每个图层代表场景中的一个元素, 每个角色都画在各自的图层上。因此, 可能有背景图层、前景景象、阴影、交通工具、伶俐的小动物等等, 都位于各自的图层上。

我们的工具的用处之一, 是让动画师检查动画的连贯度。另一个用处是检查角色的嘴唇运动是否与声轨同步。第三个用处是按导演的要求, 有声或无声地播放场景。用户可以在播放前 (或播放时) 开启或关闭某些图层。

要想让动画看起来连贯, 每秒钟至少要播放 24 帧画面 (frames)。显示画面之前可能先得进行图像合成, 也就是说如果活动图层 (active level) 不止一个, 那么这些活动图层 (各自代表一幅图像) 必须合成为一幅图像。图像合成是按一个个像素 (pixels) 将活动图层融合到一块儿。合成是动态完成的, 并将成果 (一个个 frames) 传送给播放引擎。如果“图层合成器”传送 frames 的速度过慢, 这个工具就无法以正常速度播放而不丢失 frames 或中断音效。不消说, 这两种情况都会严重激怒用户。

提高合成速度的一个方法是, 将合成后的图像暂存起来 (cache)。在调用合成器之前, 我们检查一下合成后的图像是否存于高速缓存 (cache) 中。如果没有找到完全吻合的, 就找找最吻合的——可能两个、三个、四个必要图层已经被合成出来了, 因此, 我们只要再合成余下的图层。为了能快速查找并取回图像, 我们必须能够高效地将不同的合成图像标上独一无二的标记。每幅图像独一无二的标记由其帧号 (frame number) 和其合成所用的活动图层号码组成。

BitArray object 组成的 array 能够极佳地记录上述的信息。array 的每个元素代表一个 frame, BitArray object 代表该 frame 中的活动图层。因此, 对一个 6 图层场景, 如果其 bits 分配为 00000101, 则第 6、8 图层是有效的, 对应的独一无二的标记码为 5。

让我们自行实现一个 BitVector class, 提供 BitArray 缺少的功能。我的第一个主意是让 BitVector 派生自 BitArray, 这么一来 BitVector 就继承了 BitArray 的所有基础设施, 我们只须简单地对二者的不同部分给予编码即可。但很不幸, BitArray 是密封的 (sealed), 我们无法继承它:

```
public sealed class BitArray :
    ICollection, IEnumerable, ICloneable
```

除了继承, 另一个可选方案是组合 (合成, composition), 于是我让 BitVector 内部包装一个 BitArray 成员:

```
public sealed class BitVector :
    ICollection, IEnumerable, ICloneable
{
    private BitArray m_array;
    // ...
}
```

尽管我们没有继承 BitArray, 但可以通过小小的分派 (dispatch) 函数来复制 BitArray 的功能。例如:

```
// 译注: 所谓“分派函数”就是指以下的转调用函数 (forwarding functions),
//      我特别将复合物 (内含物, m_array) 加粗, 以利阅读。
//      这也有点 “adapter” pattern 的味道。
// ICollection
public int Count { get { return m_array.Count; }}
public bool IsReadOnly { get { return m_array.IsReadOnly; }}
public object SyncRoot { get { return m_array.SyncRoot; }}
public bool IsSynchronized
{ get { return m_array.IsSynchronized; }}

public void CopyTo( Array array, int index )
{ m_array.CopyTo( array, index ); }

// IEnumerable
public IEnumerator GetEnumerator()
{
    return m_array.GetEnumerator();
}
```



```
// ICloneable
public object Clone()
{
    BitVector bv = new BitVector( m_array.Count );
    bv.m_array = (BitArray)m_array.Clone();
    return bv;
}
```

接下来我要再以一个样例展示 BitVector 对于 ToString() 的实现。此例将建立一个字符串，用以表示内部的所有 bits，然后遍历 BitArray 成员，依据当前的 bit 是 true 或 false，在字符串后附加 '1' 或 '0'。由于这个算法每遇到一个元素都得修改字符串，因此，如果直接建立 string object，将是十分低效的做法。

要知道，string class 是不可变更的 (immutable)。如果采用“在 string object 后附加字符”的做法，那么将一个带有 132 个元素的 BitArray 分析完毕之后，我们将总共创建 133 个不同的 string objects。但其实只需一个 string object 就足够了，这就是为什么我以 StringBuilder object 取而代之的原因（再以 string 建立最后结果）。

我以关键字 override 标示 ToString()，因为它覆写了继承而得的“定义于 object class 中的虚函数”：

```
override public string ToString()
{
    StringBuilder sb = new StringBuilder( m_array.Count );
    for ( int ix = m_array.Count-1; ix >= 0; --ix )
        sb.Append( m_array[ ix ] ? '1' : '0' );

    return sb.ToString(); // 译注：从 StringBuilder 中取回 string
}
```

StringBuilder 位于 System.Text 命名空间内，它的唯一目的是允许我们修改字符串（例如移除字符、替换字符、插入字符等等）时不必为了每次改动而创建新字符串。举个例子，如果要将 ies 结尾的单词改为以 y 结尾，我们可以使用 StringBuilder 的重载函数 Replace()，像这样：

```
if ( word.EndsWith( "ies" ) )
    theWord.Replace( "ies", "y", theWord.Length-3, 3 );
```

一旦修改完毕，我就通过 ToString() 取回 StringBuilder object 所代表的 string。

以下是 ToUlong() 的实现, 用来将底层的 bit vector 转换为不带正负号的 long:

```
public ulong ToUlong()
{
    ulong bv = 0ul;
    for ( int ix = 0; ix < m_array.Count; ++ix )
    {
        if ( m_array[ix] )
        {
            ulong bv2 = 1;
            bv2 <<= ix;
            bv += bv2;
        }

        Console.WriteLine( "{0} :: {1}", ix, bv );
    }

    return bv;
}
```

其中所用的 <<= 是“bit 左移”和“赋值”的复式操作符 (compound operator)。我们首先对 bv2 赋值 1, 在计算机内部, 此值被表现为“底层表述之第 1 个 bit 被设置为 1” (译注: 也就是 00000000 00000000 00000000 00000001)。接下来的左移操作将 1 向左移动 ix 个位置, 如果 ix 是 1, 就左移一个位置。左移一个位置得到的结果为 2, 左移二个位置得到的结果为 4; 依此类推。

以下便是 ToString() 和 ToUlong() 在含有 24 个元素的 BitVector 上的输出序列:

```
0000000000000000000000010 :: 2
00000000000000000000001010 :: 10
000000000000000000000101010 :: 42
000000000000000000010101010 :: 170
00000000000000101010101010 :: 682
```

为了让 BitVector 适用于任何原本能够使用 BitArray 的地方, 我们提供从 BitVector 到 BitArray 的隐式转换:

```
public static implicit operator BitArray( BitVector bv )
{ return bv.m_array; } // 译注: return by reference
```


以 *by reference* 方式返回 *private array* 成员 (译注: 因为 *array* 是个 *reference* 型别), 可能导致将来有多个 *handles* 同时指向内部 *array*. 这意味着内部的 *m_array* 有可能在 *class* 之外被改变. 为此, 另一个实现办法是: 调用相应的 *Clone()* 函数, 返回 *m_array* 的一份副本:

```
return (BitArray)m_array.Clone();
```

5

探访 System 命名空间

System 命名空间为 C#.NET 程序员提供了至少三种不同层次的支持:

1. 在最基础层次, 它提供了所有基本型别 (fundamental types) 的底层实现, 包括数值型别 (numeric types)、string、enum、delegate、array。
2. 在第二层次, 它作为一个 system class library, 提供对 input/output (输入/输出)、collection classes (群集类)、exceptions (异常)、regular expressions (正则表达式)、sockets (网络套接口)、threads (线程)、Web requests (网页请求) 等等支持。
3. 在第三层次, 它提供了“为综合性应用领域而准备”的一套 framework (框架), 包括 Windows Forms (第 6 章)、Web Forms (第 7 章)、XML (eXtensible Markup Language, 可扩展标记语言), 并可 (通常是从数据库取回的) 数据制成显示表格 (数据格, DataGrid)。

本章将概略介绍多种 classes 和 class 继承体系, 遗憾的是本章篇幅不允许我详细介绍其中任何一个主题。

5.1 支持基本型别 (Fundamental Types)

在最基础的层次上, System 命名空间提供了对 C# 语言的基本数据型别的支持。C# 的所有基本型别 (primitive types) 如 int、double、bool、string 等等, 都是 System 命名空间所提供的各种型别的别名 (aliases)。这些“同义名”列于 1.18.2 节的表 1.3 (p.51)。

我们可以通过“C# 提供的别名”所产生出来的 object, 调用这些基本型别的相关函数, 例如, 为了得到某个数值型别的数值范围, 我们可以取用该型别的 MaxValue property 和 MinValue property:

```
int imaxval = int.MaxValue;  
int iminval = int.MinValue;
```


每个数值型别都有一个十分有用的成员函数 `Parse()`。假设你有如下的一个 `string` object:

```
string bonus = "$ 12,000.79";
```

调用 `Parse()` 便会将 `myBonus` 初始化为 12000.79:

```
double myBonus = double.Parse( bonus, ns );
```

这里的 `ns` 代表枚举型别 `NumberStyles` 的某种 bitwise OR 运算结果, 用来指导如何处理“前导的空白符、货币符、小数点或逗号”等等。此处我把 `ns` 设置如下:

```
NumberStyles ns = NumberStyles.AllowLeadingWhite;  
ns |= NumberStyles.AllowCurrencySymbol;  
ns |= NumberStyles.AllowThousands;  
ns |= NumberStyles.AllowDecimalPoint;
```

我们可以使用 C# 的转型表示法 (cast notation) 在各型别之间进行显式转换:

```
int ival = (int) myBonus;
```

或者利用 `System.Convert` class 的某个转换函数, 如 `ToDouble()`、`ToInt32()`、`ToDateTime()` 等等, 例如:

```
int ival2 = Convert.ToInt32( myBonus );
```

这两种方法的差异在于, 显式转型会导致截断 (truncation), 因此 `ival` 会被赋值为 12000。Convert 的成员函数则是执行四舍五入 (rounding), 因此 `ival2` 被赋值为 12001。

5.2 所有 array 都是 System.Array

C# 的所有 array 相关型别都可以取用 `System.Array` class 的 public 函数和属性 (properties)。例如 `Length` property 用以返回 array 的元素个数:

```
for ( int ix = 0; ix < fib.Length; ++ix )
```

如果 array 是多维的 (multidimensional), Length property 就不那么有用了。例如对以下的 array 声明式而言, Length 的结果是 20:

```
static float [,] mat = new float[4,5]
{
    { 1f, 0f, 0f, 0f, 0f },
    { 0f, 1f, 0f, 0f, 0f },
    { 0f, 0f, 1f, 0f, 0f },
    { 0f, 0f, 0f, 1f, 0f }
};
```

要取回多维 array 中各个维度的大小, 可以使用 GetLength(int dim), 这里的 dim 代表第几个维度: 0, 1, 2... 等等。要想知道 array 的维度, 可查询其 Rank property:

```
int rank = mat.Rank;
Console.WriteLine( "Array of {0} dimensions", rank );

for ( int dim = 1, ix = 0; ix < rank; ++ix )
{
    dim = mat.GetLength( ix );
    Console.WriteLine( "size of dimension {0} is {1} elements",
                        ix, dim );
}
```

编译后执行, 上述代码产生以下输出:

```
Array of 2 dimensions
size of dimension 0 is 4 elements
size of dimension 1 is 5 elements
```

成员函数 CopyTo() 可以将一维 source array 的所有元素复制到一维 target array (以第一引数表示)。第二引数指出“从 target array 的第几号元素开始覆盖”。如果元素索引 (下标) 不合法, 或是 target array 过小, 会出现异常 (exception)。例如:

```
int [] fib;

// assign fib to an array ...
int [] notfib = new int[ fib.Length ];
fib.CopyTo( notfib, 0 );
```


如果要将一维 array 内某个区间的元素复制到另一个 array 中, 必须使用 `static Copy()`。只要传给它 source array、target array、欲复制的元素个数, 它便会从两个 arrays 的第一个元素处开始复制。例如:

```
Array.Copy( fib, notfib, fib.Length );
```

如果你不想从第一个元素开始复制, 就必须调用带有 5 个参数的 `Copy()`。例如以下函数会将 `fib` 内的“从索引 1 开始, 共 `Length-1` 个元素”复制到 `notfib`, 并从 `notfib` 的头一个元素开始覆盖:

```
Array.Copy( fib, 1, notfib, 0, fib.Length-1 );
```

如果两个 array 持有不同型别的元素, 而且两个型别之间存在隐式转换, 那么这一隐式转换将成为 `Copy()` 的一部分, 自动运行起来 (译注: 也就是一边复制一边转换)。

如果想要将一个 array 的所有或部分元素设为 0 (对数值型别而言)、`false` (对 `bool` 型别而言)、`null` (对 `reference` 型别而言), 可使用 `static Clear()`。例如以下动作会将 `fib` array 整体清为零:

```
Array.Clear( fib, 0, fib.Length );
```

此外, 我们还可以对一维 array 施行排序 (`sort`)、倒转 (`reverse`) 和查找 (`search`) 动作。例如以下的整数 array:

```
int [] ivalues = new int[] { 14, 8, 2, 16, 8, 7, 14, 0 };
```

想查找整数 8 的第一次出现位置吗? 请用 `static IndexOf()`:

```
int index = Array.IndexOf( ivalues, 8 );  
if ( index != -1 ) /* found! */;
```

如果找到目标, `IndexOf()` 便返回目标值第一次出现位置, 否则返回 -1。 `LastIndexOf()` 查找的是某值最后一次出现位置, 返回的是该位置的索引 (下标) 或 -1。

以下代码查找某值的所有出现位置。其中所用的伎俩是反复调用 `IndexOf()`, 直到它返回 -1。为了做到这一点, 我们传入第三参数: 起始查找位置。首先传入的是 0, 而后在每次匹配 (找到) 之后, 又将匹配位置加 1, 再传入:

```
int index = -1;
```

```

ArrayList found = new ArrayList();
while( true )
{
    index = Array.IndexOf( ivalues, search_value, index+1 );
    if ( index == -1 )
        break;

    found.Add( index );
}

Console.Write( "{0} occurrences of {1} found at ",
               found.Count, search_value );

foreach ( int ix in found )
    Console.Write( "{0} ", ix );

```

当 `search_value` 为 8 时，上述代码产生以下输出：

```
2 occurrences of 8 found at 1 4
```

`IndexOf()` 所用的查找算法的复杂度是线性的，也就是说，它会依次检视 `array` 中的每个元素，直到发现查找目标，或直到所有元素都检视完毕。对于大型 `array`，二分查找法 (binary search) 效率更高，但它要求 `array` 必须先排序：

```

Array.Sort( ivalues );
index = Array.BinarySearch( ivalues, search_value );
if ( index >= 0 ) /* found! */

```

`BinarySearch()` 返回的是找到的元素位置 (如果找到的话)，或返回一个负数 (不一定是 -1)，除此之外还有一个 `static Reverse()`，它会令 `array` 中的元素逆序存放。

5.3 查询运行环境

现在，我来实现一个小程序，查询本机环境 (local machine environment)，如用户、进程 (processes)、逻辑驱动器 (logical drives) 等等，输出结果看起来会是这个样子：


```
Hello, stanley lippman!  
Your machine PROUST is running Microsoft Windows NT 5.0.0.2195  
Service Pack 1
```

```
The current process running is 'HelloEnvironment'  
Startup Path is C:\C#Programs\HelloEnvironment\bin\Debug  
There are 46 other processes running on PROUST
```

```
Process Name: Idle  
has been running for 3 days  
total: 3.22:59:33.0130432
```

```
Machine PROUST Process Statistics
```

```
Longer than a day: 1  
Longer than an hour: 0  
Longer than a minute: 12  
Less than a minute: 34
```

```
The logical drive subdirectory structure:
```

```
A:\ :: 2 subdirectories.  
The subdirectories: ParamPassing Visibility  
C:\ :: 50 subdirectories.  
D:\ :: 4 subdirectories.  
The subdirectories: interfaces xml efficiency munich  
E:\ :: 1 subdirectories.  
The subdirectories: Art  
F:\ :: is currently unavailable.
```

5.3.1 Environment Class

利用 `System.Environment` class, 我们可以取得用户姓名、机器名称以及操作系统名称和版本号。 `GetEnvironmentVariable()` 是其中的一个 static 成员函数, 只要我们将我们感兴趣的环境变量 (environment variable) 名称传给它, 它就会返回该环境变量的 string 值 (如果有定义的话), 或是返回 null。我感兴趣的两个环境变量是 `USERNAME` 和 `COMPUTERNAME`:

```
string user_name =  
    Environment.GetEnvironmentVariable( "USERNAME" );  
  
string mach_name =  
    Environment.GetEnvironmentVariable( "COMPUTERNAME" );
```

```
OperatingSystem os_ver = Environment.OSVersion;
```

```
Console.WriteLine( "Hello, {0}!", user_name );  
Console.WriteLine( "Your machine {0} is running {1}\n",  
    mach_name, os_ver.ToString() );
```

OSVersion 会返回一个 OperatingSystem object, 其中存储了当前操作系统的各种属性。上述代码生成了以下数行问候语:

```
Hello, stanley lippman!  
Your machine PROUST is running Microsoft Windows NT 5.0.0.2195  
Service Pack 1
```

事实显示, USERNAME 只在 Windows NT、Windows 2000、Windows XP 环境下有所定义, Windows 95、Windows 98、Windows ME 没有定义这个环境变量。因此, 在后三个操作系统下 user_name 将被初始化为 null。我们当然也可以增加检测 user_name 是否为 null, 此外我们也可以根据操作系统来区分不同的问候语, 例如:

```
if ( os_ver.Platform == PlatformID.Win32NT )  
    Console.WriteLine( "Hello, {0}!", user_name );  
else Console.WriteLine( "Hello!" );
```

其中 PlatformID 是个枚举(enum)型别, Platform 是 OperatingSystem class 的一个 static property, 它持有一个 PlatformID 值。目前的 PlatformID 定义了三个值: (1) Win32NT, 代表 Windows NT-based 系列 (译注: NT, 2000, XP 等版本); (2) Win32S, 代表 Windows 95 之前的早期操作系统版本; (3) Win32Windows, 代表其他所有版本 (译注: 亦即 Windows 95/98/ME 系列)。

5.3.2 访问所有环境变量 (Environment Variable)

我们可以调用 GetEnvironmentVariables(), 取得所有环境变量并存储为一个以 key/value pairs 为元素的群集。这个函数会返回一份 IDictionary 实体。下面这份代码收集所有环境变量并加以排序, 再打印出来 (用户可选择是否查看每一个环境变量值):


```
public static void displayEnvironment()
{
    IDictionary dict =
        Environment.GetEnvironmentVariables();

    Console.WriteLine( "There are {0} environment variables",
        dict.Count );

    string [] keys = new string[ dict.Count ];
    string [] values = new string[ dict.Count ];

    int ix = 0;
    foreach ( DictionaryEntry de in dict )
    {
        keys[ ix ] = (string) de.Key;
        values[ ix ] = (string) de.Value;
        ++ix;
    }

    Array.Sort( keys, values );

    for ( ix = 0; ix < keys.Length; ++ix )
    {
        Console.Write( "Variable is {0} -- value? (y/n)",
            keys[ix]);
        string rsp = Console.ReadLine();
        if ( rsp == "y" || rsp == "Y" )
            Console.WriteLine( "\t==> {0} ", values[ix] );
        else Console.WriteLine();
    }
}
```

你也许会问，为何要将 keys 和 values 存储于各自的 string array 中。这是因为从 GetEnvironmentVariables() 获得的条目 (entries) 并不是依字母顺序排列。我将环境变量按字母顺序排序，同时还要让每个环境变量所对应的值随之变动位置，保持正确下标。以下调用的 sort() 刚好满足这样的要求：

```
// sorts the keys array --
// that is, maintains the values array
// in the order of the associated keys element
Array.Sort( keys, values );
```

存储于 `keys` 中的环境变量将按字母顺序表进行排序，存储于 `values` 中的对应元素则在 `keys` 排序时移动相应位置。两个 `arrays` 的所有元素之间的一一对应关系因而得以保持。

5.3.3 Process Class

定义于 `System.Diagnostics` 命名空间中的 `Process` class，有两个 `static` 拣取函数 (`retrieval methods`)，通过它们可访问到本地 (`local`) 或远程 (`remote`) 计算机中正在运行的程序 (称为“进程”，`process`)：

1. `GetCurrentProcess()` 返回一个 `Process` class object，其中带有当前运行的程序信息：

```
Process currProc = Process.GetCurrentProcess();
Console.WriteLine( "The current process running is \"{0}\"",
    currProc.ProcessName );
```

2. `GetProcesses()` 返回由 `Process` objects 组成的一个 `array`，代表本机 (`local computer`) 之中正在运行的所有进程 (`processes`)：

```
Process [] procs = Process.GetProcesses();
string msg="There are {0} other processes running on {1}\n";
Console.WriteLine( msg, procs.Length-1, mach_name );
```

通过 `Process` class object，我们可以获得进程的许多属性，包括 `Id` (唯一标识符) 和 `ProcessName`。我们也可以借此获得进程的运行时间，包括用户时间 (`UserProcessorTime`)，系统时间 (`PrivilegedProcessorTime`) 以及总时间 (`TotalProcessorTime`，用户时间与系统时间之和)。下面这份统计信息：

```
Process Name: Idle
    has been running for 3 days
    total: 3.22:59:33.0130432
```

```
Machine PROUST Process Statistics
    Longer than a day:      1
    Longer than an hour:   0
    Longer than a minute:  12
    Less than a minute:    34
```

就是由以下代码产生出来的：


```
foreach ( Process proc in procs )
{
    TimeSpan totalTime = proc.TotalProcessorTime;
    if ( totalTime.Days > 0 )
    {
        dayCnt++;
        Console.WriteLine( msg, proc.ProcessName,
                           totalTime.Days, totalTime.ToString() );
    }
    else
    if ( totalTime.Hours > 0 )
        hourCnt++;
    else
    if ( totalTime.Minutes > 0 )
        minCnt++;
    else lessMinCnt++;
}
```

其中所用的 `TimeSpan` 是 `System` 命名空间中的一个 `struct`，用来表示一段 (span) 时间，例如某个 process 存活时间长短或两个 `DateTime` objects 之差。`Days` 和 `Hours` 等 `properties` 返回的是整单位 (这里分别是日期和小时)。举个例子，如果 `TimeSpan` object 表示 25 小时又 20 分钟，那么 `Days` 的值是 1，`Hours` 也是 1，`Minutes` 的值是 20。以下代码将两个 `DateTime` objects 相减：

```
DateTime theNow = DateTime.Now;
DateTime backThen = new DateTime( 1945, 11, 11, 11, 11, 0, 0 );
TimeSpan soFar = theNow - backThen;
Console.WriteLine( soFar.ToString() );

Console.WriteLine( "Days since: {0}", soFar.Days );
Console.WriteLine( "Years since: {0}", soFar.Days / 365 );
```

5.3.4 查找逻辑驱动器

本程序的最后一部分是列出计算机涵括的所有逻辑驱动器，并打印每个驱动器的最上层目录 (如果其个数少于 10 的话)：

The logical drive subdirectory structure:

A:\ :: 2 subdirectories.

The subdirectories: ParamPassing Visibility

C:\ :: 50 subdirectories.

D:\ :: 4 subdirectories.

The subdirectories: interfaces xml efficiency munich

E:\ :: 1 subdirectories.

The subdirectories: Art

F:\ :: is currently unavailable.

GetLogicalDrives() 是 Directory class 的 public static 成员函数 (Environment class 也有一个同名函数)，它返回一个 string array，其内的每个元素代表一个逻辑驱动器；字符串格式如同 "c:\", 其中 c 表示驱动器字母：

```
string [] logical_drives = Directory.GetLogicalDrives();
```

Directory class 定义于 System.IO 命名空间中，稍后我们再来研究它。

5.4 System.IO

System.IO 命名空间提供了对 I/O (input/output) 的支持。有四个 classes 用来操控文件和目录。其中 Directory class 和 File class 只提供 static 成员，可以在实际目录或文件不存在时使用。如果要操控实际文件或目录，请使用 FileInfo class 或 DirectoryInfo class，两者都提供了“能作用于特定目录或文件上”的 instance (non-static) 成员函数。此外，Path class 用于处理文件和目录的路径字符串。

支持 I/O 的 classes，大致分为三个种类 (categories)：(1) byte-oriented (面向字节) Stream 阶层体系；(2) 专门用于处理字符输入输出；(3) 专门用于处理二进制输入输出。我并不打算枚举出这些种类繁多的 classes，也不想逐条详细解说各个 interface，我想做的是实现第 3 章介绍的“文字查询系统”中的文件读写部分。我们需要的两个主要例程 (routines) 如下：

1. `request_text_file()`，要求一个目录或文件路径。它会检查文件是否存在，以及后缀名是否为 `.txt` ——我们的程序只支持这种文件。
2. `handle_directory()`，它首先确认目录的存在，而后搜集文件（文件的后缀名称必须能被我们的程序识别），并显示文件清单和相关特征如长度、最后开启时间等等。

用户输入的路径名称，例如 `C:\fictions\araby.txt` 或 `C:\fictions`，有可能表示一个文件（如前者）或一个目录（如后者），也有可能是非法路径。我们如何测定这些情况呢？

```
// text_file holds a string entered by the user
file_check = File.Exists( text_file );

if ( file_check == false )
{
    if ( Directory.Exists( text_file ) )
        return handle_directory();

    Console.WriteLine( "Invalid file: {0}: ", text_file );
}
```

`File class` 提供了一些 `static` 函数，可对文件进行查询、创建、复制、删除、搬移、开启等动作。`Exists(string path)` 中的 `path` 所代表的文件如果确实存在，就返回 `true`，如果文件不存在或 `path` 代表的其实是个目录，就返回 `false`。

`Directory class` 为目录提供了类似的 `static` 函数，其中的 `Exists(string path)` 只在 `path` 所代表的目录存在时才会返回 `true`，否则返回 `false`。

5.4.1 处理文件扩展名：Path Class

一旦得到一个有效的文件名称，我们还需要确认我们的确能够支持其文件类型。“文件类型”按习惯是由文件扩展名标示出来的，例如文件后缀名 `.cs` 代表一个 C# 程序文件，`.cpp` 代表一个 C++ 程序文件，`.xml` 代表一个 XML 文件，`.txt` 代表一般的文本文件，等等。

扩展名由文件名称中的小数点 (.) 分隔出来。文件名称则是“最后一个目录分隔符 ('/' 或 '\')”之后的一串字符序列。例如以下路径:

```
@\"c:\fictions\current\word.txt\"
```

文件名称是 word.txt, 扩展名是.txt。

Path class 提供了一套 static 函数, 用来处理 (分解或组合) 目录和文件的路径字符串——这个字符串并不保证代表真实存在的文件或目录。我们可以使用以下三个 static 函数来查询、取回、修改某个文件或目录的扩展名:

1. bool HasExtension(string path), 如果文件名称中包含小数点 (.), 此函数就返回 true, 否则返回 false。
2. string GetExtension(string path), 返回文件扩展名, 包括小数点 (.), 例如.txt, 或者返回 String.Empty。
3. string ChangeExtension(string path, string newExt), 若第二个参数是 null, 此函数会把文件名称的扩展名除掉, 否则就以新扩展名取代旧名。如果原文件没有扩展名, 则为它添加新扩展名。

以下代码简单扼要地介绍了 Path 的上述成员, 以及其他一些成员。你可以从这些成员的名称中轻易猜出其作用:

```
string thePath = @"C:\fictions\Phoenix\alice.txt";
```

```
Console.WriteLine( "The file is named " +  
    Path.GetFileNameWithoutExtension( thePath ));
```

```
if ( Path.HasExtension( thePath ))  
    Console.WriteLine( "It has the extension: " +  
        Path.GetExtension( thePath ));
```

```
Console.WriteLine( "The full path is " +  
    Path.GetFullPath( thePath ));
```

```
if ( Path.IsPathRooted( thePath ))  
    Console.WriteLine( "The path root is " +  
        Path.GetPathRoot( thePath ));
```

```
string tempDir = Path.GetTempPath();
```



```

Console.WriteLine( "The temporary directory is " + tempDir );

// combine two path strings ...
string tempCombine =
    Path.Combine( tempDir, Path.GetFileName( thePath ) );

Console.WriteLine( "Path of file copy " + tempCombine );

// creates the path of a unique file name within the
// system's temporary directory ...

string tempFile = Path.GetTempFileName();
Console.WriteLine( "Temporary file is " + tempFile );

```

把以上代码置入某个成员函数中并执行之，便会生成以下输出（译注：请注意其中路径仅供参考）：

```

The file is named alice
It has the extension: .txt
The full path is C:\fictions\Phoenix\alice.txt
The path root is C:\
The temporary directory is C:\DOCUME~1\STANLE~1\LOCALS~1\Temp\
Path of file copy C:\DOCUME~1\STANLE~1\LOCALS~1\Temp\alice.txt
Temporary file is C:\DOCUME~1\STANLE~1\LOCALS~1\Temp\tmp337.tmp

```

5.4.2 操控目录 (Directories)

已知某个字符串，代表一个合法目录，我们想要 (1) 找出这个目录下具有特定扩展名（例如.txt）的所有文件；(2) 找出这个目录的所有子目录，并逐一检查这些子目录。下面的代码可以完成使命：

```

try
{
    // if unable to open for any reason, throws exception
    DirectoryInfo dir = new DirectoryInfo( text_file );

    // holds all supported file types

    ArrayList candidate_files = new ArrayList();

    // holds array of files returned from GetFiles()
    FileInfo [] curr_files;

```

```
foreach ( string ext in m_supported_files )
{
    // returns a file list from the current directory
    // that matches the given search criteria,
    // such as "*.txt"
    curr_files = dir.GetFiles( "*" + ext );
    candidate_files.AddRange( curr_files );
}

// get all subdirectories within our directory
DirectoryInfo [] directories = dir.GetDirectories();

// OK: let's do it again
foreach ( DirectoryInfo d in directories )
    foreach ( string ext in m_supported_files )
    {
        curr_files = d.GetFiles( "*" + ext );
        candidate_files.AddRange( curr_files );
    }
}
```

DirectoryInfo class 的成员函数 GetFiles() 有两份重载实体。不带参数的那一份实体返回一个由 FileInfo objects 构成的 array，代表目录中的所有文件。第二份实体接受一个“条件查找字符串 (search criteria string)”，并且只返回满足该条件的文件。在这里，我们要求所有文件以特定的扩展名结尾，例如以下程序返回扩展名为.txt 的所有文件：

```
FileInfo [] curr_files = dir.GetFiles( "*.txt" );
```

星号 (*) 作为查找过程中使用的通配符 (wild card)。上述条件语句的意思是，只要文件名称以.txt 做为后缀，便合乎条件。

GetDirectories() 也有两份函数重载实体，不带参数的那份实体返回所有子目录，带“条件查找字符串”的那份实体，仅返回满足条件的目录。

以下代码创建一个目录，并于其中创建一个文件和一个子目录，而后删除三者。其中用到的 static 函数如 Exists()，是通过 Directory class 调用的；instance 函数如 CreateFile() 和 CreateSubdirectory() 则是通过一个 DirectoryInfo object 调用的：


```

public static void testDirCreateDelete( string workDir )
{
    DirectoryInfo wd;

    // create the directory if it does not exist
    if ( ! Directory.Exists( workDir ) )
        wd = Directory.CreateDirectory( workDir );
    else wd = new DirectoryInfo( workDir );

    // create a file and a subdirectory
    FileStream f = wd.CreateFile( workDir + "test.txt" );
    DirectoryInfo d = wd.CreateSubdirectory( "subdir" );

    // delete directory and its contents
    d.Delete();

    // delete directory and all subdirectories
    f.Close();
    wd.Delete( true );
}

```

以上函数可以这么调用:

```

testDirCreateDelete( Path.GetTempPath() + "foobar" +
    Path.DirectorySeparatorChar );

```

其中 `DirectorySeparatorChar` 是 `Path` class 的只读 (read only) property, 在 Windows 之下是反斜线 (\), 在 Unix 之下是斜线 (/), 在 Macintosh 操作系统下是冒号 (:).

`DirectoryInfo` class 提供了一套 properties, 用以封装不同的目录特征, 例如目录是否存在等等. 举例来说, 软盘或 CD-ROM 驱动器不存在时, 如果对它们查询目录, 会返回 `false`. 以下代码运用了数个 `DirectoryInfo` properties (属性), 它们很大程度上可以顾名思义:

```

public static void testDirProperties( string workDir )
{
    DirectoryInfo dir = new DirectoryInfo( workDir );
    if ( dir.Exists )
    {
        Console.WriteLine("Directory full name {0}", dir.FullName);
    }
}

```

```

// refreshes object --
dir.Refresh();

DateTime createTime = dir.CreationTime;
DateTime lastAccess = dir.LastAccessTime;
DateTime lastWrite = dir.LastWriteTime;
// ... display these ...

DirectoryInfo parent = dir.Parent;
DirectoryInfo root = dir.Root;

FileInfo [] has_files = dir.GetFiles();
DirectoryInfo [] has_dirs = dir.GetDirectories();
}
}

```

5.4.3 操控文件 (Files)

和目录类似，文件的创建、复制、搬移、删除、查询、修改（文件特征）等等操作，也被分入两个 classes。File class 提供一套 static 成员函数，即使不存在实际文件，也可以调用它们。FileInfo class 则只能提供给实际存在的文件使用。

FileInfo class 也提供了一套 properties（属性），它们很类似 DirectoryInfo 所提供者。例如以下输出：

```

Creation Time       : 11/29/2000 7:02 PM
Last Access Time    : 5/7/2001 12:00 AM
Last Write Time     : 3/13/2001 1:59 PM
File Size in Bytes  : 703

```

是由这样的代码生成的：

```

DateTime createTime = fd.CreationTime;
DateTime lastAccess  = fd.LastAccessTime;
DateTime lastWrite   = fd.LastWriteTime;
long    fileLength    = fd.Length;
...

```

其中的 fd 表示一个 FileInfo object。

我们可以运用 `Attributes` property 来查询某个文件的属性 (attributes), 这些文件属性包括 `Archive` (存档)、`Compressed` (压缩)、`Encrypted` (加密)、`Hidden` (隐藏)、`Normal` (正常)、`ReadOnly` (只读) 等等, 它们都是 `FileAttributes` enum 型别的枚举元 (enumerators)。你可以把 `Attributes` property 所返回的 object 想象为一个 bit vector, 其中每个文件属性 (attributes) 不是设为 1 就是设为 0。例如我们可以按文件属性来查询某个 `FileInfo` object:

```
public static void displayFileAttributes( FileInfo fd )
{
    // Attributes returns a FileAttributes object
    FileAttributes fs = fd.Attributes;

    // use bitwise operators to see if attribute is set
    if ( ( fs & FileAttributes.Archive ) != 0 )
        // OK: file is archived ...

    if ( ( fs & FileAttributes.ReadOnly ) != 0 )
        fd.Attributes -= FileAttributes.ReadOnly;

    // ... and so on ...
}
```

如果想要修改文件的某个属性 (attributes), 可以对它减去或加上对应的 enum 值 (译注: 这个说法恐怕有问题, 比如连续两次加上某个 enum 值, 会影响其他文件属性), 当然我们得拥有必要的权限 (permissions) 才行。

5.4.4 读写文件 (Files)

开启某个已存在的文本文件 (text file) 并读写之, 办法很多。文本文件的实际读写任务分别由 `StreamReader` class 和 `StreamWriter` class 完成, 例如:

```
public static void StreamReaderWriter()
{
    StreamReader ifile =
        new StreamReader( @"c:\fictions\word.txt" );

    StreamWriter ofile =
        new StreamWriter( @"c:\fictions\word_out.txt" );

    string str;
    ArrayList textLines = new ArrayList();
```

```

while ( ( str = ifile.ReadLine() ) != null )
{
    Console.WriteLine( str );    // echo to Console
    textLines.Add( str );        // add to back ...
}

textLines.Sort();

foreach ( string s in textLines )
    ofile.WriteLine( s );

ifile.Close();
ofile.Close();
}

```

如果文件无法开启，无论是什么原因，都会导致异常被抛出。因此，在产品程序代码中，我们应该在传递文件至 `StreamReader` 和 `StreamWriter` 之前，先检测能否开启文件。以下就是一种策略，运用了 `FileInfo` class 的 `OpenText()` 和 `CreateText()` 两个成员函数：

```

public static void FileOpen( string inFile, string outFile )
{
    FileInfo ifd = new FileInfo( inFile );
    FileInfo ofd = new FileInfo( outFile );

    if ( ifd.Exists && ofd.Exists &&
        ((ofd.Attributes & FileAttributes.ReadOnly)==0) )
    {
        StreamReader ifile = ifd.OpenText();
        StreamWriter ofile = ofd.CreateText();
        // rest is the same as above
    }
}

```

如果你想在既有的输出文件末尾附加 (append) 一些文本，而不是覆盖文件中的原有文本，那么就不要使用上述的 `CreateText()`，改而使用 `FileInfo` 的 `AppendText()`：

```

StreamWriter ofile = ofd.AppendText();

```


如果使用 Stream 继承体系, 可以将文件当做字节序列 (sequence of bytes) 而非文本文件来读写, 例如:

```
FileInfo ifd = new FileInfo( inFile );
FileInfo ofd = new FileInfo( outFile );

Stream ifile = ifd.OpenRead();
Stream ofile = ofd.OpenWrite();
```

FileInfo 的 OpenRead() 和 OpenWrite() 都会返回一份 Stream 派生实体。Stream 的 Read() 函数接受三个参数, 第一参数是个 array, 用以存放被读取的 bytes, 第二参数指明从上述 bytes array 的哪个位置开始置入内容, 第三参数表明希望读取的最大 bytes 数: 返回的是实际读取的 bytes 个数, 返回 0 则表示已到达文件尾端。类似道理, 我们可调用 Write() 来写至 Stream。Write() 接受三个参数, 第一参数是个 array, 内含待被写出的 bytes, 第二参数指明从这个 array 的哪个位置开始写出, 第三参数表示希望写出的 bytes 个数:

```
const int max_bytes = 124;
byte []  buffer      = new byte[ max_bytes ];
int      bytesRead;

while ( ( bytesRead =
        ifile.Read( buffer, 0, max_bytes ) ) != 0 )
{
    Console.WriteLine( "Bytes read: {0}", bytesRead );
    ofile.Write( buffer, 0, bytesRead );
}
```

System.IO 命名空间中定义的若干 enum (枚举) 型别, 可用来指示文件被开启时的读/写/共享 (read/write/sharing) 等属性 (attributes), 例如:

```
Stream ifile = ifd.Open( FileMode.Open, FileAccess.Read,
                        FileShare.Read );

// default FileShare of None
Stream ofile = ofd.Open( FileMode.Truncate, FileAccess.ReadWrite );
```

FileMode 共有 6 个枚举元 (enumerators):

(译注: File.Open() 和 FileInfo.Open() 的区别在于, 后者无需指定文件名称)

1. Append, 如果文件存在, 就开启之, 并跳至文件末尾; 否则创建一份新文件。
2. Create, 创建一份新文件。如果文件已存在则覆盖之 (overwritten)。
3. CreateNew, 创建一份新文件。如果文件已存在则抛出异常。
4. Open, 开启一份既有文件。
5. OpenOrCreate, 如果文件存在, 就开启之; 否则创建一份新文件。
6. Truncate, 开启一份既有文件, 并将它截断 (truncate) 为 0 bytes。

FileAccess 枚举元 (enumeration) 定义了对已开启文件的三种访问模式 (缺省情况下文件是以 "ReadWrite" 方式被开启):

1. Read, 可读, 可从文件中读出数据; 可移动当前位置。这是 File.OpenRead() 所采用的模式。
2. ReadWrite, 可读可写, 可从文件读出数据, 也可对文件写入数据; 可移动当前位置。
3. Write, 可写, 可对文件写入数据; 不可移动当前位置。这是 File.OpenWrite() 所采用的模式。

FileShare 枚举元 (enumeration) 定义有若干值, 这些值用来控制“其他 Stream objects 可以何种方式访问这份文件”。如果你在开启某份文件时设置了 FileShare.Read, 那么其他用户也可以开启那份文件, 但只能读不能写。以下是 FileShare 的访问模式:

- None, 不允许共用 (share) 当前文件。随后针对这份文件的任何开启请求 (包括相同 process 内或不同 processes 内) 都会被拒绝; 直到这份文件被关闭, 才能允许其他请求。
- Read, 允许连续多次开启这份文件, 但只能读取。
- ReadWrite, 允许连续多次开启这份文件, 且可读可写。
- Write, 允许连续多次开启这份文件, 但只能涂写。

Stream 阶层体系允许我们以 Seek() 调整流 (stream) 的当前位置。我们可以一次读取一个 byte 或一组 bytes (存储在 array 中), 也可以写入一个 byte 或一组

bytes。如果想知道你对你手上的 Stream object 是否有读取、写入、定位 (seek) 的权限，可以看看下面这些 Boolean properties:

```
public void StreamSeek( Stream file, int offset )
{
    Console.WriteLine( "CanRead: {0}", file.CanRead );
    Console.WriteLine( "CanWrite: {0}", file.CanWrite );
    Console.WriteLine( "CanSeek: {0}", file.CanSeek );

    if ( ! file.CanWrite || ! file.CanSeek )
        return;

    // ...
}
```

举个例子，我们可以使用 File class 的 Open() 开启两个 streams，其中用到的 "mode"、"access"、"share" 三种枚举元 (enum) 如前所论：

```
public void BytesReadWrite( string inFile, string outFile )
{
    FileInfo ifd = new FileInfo( inFile );
    FileInfo ofd = new FileInfo( outFile );

    if ( ifd.Exists && ofd.Exists &&
        (( ofd.Attributes & FileAttributes.ReadOnly ) == 0) )
    {
        Stream ifile = ifd.Open( FileMode.Open,
                                FileAccess.Read,
                                FileShare.Read );

        Stream ofile = ofd.Open( FileMode.Truncate,
                                FileAccess.ReadWrite );

        // ... we read the same as before ...
    }
}
```

以下代码展示如何在“同时进行读、写操作”的文件中重新定位 (reposition)。其中 Length 和 Position 都是 Stream properties。Length 代表文件大小 (以 bytes 计)，Position 表示 stream 的当前位置：

```
// it's just been written to by ifile;
// let's flush it, just to be on the safe side
ofile.Flush();

long offset = ofile.Length/4 - 1,
    position = 0L;

for ( ; position < ofile.Length; position += offset )
{
    ofile.Seek( position, SeekOrigin.Begin );
    int theByte = ofile.ReadByte();
    ofile.WriteByte((byte)'X' );

    Console.WriteLine( "Position: {0} -- byte replaced: {1}",
        ofile.Position, theByte.ToChar() );
}

ifile.Close(); ofile.Close();
```

Seek() 接受两个引数。第二引数表示一个参考位置（基准点），第一引数表示相对于该参考位置的 bytes 偏移量。SeekOrigin 是个 enum 型别，它有三个枚举元（enumerators）：(1) Begin，用来将“参考位置”定为 stream 起始处；(2) Current，用来将“参考位置”定为“当前位置”；(3) End，用来将“参考位置”定为 stream 尾端。

5.5 System 杂项讨论

本节我们要研究数个有用的 classes，除了称之为“杂项”，我无法将这些 classes 归于某一类。这里提供的并非正式而详尽的介绍，我采用的表述方式更像是惊鸿一瞥。除非特别注解，这里提到的 classes 都直接位于 System 命名空间内。

5.5.1 System.Collections.Stack 容器

System.Collections 命名空间内含多种容器类（container classes），像 ArrayList（1.13 节）、Hashtable（1.16 节）、BitArray（4.9 节）等等。一般而言，容器存储的元素都是型别为 object 的 objects。Value 型别在每次读写操作时，会被装箱（boxed）与开箱（unboxed）。让我们粗略看一下 Collections 命名空间中的 Stack class。

Stack class 的 interface (接口) 描述如下:

- Clear(), 移除 (remove) stack 中的所有元素。
- Contains(object), 在 stack 中查找 object。
- Count, 返回 stack 中的元素个数。
- Peek(), 返回 stack 顶端元素, 但不移除之。
- Pop(), 返回 stack 顶端元素, 并移除之。
- Push(), 在 stack 顶端插入元素。
- ToArray(), 将 stack 内的元素复制到 array 之中。

为了说明如何运用 Stack, 让我们考虑以下问题。当 "TextQuery" 应用程序需要评估 (核定) 一个复式查询如下时:

```
Alice && ( fiery || untamed )
```

它会遇到“尚未评估完毕的操作符”和“独立的操作数”, 这些操作符和操作数有必要先暂存起来, 以备日后取回。因此, 复式查询的评估办法之一是, 定义两个 stacks, 一个持有当前操作符, 像是 && 或 ||, 另一个持有独立操作数。例如:

```
public class QueryManager
{
    // query operand and operator stacks for
    // processing the user query

    private Stack m_query_stack;
    private Stack m_current_op;
}
```

处理前述查询时, 我们先遇到 *Alice*, 并以 NameQuery 处理之。由于这时并没有与 NameQuery 相关的操作符, 所以我们将 NameQuery 压入 m_query_stack 内。一旦遇到一个操作符 (例如 AndQuery), 就从 m_query_stack 弹出一个元素, 作为 AndQuery 的左操作数。此时 AndQuery 的右操作数尚不可寻, 于是我们将 AndQuery 压入 m_current_op 中。举个例子:

```
case ')':
{
    // ...
```

```
if ( m_paren < m_current_op.Count )
{
    if ( m_query_stack.Count==0 ||
        m_current_op.Count==0 )
    {
        throw new Exception( "Internal Error: " +
            "Empty query or operator stack " +
            "for closing right paren!" );
    }

    Query operand = (Query) m_query_stack.Pop();
    Query op       = (Query) m_current_op.Pop();

    op.add_op( operand );
    m_query_stack.Push( op );
}

break;
}
// 注: 考配套代 chapter3\QueryManager\QueryManager.cs
```

Collections 中还有一个 Queue class, 它是以环状数组 (circular array) 实现的。也就是说, object 将被插入 queue 的一端; 而在另一端被移除。其容量 (capacity) 会自动增长。Queue class 的 interface (接口) 描述如下:

- Clear(), 移除 queue 内的所有元素。
- Contains(object), 在 queue 中查找 object。
- Count, 返回 queue 内的元素个数。
- Peek(), 返回 queue 头部对象, 但并不移除之。
- Dequeue(), 返回 queue 头部元素, 并移除之。
- Enqueue(), 在 queue 尾部插入元素。
- ToArray(), 将 queue 中的元素复制到 array 内。

5.5.2 System.Diagnostics.TraceListener Class

System.Diagnostics 命名空间提供了一整族的 TraceListener classes。利用它们, 用户可以开启并控制 (引导) 诊断信息的输出。例如第一章的 WordCount 程序中, 用户可以决定是否生成“追踪性输出”, 也可以决定在何处生成这些输出:


```

if ( m_trace != traceFlags.turnOff )
    switch ( m_trace )
    {
        case traceFlags.toConsole:
            cout = new TextWriterTraceListener( Console.Out );
            Trace.Listeners.Add( cout );
            break;

        case traceFlagsToFile:
            m_tracer =
                File.CreateText( startupPath + @"\trace.txt" );
            cout = new TextWriterTraceListener( m_tracer );
            Trace.Listeners.Add( cout );

            break;
    }

```

其中的 `TextWriterTraceListener` class 可以接受 `Stream` 或 `TextWriter` object 获得初始化。本例之中我们将它初始化为 `console` (控制台) 或一个名为 `trace.txt` 的文件。接着将 `TextWriterTraceListener` 添加到 `Listeners` 群集中, 该群集的功用是监听追踪性输出 (trace output), 后者由我们填写。例如:

```

private void writeWords()
{
    Trace.WriteLine( "!!! WordCount.writeWords: " +
        m_file_output );
    timer tt = null;

    if ( m_spy ){
        tt = new timer();
        tt.context = "Time to write file ";
        tt.start();
    }

    ArrayList aKeys = new ArrayList( m_words.Keys );
    aKeys.Sort();
    foreach ( string key in aKeys )
    {
        m_writer.WriteLine("{0} : {1}", key, m_words[ key ]);
        Trace.WriteLine( "!!! " + key + " : " +
            m_words[ key ].ToString() );
    }
}

```

```
if ( m_spy )
{
    tt.stop();
    m_times.Add( tt.ToString() );
}
```

其中的 `Trace.WriteLine()` 表示要向主管监视机关 (某个 `Listeners`) 输出信息。如果用户已向 `Trace` 中的 `Listeners` 群集添加了某个监视设备 (译注: 例如添加了 `TextWriterTraceListener`), 则 `WriteLine()` 的输出将被导至该处。

缺省情况下, 在 `Visual Studio` 中无论是调试 (`Debug`) 或发布 (`Release`) 模式, 都会开启追踪器。这意味着追踪用的代码总是存在。这样安排的好处是任何人都可以“窃听”程序代码而不必重新编译任何东西。当然了, 不利的一面是: 这些程序代码总是存在!

如果想在出货 (`Release`) 模式中消除这些起追踪作用的程序代码, 我们可利用 `Debug class` 来取代 `Trace class`。发布 (`Release`) 模式会关闭调试功能, 因此 `Debug class` 的任何成员函数都不会产生任何调试代码。

5.5.3 System.Math

`System` 命名空间中的 `Math class` 提供了许多 `static` 成员函数, 例如三角函数 (`Cos()`、`Sin()`、`Tan()`)、对数函数 (`Log()`、`Log10()`) 以及其他常用的数学函数 (`Abs()`、`Ceiling()`、`Floor()`、`Exp()`、`Min()`、`Max()`、`Pow()`、`Round()` 等等)。它还定义了两个属性 (properties): `E` 和 `PI` (分别是自然对数的底和圆周率)。以下是 `Vector class` 示例 (译注: 此处的 `Vector` 做为数学上的向量, 而非一种容器), 其中调用平方根函数 `Sqrt()`:

```
public class Vector
{
    public double length()
    { return Math.Sqrt( length_() ); }

    public double distance( Vector v )
    { return Math.Sqrt( distance_( v )); }
    // ...
}
```


Math class 用来收容东一样西一样的操作，这些操作之间彼此关联不大，唯一的共通性质仅仅在于：它们都是数学例程（routine）。在过程式（procedural）语言例如 C 中，这些 static 成员函数通常以彼此独立的全局函数出现，并聚合于一个名为 math 的程序库中。现在，你只要知道存在这么一个 Math class 即可，一旦需要某个标准数学例程，请先来这里找一找。

5.5.4 DateTime Class

DateTime class 用来表示日期和时间。它有两个 static 属性（properties）：Now 和 Today，前者表示当前的日期和时间，后者表示当前的日期（时间则设为午夜 0 点）。例如：

```
DateTime theDate = DateTime.Now;
```

如果要以字符串方式取用时间和日期，可调用以下的 DateTime 成员函数之一：

```
Console.WriteLine( "The current long date is {0}",  
                    theDate.ToLongDateString() );
```

```
Console.WriteLine( "The current short date is {0}",  
                    theDate.ToShortDateString() );
```

```
Console.WriteLine( "The current long time is {0}",  
                    theDate.ToLongTimeString() );
```

```
Console.WriteLine( "The current short time is {0}",  
                    theDate.ToShortTimeString() );
```

执行后，上述语句会生成类似下面的输出（译注：在 Windows 中文版环境下，上述第一个语句通常会输出中文）：

```
The current long date is Tuesday, February 27, 2001  
The current short date is 02/27/2001  
The current long time is 14:31:30  
The current short time is 14:31
```

有许多种格式字符串可供我们选择。DateTime 的各种输出格式列于表 5.1。举个例子，我最喜欢 *full*（完整）格式，因此我在代码中以大写字母 F 表示这个格式：

```
// generates this format: Tuesday, May 08, 2001 6:23:51 PM  
// （译注：中文 Windows 的输出结果像这样：2002 年 10 月 17 日 19:03:22）  
Console.WriteLine( "{0:F}", theDate );
```

表 5.1 DateTime 格式字符串 (Format Strings)

格式	输出	描述
"{0:d}"	5/8/2001	短日期
"{0:D}"	Tuesday, May 08, 2001	长日期
"{0:f}"	Tuesday, May 08, 2001 6:23 PM	完整 (长日期 + 短时间)
"{0:F}"	Tuesday, May 08, 2001 6:23:51 PM	完整 (长日期 + 长时间)
"{0:g}"	5/8/2001 6:23 PM	一般 (短日期 + 短时间)
"{0:G}"	5/8/2001 6:23:51 PM	一般 (短日期 + 长时间)
"{0:M}"	May 08	月/日
"{0:R}"	Wed, 09 May 2001 01:23:51 GMT	RFC 标准
"{0:s}"	2001-05-08T18:23:51	无时区, 可排序
"{0:t}"	6:23 PM	短时间
"{0:T}"	6:23:51 PM	长时间
"{0:u}"	2001-05-09 01:23:51Z	国际化短格式
"{0:U}"	Wednesday, May 09, 2001 1:23:51 AM	国际化完整格式
"{0:Y}"	May, 2001	年/月

DateTime 支持多种属性 (properties), 用以返回当前的 DateTime object 的各个部分: Date、Day、Hour、DayOfWeek、DayOfYear、Millisecond、Minute、Month、Second、Ticks、TimeOfDay、Year。

DateTime class 提供了大量成员函数: AddMinutes()、AddMilliseconds()、Parse() (从字符串中获得一个 DateTime object)、IsLeapYear()、AddDays()、AddHours()、DaysInMonth() 等等。

DateTime 有七个重载构造函数, 以下是其中四个:

```
DateTime( long Ticks );
DateTime( int year, int month, int day );
DateTime( int year, int month, int day,
          int hour, int min, int sec );
DateTime( int year, int month, int day,
          int hour, int min, int sec,
          int millisecond );
```


5.6 正则表达式 (Regular Expressions)

《狂想曲 2000》(Fantasia 2000) 电影中的火鸟 (Firebird) 段只有三个角色：一个小妖精、一只火鸟、一只颇爱挑剔的麋鹿。尽管这头麋鹿是三个角色中最乏善可陈者，不过它的画面却最值得一提。它的身体和面部表情是以迪斯尼传统动画技术手工绘制的，它的一对鹿角则是计算机三维模型。我们遭遇的挑战是，如何将鹿角的运动和手绘的二维动画协调好。这个问题的解决，不仅将这头麋鹿送入了《狂想曲 2000》，也让我长进了不少。

我的计算机图像总监 Chyuan 想出一个绝妙主意：将手绘的麋鹿头部的运动先以摄像机捕获下来，再将这些运动转换为一组曲线，反馈给三维动画软件。在场景的开始处，先手工把鹿角安放在麋鹿的头上，鹿角按先前输入的曲线运动，而与麋鹿头部的运动相吻合，于是整个场景就天衣无缝了。

Chyuan 耐心地向我解释其中的数学 (算法)，我完成实际编程工作。先是采用 C++，后来改用 Perl，这是一种脚本语言 (scripting language)。Perl 程序比 C++ 程序小得多，这很大程度上得益于 Perl 的重要组成——正则表达式 (regular expression)。.NET 也支持正则表达式，位于 `System.Text.RegularExpressions` 命名空间内。这正是本节的主题。

何谓正则表达式？这是一个由字符和符号组成的模式 (patterns)，用来代表任意长度的字符序列。举个例子，我们想找出满足下列条件的所有文本行：以整数起头，该整数的最高位是 5 且长度不限，其后紧跟一个连字号 (-)，而后是 a、b、c 三个字母之一，之后再跟着一个或多个字母或其他字符；整个文本行最后以 2001 结尾。为了能够使用正则表达式，我们需要一些记号 (symbols) 用以实现以下工作：

- 指明希望从文本行起始处进行查找。我们以字符 '^' 标明这一企图，因此 ^5 意味着我们想找出“以字面值 5 开头”的文本行。

- 指明希望匹配特定字符。我们以 `\d` 表示想要匹配单个数字 (single digit, 0 至 9), `\D` 表示希望匹配非数字 (nondigit) 的单一字符, `\s` 表示希望匹配单个空白字符 (white-space), `\S` 表示希望匹配非空白 (not white-space) 的单一字符, `\w` 表示希望匹配单个任意文数字 (alphanumeric, a 至 z, A 至 Z, 下划线 '_', 0 至 9), `\W` 表示希望匹配任何非文数字 (not alphanumeric, 译注: 例如标点符号、空白等等)。
- 指明希望匹配任意字符, 不论其类型为何。例如以小数点 (.) 表示希望匹配任何非换行 (non-newline) 字符。
- 指明希望匹配多个 (或零个) 特定类型的字符。我们以加号 (+) 表示希望匹配一个 (含) 以上同类字符, 例如 `\d+` 意指匹配 2、22、1217 等等。星号 (*) 则表示我们允许无匹配 (no matches) 情况发生。例如以下正则表达式:

```
^5\d+\D+2001
```

能够匹配“以 5 开头, 后跟一个或多个数字, 而后跟着一个或多个非数字字符, 而后是字面值 2001”的文本行。以下正则表达式:

```
^5\d*\D*2001
```

则能够匹配“以 5 开头、以字面值 2001 结尾, 其间可带其他字符, 这些字符必须以零个或多个数字开头, 其后跟着零或多个非数字字符”的所有文本行。

- 指明希望匹配固定数量的字符, 例如以下正则表达式:

```
\d{3}-\d{4}
```

能够匹配“一个三位数和一个四位数, 其间有个连字号 (-)”的字符串, 例如 375-4128。

- 指明希望匹配“若干字符所组成的集合”中的一个字符。做法是将一组字符以圆括弧括起来, 字符之间以 `bool OR` 操作符 (|) 分隔。正则表达式 `a|e|i|o|u` 表示匹配英语五个元音字母之一。如果加上 + 号, 亦即 `(a|e|i|o|u)+`, 就表示匹配一个元音字母或多个连续出现的元音字母。如果再在最后附加一个星号 (*), 表示允许“无匹配”情况发生。

正则表达式必须多多使用才能熟悉。初学的时候, 这些简洁符号可能看起来十分复杂。为了促进你对正则表达式的探究, 我写了一个小小的测试程序。对它输入一个字符串、一个正则表达式, 或二者都输入, 该程序能够找出与正则表达式相匹

配的字符串的出现位置。例如（请注意，我的控制台输入以黑体显示）：

```
Would you like to enter a string to match against? (Y/N/?) y
Please enter a string, or 'quit' to exit.
==> 5abc2001
```

```
Would you like to change regular expressions? (Y/N/?) y
Please enter regular expression:
**> ^5\d*(a|d|e)\w+2001
```

```
original string: 5abc2001
attempt to match: ^5\d*(a|d|e)\w+2001
```

The characters 5abc2001 match beginning at position 0

```
Would you like to enter a string to match against? (Y/N/?) y
Please enter a string, or 'quit' to exit.
==> 527ar2001
```

```
Would you like to change regular expressions? (Y/N/?) n
```

```
original string: 527ar2001
attempt to match: ^5\d*(a|d|e)\w+2001
The characters 527ar2001 match beginning at position 0
```

当然，也可能出现多处匹配的情况，例如：

```
original string: r24d2
attempt to match: \d+
The characters 24 match beginning at position 1
The characters 2 match beginning at position 4
```

现在，我们开始着手在 .NET 之下使用正则表达式来编程。首先我向你展示进行“匹配”的程序代码，然后解释它：

```
public static void doMatch()
{
    Console.WriteLine( "original string: {0}", inputString );
    Console.WriteLine( "attempt to match: {0}", filter );

    Regex regex = new Regex( filter );
    Match match = regex.Match( inputString );
```

```
if ( ! match.Success )
{
    Console.WriteLine( "Sorry, no match of {0} in {1}",
        filter, inputString );
    return;
}

for ( ; match.Success; match = match.NextMatch() )
{
    Console.WriteLine(
        "The characters {0} match beginning at position {1}",
        match.ToString(), match.Index
    );
}
```

Regex class 代表我们的正则表达式。我将代表正则表达式的字符串传给 Regex 的构造函数，而后这个字符串被转换为一种内部表述形式 (internal representation)，而这种内部表述法是一次成型不可更改的 (immutable) (译注：通常可将正则表达式转换为等价的有限状态机 (Finite-State Machine, FSM)，也就是不能更改任何 Regex object 所关联的正则表达式)。构造函数也可接受两个参数，第二个字符串引数内含“选项字符”，用来修改匹配模式 (译注：目前第二参数的型别为 RegexOptions)。

Match() 会对其 string 引数执行正则表达式匹配算法。它会返回一个 Match class object，其中持有模式 (patterns) 匹配结果。Match object 也是不可更改的。

如果想知道是否匹配成功，我们应该查询 Match class 的 Success property。每次匹配我们称为一次 "capture" (捕获)。Index property 会返回原始字符串中的一个位置，被捕获之子字符串的第一个字符就位于该处。Length 会返回被捕获之子字符串长度。ToString() 会返回被捕获之子字符串。

Match() 返回的 Match object 持有第一次捕获结果。如果正则表达式能够捕获多个子字符串，我们可使用 NextMatch() 取得第二次 (以及更后面的) 捕获结果。在操控第二次取回的 Match object 之前，我们必须检测匹配是否成功。有一个作为哨兵用途的 Match object，其 Success 被评估 (核定) 为 false，表示没有更多捕获结果了。典型的 for 循环如下：


```

for ( Match match = regex.Match( inputString );
      match.Success;
      match = match.NextMatch() )
{ ... }

```

现在，考虑以下三行文本：

```

5040      bez( 99,   -3.194,  43.8,  85 )
4930.7823 bez( 10.7, 19.59, -20,   -20.48 )
-5123     bez( -3.5,  2.46,  89,    0.02 )

```

这些是我们想要匹配的文本行样例。首先我们要构造一个能够匹配以上文本行的正则表达式。

我们发现，每行文本都以一个数起头。这个数可正可负，既可以是整数也可以是浮点数。其后跟着一个空格（space），然后是字面字符串 bez，其后是一对圆括弧，括弧中有四个以逗号分隔的数。这四个数可正可负，既可以是整数也可以是浮点数。在观看解答之前，请你动手试试能否构造出一个正则表达式来捕获（匹配）上述每一行。

即使你完成了那样一个正则表达式，事情离结束也还早得很。下一个问题是如何取得文本行中的个别部分。也就是说，正则表达式捕获了整个字符串，而现在我们需要取出 5 个数值段（numeric fields）。

正则表达式的语法支持一种分组机制，可将一个匹配的特定子段（subfield）赋予一个标识数目，然后可用这些数目来访问这些子段，例如以下便是使用特殊的 ?<1> 语法，将一个“子段”标识为 1：

```
(?<1>(-*\d+.\d+)|(?*\d+))
```

你能读懂它吗？这表示一组“二选一”的正则表达式。头一个表达式：

```
-*\d+.\d+
```

匹配一个浮点数（无论有无负号）（译注：严格地说这个表达式能匹配 "---1.2" 这样的文本，但那并不是个有效的浮点数。因此欲匹配一个“正负皆可的浮点数”，比较好的正则表达式或许应该是 `-?\d+.\d+`，但也未臻完美）。第二个表达式：

```
-*\d+
```

能匹配一个整数（无论正负）。整个正则表达式，带有五个具备识别符的子段，看起来像是下面这个样子。为了清楚起见，我将每个子段列于一行。为了保证真实性，我将这个表达式列为一个字符串字面值，并使用必要的双反斜线（\\）转义字符（`escape`）（译注：以下与原书稍有差异。原书竟连同代码注释都放进正则表达式的字符串内）：

```
//译注：这里作者似乎没有考虑各数之间的空白字符
string filter =
    // the digit before the bez literal
    "(?<1>(-*\\d+\\.\\.\\d+)|(-*\\d+))" +

    // arbitrary white space, bez literal, and open paren
    "\\s*bez\\(" +

    // the four internal numeric values and literal comma
    "(?<2>(-*\\d+\\.\\.\\d+)|(-*\\d+)), " +
    "(?<3>(-*\\d+\\.\\.\\d+)|(-*\\d+)), " +
    "(?<4>(-*\\d+\\.\\.\\d+)|(-*\\d+)), " +
    "(?<5>(-*\\d+\\.\\.\\d+)|(-*\\d+))";
```

现在我们尝试以这个正则表达式来匹配前述那些文本行：

```
Regex regex = new Regex( filter );
Match match = regex.Match( line );
```

如果匹配成功，我们要取得前述文本行的五个数值段，并将它们转换为 `float` 值：

```
float loc = Convert.ToSingle(match.Groups[1].ToString());
float m_xoffset1 = Convert.ToSingle(match.Groups[2].ToString());
float m_yoffset1 = Convert.ToSingle(match.Groups[3].ToString());
float m_xoffset2 = Convert.ToSingle(match.Groups[4].ToString());
float m_yoffset2 = Convert.ToSingle(match.Groups[5].ToString());
```

其中 `Group` class 表示 `Match` object 中的一个被捕获群（`capturing group`），我们通过下标（索引）访问其中各元素。`ToString()` 会返回被捕获的子字符串。本例中我对每个字符串调用 `Convert.ToSingle()`，借此将字符串所代表的值转换为 `float` 型别。

`Regex` class 的另一个非常有用的成员函数是 `Split()`。这个函数与 `String` class 的 `Split()` 成员函数非常相似，两者都返回一个 `string array`。然而与 `String` class 的 `Split()` 不同的是，`Regex` class 的 `Split()` 按照某个正则表达式（而非一组字符）来分割输入的字符串，例如：


```

string textLine =
    "Danny%Lippman%%Point Guard%Shooting Guard%%floater";
string splitMe = "%+";
Regex regex = new Regex( splitMe );

foreach ( string capture in regex.Split( textLine ) )
    Console.WriteLine( "capture: {0}", capture );

```

本例我们将 textLine 进行分割, 以一个或多个 % 字符的出现处为分割点, 执行这段代码, 会生成以下输出:

```

capture: Danny
capture: Lippman
capture: Point Guard
capture: Shooting Guard
capture: floater

```

Regex class 还有一个很有用的成员函数 Replace(), 可用来将被捕获的子字符串替换为其他文字。以下是简单示例:

```

public static void testReplace()
{
    string re = "XP.\\d+";
    Regex regex = new Regex( re );

    string textLine =
        "XP.109 is currently in alpha. " +
        "XP.109 represents a staggering leap forward";

    string replaceWith = "ToonPal";

    Console.WriteLine ( "original text: {0}", textLine );
    Console.WriteLine ( "regular expresion : {0}", re );

    string replacedText = regex.Replace( textLine, replaceWith );
    Console.WriteLine ( "replacement text: {0}", replacedText );
}

```

编译并执行这段代码, 会生成以下输出 (为了美观, 我稍微重排了一下格式):

```

original text: XP.109 is currently in alpha. XP.109 represents a
               staggering leap forward
regular expresion : XP.\\d+
replacement text: ToonPal is currently in alpha. ToonPal
                  represents a staggering leap forward

```

5.7 System.Threading

多线程 (Multithreading) 可使一个进程 (process) 能够并行 (parallel) 处理多个任务 (tasks)。每个任务都有自己的“执行线程”。例如在先前的文字查询系统中, 我们也许想要同时运行各个独立子查询 (subqueries), 这可以通过“为每个子查询的评估动作 (evaluation) 分配一个线程”来实现。至于主控程序 (或称主线程), 既可以让它暂停工作直到所有负责评估的线程完成任务, 也可以让它在那些线程正在运行时, 做一些辅助性的工作。

多线程的好处之一是, 它有可能提高性能。在一个单线程程序中, 查询时间是每个子查询的运行时间总和, 而每个子查询是一个接一个依次运行的。在多线程程序中, 时间消耗将减为“耗时最久的那个子查询所用去的时间”加上“因支持多线程而招致的额外开销”, 因为所有子查询都是并行 (parallel) 运行的。(译注: 这种说法只在多 CPU 机器上才成立。在单 CPU 机器上, 虽然多线程的逻辑概念是并行处理, 但其实每一件事、每一个动作、每一个任务最终都要落到唯一那颗 CPU 身上, 并没有“并行”发生。多线程在单 CPU 机器上一定比单线程耗费更多时间, 因为它还需要一些额外开销。多线程在单 CPU 机器上的最大好处在于改善 UI, 如下所述)

第二个好处是, 主线程 (main thread) 得以将子任务分派给并行的线程去执行。主线程因而空闲下来, 可以维持与用户良好的 (不必太多等待的) 交互行为。

多线程程序设计的主要缺点是, 必须维持共用数据和资源 (包括窗体 (forms)、文件, 甚至控制台 (console) 的完善性 (integrity)。稍后我们即将看到, 所谓的“monitors”能够在多个线程访问临界区段 (critical section) 时, 维持这些线程的同步 (synchronization)。

System.Threading 命名空间提供了对线程的支持, 其中包含诸如 Thread、Mutex、Monitor、Timer、ThreadPool 之类的 classes。首先让我们看一个简单程序, 从中练习如何使用 Thread 的接口 (interface)。这个程序将要创建两个线程, 一个负责填写单词 ping、另一个负责填写单词 PONG¹⁴。

¹⁴ 这是一个极佳的线程示例的变体, 原始程序出自 *The Java Programming Language*, by Ken Arnold and James Gosling, Addison-Wesley, 1996.


```

using System.Threading;

public static void Main()
{
    PingPong p1 = new PingPong( "ping", 33 );
    Thread ping = new Thread( new ThreadStart( p1.play ) )
    ping.Start();

    while ( ! ping.IsAlive )
        Console.Write( "." );

    // OK: now Main() and ping are executing in parallel

    PingPong p2 = new PingPong( "PONG", 100 );
    Thread PONG = new Thread( new ThreadStart( p2.play ) );
    PONG.Start();

    // OK: now Main(), ping, and PONG are executing ...
    // OK: let's rest this puppy for 100 milliseconds;
    //      both ping and PONG continue to run in parallel
    Thread.Sleep( 100 );

    /*
    * OK: another way of resting this puppy
    *
    * this main thread waits until either
    *      ping completes or 100 milliseconds pass ...
    *
    * the PONG thread is unaffected;
    * both threads continue to run in parallel
    */
    ping.Join(100);

    // let's suspend PONG for a moment while we determine
    // if ping completed or Join() timed out ...
    PONG.Suspend();

    if ( p1.Count != PingPong.Max )
        // Join() timed out ...
        // ping must still be executing

    // OK: let's resume PONG
    PONG.Resume();
}

```

```
// let's absolutely wait for PONG to complete
PONG.Join();

Console.WriteLine( "OK: ping count: {0}", p1.Count );
Console.WriteLine( "OK: PONG count: {0}", p2.Count );
}
```

每个 PingPong class object 都以“欲显示之字符串”和“两次显示之间的延迟时间（毫秒）”进行初始化：

```
PingPong p1 = new PingPong( "ping", 33 );
PingPong p2 = new PingPong( "PONG", 100 );
```

play() 是 PingPong class 的成员函数，用于打印相应字符串。在其函数实现中，我们唯一还不熟悉的是，利用 Thread class 的 static Sleep() 实现以毫秒计的延迟：

```
public void play()
{
    for ( ; ; )
    {
        Console.Write( theWord+ " " );

        Thread.Sleep( theDelay );
        if ( ++theCount == theMax )
            return;
    }
}
```

theCount 记录了字符串的显示次数。theMax 是个 const 成员，内置字符串的最高显示次数。

Sleep() 将当前线程（current thread）挂起（suspend）一段指定时间（以毫秒计）。所谓当前线程一般而言是指目前运行中的那一个。当然，当 ping 线程调用 play() 时，ping 就是当前线程。theDelay 被设为 33，表示 ping 将休眠 33 毫秒。

“当前线程”并不总是一个具名的 thread object（像我们这里所讨论的东西）。如果我们的程序中根本不使用线程，我们不能说它是个无线程程序，我们称此为单线程（single-threaded）程序。主线程永远伴随 Main() 而产生。我们的程序在 Main() 内部调用 Sleep()，将主线程（而非 ping 线程）休眠 100 毫秒，如下所示：


```
public static void Main()
{
    PingPong p1 = new PingPong( "ping", 33 );
    Thread ping = new Thread( new ThreadStart( p1.play ) );
    ping.Start();

    Thread.Sleep( 100 );
}
```

Thread 构造函数接受单个引数：一个名为 ThreadStart 的 delegate 型别。当操作系统启动一个线程时，会调用 ThreadStart 所指的那个函数。我们的两个 thread objects 都被初始化为“调用 play()”，而且是通过不同的 PingPong class object 来调用 play()：

```
Thread ping = new Thread( new ThreadStart( p1.play ) );
Thread PONG = new Thread( new ThreadStart( p2.play ) );
```

除非我们明确（显式）调用 Start()，否则线程不会开始运行：

```
ping.Start();
```

一旦 ping 线程开始运行，便通过 p1 调用 play()。一旦 play() 运行完毕，ping 线程将不复存在，我们称这个线程死了（dead）。

启动一个线程之后，我们能做些什么呢？以下是若干选择：

- 完全不理睬，听其自然发展。球一旦开出去（调用 Start()），我们就不管了。
- 调用 Suspend() 来停止（暂停、挂起）线程。被挂起的线程将被集中于某个地方，我们无须操心。
- 如果线程被挂起（suspend），我们可以调用 Resume() 再次启动它。如果线程未被挂起，对它调用 Resume() 并不会招致损害。我们可以对已被挂起的线程再次调用 Suspend()，这不会导致不利结果。同样道理，针对一个未被挂起的线程调用 Resume()，也不会导致不利结果。
- 如果“挂起”（suspend）过于严厉，我们可以调用 Sleep() 使线程进入休眠，但这只对当前线程有效。没有哪个 Sleep() 函数可施行于你所指定的线程身上，它只能用于“当前线程”。
- 我们可以通过 IsAlive property 来查询一个线程是否处于活动状态。IsAlive 只在“线程被启动且尚未死亡”的这段期间才返回 true。一旦 ThreadStart

object 执行完毕，线程也就死亡了，我们也可以调用 `Abort()` 直接杀死一个线程。

通过 `Join()` 成员函数，我们还可以令当前线程进入等待状态，直到某个特定线程运行完毕并终止（译注：这个函数有连接的意思，表示将两根“线”头尾相连）：

```
public static void Main()
{
    // OK: we resume PONG
    PONG.Resume();

    // OK: we sit here until PONG completes
    PONG.Join();
}
```

本例中，`Main()` 一步不移地等待，直至 PONG 运行完毕，并且（想必）其运行结果可资运用。当然，如果 PONG 陷入无限循环，`Main()` 就永远等待下去，因此有时候我们得为我们的等待附加一个条件，也就是为 `Join()` 提供一个等待时限：

```
// OK: let's rest this puppy for 100 milliseconds;
//      both ping and PONG continue to run in parallel
Thread.Sleep( 100 );

/*
 * OK: another way of resting this puppy
 *
 * this main thread waits until either
 *      ping completes or 100 milliseconds pass ...
 *
 * the PONG thread is unaffected;
 * both threads continue to run in parallel
 */
```

```
ping.Join(100);
```

线程编程的难点在于，如何维持内存和其他共享资源（shared resources）的完善性（健全性，integrity）。例如，以下是上述程序（译注：请见本书配套源码）输出的一部分，以 OK 起头的非黑体文字是 `Main()` 生成的，以黑体突出显示的文字是由 ping、PONG 两个异步（asynchronous）线程生成的：


```

OK: about to start ping thread
OK: ping thread is now alive!
OK: ping is now running in parallel
OK: 0 Within PingPong.play() for ping!!!
ping 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
OK: creating PONG object
OK: creating PONG thread object
OK: about to start PONG thread
OK: ping and PONG are now running in parallel
OK: 0 1 2 3 4 5 6 Within PingPong.play() for PONG!!!
PONG 7 8 9 10 11 12 13 14 15 16 17 18 19

OK: about to put main thread to sleep for 600 milliseconds
ping ping PONG ping ping ping PONG ping ping PONG
ping ping ping PONG ping ping PONG ping ping ping
PONG OK: here after sleep

OK: about to wait to Join(400) ping
ping ping PONG ping ping ping PONG ping ping PONG
ping ping ping OK: back now -- hi.

```

这个程序中，三个并行线程（parallel threads）的交错输出非常有趣——好吧，保守点地说是稍微有趣吧。在真实应用中，我们必须在“并行的独立线程”所发出的竞争需求（competing demands）下保证资源完善性（健全性，integrity）。办法之一是利用所谓的 monitors（管程、监控器）。

Monitor class 可以通过 Enter() 和 Exit() 这一对上锁（lock）与解锁（unlock）函数，使线程对临界代码块（critical code blocks）的访问达到同步（synchronize）。例如：

```

Monitor.Enter( this );
try
{
    // critical text goes here ...
}
finally{
    Monitor.Exit( this );
}

```

Enter() 是个 static 函数，一旦被调用，便将它所收到的 object 上锁（lock）。如果这个 object 先前已经被（别的 monitor）上了锁，目前的这个线程就不能继续运行下去。这时候我们称此线程发生了阻塞（blocks）。如果 object 目前没有上锁，

就给它上锁并执行 `Enter()` 之后的程序代码。在解除 `object` 身上的枷锁之前，我们的线程对于 `Enter()` 之后的程序代码享有独占访问权 (exclusive access)。

`Exit()` 也是个 `static` 函数，它会将它所收到的 `object` 身上的枷锁解除。如果此时有一个或多个线程正等着这个枷锁被解除，那么这些线程之一将会因此“解除阻塞” (unblocked) 并接续运行下去。如果我们对某个 `object` 未能提供成对的 `Enter()/Exit()`，代码段有可能无限期地处于锁定状态。

`TryEnter()` 并不会造成阻塞 (block)；或者它只阻塞特定时间 (以毫秒计)，此后便放弃进入并返回 `false`。例如：

```
if ( ! Monitor.TryEnter( fout, maxWait ))
    { logFailure( file_name ); return; }
```

`Monitor` 通过“给 `object` 上锁”来保证对临界代码块的独占访问。为了方便程序“成对地调用 `Monitor.Enter()/Monitor.Exit()`”，C# 提供了另一种简化记号：关键字 `lock`。例如以下代码：

```
// equivalent to the earlier
// Monitor.Enter()/Monitor.Exit() code block
lock( this )
{
    // critical text goes here ...
}
```

这一段代码等价于先前 (上一页) 以 `Monitor.Enter()` 开头的那一段代码。

最后，让我简略提一下 `ThreadPool` class。这个 class 管理一池子 (pool) 预先分配的线程 (preallocated threads)。 `ThreadPool` 可将一个 `callback` (回调) 函数关联到某个 `object` 的等待状态。一旦等待结束 (亦即某个操作完成)，池中某个可用线程就会调用相应的 `callback` 函数。我们也可以将函数以队列 (queue) 方式加入线程池，并当某个线程变为可用时，调用该函数。

5.8 Web 的请求/响应模型 (Request/Response Model)

本节让我们体验一下访问 Internet 上的网页 (页面) 需要经历哪些步骤。我们将使用 `WebRequest` class、`WebResponse` class、`Uri` class。这些 classes 都定义于 `System.Net` 命名空间内 (译注：Uri 其实是定义于 `System` 命名空间)。让我们先创建一个名为 `PageReader` 的 class。首先需要有一个 URL (Uniform Resource

Locator, 统一资源定位符) 地址, 用户可以传给 PageReader 构造函数一个“表示某个 URL 地址”的字符串; 缺省情况下采用作者本人的公司主页地址:

```
public class PageReader
{
    private Uri uri;
    private bool validateUrl( string url ) { ... }

    public PageReader( string url )
    {
        if ( url != null && url != String.Empty &&
            validateUrl( url ) )
            uri = new Uri( url );
    }

    public PageReader()
        : this( "http://www.objectwrite.com" ) {}

    // ...
}
```

Uri 构造函数会分析字符串, 将它转换为小写规范形式 (译注: 例如将主机名称转为小写), 例如将 c# 转为 c%23。如果传入一个非法字符串, 会导致抛出异常 UriFormatException。字符串必须表示绝对路径, 例如它不接受 www.amazon.com 这样的字符串。字符串之中必须有 http:// (或 file://) 这样的前缀。

Uri class 提供“资源” (译注: URI 的本意就是统一“资源”标识符) 的各个只读属性供外界访问, 例如 Host、HostNameType、Port、Query 等等。例如以下的一个 url 字符串 (其中有个内嵌查询):

```
string url = @"http://www.amazon.com/exec/obidos/search-handle-form/002-9257402-0511232?index=books&field-keywords=C#";
```

我们可以运用 Uri 查询这个“资源”的各个属性, 像这样:

```
Uri uri = new Uri( url );
Console.WriteLine( "Uri: " + uri.AbsoluteUri );
Console.WriteLine( "Uri host: " + uri.Host );
Console.WriteLine( "Uri host type: " + uri.HostNameType );
```

```
Console.WriteLine( "Uri port: "      + uri.Port );
Console.WriteLine( "Uri path: "      + uri.AbsolutePath );
Console.WriteLine( "Uri query: "     + uri.Query );
Console.WriteLine( "Uri toString: " + uri.ToString() );
```

编译并运行这段程序后, 生成以下输出:

```
Uri: http://www.amazon.com/exec/obidos/search-handle-form/002-925
7402-0511232?index=book&field-keywords=C%23
Uri host: www.amazon.com
Uri host type: Dns
Uri port: 80
Uri path: /exec/obidos/search-handle-form/002-9257402-0511232
Uri query: ?index=book&field-keywords=C%23
Uri toString: http://www.amazon.com/exec/obidos/search-handleform/
002-9257402-0511232?index=book&field-keywords=C#
```

Uri object 不可被更改。如果想修改其属性 (properties), 应使用 UriBuilder class。在使用策略上, Uri 和 UriBuilder 的关系类似于 String 和 StringBuilder。

一旦有了 Uri object, 下一步就是创建 WebRequest object。办法是将一个 Uri object 传给 WebRequest 的 static 成员函数 Create(), 例如:

```
WebRequest wreq = WebRequest.Create( uri );
```

Create() 返回的 WebRequest object, 是一个支持特定网络协议 (如 HTTP 或 FTP) 的 derived class 实体。当然了, 协议细节被封装于 WebRequest 所代表的那个 class 继承体系中。

我们使用一套通用操作来编写程序, 这套与协议无关的操作并不复杂, 而且不易出错; 这样便能让我们从“错综复杂的底层 Internet 连接技术”中解脱出来。网络专门技术被封装于 derived classes 内, 缺省情况下我们不需要直接使用这些 classes。

如果高阶接口 (interface) 的灵活性不足以满足我们的请求 (request), 我们可以将 WebRequest object 显式向下转型 (downcast) 为某个 derived class。举个例子, HttpWebRequest class 能够管理 HTTP Internet 连接细节:


```

WebRequest wreq = WebRequest.Create( uri );

// the downcast to the specific derived instance
HttpWebRequest hwreq = ( HttpWebRequest )wreq;

```

下面的 `GetResponse()` 将 client 程序端的请求信息传给 (`Uri` object 代表的那个) server, 返回一个 `WebResponse` object, 用以访问 server 返回的数据。像这样:

```
WebResponse wresp = wreq.GetResponse();
```

现在我们可以像下面代码那样, 通过 “`GetResponseStream()` 返回的 `System.IO.Stream` class object” 来访问 server 返回的数据。这里我们仿佛又回到了 “从 stream (串流) 读入” 的处理模式。以下代码将各行文本存入一个 `ArrayList` object 内:

```

Stream      wrespStream = wresp.GetResponseStream();
StreamReader wsrdr      = new StreamReader(wrespStream);
ArrayList   webData      = new ArrayList();
string      data          = null;

while ( ( data = wsrdr.ReadLine() ) != null )
    webData.Add( data );

```

下面这段代码检验 array 的每个元素, 查找 “以 `http://` 起始” 的 URL 地址:

```

Console.WriteLine( "read {0} lines of text from {1}",
    webData.Count, uri.AbsoluteUri );

ArrayList webUrls = new ArrayList();
foreach ( string s in webData )
{
    int pos, nextPos;
    int spacePos, quotePos;
    char space = ' ', quote = '\"';

    if ( ( pos = s.IndexOf( "http://" ) ) != -1 )
    {
        spacePos = s.IndexOf( space, pos );
        quotePos = s.IndexOf( quote, pos );
        nextPos = spacePos < quotePos
            ? spacePos : quotePos;
    }
}

```

```
        if ( nextPos > pos )
        {
            string surl = s.Substring( pos, nextPos - pos );
            if ( ! webUrls.Contains( surl ) )
                webUrls.Add( surl );
        }
    }

Console.WriteLine("There are {0} url references",webUrls.Count);
webUrls.Sort();

foreach ( string s in webUrls )
    Console.WriteLine( "\t{0}", s );
```

如果以我的个人主页为分析目标，上述代码运行后将生成以下输出：

```
read 117 lines of text from http://www.objectwrite.com
There are 5 url references
http://cseng.awl.com/bookdetail.qry?ISBN=0-201-30993-9&ptype=0
http://www.amazon.com/exec/obidos/ASIN/0135705819/
qid%3D902875557/sr%3D1-6/002-5252584-4839230
http://www.awl.com/cseng/titles/0-201-82470-1/
http://www.awl.com/cseng/titles/0-201-83454-5/
http://www.objectwrite.com/
```

5.9 System.Net.Sockets

System.Net.Sockets 命名空间提供了一个 Socket class，其中封装 Windows Sockets 的各种功能（译注：Network Programming for Microsoft Windows, Second Edition by Anthony Jones and Jim Ohlund 是 Windows 网络编程方面相当值得参考的一本专著）。TcpListener class 与 TcpClient class 提供了对 TCP（Transmission Control Protocol，传输控制协议）的支持。为了展示这些 classes 的使用方法，我们以 Northwind SQL 数据库为数据源，创建一个 client/server 式的电话目录程序。Server 使用两个线程分别实现“取回数据库信息”和“创建 TcpListener object 以接受送进来的请求（incoming requests）”。Client 创建一个 TcpClient object，以此获得一个 NetworkStream object，然后包装一个请求（request）并通过网络发送这个请求。图 5.1 所示的是某个 client session 画面，方括弧内显示的是 client request 的端口号（port number）。标有“Server”和电话号码的那一行是从 server 取回的数据，这些数据是通过“从 TcpClient 取得之 NetworkStream object”获得的。


```

C:\AC#Programs\trySockets\bin\Debug\trySockets.exe
Beginning: send and receive socket demo ...
Usage: SocketDemo <data>
Please enter data : Fuller

Client: About to access socket with data: Fuller
Client: Time now: 7/23/2001 7:02 PM

Client[4554]: got TcpClient: System.Net.Sockets.TcpClient
Client[4554]: got NetworkStream: System.Net.Sockets.NetworkStream
Client[4554]: write data to server stream.

Server[4554]: Phone number for Fuller: <206> 555-9482

Ending: send and receive socket demo ...
Press any key to continue

```

图 5.1 Socket 示例程序, Client 端

图 5.2 所示的是某个 server session 的截图, 这个 server session 处理了三个传入请求 (requests)。Server 启动时对各项操作都有追踪记录 (trace), 包括启动 TcpListener 和取回 SQL 数据库信息等等, 方括弧内显示的是 server 端口号。让我们先来解决 server 端的实现。

5.9.1 服务器端 (Server-Side) 的 TcpListener

TcpListener 用来侦听 (listen) TCP clients 的连接 (connections)。首先请创建一份 TcpListener 实体, 此处我们传给构造函数一个 port (端口) 号, 这个号码在 clients 发送请求 (requests) 时也会用到。一旦建构出 TcpListener object, 接着就调用其 Start() 成员函数, 令它开始侦听传入的请求:

```
public class SocketDemo_Server
```

```
{
    static private int port = 4554;
    private TcpListener tcpl;
```

```
public SocketDemo_Server()
```

```
{
    // start listening on the assigned port
    tcpl = new TcpListener(port);
    tcpl.Start();
}
```

```
C:\VC#Programs\socketServer\bin\Debug\socketServer.exe
Server[4554]: OK: started TcpListener ...
Server[4554]: OK: actively listening for connections ...
Server[4554]: OK: retrieved SQL database info ...
Server[4554]: OK: a client connected ...
Server[4554]: OK: client requested phone # for Fuller
Server[4554]: OK: first request for Fuller
Server[4554]: Phone number for Fuller: <206> 5555-9482
Server[4554]: OK: a client connected ...
Server[4554]: OK: client requested phone # for King
Server[4554]: OK: first request for King
Server[4554]: Phone number for King: <21> 5555-4498
Server[4554]: OK: a client connected ...
Server[4554]: OK: client requested phone # for Musil
Server[4554]: OK: first request for Musil
Server[4554]: Phone number for Musil: Sorry. Cannot be found.
```

图 5.2 Socket 示例程序, Server 端

下一步是实现“连接逻辑”(connection logic),用以处理传入的请求(request)。此处假设我们认定外界输入的是某人的姓氏,我们将抓取数据,稍做整理,并试着取得与此人相关的一个电话号码;接下来将响应(response)加以打包(package),递送给 client。让我们看看如何实现上述功能。

client 是个短命的家伙。它只负责递交请求(request),等待响应(response),然后处理响应。做完这一切,clients 就退出舞台。相反地,server 能够运行很长时间,几乎与主机(host machine)的运行时间相比拟。这意味着 server 必须不断轮询(poll),查看各端口是否申请连接。成员函数 AcceptSocket() 就是扮演轮询机制(poll mechanism);如果调用它,要不就发生阻塞(block),要不就有个 client 被连接上来——此时它会返回一个 Socket connection object。以下是部分代码:

```
public void handleConnection()
{
    while( true )
    {
        // blocks until a client connects

        Socket aSocket = tcp1.AcceptSocket();
        if( aSocket.Connected ) {
```


接下来我要创建一个 byte array 用以接受 client 数据，并将这个 byte array 连同其大小和一个下标（索引，用以指明在何处放置收到的数据）一起传给 Socket class 的 Receive() 成员函数。Receive() 返回已传送的 bytes 个数，我们将 byte array 转换为字符串，并去掉填充于此字符串内的无关字符：

```
Byte [] packetBuffer = new Byte[ maxPacket ];
int byteCnt =
    aSocket.Receive( packetBuffer, packetBuffer.Length, 0 );

string clientPacket =
    Text.Encoding.ASCII.GetString( packetBuffer );

// get rid of unused buffer space ...
char [] unusedBytes = { (char)clientPacket[ byteCnt ] };
clientPacket = clientPacket.TrimEnd( unusedBytes );
```

一旦所有请求（request）处理完毕，下一步是将响应结果打包，并返回给 client。我以字符串表示响应（response），再转换为 byte array，然后将此 byte array 传给 Socket class 的 Send() 成员函数：

```
Byte [] resultBuffer =
    Text.Encoding.ASCII.GetBytes( response.ToCharArray() );
aSocket.Send( resultBuffer, resultBuffer.Length, 0 );
```

到目前为止，我们见到的代码片段实现了一个 "client connection" 的完整处理过程。此后继续执行 while 循环，再次调用 TcpListener 的 AcceptSocket()，等待下一个 "client connection"。

5.9.2 客户端（Client-Side）的 TcpClient

TcpClient class object 提供了对特定主机(host)之特定 port (端口)的网路连接。本例中 TcpClient object 以主机名称和 port 号码初始化。构造函数会自动做好连接，然后调用 TcpClient 的 GetStream() 函数，获取一个 NetworkStream object。client 与 server 之间的所有数据交换都通过此 NetworkStream object 完成。例如：

```
public class SocketDemo_Client
{
    private TcpClient tcpcl;
    private static int port = 4554;
    private static string host = "localhost";
```

```
public bool sendRequest( string data )
{
    tcpc = new TcpClient( host, port);
    NetworkStream netstream = tcpc.GetStream();
```

接下来,我们要准备好待传输数据,和 server 端类似,我们创建一个 byte array,而后调用 Write(), 同时传给它 byte array、起始下标(索引)、欲传输的 bytes 数量:

```
Byte[] outPacket =
    Text.Encoding.ASCII.GetBytes( data.ToCharArray() );

netstream.Write( outPacket, 0, outPacket.Length );
netstream.Flush();
```

数据传输完毕后,我们不断查询 DataAvailable property, 等待 NetworkStream 内的数据变为可用。当数据变得可用时,我创建一个 byte array 以接收 server 返回的数据。我将此 array 连同其大小、起始位置一起传给 Read() 函数。最后,我们将 array 中接收到的数据转换为字符串,并在控制台(console)上显示出来(译注:原程序有重大逻辑错误,下为修正版):

```
while( ! netstream.DataAvailable )
    ; // 译注: empty loop

byte[] packet = new byte[ max_packet ];
int byteCnt = netstream.Read( packet, 0, max_packet );

string dataRcd = Text.Encoding.ASCII.GetString( packet );
Console.WriteLine( dataRcd );
```

5.10 System.Data

译注: 此处用到的 Access 样例数据库可从美国农业部网站下载获得:
<http://www.nal.usda.gov/fnic/foodcomp/Data/SR13/dnload/sr13dnld.html>.

位于 System.Data 命名空间内的 DataSet class 被当做数据的“内存存储区”(in-memory storage), 这些数据存于一个或数个 DataTable 成员中。这些数据往往是从数据库取回的信息(这正是本节主题), 或是存储于 XML 文档内的信息(这是下一节主题)。DataTable 可被视为一个群集(collection), 装有一些 DataRow objects, 每个 DataRow 又是由一个或多个 DataColumn 字段(fields)组成的集合。

这一节我将使用美国农业部维护并免费发布的 USDA Nutrient Database 当做样例数据库。我用的这一版（第 13 版，SR13）囊括了 6000 种以上的食品数据，如奶酪、土豆片、软性饮料等等。我将使用这个关系数据库（relational database）中的三张表（tables）。

5.10.1 数据库表格（Database Tables）

这个数据库中的食品描述表（FOOD_DES）内的食品项有 6210 行，每一行数据有 11 个栏位（columns，字段）：表格主键（primary key）是 NDB_NO 栏位，持有每个食品项的唯一标识号。我们感兴趣的第二栏位是 DESC，它简短描述了食品项。例如以下是 FOOD_DES 数据表的前 11 行（只列出 NDB_NO 项与 DESC 项）：

NDB_NO	DESC
01001 ::	Butter, with salt
01002 ::	Butter, whipped, with salt
01003 ::	Butter oil, anhydrous
01004 ::	Cheese, blue
01005 ::	Cheese, brick
01006 ::	Cheese, brie
01007 ::	Cheese, camembert
01008 ::	Cheese, caraway
01009 ::	Cheese, cheddar
01010 ::	Cheese, cheshire
01011 ::	Cheese, colby

数据库中另有一张表，记录每种食品的营养价值，那就是营养数据表（NUT_DATA）。我们对其中的 3 个栏位感兴趣：(1) NDB_NO，用以确认食品项；(2) NUTR_NO，营养成分（如蛋白质、脂肪、碳水化合物等等）的唯一标识号；(3) NUTR_VAL，营养价值。以下是 NUT_DATA 表（只取 3 个栏位）：

NDB_NO	NUTR_NO	NUTR_VAL
01001 ::	203 ::	0.85
01001 ::	204 ::	81.11
01001 ::	205 ::	0.06

```

01001 :: 207 :: 2.11
01001 :: 208 :: 717
01001 :: 255 :: 15.87
01001 :: 268 :: 3000
01001 :: 291 :: 0
01001 :: 301 :: 24
01001 :: 303 :: 0.16
01001 :: 304 :: 2

```

每张表单独看都不是完整的。食品描述表 (FOOD_DES) 记录了每项食品的名称，但没有包含其营养价值——它被记录在营养数据表 (NUT_DATA) 中。NDB_NO 栏位是连接两张表格的纽带。要想在两张表之间航行，我们需要定义一个数据关系 (data relationship)。在 .NET Framework 之下，我们为 DataSet.DataRelation 添加一个 DataRelation property，借此定义两张表格之间的关系（根据某共享栏位）。

上述两张表即便合在一起，仍旧不完善，我们还无法确知“每一项食品所代表的营养值”到底属于何种营养成分。这些相关信息存储在营养定义表 (NUTR_DEF) 中，我们对这张表内的两个栏位颇感兴趣：(1) NUTR_NO，这是这张表的主键 (primary key)；(2) NUTRDESC，描述该种营养素。以下样例取自 NUTR_DEF 表格：

```

NUTR_NO  NUTRDESC
203 ::   Protein (蛋白质)
204 ::   Total lipid (fat) (总脂肪含量)
205 ::   Carbohydrate, by difference (碳水化合物)
207 ::   Ash (大概是无机盐)
208 ::   Energy (能量)
221 ::   Alcohol (酒精)
255 ::   Water (水份)
262 ::   Caffeine (咖啡因)
263 ::   Theobromine (可可碱)
268 ::   Energy (能量)
269 ::   Sugars, total (糖总含量)

```

营养数据表 (NUT_DATA) 与营养定义表 (NUTR_DEF) 之间的联系，倚赖 NUTR_NO 栏位，这个“联系”也可以采用 DataRelation object 表示。

这份营养数据库由三个 `DataTable` objects、两个 `DataRelation` objects (统统包含于一个 `DataSet` object 内), 就我们所需要的值而言完整地于内存中表示出来。这些 classes 以及其他位于 `System.Data` 命名空间内的 classes 构成了整个 ADO.NET 体系结构 (architecture)。

5.10.2 开启数据库: 选择一个数据供应器 (Data Provider)

所有与数据库的互动都要通过数据供应器 (data provider) 来进行。数据供应器主要由以下四种服务 (services) 组成, 每种服务均以 `class` 代表:

1. `Connection`, 负责处理与某特定数据源 (data source) 的连接 (connection)。
2. `Command`, 对数据源 (data source) 执行一条命令, 命令包括 `select`, `update`, `insert`, `delete` 等数种。
3. `DataReader`, 提供只能单向前进且只读 (read-only) 的数据流 (data stream), 用以读取数据源 (data source)。
4. `DataAdapter`, 以选中的数据 (所构成的 in-memory cache) 充填 `DataSet`。
`DataAdapter` 也能用来更新数据源 (data source)。

目前的 .NET Framework 提供两个数据供应器 (data providers): (1) SQL Server .NET 数据供应器, 定义于命名空间 `System.Data.SqlClient` 中, 适用 Microsoft SQL Server 7.0 及其后续版本; (2) OLE DB .NET 数据供应器, 定义于命名空间 `System.Data.OleDb` 中, 适用其他所有数据库如 Microsoft Access。

本节中我将使用 Microsoft Access 数据库和 OLE DB 数据供应器。7.7 节我将介绍如何使用 SQL Server 数据供应器来支持 ASP.NET。

以下是使用 OLE DB 数据供应器的必要步骤, 首先连接 (connect) 数据库, 并选取 (select) 数据, 然后将取回的数据拿来填充一个 `DataSet` object:

1. 建立一个 `connection` 字符串。这个字符串由两部分组成, 第一部分定义数据库供应器——Microsoft Access 使用 `Microsoft.JET.OLEDB.4.0`, Oracle 使用 `MSDORA`, 7.0 版之前的 SQL Server 则是使用 `SQLOLEDB`。字符串的第二部分定义数据源文件 (source file) 及其路径, 例如:

```
string connect = "Provider=Microsoft.JET.OLEDB.4.0;" +
    @"data source=C:\nutrition\database\FOOD_DES.mdb";
```

2. 建立一个 *selection command* 字符串。这个字符串最少由两部分组成：SELECT 部分与 FROM 部分，前者决定从每行 (row) 数据中读取哪些栏位 (columns)，后者决定读取哪一张数据表。以下第一个例子选取 FOOD_DES 数据表的所有栏位，第二个例子只选取 NDB_NO 栏位与 DESC 栏位。我们还可通过 WHERE 子句 (以及 GROUP 子句、HAVING 子句、ORDER BY 子句等等) 来决定数据的选取条件。下面的第三个例子中，我们从满足“UNITS (单位) 是 'g' (克)”的那些数据行中选取 NUTR_NO 栏位和 NUTRDESC 栏位：

```
string command1 = "SELECT * FROM FOOD_DES";
string command2 = "SELECT NDB_NO, DESC FROM FOOD_DES";
string command3 =
    "SELECT NUTR_NO, NUTRDESC FROM NUTR_DEF WHERE UNITS = 'g'";
```

3. 创建 OleDbConnection object，并以前述的 *connection* 字符串加以初始化。用完 *connection* 之后要记得关闭 (利用 Close() 或 Dispose() 均可)：

```
using System.Data.OleDb;
OleDbConnection db_conn = new OleDbConnection( connect );

// OK: access the data source ...
db_conn.Close();
```

4. 创建一个 OleDbDataAdapter data adapter object。接着创建一个 OleDbCommand object 并以“*selection command* 字符串”和先前创建的 *connection* object 加以初始化。OleDbCommand 会取得数据。最后再将 *command* object 赋值给 data adapter 的 SelectCommand property：

```
OleDbDataAdapter adapter = new OleDbDataAdapter();

string command = "SELECT * FROM FOOD_DES";
adapter.SelectCommand = new OleDbCommand( command, db_conn );
```

5. 创建一份 DataSet object，并将它连同数据表名称 (该表格用来放置取得的数据) 传给 data adapter 的 Fill()。Fill() 执行 “select” 命令，并将数据置于指定的 DataTable 中。如果指定的 DataTable 不存在，就创建一份新表：

```
DataSet ds = new DataSet();
adapter.Fill( ds, "FOOD_DES" );
```



```
<?xml version="1.0" encoding="utf-8" ?>
<!-- Place keywords in the PI below -->
<?calories diabetes?>
<food_composition>
```

根元素 (root element) 之后跟着 4 项“食品元素”，每个元素由 9 个子元素组成。每个“食品元素”大致看起来如下：

```
<food>
  <category>dairy</category>
  <family>cheese</family>
  <name>mozzarella</name>
  <calories weight="-1" diabetes="-1">281</calories>
  <fat>21.60</fat>
  <cholesterol>78</cholesterol>
  <carbohydrates>2.28</carbohydrates>
  <protein>19.42</protein>
  <fiber>0</fiber>
</food>
```

以上文件被命名为 food.xml。如果我们愿意，可以直接将它读入 DataSet object：

```
DataSet ds = new DataSet();
ds.ReadXml( @"c:\C#Talks\food.xml" );
```

然后，既可以直接通过 DataSet object 操控之，也可以根据 DataSet 构建一个 XmlDocument，就像本节一开始所做的那样。

DataSet class 拥有一对成员函数：WriteXml() 和 WriteXmlSchema()。利用这些函数，我们可以为任何 DataSet object 生成对应的.xml 文件和.xsd schema 文件：

```
DataSet ds = new DataSet();
adapter.Fill( ds, "FOOD_DES" );
ds.WriteXml( @"C:\C#Talks\northwind.xml" );
ds.WriteXmlSchema( @"C:\C#Talks\northwind.xsd" );
```

5.11.1 在程序中使用 XML

现在，假设我们已经获得了用户录入的一顿饭的营养数据。我想把这些信息保存为一个 XML 文档。这可以分两个步骤办到。第一步是根据已获得的数据创建一份文档，这项工作交给 registerMeal() 去完成：

```
public static void  
registerMeal( string client, MealTime mt, ArrayList foods )  
( ... )
```

XML 文档 (document) 是由 XmlDocument object 来表现的。一份 XmlDocument 就是数据在 DOM 模型下的一种 *in-memory* 表示法。XmlDocument 定义于命名空间 System.Xml。首先我们创建一份空文档：

```
XmlDocument theDoc = new XmlDocument();
```

然后运用这个 document class 的一系列 Create 成员函数，手工创建其 *children*。例如以下创建 XML 声明：

```
XmlNode theChild;  
theChild = theDoc.CreateXmlDeclaration( "1.0", null, null );  
theDoc.AppendChild( theChild );
```

XmlNode 是“所有能够出现于 XML 文档内”之各种节点型别 (node types) 的 base class。CreateXmlDeclaration() 的三个字符串引数分别用来表示 *version*、*standalone*、*encoding* 三个属性 (attributes)。如果第二或第三个引数为 null，则对应的属性不会被打印出来。AppendChild() 可以将特定节点 (node) 添加到 *children* 链表末尾。生成的声明看起来是这个样子：

```
<?xml version="1.0"?>
```

以下是创建“注解节点 (comment node)”和“处理指示节点 (processing instruction node)”的办法：

```
theChild = theDoc.CreateComment( client + ":" +  
    DateTime.Now.ToShortDateString() );  
theDoc.AppendChild( theChild );  
  
theChild = theDoc.CreateProcessingInstruction(  
    "nutrition", "diabetes heart" );  
theDoc.AppendChild( theChild );
```

如果其中的 client 字符串是 *Alice Emma Weeks*，上述代码会生成以下 XML 节点：


```
<!--Alice Emma Weeks:8/17/2001-->
<?nutrition diabetes heart?>
```

最后, 让我们创建根元素 (root element) 和它的各个 *children* 如下:

```
// 译注: 原书有些笔误, 已修正。
// create the root element ...
theChild = theDoc.CreateElement("k", "nutrition", "urn:nutrition");
theDoc.AppendChild(theChild);

// OK: let's populate this puppy
theChild = theDoc.CreateElement("client");
theChild.InnerText = client;
theDoc.DocumentElement.AppendChild(theChild);

theChild = theDoc.CreateElement("meal");
theChild.InnerText = mt.ToString();
theDoc.DocumentElement.AppendChild(theChild);

for( int ix = 0; ix < foods.Count; ++ix )
{
    theChild = theDoc.CreateElement("item" + ix.ToString());
    theChild.InnerText = (string) foods[ ix ];
    theDoc.DocumentElement.AppendChild(theChild);
}
```

以下是调用这个函数的方法:

```
static public void Main()
{
    string client = "Alice Emma Weeks";
    ArrayList foods = new ArrayList();

    foods.Add("banana");
    foods.Add("bread, whole wheat");
    foods.Add("Ben & Jerry, Quart, RockyRoad");

    ClientMeal.registerMeal( client,
        ClientMeal.MealTime.SnackNight, foods );
}
```

下面是运行结果:

```
<?xml version="1.0"?>
<!--Alice Emma Weeks:8/17/2001-->
<?nutrition diabetes heart?>
```

```
<k:nutrition xmlns:k="urn:nutrition">
  <client>Alice Emma Weeks</client>
  <meal>SnackNight</meal>
  <item0>banana</item0>
  <item1>bread, whole wheat</item1>
  <item2>Ben & Jerry, Quart, RockyRoad</item2>
</k:nutrition>
```

上页程序用到的 MealTime 是定义于 ClientMeal class 的一个 enum, 如下:

```
class ClientMeal
{
    public enum MealTime
    {
        NotSure,
        Breakfast, Lunch, Dinner, SnackMorning,
        SnackAfternoon, SnackEvening, SnackNight
    };
    // ...
}
```

接下来, 第二步骤是生成 XML 文档 (document)。我们运用 XmlTextWriter 来生成 XML 文档, 它也被定义于命名空间 System.Xml 中。这里又需分两个步骤来进行。首先创建一个 XmlTextWriter, 并将它绑定到“XML 文档的输出位置”, 然后调整格式。接下来, 调用 XmlDocument 的 Save() 成员函数, 传给它 XmlTextWriter object, 例如:

```
XmlTextWriter theWriter = new XmlTextWriter(Console.Out);
theWriter.Formatting = Formatting.Indented;
theDoc.Save( theWriter );
```

XmlSerializer class (也定义于命名空间 System.Xml 中) 能够让我们对 objects 做 serialize / deserialize (序列化/反序列化) 动作, 读写的目标是 XML 文档。XML serialization 的过程是将 objects 的 public properties (属性) 和 public fields (数据成员) 转换为“序列格式” (serial format, 此处是指 XML), 以便存储和传送。XML deserialization 动作则是根据 XML 的输出, 将 objects 重建起来, 并恢复其原本状态。以下是我们想要序列化的一个 Point class:


```
[XmlRoot( "rootElement" )]
public struct Point
{
    public float m_x, m_y, m_z;
    public string m_now;

    public Point( float x, float y, float z )
    {
        m_x = x; m_y = y; m_z = z;
        m_now = DateTime.Now.ToString();
    }
}
```

Point class 定义式最上方以中括弧围起来的奇怪东西，我们称为 *attribute* (特征属性)：

```
[XmlRoot( "rootElement" )]
```

其作用是将 metadata (元数据) 附加于 object 身上，然后可在运行期查询或取回之。XmlRoot *attribute* 指明文档中的根元素 (root element) 名称。

创建 XmlSerializer object 时，应该将我们打算加以序列化 (serialize) 的 class 的 Type node 传给它构造函数。Type class 可说是 .NET Framework 带领我们通向“运行期反射” (runtime reflection) 的大门，例如：

```
XmlSerializer xs = new XmlSerializer( typeof( Point ) );
```

序列化 (serialization) 的施行需要两样东西：动作的对象 (某个 object) 和动作目的地 (某个数据流, data stream)。下面就是对 Point object 的序列化动作：

```
Point pt = new Point( 0.5f, 1.5f, 4.5f );
FileStream fs =
    new FileStream( @"C:\temp\xs.xml", FileMode.Create );
```

对 Point class 而言，要做的就是这些。余下的工作由 XmlSerializer object 帮我们完成——它的 Serialize() 成员函数接受诸如上述的 Point class object 和一个文件流 (file stream)，例如：

```
xs.Serialize( fs, pt );
fs.Close();

xs.Serialize( Console.Out, pt );
fs.Close();
```

编译并执行以上代码，会生成下面的输出结果：

```
<?xml version="1.0" encoding="IBM437"?>
<rootElement xmlns:xsi="http://www.w3.org/2001/XMLSchemainstance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <m_x>0.5</m_x>
  <m_y>1.5</m_y>
  <m_z>4.5</m_z>
  <m_now>8/17/2001 3:22:25 AM</m_now>
</rootElement>
```

5.11.2 XmlTextReader

处理 XML 文档的办法之一是“一次读一个节点”：抓取节点、处理、控制必要状态 (state context)，然后处理下一个节点。NET 的 `XmlTextReader` class 可以办到这一点，此外它还可以跳过我们不感兴趣的节点。这个 class 定义于 `System.Xml` 命名空间中。以下是依序处理各个节点的惯常写法：

```
public static void Main()
{
    string fname = @"C:\c#primer\Food.xml";
    FileStream fs = new FileStream( fname, FileMode.Open );

    XmlTextReader xrd = new XmlTextReader( fs );
    while ( xrd.Read() == true )
    {
        // OK: process each node ...
    }
}
```

`FileStream` object 用来读取文本，然后 `XmlTextReader` 理解其内容并按其含义进行组织。每次调用 `Read()` 就从 `FileStream` 读取下一个节点。如果成功读取，就返回 `true`；如果遇到 stream 末尾，就返回 `false`。我们通过所谓的 *reader object* 来访问节点内容。节点所拥有的属性 (properties) 包括 `Name`、`HasValue`、`HasAttributes`、`NodeType`、`Prefix`、`AttributeCount`、`Value`、`NamespaceURI`：


```

while ( xrd.Read() == true )
{
    Console.WriteLine("node type is {0}", xrd.NodeType);

    if ( xrd.Name != String.Empty )
        Console.WriteLine( "\tname is {0}", xrd.Name);
    else
        Console.WriteLine("\tThis node has no name");

    if ( xrd.HasValue )
        Console.WriteLine("\tvalue is {0}", xrd.Value );

    if ( xrd.NodeType == XmlNodeType.Element )
        if ( xrd.HasAttributes )
        {
            Console.Write( "\thas {0} attributes: ",
                           xrd.AttributeCount );

            while ( xrd.MoveToNextAttribute() )
                Console.Write( "{0} = {1} ",
                               xrd.Name, xrd.Value );

            Console.WriteLine();
        }
        else Console.WriteLine( "\thas no attributes" );
}

```

XML 元素的属性 (attributes) 并不被视为此元素的 *children*。我们必须明确访问它们。MoveToAttribute() 可以将我们的关注点移动到特定的属性 (attributes) 去；共有以下三个重载实体：

```

// move to the index attribute
void MoveToAttribute(int index)

// move to the name attribute
bool MoveToAttribute(string name)

// move to the attribute with these characteristics
bool MoveToAttribute( string localName, string namespaceURI )

```

如果想要遍历某个元素的所有属性 (attributes)，也可以采用 MoveToNextAttribute() 函数。如果当前节点是个元素节点 (element node)，MoveToNextAttribute() 会将关注点移动到第一个属性 (attributes) 处。此后每次调用 MoveToNextAttribute() 就依次移动到下一个属性，直到遍历所有属性。

`MoveToElement()` 会将关注点退回到“此属性 (attributes) 所隶属的”元素节点处。如此一来, 尽管是以单向、无缓冲区方式读取 XML 文档, 我们仍有一定的灵活性。例如下面这个 XML 文件片段:

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- Place keywords in the PI below -->
<? calories diabetes ?>
<food_composition>
// ...
<carbohydrates diabetes="-1">15</carbohydrates>
```

以下是输出结果。请注意 XML 声明中的 `version` 和 `encoding` 两个属性 (attributes) 是作为此声明的 `value properties` 出现, `carbohydrates` 元素的属性 (attributes) 则是存储于独立的属性节点 (attribute node) 中, `carbohydrates` 的 `value property` 是空的:

```
node type is XmlDeclaration
  name is xml
  value is version="1.0" encoding="utf-8"
node type is Comment
  This node has no name
  value is Place keywords in the ...
node type is ProcessingInstruction
  name is calories
  value is diabetes
node type is Element
  name is food_composition
  has no attributes
node type is Element
  name is carbohydrates
    has 1 attributes: diabetes = -1
node type is Text
  This node has no name
  value is 15
node type is EndElement
  name is carbohydrates
```

缺省情况下, `XmlTextReader` 将返回一个 `whitespace node`, 表示每项结尾处的换行 (`new-line`) 字符。举个例子, 以下是读取上述 XML 文件后的输出结果, 其中对于空白符 (`white space`) 采取缺省处理办法:


```
node type is XmlDeclaration
node type is Whitespace
node type is Comment
node type is Whitespace
```

如果想要移除不重要的 `Whitespace` 节点，我们可以设置 `reader` 的 `WhitespaceHandling` property。"White space" 分两种，一是 `Whitespace`，例如 `new-line` 字符，二是 `SignificantWhitespace`，代表元素或属性（`attributes`）的实际内容。缺省情况下这两种都会被返回。如果你要求两者皆不返回，可将 `WhitespaceHandling` property 设为 `None`。如果只想要返回有意义的（`significant`）`whitespace`，则请将 `WhitespaceHandling` property 设为 `Significant`。例如：

```
xrd.WhitespaceHandling = WhitespaceHandling.Significant;
```

这么一来，得到的输出结果是下面这个简单得多的序列：

```
node type is XmlDeclaration
node type is Comment
```

如果要找出当前节点的型态，可查看 `NodeType` property，例如：

```
if ( x.NodeType == XmlNodeType.Whitespace )
    continue;
```

其中 `XmlNodeType` enum 是一种标记（`flag`），用来确认当前节点的种类，就像以下所展示的那样：

- `enum XmlNodeType(Element, EndElement, ...)`

`Element` 节点可拥有以下各类子节点：`Text`、`EntityReference`、`Element`、`CData`、`ProcessingInstruction`、`Comment`。`Element` 节点自身可以是 `Document`、`DocumentFragment`、`Element`、`EntityReference` 等节点的子节点。如果返回 `EndElement`，就表示到达了某个元素的结尾处。

- `enum XmlNodeType(Attribute, ...)`

`Attribute` 节点不能作为其他类型的节点的子节点。请特别注意，`Attribute` 并不是 `Element` 的子节点；它可以拥有一个 `Text` 节点或 `EntityReference` 节点作为其子节点。

- `enum XmlNodeType{ XmlDeclaration, ... }`

`XmlDeclaration` 应该是 XML 文档的第一个节点。它没有子节点，它是根节点的子节点。它可以拥有一些属性 (attributes) 用来表示 *version* 和 *encoding* 信息。

- `enum XmlNodeType{ ProcessingInstruction, ... }`

`ProcessingInstruction` 不能拥有任何子节点。它可作为 `DocumentFragment`、`EntityReference`、`Element`、`Document` 等等的子节点。“处理指示” (processing instruction) 的第一个单词将被视为“名称”，余下的文本被视为“值”，例如：

```
<?SQLQuery SELECT * FROM Food_Des?>
```

其中的 `SQLQuery` 表示“名称”，`select` 语句表示“值”。

- `enum XmlNodeType{ Comment, ... }`

`Comment` 节点不能拥有任何子节点。它可作为 `DocumentFragment`、`Element`、`EntityReference`、`Document` 等的子节点。

- `enum XmlNodeType{ Text, ... }`

`Text` 节点不能拥有任何子节点。它可作为 `DocumentFragment`、`Element`、`EntityReference`、`Document` 等的子节点。

每次 `Read()` 之后，我们通常使用 `switch` 语句来分辨 `XmlTextReader` 所持有的节点种类，这一 `switch` 语句的各个 `case` 标签应该对应一个个 `XmlNodeType` 枚举元。为示范起见，让我们累计“处理指示” (processing instruction) 中指定的元素值总和，并找出同样在“处理指示”中指定的“健康种类” (health category) 相关属性 (attributes)。输出结果是一张列有匹配元素的表，同时列出所关心的健康问题，然后列出这些元素值的总和。以下处理“处理指示” (processing instruction) 以取得我们感兴趣的元素名称和健康属性 (译注：这里的“处理指示”是指 `<calories diabetes>`)：

```
string filter = null;
string health = null;
XmlTextReader xrd = new XmlTextReader( fs );
```



```

while ( xrd.Read() == true ){
    switch( xrd.NodeType )
    {
        case XmlNodeType.ProcessingInstruction:
            filter = xrd.Name;
            health = xrd.Value;
            break;

        case XmlNodeType.Element:
            // ...
            break;
    }
}

```

面对一个元素，首先应该判断它是否为“处理指示”中指定的元素：

```

case XmlNodeType.Element:
{
    if ( xrd.Name.Equals( filter ) )
    {
        // ... ok, found it
    }
}

```

如果符合要求，就遍历该元素的所有属性（attributes；如果有的话），以找寻符合“处理指示”的值（value）所指出的（也就是我们希望关心的）健康问题的属性：

```

if ( xrd.HasAttributes )
{
    while ( xrd.MoveToNextAttribute() )
        if ( xrd.Name.Equals( health ) )
        {
            suffix = "\t*** " + health + " : " + xrd.Value;
            break;
        }
}

```

此外还需取回此元素的相关值。哎呀，*reader* 目前位于此元素的属性（attributes）中，因此先得将 *reader* 回置到当前元素身上：

```

// restore reader to element, from attribute
xrd.MoveToElement();

```

接下来，取回元素值并转换为 `int`，并将数值添加到总和 `total` 中：

```
// this is the value to compute
text = xrd.ReadString();

// convert and total
total += Convert.ToInt32( text );
```

其中用到的 `ReadString()` 会将所有的 *text*、*significant white space*、*white space*、`CDate` 节点型别 (node types) 串接到一起，并以字符串返回所串接的数据。这么做很奏效，省得我们还得显式遍历此元素的 *text* 子节点。

对每一个吻合的元素，我们要创建一个字符串来持有食物名称、筛选种类、相关的健康属性 (attributes)。例如我们也许想让输出看起来像这个样子：

```
whole milk      calories : 150
mozzarella     calories : 281 *** diabetes : -1
muenster        calories : 368 *** diabetes : -1
provolone       calories : 351 *** diabetes : -1

*** Total calories : 1150
```

(译注：whole milk 是全脂牛奶，mozzarella 是意大利白干酪，muenster 是一种产于法国 Muenster (门斯特) 的淡黄色的半软干酪，diabetes 是糖尿病)

然而我们遇到了一个问题，持有食物名称的那个元素已经读过去了，并没有保留内容。而且在这种“勇往直前”的解析模型 (parse model) 下我们无法回头读取其值。所以只好添加一个对食物名称的元素检测，并把该值保存下来，像这样：

```
// need to tuck away the food's name
if ( xrd.Name.Equals( "name" ) )
{
    food = xrd.ReadString();
    break;
}
```

现在终于可以生成字符串，并将它添加到代表“吻合元素”的 `ArrayList` object 中了：

```
// tuck it away in a container
elements.Add( food + "\t" + xrd.Name + " : " + text + suffix );
```


这种“分别对不同节点种类进行处理”的编码风格，从本质上说，各节点类型并无良好区分。下一节将探讨 Document Object Model (DOM)，在此模型下整个 XML 文档以 class node types 阶层架构 (hierarchy) 的形式存储于内存中，可供随机访问。

5.11.3 Document Object Model (DOM, 文档对象模型)

本节探讨“对 W3C Document Object Model (DOM) 提供支持”的一些 classes。在 DOM 模型下，整个 XML 文档以树状结构存储于内存中，我们可以浏览并编辑树的分枝 (branches)。XML 文档以 XmlDocument class 表示，其中各个节点型别则以 XmlNode 阶层体系表示，例如 XmlElement 和 XmlAttribute 等。

我们仍旧沿用 FileStream 和 XmlTextReader 来读取并构造 XML。区别在于我们对 XmlTextReader object 的使用方式。我们不直接操作 XmlTextReader，而是将它传给 XmlDocument class 的 Load()。Load() 构造出文档在内存中的树状结构 (XmlTextReader 对它提供了节点信息)。一旦 XML 文档加载完毕，我们就可以浏览之、查询之、编辑之：

```
using System;
using System.Xml;
using System.IO;
class testXML
{
    public static void Main()
    {
        string xfile = @"C:\c#primer\food.xml";
        FileStream fs =
            new FileStream( xfile, FileMode.Open );

        // so far, same as before ...
        XmlTextReader xr = new XmlTextReader( fs );

        // OK: here is the difference
        XmlDocument xd = new XmlDocument();

        // build the in-memory representation
        xd.Load( xr );
```

```
// this is our program logic  
processDom( xd );  
}  
}
```

XmlDocument class 提供了一套属性 (properties)，用作查询内存中的 XML 文档，包括以下这些：

- Attributes, 返回此节点的所有属性所组成的一个群集 (collection)。
- ParentNode, 返回此节点的父节点。
- HasChildNodes, 如果节点有子节点，就返回 true (ChildNodes 则是返回此节点的所有子节点)。
- FirstChild, 返回此节点的第一个子节点。
- LastChild, 返回此节点的最末一个子节点。
- NextSibling, 返回紧跟在此节点之后的节点。
- PreviousSibling, 返回紧挨在此节点之前的节点。
- DocumentElement, 返回文档的根节点 XmlElement。
- DocumentType, 返回内含 DOCTYPE 声明的节点。
- OwnerDocument, 返回此节点所属的 XmlDocument。
- InnerText, 读取 (get) 或设置 (set) 此节点 (及其子节点) 的值。
- InnerXml, 读取 (get) 或设置 (set) “代表此节点之子节点”的标记 (markup)。
- OuterXml, 返回表示此节点及其所有子节点的标记 (markup)。
- Name, 返回此节点的“全名” (qualified name)。
- LocalName, 返回此节点的局部名称 (local name)。
- NamespaceURI, 返回此节点之命名空间的 Uniform Resource Identifier (URI)。
- NodeType, 返回当前节点的型别。
- Value, 读取 (get) 或设置 (set) 此节点的值。

举例来说, 前面那个例子的最后一个语句调用的 `processDom()` 函数如下。文档内部各节点均以 `class object` 表示, 这些 `classes` 都派生自 `XmlNode abstract base class`, 事实上 `XmlDocument class` 自身也继承自 `XmlNode` (本节末尾我会大略说说这些 `classes`) :

```
static public bool processDom( XmlDocument xd )
{
    Console.WriteLine( "XmlDocument {0} :: {1} ",
                        xd.Name, xd.Value );

    XmlAttributeCollection xac = xd.Attributes;
    if ( xac.Count != 0 ) // ...
    // 译注: 上一句似应改为 if ( xac != null && xac.Count != 0 )

    Console.WriteLine(
        "Retrieving the {0} XmlDocument Children\n",
        xd.ChildNodes.Count );

    XmlNodeList children = xd.ChildNodes;
    foreach ( XmlNode node in children )
    {
        Console.WriteLine( "Child node: {0} of type {1}",
                            node.Name, node.NodeType );
        Console.WriteLine(
            "Child has children? {0} :: Node's parent: {1}\n",
            node.HasChildNodes, node.ParentNode.Name );
    }

    // 译注: 应该添加以下这行, 才能呼应下页结果。
    Console.WriteLine( "OK: grabbing root element using
                        DocumentElement" );
    XmlElement root_elem = xd.DocumentElement;
    Console.WriteLine( "OK: {0} is the root element: ",
                        root_elem.Name );

    if ( root_elem.HasChildNodes )
        Console.WriteLine(
            "OK: {1}'s first child: {0}\n",
            root_elem.FirstChild.Name, root_elem.Name );

    XmlNodeList foodElem = root_elem.GetElementsByTagName( "food" );
    Console.WriteLine( "{0} foodElem retrieved", foodElem.Count );

    return true;
}
```

执行这个程序后，它将根据 food.xml 文件，生成如下输出结果：

```
XmlDocument #document ::  
    Document has no attributes.  
  
Retrieving the 4 XmlDocument Children  
  
Child node: xml of type XmlDeclaration  
    Child has children? False :: Node's parent: #document  
  
Child node: #comment of type Comment  
    Child has children? False :: Node's parent: #document  
  
Child node: calories of type ProcessingInstruction  
    Child has children? False :: Node's parent: #document  
  
Child node: food_composition of type Element  
    Child has children? True :: Node's parent: #document  
  
OK: grabbing root element using DocumentElement  
OK: food_composition is the root element:  
OK: food_composition's first child: food
```

每个节点型别 (node type) 均以“派生自抽象基类 XmlNode”的 class 表示。XmlElement 表示一个元素，XmlAttribute 表示一个属性，XmlComment 表示一条注解……如此等等。ChildNodes property 返回的是节点的子节点，以 XmlNodeList 表示。

GetElementsByTagName() 成员函数可以取回“符合标签名称”的所有子元素。例如以下式子将取回文档中的所有 "food" 元素：

```
XmlNodeList foodElem = root_elem.GetElementsByTagName( "food" );
```

上式返回一个 XmlNodeList object，其内元素将按照原始文档的出现顺序排列。

XmlDocument 的 LoadXml() 成员函数可以根据一个字符串，将一份 XML 文档加载到内存中，例如：


```

XmlDocument doc = new XmlDocument();

// OK: let's explicitly load the elements ...

doc.LoadXml(
    "<food id=\"123\">" +
    "    <category>dairy</category>" +
    "    <family>cheese</family>" +
    "    <name>mozzarella</name>" +
    "    <calories>281</calories>" +
    "    <fat>21.60</fat>" +
    "    <isEmpty/>" +
    "    <cholesterol>78</cholesterol>" +
    "    <carbohydrates>2.28</carbohydrates>" +
    "    <protein>19.42</protein>" +
    "    <fiber>0</fiber>" +
    "</food>"
);

```

尽管 `XmlTextReader` 和 DOM 的 `XmlDocument` 对单个节点的表达法不尽相同，然而在这两种表示法之间进行转换却并不困难，只要将 `XmlTextReader` object 当做 `XmlDocument` class 的 `ReadNode()` 成员函数的引数即可。为了能够返回任何种类的节点元素，所以 `ReadNode()` 返回一个 `XmlNode`（这是抽象基类）：

```

XmlTextReader xrd;
// ...

// position reader on a node or attribute
xrd.Read();

XmlDocument xdoc = new XmlDocument();

// OK: convert reader data into equivalent node
// returned through an abstract base-class object

XmlNode xn = xdoc.ReadNode( xrd );

```

只有在 `XmlNode` 的接口不能满足节点的处理需求时，才需要将节点的类型别向下转型（downcast），例如转为 `XmlElement`。

最后,我们还可以利用 `XmlNodeReader` class object 在“整个 `XmlDocument` 树”或者“以某个 `XmlNode` 为根的子树”之间来回穿梭。`XmlNodeReader` 和 `XmlTextReader` 皆派生自抽象的 `XmlReader`, 二者都采取单向前进、勇往直前式的读取策略。

5.11.4 System.Xml.Xsl

XSLT 代表的是“Extensible Stylesheet Language (XSL) Transformations”, 一个转换 (transformation) 动作由三个部件 (parts) 构成:

1. 一份 XML 文档, 亦即“将被转换的源文档”。
2. 一张 XSLT “style sheet” (样式表)。
3. 转换引擎 (一个 `XslTransform` class object)

`XslTransform` class 是个 XSLT 处理器, 实现了 XSL Transformations (XSLT) Version 1.0 推荐标准。`XslTransform` 的两个成员函数: `Load()` 用来加载即将被使用的 style sheet, `Transform()` 用来施行转换动作。二者都被重载, 支持多种参数组合。例如:

```
using System;
using System.IO;
using System.Xml;
using System.Xml.Xsl;
using System.Xml.XPath;

public static void
Transform( string xmldoc, string xsltdoc )
{
    // get a processor object
    XslTransform xslt = new XslTransform();

    // OK: load the .xslt style sheet
    xslt.Load( xsltdoc );

    // OK: load the source .xml document
    XPathDocument xdoc = new XPathDocument( xmldoc );

    // OK: we need an output target
    TextWriter writer = Console.Out;
```



```
// OK: let's transform this puppy ...
xslt.Transform( xdoc, null, writer );
}
```

我们的转换目的是将 food.xml 中的 name 元素和 calories 元素析取出来, 新的 meal.xml 文档看起来是这个样子:

```
<meal>
  <item>
    <name> xxx </name>
    <calories> xxx </calories>
  </item>
```

XSLT 文件由两部分组成 (至少概念上如此)。其中的“声明部分”在必不可少的 xsl:stylesheet 数据项中提供版本验证信息, 并在 xsl:output 数据项中选择性地修改转换结果的输出格式:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" indent="yes" encoding="UTF-8" />
  // ... template definitions go here
</xsl:stylesheet>
```

至于 XSLT 的另一部分: “模板部分”(template portion), 以各个 xsl:template 项标出, 选取每个 food 元素的名字和 calories, 并将选中结果输出至“成果树”中:

```
<xsl:template match="/">
  <meal>
    <xsl:for-each select="/food_composition/food">
      <item>
        <name>
          <xsl:value-of select="name"/>
        </name>
        <calories>
          <xsl:value-of select="calories"/>
        </calories>
      </item>
    </xsl:for-each>
  </meal>
</xsl:template>
```

调用 `Transform()` 之后, 将生成以下输出结果 (译注: 为了得到这个结果, 需要将配套代码的 `food.xml` 第四行 `<food_composition xmlns="http://tempuri.org/food.xsd">`) 改为 `<food_composition>`):

```
<?xml version="1.0" encoding="IBM437"?>
<meal>
  <item>
    <name>whole milk</name>
    <calories>150</calories>
  </item>
  <item>
    <name>mozzarella</name>
    <calories>281</calories>
  </item>
  <item>
    <name>muenster</name>
    <calories>368</calories>
  </item>
  <item>
    <name>provolone</name>
    <calories>351</calories>
  </item>
</meal>
```

5.11.5 System.Xml.XPath

XPath 表达式可让我们不按次序地 (randomly) 到达 XML 文档树的任意分支。一个 XPath 表达式由两部分组成:

- *location path*, 指明文档之中我们想要到达的目的地。
- *context node*, 指明我们在文档之中目前停驻的节点。

在文档树之间移动, 需要借助 `XPathNavigator` object, 也就是运用它的一组 `Select()` 成员函数。`XPathNavigator` 可以浏览 `XmlDocument`、`XmlDataDocument`、`XPathDocument`。其中 `XPathDocument` 为节点之间的随机往返移动做了特殊优化处理。例如以下是创建“绑定至 `XPathDocument`”的 `XPathNavigator` object 的办法:

```
XPathDocument xpdoc =
    new XPathDocument( @"C:\C#Talks\Food.xml" );

XPathNavigator navi = xpdoc.CreateNavigator();
```


接下来调用 `Select()`，传给它我们希望遍历的元素。例如以下选取行为选中了 `food.xml` 样例文件的四个 `food` 元素：

```
XPathNodeIterator iter =  
    navi.Select( "/food_composition/food" );
```

`Select()` 返回一个 `XPathNodeIterator`，这个 `class` 实现出 `IEnumerator`，可以通过其 `Current` `property` 访问单个节点，并调用 `MoveNext()` 移至下一个节点：

```
while ( iter.MoveNext() )  
{  
    Console.WriteLine( iter.Current.Name );  
    if ( iter.Current.HasChildren )  
    {  
        iter.Current.MoveToFirstChild();  
        Console.WriteLine( iter.Current.Name );  
  
        while ( iter.Current.MoveToNext() )  
            Console.WriteLine( iter.Current.Name );  
    }  
}
```

`Current` 代表一个定位于当前节点的 `XPathNavigator` `object`。 `HasChildren` `property` 和 `MoveToFirstChild()`、`MoveToNext()` 都是 `XPathNavigator` 的成员。外层的 `while` 循环遍历选中的四个 `food` 元素。内层的 `while` 循环遍历每个 `food` 元素的几个子节点。

另一个引人注目的选取方法是 `SelectChildren()`，它接受“`XPathNodeType` 枚举元”为其引数：

```
enum XPathNodeType  
{  
    All,  
    Attribute, Comment, Element, Namespace,  
    ProcessingInstruction, Root, SignificantWhitespace,  
    Text, Whitespace  
}
```

下面这个例子中,我们传入 All,因此取回了文档中的所有子节点.对 food.xml 文件而言,我们取回了三个子节点:一条注解、一条“处理指示”和一个根元素 food_composition (那是 food 元素的父节点):

```
iter = navi.SelectChildren( XPathNodeType.All );
int icnt = 0;
while ( iter.MoveNext() )
    Console.WriteLine( (++icnt).ToString() + ". " +
        iter.Current.Name + " ==> " +
        iter.Current.ToString() );
```

这段代码生成以下三行输出.请注意, XPathNavigator class 的 ToString() 将当前节点及其所有子节点的值全部串接到一起:

```
1. ==> Place keywords in the PI below
2. calories ==> diabetes
3.
food_composition==>dairymilkwholemilk150801580dairycheesemozzar
ella28121.60782.2819.420dairycheesemuenster36830.04961.1223.410da
irycheeseprovolone35126.62692.1425.580
```

余下两个主要的 XML 功能是位于 System.Xml.Schema 命名空间的 *schema* 支持功能,以及位于 System.Xml.Serialization 命名空间的 *Simple Object Access Protocol (SOAP)* 支持功能.限于篇幅,本书并未涉及这两个题目.

6.1 我们的第一个 Windows Forms 程序

我们的第一个 Windows Forms 程序将向用户显示问候(就像第 1 章所做),并将以文本通过图形用户界面(GUI)来得到.版本 1 是程序的初始版本.

在这个程序中,窗口(Window)以 Form class object 形式,标题栏(title bar)设置为“Hello, Windows Forms!”,当用户把鼠标放在文本框(text box)时,我们让其中的文本更改为“Hello, Windows Forms!”.这是通过文本框的文本改变事件(text change event)来做的.一旦文本框的内容发生改变,就触发一个事件(event).在这个窗体(form)中,我放置了以下控件(control):

如果想取回“目前运行的程序”对应之 Assembly object, 我们就调用 Assembly class 的 static 成员函数 `GetExecutingAssembly()`:

```
Assembly myAssembly = Assembly.GetExecutingAssembly();
```

如果想取回“定义了某特定型别”的 Assembly object, 我们调用 Assembly class 的 static 成员函数 `GetAssembly()`, 传给它你想找的那个特定型别所对应的 Type object.

欲获得 Type object, 有两个办法. 如果我们有该型别的 object, 只需调用它 (从 Object class 继承而来) 的 `GetType()` 成员函数即可. 另一种办法是, 我们可以调用 Type class 的 static 成员函数 `GetType()`, 传给它型别名称 (一个字符串):

```
Object o = new Fibonacci();
Assembly fibAssem = Assembly.GetAssembly( o.GetType() );

Type calType = Type.GetType("System.Globalization.JulianCalendar");
// 译注: 英文版上一行缺少 "System.Globalization."
Assembly calAssem = Assembly.GetAssembly( calType );
```

通过 Assembly class, 几乎可以获知组件的任何信息. 如果要取得 Assembly object 入口 (entry point: 如果有的话), 可询问 Assembly class 的 `EntryPoint` property:

```
MethodInfo ep = fibAssem.EntryPoint;
```

其中的 MethodInfo class 定义于 System.Reflection 命名空间, 内含它所指向 (代表) 的成员函数的详细信息, 诸如名称、访问级别、返回型别、参数列等等. 8.2 节将更深入地研究它.

如果要取回装配件中定义的所有型别, 应该调用 `GetTypes()` 成员函数:

```
Type [] types = fibAssem.GetTypes();
```

此外亦可通过 `GetType()` 成员函数, 按名称取回特定型别:

```
Type fibType = fibAssem.GetType( "Fibonacci" );
```

如果没有找到吾人指定的型别, `GetType()` 返回 `null`. 如果想令它在未能找到指定的型别时抛出一个异常, 可以将字面值 `true` 作为第二引数传入:

```
Type fibType = fibAssem.GetType( "Fibonacci", true );
```

举个例子, 以下的输出结果:

译注: 从以下的 Ver=1.0.2204.21 可看出作者用的 .NET Framework 是 Beta 版, 而本章涉及底层机制, 可能与最新的 .NET Framework 有不符之处. 我在翻译时尽力找出不相符的地方.

```
The current AppDomain has 2 Assemblies
```

```
The Assembly FullName Property:
```

```
mscorlib, Ver=1.0.2204.21, Loc="", SN=03689116d3a4ae33
```

```
assembly name: mscorlib
```

```
manifest loc: C:/WINNT/Microsoft.NET/Framework/v1.0.2204/mscorlib.dll
```

```
entry point: Not Defined
```

```
number of types: 1205
```

```
The Assembly FullName Property:
```

```
Assemblies, Ver=1.0.471.38017, Loc=""
```

```
assembly name: Assemblies
```

```
manifest loc: C:/C#Programs/Assemblies/bin/Debug/Assemblies.exe
```

```
entry point: Void Main()
```

```
number of types: 2
```

```
The types contained within the assembly are:
```

```
type name: AssemblyEntryPoint
```

```
type name: AssemblyExplore
```

便是由以下函数生成的:

```
public static void getAssemblies()
{
    AppDomain theApp = AppDomain.CurrentDomain;
    Assembly [] currAssemblies = theApp.GetAssemblies();

    Console.WriteLine( "The current AppDomain has {0} Assemblies",
                       currAssemblies.Length );

    foreach ( Assembly a in currAssemblies )
    {
        message( "\nThe Assembly FullName Property: \n\t",
                 a.FullName );
    }
}
```



```

int    pos  = a.FullName.IndexOf( ',' );
string name = a.FullName.Substring(0, pos);
Type [] t   = a.GetTypes();

string    manifestLocation = a.Location;
MethodInfo theEntry        = a.EntryPoint;

message( "assembly name: ", name );
message( "manifest loc:   ", manifestLocation );
message( "entry point:    ",
        theEntry != null ? theEntry.ToString() : "Not Defined" );
message( "number of types: ", t.Length );

if ( t.Length < 10 )
{
    Console.WriteLine(
        "The types contained within the assembly are:");
    foreach ( Type tt in t )
        message( "\ttype name: ", tt.Name );
}
}
}

```

你也可以使用 `Assembly` class 的 static `Load()` 或 `LoadFrom()` 自己动手加载某个装配件。如果想要为“定义于 `Assembly` object 中的某个型别”创建一份实体，可以使用 `CreateInstance()`。下一节讨论反射 (reflection) 时我将展示一个样例。

许多管理方面的议题我并没有提到，例如应该将装配件包装于单一模块 (module) 或包装于多个模块内 (亦即装配件由单一模块或多个模块组成)？装配件应该是 `private` 或 `shared`？此外还有版本管理、安全设施等议题。缺省情况下，Visual Studio 会为我们生成 `private` 装配件，并包装于单一模块中 (译注：事实上 Visual Studio 无法生成“多模块装配件”)。

8.2 Reflection (运行期型别反射)

`System.Reflection` 命名空间内定义了一套特殊的 *Info classes*，例如上一节见过的 `MethodInfo` class。这些 classes 针对以下各种东西提供访问能力：constructors (构造函数)、events (事件)、fields (数据成员)、methods (成员函数)、parameters (参数)、properties (属性) 等等所拥有之 metadata (元数据) 和 attributes (特征属性)。所有这些 classes 列于表 8.1。

表 8.1 System.Reflection 命名空间中的 Info classes

※译注：以下保留众多英文术语，对应之中文术语请见上页底

class	描述
MemberInfo	获取某个 member 的 attributes, 并提供对 "member metadata" 的访问. 这是一个抽象基类 (abstract base class)
MethodInfo	获取某个 method 的 attributes, 并提供对 "method metadata" 的访问
ParameterInfo	获取某个 parameter 的 attributes, 并提供对 "parameter metadata" 的访问
ConstructorInfo	获取某个 class constructor 的 attributes, 并提供对 "constructor metadata" 的访问
PropertyInfo	获取某个 property 的 attributes, 并提供对 "property metadata" 的访问
FieldInfo	获取某个 field 的 attributes, 并提供对 "field metadata" 的访问
EventInfo	获取某个 event 的 attributes, 并提供对 "event metadata" 的访问

所有的 reflection (反射) 操作都归根于 Type abstract class。表 8.1 Info classes 的访问操作是通过某型别相应之 Type object 完成的。Type class 是“运行期间访问 objects 型别信息”的关键。

获取 Type object 主要有三种方法, 如果我们拥有任何一个 object, 可以调用“从 Object class 继承而来的 nonvirtual GetType()”, 取回其对应的 Type object:

```
public static
void TypeDisplay( object o )
{
    Type t = o.GetType();

    // ... OK: now we have metadata access
}
```

```
// example of TypeDisplay() invocation
```

```
TextQuery tq = new TextQuery();
TypeDisplay( tq );
```

我们可以通过型别名称来获取相应之 Type object, 也就是将内含型别名称的字符串传给 Type 的 static GetType():


```
public static void TypeDisplay( string t_name )
{
    Type t = Type.GetType( t_name );

    if ( t == null )
        // couldn't find it ...
        return;

    // ... OK: now we have metadata access

    // an unqualified name ... note that it won't be found
    // if the type is defined within a namespace
    TypeDisplay( "Math" );

    // a fully qualified name
    TypeDisplay( "System.Math" );

    // a fully qualified name and assembly
    TypeDisplay( "System.Math, mscorlib" );
}
```

我们可以从某个 Assembly object 身上获得其中定义的所有 Type 实体(s)或某一特定 Type 实体。上一节我们已经见过这样的例子。

一旦拿到 Type object, 下一步是取得相应的 Info class objects. 举个例子, 假设我们对 string class 的 Length property 的型别及 get/set 访问器 (accessor) 感兴趣, 该如何获得这些信息呢?

首先我们应该获取某个 string 实体的 Type object:

```
string s = "a simple string";
Type t = s.GetType();
```

接下来调用 Type class 的 GetProperty(), 并指明 Length property 的名称:

```
PropertyInfo pi = t.GetProperty( "Length" );
```

拿到 Length property 对应的 PropertyInfo object 之后, 我们就可以进行深度探索了。例如下面输出结果:

```
Length is of type Int32
Can Read? True
Can Write? False
Actual value is 15
```

是由以下代码生成的:

```
Console.WriteLine( "{0} is of type {1}",  
                    pi.Name, pi.PropertyType );  
  
Console.WriteLine( "Can Read? {0}\nCan Write? {1}",  
                    pi.CanRead, pi.CanWrite );  
  
Console.WriteLine( "Actual value is {0}",  
                    pi.GetValue( s, null ) );
```

如果我们对 `string` class 所包含的 `properties` 一无所知, 又该如何? 如何获得这些信息? `Type` class 提供了一些成对的 `Get` 函数 (译注: 例如 `GetEvent / GetEvents`、`GetField / GetFields`), 其中之一返回与某一指定成员相应的 `Info` object, 就像先前 `GetProperty()` 的作为那样; 另一个 `Get` 函数在缺省情况下返回某型别的所有 `public` 成员——通过一个由 `Info` objects 组成的 `array`. 如果未能取回成员, 则返回一个空数组 (注意, 不是 `null`). 因此, 如果要获得所有的 `properties` 信息, 首先使用 `GetProperties()`:

```
PropertyInfo [] parray = t.GetProperties();  
Console.WriteLine( "{0} has {1} properties",  
                    t.FullName, parray.Length );
```

然后将取回的这个 `array` 遍历一遍, 依次检视每个元素:

```
foreach ( PropertyInfo pinfo in parray )  
{  
    // same Console.WriteLine as above ...  
    Console.WriteLine( "Actual value is {0}",  
                        pinfo.GetValue(s,null));  
}
```

不幸的是以上调用 `GetValue()` 时会引发异常 (`exception`)。原因是 `GetProperties()` 不仅取回 `properties` (属性) 还取回了 `indexers` (索引器)。对 `property` 而言, 传给 `GetValue()` 的第二引数应该是 `null`, 对 `indexer` 而言, 第二引数应该是个 `array` (译注: 型别为 `object[]`), 其所持有的元素个数应该和 `indexer` 所支持的维度相等 (译注: 也就是说 `array` 的大小等于调用此 `indexer` 时的引数个数)。 `array` 内的元素值(s)依次与调用 `indexer` 所用的引数值对应。因此, 为了让代码不失一般性, 我们必须做以下工作。

首先, 确定 `PropertyInfo` object 代表的究竟是 `property` 还是 `indexer`. 为了验证这一点, 我们可以调用 `GetIndexParameters()`:

```
ParameterInfo [] pif = pinfo.GetIndexParameters();
```

如果 `pif` array 长度为零, 则 `pinfo` 所代表的成员是 `property`. 这么一来事情就简单多了: 将 `null` 作为第二引数传给 `GetValue()` 即可. 否则应该以一个 `array of object` 作为 `GetValue()` 的第二引数, `array` 中的各元素持有每个索引对应的值. 那么, 我们如何确定这个 `indexer` 的参数个数及其型别呢?

`GetIndexParameters()` 返回的 `array` 长度代表了这个 `indexer` 所支持的索引参数个数. `ParameterInfo` 提供有 `ParameterType` property, 后者返回此参数对应的 `Type` object. `Position` property 指出参数的位置 (从 0 起算, 第一个参数的位置是 0).

```
ParameterInfo [] pif = pinfo.GetIndexParameters();
```

```
if ( pif.Length != 0 )
```

```
{
```

```
    Console.WriteLine( "{0} is an indexer of {1} indices",  
                        pinfo.Name, pif.Length );
```

```
    foreach ( ParameterInfo parm in pif )
```

```
        Console.WriteLine( "index {0} is of type {1}",  
                            parm.Position+1,  
                            parm.ParameterType );
```

```
}
```

运行结果显示, `String` class 有两个 `public` properties: 一个是名为 `Chars` 的“一维 `char` indexer”, 一个是型别为 `int` 的 `Length` property. 以下是查询结果: (译注: 完整代码在 `chapter8\reflection\reflect.cs` 的 `simpleTest()` 函数内, 请读者随时查看代码, 以弥补书上文字表现的不足)

```
System.String has 2 properties
```

```
Chars is an indexer of 1 indices
```

```
index 1 is of type Int32
```

```
Chars is of type Char
```

```
Can Read? True
Can Write? False
First index value is A // 译注: 应为小写 a
```

```
Length is of type Int32
Can Read? True
Can Write? False
The Property value is 15
```

但愿以上讨论至少能够阐明本章所言的运行期编程 (runtime programming) 和传统的编译期编程 (compile-time programming) 是何等不同。运行期编程 (runtime programming) 要求程序员更关心型别系统 (type system) 的实现细节。它带来了巨大灵活性, 但同时也因为它与运行期环境 (runtime environment) 的交互作用, 使得性能明显下降。

8.3 通过 BindingFlags 修改拣取策略 (Retrieval)

string class 有 5 个 properties, 其中只有两个是 public。缺省情况下, Get() 只拣取 public instance 成员, 包括继承而来的成员和 class 自己声明的成员。我们可利用 BindingFlags enum object 改变上述缺省策略, 例如我们可以要求拣取 static 成员和 nonpublic 成员, 或者不拣取继承而来的任何成员。

BindingFlags 枚举元 (enumerators) 用来告诉“拣取算法”应考虑哪些成员。缺省的 BindingFlags 枚举元包含 Public 和 Instance (译注: 正如上一段所言, 在缺省情况下)。如果想要拣取 nonpublic 成员, 你必须将 NonPublic 枚举元传给 GetProperties()²²。然而下面的写法无法奏效:

```
PropertyInfo[] parray = t.GetProperties(BindingFlags.NonPublic);
```

以上动作总是返回 null, 表示未能取回任何东西。要想让它有效运作, 你还必须告诉“拣取算法”拣选何种成员。欲找出所有 nonpublic instance 成员, 你应该以 bitwise OR (按位或) 操作符连接两个枚举元 (enumerators):

```
BindingFlags f = BindingFlags.NonPublic | BindingFlags.Instance;
PropertyInfo[] parray = t.GetProperties(f);
```

²² 前提是 ReflectionPermission security code 允许访问 nonpublic 成员。如果禁止访问, 会抛出 SecurityException 异常。

如果要找出所有 static 和 instance properties, 包括 public 和 nonpublic, 可以这么写:

```
// OK: retrieve all static and instance members
// that are public and nonpublic

BindingFlags bitmap = BindingFlags.Public |
    BindingFlags.NonPublic;

// consider all static members
bitmap |= BindingFlags.Static;

// consider all instance members
bitmap |= BindingFlags.Instance;

PropertyInfo [] parray = t.GetProperties( bitmap );
```

如果要拣取所有 public、nonpublic、static、instance 成员, 可使用 LookupAll 枚举元。以下调用与上面所写的完全等价:

```
// OK: this is a shorthand notation:
// it also retrieves all public
// and nonpublic static and instance members

PropertyInfo[] parray = t.GetProperties(BindingFlags.LookupAll);
```

一般来说, 只有当我们需要在“缺省策略(也就是仅拣取 public instance 成员)”和“以 LookupAll 拣取所有成员”之间做细微调整时, 才需要对一个个枚举元(enumeralators)施以 bitwise OR 运算。

NonPublic 和 LookupAll 取回的是 base class 的 protected 成员, 但不取回 base class 的 private 成员。

以下, 让我们分别以“使用 BindingFlags”和“不使用 BindingFlags”两种情况来查询 String class 的数据成员。

FieldInfo class 负责持有某个数据成员的信息。缺省情况下, GetFields() 可能不会取回大部分 class 成员, 因为通常我们将数据成员定义为 private。以下是对某个字符串(string)调用 GetFields() 的结果:

```
number of public string fields: 1
```

很令人惊讶吧。我们原以为 String class 拥有不只一个数据成员。当我改以 LookupAll 为引数再次调用 GetFields():

```
FieldInfo[] fi = t.GetFields(BindingFlags.LookupAll);
Console.WriteLine("total number of string fields: {0}", fi.Length);
```

这次取回的结果就好多了:

```
total number of string fields: 13 // 译注: 我的实际运行结果是 8
```

FieldInfo class 的 properties 有 Name、IsPublic、IsPrivate、IsStatic 等等。出于某种理由, 目前尚未有“与 protected 访问级别相应”的 property, 以下的 foreach 循环用来读取挑选回来的各个成员:

```
foreach ( FieldInfo f in fi )
{
    Console.Write( "\t{0} :: ", f.Name );
    Console.Write( "{0} ",
        f.IsPublic
            ? "public"
            : f.IsPrivate
            ? "private" : "property?" );
    // 译注: 我想作者在上一行应该是想写成 ?"private": "protected"

    Console.Write( "{0}\n", f.IsStatic ? "static" : "" );
}
```

这个循环将生成以下输出:

The 13 fields are as follows:

```
m_arrayLength :: private
m_stringLength :: private
m_firstChar :: private
Empty :: public static
WhitespaceChars :: private static
MASK_LENGTH :: private static
MASK_CHARS :: private static
HAS_NO_HIGH_CHARS :: private static
HAS_HIGH_CHARS :: private static
HIGH_CHARS_UNDETERMINED :: private static
TrimHead :: private static
TrimTail :: private static
TrimBoth :: private static
```

译注: 我的实际运行结果是:

```
total number of string fields: 8
```

The 8 fields are as follows:

```
m_arrayLength :: private
m_stringLength :: private
m_firstChar :: private
Empty :: public static
WhitespaceChars :: protected static
```



```
TrimHead :: private static  
TrimTail :: private static  
TrimBoth :: private static
```

BindingFlags 总共定义了 20 个枚举元 (译注: 我的统计结果是 18 个)。此处再提两个, 一个是 DeclaredOnly, 只拣取定义于 class 中的成员 (不含继承而来者), 另一个是 IgnoreCase, 当我们通过名称来拣取成员时可能用得上。每个 Get 函数都有另一份重载实体, 以 BindingFlags 作为第二引数。(译注: 这里应是指取回单个成员的那些 Get 函数, 因为对于取回多个成员的 Get 函数而言, BindingFlags 常被作为第一引数来传递)

GetConstructors() 只取回 public instance 构造函数。如果要同时取回 static 构造函数, 必须显式要求拣取 static, 像这样:

```
// retrieves public constructors and  
// the static constructor, if present  
BindingFlags f = BindingFlags.Instance;  
f |= BindingFlags.Static;  
f |= BindingFlags.Public;  
  
ConstructorInfo[] ci = t.GetConstructors(f);
```

如果要同时取回 nonpublic 构造函数, 可以使用 Private 枚举元搭配 bitwise OR 操作符, 或使用 LookupAll 枚举元。由于 base-class 构造函数不会被继承, 所以不会被拣取。

我们如何选取某个构造函数? 不, 不能只用名称, 因为所有构造函数共用相同名称。构造函数的区分倚赖的是其 signature (标记式), 这正是我们必须传入的参数。如果想要拣取某个构造函数, 我们必须创建一个由 Type objects 组成的 array 用以表示构造函数的每一个参数, 并将它传给 GetConstructor()。以下代码展示两个例子, 第一个例子取回不带引数的构造函数, 如果不存在这样的构造函数就返回 null。第二个例子取回“以 string 为第一引数、int 为第二引数”的构造函数:

```
// first we define the two arrays  
static Type [] nullSignature = new Type[0];  
  
static Type [] stringIntSignature = new Type[] {  
    Type.GetType("System.String", true),  
    Type.GetType("System.Int32")  
};
```

```

ConstructorInfo ctor = t.GetConstructor( nullSignature );
if ( ctor != null ){ ... }

ctor = t.GetConstructor( stringIntSignature );
if ( ctor != null ){ ... }

```

以上概念对所有重载函数都成立。我们不能仅凭名称就区分名为 `Print` 的两份重载实体。我们必须传入第二引数，因而确认想拣取的是哪一份重载函数实体。这里的第二引数是指由 `Type objects` 组成的一个 `array`，其元素依次代表函数的各个参数。例如：

```
MethodInfo mi = t.GetMethod( "Print", stringIntSignature );
```

如果要取回某个 `class` 的所有成员函数（不含构造函数），请用 `GetMethods()`，它会返回一个 `array of MethodInfo objects`。如果该 `class` 没有任何成员函数，则返回一个空空如也的 `array`。

截至目前我们取回的成员函数均为同一类型，或许是 `properties`（属性）或许是 `constructors`（构造函数）。如果要取回所有不同类型的成员，可用 `GetMembers()`。缺省情况下这个函数只返回 `public static` 成员和 `public instance` 成员。当然，你可以传给它 `BindingFlags` 引数，改变它的缺省行为。

另有一个 `GetMember(string)`。如果的确找到了名称吻合的成员，`GetMember()` 返回一个 `MemberInfo array`。注意，不是返回单个 `object`，因为成员函数的名称可能被重载。如果没有找到名称相符的成员就返回 `null`（注意，不是返回空的 `array`）。

8.4 在运行期 (runtime) 调用某个成员函数

`MethodInfo` 和 `ConstructorInfo` 两个 `classes` 都重载了 `Invoke()`，我们可通过它来运行 `MethodInfo object` 或 `ConstructorInfo object` 所反射 (*reflected*，所代表) 的成员（译注：这里的成员指的是 `methods` 和 `constructors`）。首先应该构造一个 `object array`，代表“传递给该成员函数”的参数。如果该成员函数是个 `instance`（而非 `static`）函数，为了调用它我们还必须提供一个相应型别的 `object`。现在来看一个例子。

假设我所设计的函数有着如下的标记式 (*signature*) 和返回型别：

```

static public bool
invokeMethod( string type, string method, object[] args ){ }

```


首先, 我要拣取某个指定型别的 `Type object`. 如果找不到这个型别就返回 `false`:

```
Type t = Type.GetType( type );  
if ( t == null )  
    return false;
```

然后我要拣取我希望调用的成员函数. 如果找不到, 还是返回 `false` (我们很容易就想到, 可以采用“抛出异常”的办法来取代上述“返回 `false`”的做法. 本节末尾将考虑采用哪种办法):

```
MethodInfo mi = t.GetMethod( method );  
if ( mi == null )  
    return false;
```

如果这个成员函数是 `static`, 我们可以直接调用, 因为调用一个 `static` 成员函数并不需要借助 `class object`:

```
if ( mi.IsStatic )  
    mi.Invoke( null, args );
```

但如果这个成员函数是 `nonstatic`, 就得借助 `class object` 才能调用. 欲获得这样一个 `class object` 很简单:

```
if ( mi.IsStatic == false ) {  
    object o = Activator.CreateInstance( t );  
    mi.Invoke( o, args );  
}
```

其中用到的 `Activator class` 定义于 `System` 命名空间内, 提供一组函数, 用来根据特定 `Type object` 创建一个 `local object` 或一个 `remote object`. 缺省情况下它会调用相应型别的“无引数构造函数” (如果有的话) 以进行初始化. 此外我们可以创建一个由引数值组成的 `object array`, 传递给吻合的构造函数.

`Invoke()` 返回一个 `object`, 其中持有该函数的返回值: 如果该函数的返回值型别为 `void`, 则返回 `null`. 目前我们的实现忽略了该返回值, 这可能会招致用户的抱怨. `invokeMethod()` 的一个更加适当的实现方法是, 返回 `Invoke()` 所返回的那个值. 我们以“抛出异常” (而非返回 `false` 值) 的方式来表现运行失败:

```

static public object
invokeMethod( string type, string method, object[] args )
{
    Type t = Type.GetType( type );
    if ( t == null )
        throw new Exception("Unable to find type "+type); //原少 'new'

    MethodInfo mi = t.GetMethod( method );
    if ( mi == null )
        throw new Exception("Unable to find method "+method);

    if ( mi.IsStatic )
        return mi.Invoke( null, args );

    object o = Activator.CreateInstance( t );
    return mi.Invoke( o, args );
}

```

除了 MethodInfo class 的 Invoke(), FieldInfo class 也有一对 GetValue() 和 SetValue(), 可用来读写反射出来的数据成员 (fields)。

如果想要调用某个 property 的 get 或 set 访问器 (accessor), 可分两个步骤进行:

1. 通过 PropertyInfo 的 GetGetMethod() 取回 get 访问器所对应的 MethodInfo object. 如果是针对 set 访问器, 则应使用 GetSetMethod()。
2. 面对取回的 MethodInfo object, 调用其 Invoke() 函数。

8.5 将测试委托 (Delegating) 给 Reflection

2.12 节介绍 delegate 型别时, 曾经大致讨论了一个 testHarness class 的实现. testHarness 拥有一个 static delegate 成员, 别的 classes 可以将想要运行的测试函数注册到这个 delegate 成员身上. 这个 class 看起来像这样 (译注: 同 p.88):

```

public delegate void Action();
public class testHarness
{
    static private Action theAction;
    static public Action Tester
    {
        get{ return theAction; }
        set{ theAction = value; } }
}

```



```

static private void reset() { theAction = null; }
static public int count()
{ return theAction != null
    ? theAction.GetInvocationList().Length : 0; }

// ...
}

```

按约定, 某个 class 应该在其 static 构造函数内将自己的“某个”或“数个”成员函数注册到 Tester 身上, 例如:

```

public class testHashtable
{
    public void test0(){ ... }
    public void test1(){ ... }

    static testHashtable()
    {
        testHarness.Tester += new testHarness.Action( test0 );
        testHarness.Tester += new testHarness.Action( test1 );

        // 译注: 以上两处似应修改如下。见 2.12 节 (p.90)
        // testHarness.Tester += new Action( test0 );
        // testHarness.Tester += new Action( test1 );
    }
    // ...
}

```

并在程序入口处调用 testHarness 的 static run().run() 先测试这个 delegate 是否真的指向某函数; 如果是, 就运行这些注册过的函数, 并重置 (reset) delegate:

```

public class EntryPoint
{
    public static void Main(){ testHarness.run(); }
    // ...
}

public class testHarness
{
    public static void run()
    {
        if ( Tester != null )
            ( Tester(); reset(); )
    }

    // ...
}

```

然而这样的设计有一丝考虑不周。我们的策略取决于“想注册 test 函数”的每一个 class 的 static 构造函数的调用——这些构造函数必须在 run() 运行之前先被调用。由于 run() 是程序入口 Main() 的第一条语句（见上页），我们没有多少地方可以开展这项工作。因此我们面临两个问题：

1. 我们应该在何处调用各个 class 的 static 构造函数，从而确保这项任务在调用 run() 之前完成（亦即运行 Main() 的第一条语句之前完成）？
2. 到底要做什么工作，才能确保调用相应的 static 构造函数？

要完成以上工作，只有一个地方是绝对保险的。我们得到的保证是 testHarness 的 static 构造函数一定会在“Main() 内的 run() 函数被执行前”先被调用。如果这样，那么 testHarness 的 static 构造函数便是我们完成工作所需的时间点和地点。

到底要做些什么工作呢？首先我们要找出应用程序中的所有型别，因为它们都有可能想要注册测试函数。

接下来我们需要引发该型别的 static 构造函数的执行。这可以通过“为该型别创建一份实体”而办到。为求优化起见，我们先检测该型别是否为预定义之 .NET Framework 的一部分，如果是就不必创建这份实体。

当然，这会耗去一些时间，但因为我们处于测试阶段，所以时间不是太大问题。此外，将测试过程自动化，可以节约劳动时间。

要想找出某个可执行文件（executable）中出现的所有型别，首先必须取回程序相应的所有装配件(s)，例如（译注：请参考 p.350）：

```
AppDomain appdomain = AppDomain.CurrentDomain;  
Assembly [] assemblies = appdomain.GetAssemblies();
```

GetAssemblies() 返回一个由 Assembly class objects 组成的 array，用以表示加载于当前应用程序中的装配件。接下来滤除所有 System 装配件：

```
foreach ( Assembly a in assemblies )  
{  
    if ( isSystemAssembly( a ) )  
        // 译注：见配套源码 chapter8\tester\tester.cs 的具体做法  
        continue;  
}
```


余下的装配件有可能包含“想要注册测试函数”的 classes。因此，对每个装配件，我们调用 `GetTypes()` 取回一个 `Type objects array` (持有装配件中定义的所有型别)。然后遍历这个 `array`，并为其中记录的每个型别创建一份实体：

```
Type [] t = a.GetTypes();  
  
foreach ( Type tt in t )  
    if ( tt.IsClass )  
        Activator.CreateInstance( tt );
```

以上调用 `CreateInstance()` 将触发相应 class 的 `static` 构造函数 (如果有的话)，这些构造函数将依次向 `test.Tester delegate` 中添加东西，这一切都在执行 `test.run()` 之前发生 (译注：test 是个 `testHarness object`)。

当然，还有其他设计策略可用。例如令 class 注册其名称，或传入其型别，或以某种方式让自己被 `testHarness` 知晓。但我希望这些 classes 被动些，我希望它们只需知道它们的实体将被神奇地创建出来，而后测试动作会自动被执行。

8.6 Attributes (特征属性)

Attributes 被视为“元声明信息” (*metadeclarative information*)。C# 支持若干预定义的 attributes (又称 *intrinsic attributes*，固有属性、内置属性)。程序员也能定义新的 attribute 型别，并可在运行期通过 *type reflection* (型别反射) 取得或查询这些 attributes。“固有型 attributes”和“用户自定义型 attributes”都是 classes，尽管它们的语法看起来是基于文本 (text based)。在学习“自定义 attribute”之前，我们先大致研究一下“固有型 attributes”。

8.6.1 固有型 Attribute: Conditional

Conditional attribute 使我们能够定义一些 class 成员函数，这些函数将根据某字符串是否被定义而决定是否被调用 (然而我们不能将 Conditional attribute 置于数据成员或 properties 身上)。attribute 被置于一对方括弧 ([]) 之间，位于它所修饰的成员函数前。例如 `open_debug_output()` 和 `display()` 就附有一个 Conditional attribute:

```

using System.Diagnostics;
public class string_length : IComparer
{
    [Conditional("DEBUG")]
    private void open_debug_output()
    {
        FileStream fout =
            new FileStream("debug.txt", FileMode.Create );
        of = new StreamWriter( fout );
    }

    [Conditional("DEBUG")]
    public void display( string xs, string ys, int ret_val )
    {
        of.WriteLine("inside conditional function display()!");
        of.WriteLine("word #1: {0} : {1} ", xs, xs.Length);
        of.WriteLine("word #2: {0} : {1} ", ys, ys.Length);
        of.WriteLine("return value: {0} ", ret_val);
    }

    // not allowed: conditional data member
    // [Conditional("DEBUG")]
    private StreamWriter of;
}

```

方括弧内的字符串总是与 #define 语句相关。如果要想让上述的 Conditional attribute 的值为 true，你必须在程序中写下：

```
#define DEBUG
```

你也可以使用 #undef 取消某个定义，例如：

```
#undef DEBUG
```

由于预处理器 (preprocessor) 命令必须出现在 C# 语句之前，所以代码中不能嵌套使用 #undef 命令。调用上述成员函数并不需要什么条件，虽然它们有可能根本不会被调用，例如：

```

public class string_length : IComparer
{
    public string_length() {
        // if DEBUG is defined, this executes
        open_debug_output();
    }
}

```



```

public int Compare( object x, object y )
{
    if (( ! ( x is string )) || ( ! ( y is string )))
        throw new ArgumentException("both must be strings");

    string xs = (string) x, ys = (string) y;
    int ret_val = 1;

    // calculate result in ret_val

    // if DEBUG is defined, this executes
    display( xs, ys, ret_val );

    return ret_val;
}
}

```

如果 Conditional 内的字符串没有被定义出来, 则执行期会忽略代码之中那些 Conditional 成员函数的调用行为。

8.6.2 固有型 Attribute: Serializable

Serializable attribute 表示某个 class 可被序列化 (serialized)。serialization 的意思是让某个 object 超越可执行文件之运行寿命而持续存在 (persisting), 并保持 object 的当时状态, 以便日后恢复。我们可以将 objects 序列化至某个存储设备, 例如硬盘或远程 (remote) 计算机。此外也可以指明某些数据成员 (fields) 为 NonSerialized (不可序列化)。以下是定义为 Serializable 的 Matrix class, 它的 out_stream 成员被标示为 NonSerialized:

```

[Serializable]
class Matrix
{
    [NonSerialized]
    private string out_stream;

    float [,] mat;
}

```

既然将这个 class 标记为 Serializable, 那么该如何进行序列化呢? 以下样例将某个 class object 保存到磁盘文件中, 再读取回来 (没有做任何错误检查)。留给我们的工作少得惊人。

首先将 object 序列化 (serialize) 至磁盘, 这会用到定义于 Binary 命名空间中的 BinaryFormatter class. 序列化行为 (serialization) 是通过 BinaryFormatter 的 Serialize() 完成的:

```
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;
public void ToDisk()
{
    Stream s = File.Open(out_stream, FileMode.Create);
    // 译注: 原书上一行有误, 已改正。
    BinaryFormatter bfm = new BinaryFormatter();
    bfm.Serialize(s, this);
    s.Close();
}
```

涂写动作的反向操作 (反序列化) —— 将写至磁盘的 objects 恢复原样 —— 是通过 BinaryFormatter 的 Deserialize() 完成的:

```
public void FromDisk( string filename )
{
    Stream s = File.Open(filename, FileMode.Open);
    BinaryFormatter bfm = new BinaryFormatter();
    Matrix m = (Matrix) bfm.Deserialize(s);
    s.Close();

    mat = m.mat;
    out_stream = filename;
    // 译注: out_stream 是 NonSerialized, 所以必须由我们自行恢复
}
```

为了能够对所有型别起作用, Deserialize() 必须返回一个 object object. 因此程序中必须将它显式向下转型为 Matrix.

8.6.3 固有型 Attribute: DllImport

DllImport attribute 可让我们调用一个“不是由 .NET 产生的函数”. 举个例子, 我们想要实现一个简单的“起/停计时器” Timer class, 用以测量某个例程 (routine) 的运行耗时²³. 这里所用的底层例程是 unmanaged (非受控) Win32 API. Timer class 的两个函数声明如下:

²³ 当 Microsoft 从 .NET Framework beta 版本中拿掉了 Counter class 后, *A Programmer's Introduction to C#* (APress, 2000) 的作者 Eric Gunnerson 慷慨地与大家共享了他的 Counter class, 这里的 Timer 就是从 Eric 版本简化来的.


```

public class Timer
{
    private long    m_elapsedCount;
    private long    m_startCount;
    private string  m_context;

    [System.Runtime.InteropServices.DllImport("KERNEL32.dll")]
    private static extern bool
        QueryPerformanceCounter(ref long cnt);

    [System.Runtime.InteropServices.DllImport("KERNEL32.dll")]
    private static extern bool
        QueryPerformanceFrequency(ref long freq);
}

```

这两个函数被声明为 `extern`，因为它们是外部定义的——我们只声明它们，并不为它们提供定义。它们可以是 `private`、`public` 或 `protected`。我们就当它们是普通成员函数似地调用它们，例如：

```

public class Timer
{
    public void start() {
        m_startCount = 0;
        QueryPerformanceCounter(ref m_startCount);
    }

    public void stop() {
        long stopCount = 0;
        QueryPerformanceCounter(ref stopCount);
        m_elapsedCount = (stopCount - m_startCount);
    }

    public override string ToString()
    {
        long freq = 0;
        QueryPerformanceFrequency(ref freq);
        float seconds = (float) m_elapsedCount / (float) freq;
        //译注：原书上一行有误，已改正
        return m_context + " : " +
            seconds.ToString() + " secs.";
    }
    // ...
}

```

动态添加的控件并不是页面外观状态 (view state) 的一部分, 因此不会被自动保存, 你必须自行管理它们, 否则经过一次 round-trip 之后, 这些控件就丢失了。例如用户在 Panel 内的 Calendar 控件上选择一个日期, 回送 (post back) 之后这些控件就不再出现了。

通常我们会在 Page_Load() event handler 中创建动态控件, 以此确保在 round-trip 时能恢复这些控件。如果这些控件之中有着程序所需的数据需要维护, 我们可以在 Session object 中管理 (维护) 这些数据。

在 Page_Load() 事件中, 我们使用 Page.FindControl() 方法来查找控件。如果找到了控件, 我们就使用 Page.FindControl() 方法来查找控件。如果找到了控件, 我们就使用 Page.FindControl() 方法来查找控件。

在 Page_Load() 事件中, 我们使用 Page.FindControl() 方法来查找控件。如果找到了控件, 我们就使用 Page.FindControl() 方法来查找控件。如果找到了控件, 我们就使用 Page.FindControl() 方法来查找控件。

在 Page_Load() 事件中, 我们使用 Page.FindControl() 方法来查找控件。如果找到了控件, 我们就使用 Page.FindControl() 方法来查找控件。如果找到了控件, 我们就使用 Page.FindControl() 方法来查找控件。

在 Page_Load() 事件中, 我们使用 Page.FindControl() 方法来查找控件。如果找到了控件, 我们就使用 Page.FindControl() 方法来查找控件。如果找到了控件, 我们就使用 Page.FindControl() 方法来查找控件。

在 Page_Load() 事件中, 我们使用 Page.FindControl() 方法来查找控件。如果找到了控件, 我们就使用 Page.FindControl() 方法来查找控件。如果找到了控件, 我们就使用 Page.FindControl() 方法来查找控件。

在 Page_Load() 事件中, 我们使用 Page.FindControl() 方法来查找控件。如果找到了控件, 我们就使用 Page.FindControl() 方法来查找控件。如果找到了控件, 我们就使用 Page.FindControl() 方法来查找控件。

在 Visual Studio.NET 中，程序开发单元是装配件（assembly），那是一种“component DNA”。装配件是 .NET 应用程序中的一个“自成一体（self-contained）、版本可控制（versionable）、自我描述（self-describing）”的单元。当然，它包含了我们写下的所有可运行代码。这些代码以中间语言（intermediate language，稍后介绍）的方式存储。此外，装配件持有 metadata，其中记录了“程序所定义的类型”和“程序编译链接环境”，这些信息是在编译链接程序代码时生成的。装配件也包含一张清单（manifest），描述此装配件的内容，其中包含运行条件：它所倚赖的组件、它的版本管理，以及它的安全设施。

装配件是按需加载（load on demand），换句话说，只有当我们的程序用到某个装配件内的类型，或用到该类型的相关 metadata 时，运行期环境才会加载这个装配件。这件事将发生在“这一类型的 static 构造函数被执行”时。

我们可以访问“运行中的装配件”以及“与吾人程序相关的任何装配件”的运行期表述（runtime representation）。办法有好几种，取决于我们的需求。

如果需要访问“与应用程序相关”的所有装配件，该怎么办？首先我们要能够通过 AppDomain class 的 CurrentDomain property，访问当前线程（current thread）。App domain 被当做装配件的容器，装配件被加载到 app domain 中。以下代码以 array of Assembly class objects 的形式取回当前线程的所有相关装配件：

```
AppDomain theApp = AppDomain.CurrentDomain;  
Assembly [] currAssemblies = theApp.GetAssemblies();  
  
Console.WriteLine( "The current AppDomain has {0} Assemblies",  
                    currAssemblies.Length );
```

为了编译这段代码，我们必须开启 System.Reflection 命名空间，Assembly class 正是定义于其中（8.2 节将谈到 Reflection 命名空间）：

```
using System.Reflection;
```

如果想取回“目前运行的程序”对应之 `Assembly` object, 我们就调用 `Assembly` class 的 static 成员函数 `GetExecutingAssembly()`:

```
Assembly myAssembly = Assembly.GetExecutingAssembly();
```

如果想取回“定义了某特定型别”的 `Assembly` object, 我们调用 `Assembly` class 的 static 成员函数 `GetAssembly()`, 传给它你想找的那个特定型别所对应的 `Type` object.

欲获得 `Type` object, 有两个办法. 如果我们有该型别的 object, 只需调用它 (从 `Object` class 继承而来) 的 `GetType()` 成员函数即可. 另一种办法是, 我们可以调用 `Type` class 的 static 成员函数 `GetType()`, 传给它型别名称 (一个字符串):

```
Object o = new Fibonacci();
Assembly fibAssem = Assembly.GetAssembly( o.GetType() );

Type calType = Type.GetType("System.Globalization.JulianCalendar");
// 译注: 英文版上一行缺少 "System.Globalization."
Assembly calAssem = Assembly.GetAssembly( calType );
```

通过 `Assembly` class, 几乎可以获知组件的任何信息. 如果要取得 `Assembly` object 入口 (entry point; 如果有的话), 可询问 `Assembly` class 的 `EntryPoint` property:

```
MethodInfo ep = fibAssem.EntryPoint;
```

其中的 `MethodInfo` class 定义于 `System.Reflection` 命名空间, 内含它所指向 (代表) 的成员函数的详细信息, 诸如名称、访问级别、返回型别、参数列等等. 8.2 节将更深入地研究它.

如果要取回装配件中定义的所有型别, 应该调用 `GetTypes()` 成员函数:

```
Type [] types = fibAssem.GetTypes();
```

此外亦可通过 `GetType()` 成员函数, 按名称取回特定型别:

```
Type fibType = fibAssem.GetType( "Fibonacci" );
```


如果没有找到吾人指定的型别, `GetType()` 返回 `null`。如果想令它在未能找到指定的型别时抛出一个异常, 可以将字面值 `true` 作为第二引数传入:

```
Type fibType = fibAssem.GetType( "Fibonacci", true );
```

举个例子, 以下的输出结果:

译注: 从以下的 `Ver=1.0.2204.21` 可看出作者用的 .NET Framework 是 Beta 版, 而本章涉及底层机制, 可能与最新的 .NET Framework 有不符之处。我在翻译时尽力找出不相符的地方。

```
The current AppDomain has 2 Assemblies
```

```
The Assembly FullName Property:
```

```
mscorlib, Ver=1.0.2204.21, Loc="", SN=03689116d3a4ae33
```

```
assembly name: mscorlib
```

```
manifest loc: C:/WINNT/Microsoft.NET/Framework/v1.0.2204/mscorlib.dll
```

```
entry point: Not Defined
```

```
number of types: 1205
```

```
The Assembly FullName Property:
```

```
Assemblies, Ver=1.0.471.38017, Loc=""
```

```
assembly name: Assemblies
```

```
manifest loc: C:/C#Programs/Assemblies/bin/Debug/Assemblies.exe
```

```
entry point: Void Main()
```

```
number of types: 2
```

```
The types contained within the assembly are:
```

```
type name: AssemblyEntryPoint
```

```
type name: AssemblyExplore
```

便是由以下函数生成的:

```
public static void getAssemblies()
```

```
{
```

```
AppDomain theApp = AppDomain.CurrentDomain;
```

```
Assembly [] currAssemblies = theApp.GetAssemblies();
```

```
Console.WriteLine( "The current AppDomain has {0} Assemblies",  
currAssemblies.Length );
```

```
foreach ( Assembly a in currAssemblies )
```

```
{
```

```
message( "\nThe Assembly FullName Property: \n\t",  
a.FullName );
```

```

int    pos  = a.FullName.IndexOf( ',' );
string name = a.FullName.Substring(0, pos);
Type [] t   = a.GetTypes();

string    manifestLocation = a.Location;
MethodInfo theEntry        = a.EntryPoint;

message( "assembly name:  ", name );
message( "manifest loc:   ", manifestLocation );
message( "entry point:    ",
        theEntry != null ? theEntry.ToString() : "Not Defined" );

message( "number of types: ", t.Length );

if ( t.Length < 10 )
{
    Console.WriteLine(
        "The types contained within the assembly are:");

    foreach ( Type tt in t )
        message( "\ttype name: ", tt.Name );
}
}
}

```

你也可以使用 `Assembly class` 的 static `Load()` 或 `LoadFrom()` 自己动手加载某个装配件。如果想要为“定义于 `Assembly object` 中的某个型别”创建一份实体，可以使用 `CreateInstance()`。下一节讨论反射 (reflection) 时我将展示一个样例。

许多管理方面的议题我并没有提到，例如应该将装配件包装于单一模块 (module) 或包装于多个模块内 (亦即装配件由单一模块或多个模块组成)？装配件应该是 `private` 或 `shared`？此外还有版本管理、安全设施等议题。缺省情况下，Visual Studio 会为我们生成 `private` 装配件，并包装于单一模块中 (译注：事实上 Visual Studio 无法生成“多模块装配件”)。

8.2 Reflection (运行期型别反射)

`System.Reflection` 命名空间内定义了一套特殊的 *Info classes*，例如上一节见过的 `MethodInfo class`。这些 *classes* 针对以下各种东西提供访问能力：constructors (构造函数)、events (事件)、fields (数据成员)、methods (成员函数)、parameters (参数)、properties (属性) 等等所拥有之 *metadata* (元数据) 和 *attributes* (特征属性)。所有这些 *classes* 列于表 8.1。

表 8.1 System.Reflection 命名空间中的 Info classes

※译注：以下保留众多英文术语，对应之中文术语请见上页底

class	描述
MemberInfo	获取某个 member 的 attributes，并提供对 "member metadata" 的访问。这是一个抽象基类 (abstract base class)
MethodInfo	获取某个 method 的 attributes，并提供对 "method metadata" 的访问
ParameterInfo	获取某个 parameter 的 attributes，并提供对 "parameter metadata" 的访问
ConstructorInfo	获取某个 class constructor 的 attributes，并提供对 "constructor metadata" 的访问
PropertyInfo	获取某个 property 的 attributes，并提供对 "property metadata" 的访问
FieldInfo	获取某个 field 的 attributes，并提供对 "field metadata" 的访问
EventInfo	获取某个 event 的 attributes，并提供对 "event metadata" 的访问

所有的 reflection (反射) 操作都归根于 Type abstract class。表 8.1 Info classes 的访问操作是通过某型别相应之 Type object 完成的。Type class 是“运行期间访问 objects 型别信息”的关键。

获取 Type object 主要有三种方法。如果我们拥有任何一个 object，可以调用“从 Object class 继承而来的 nonvirtual GetType()”，取回其对应的 Type object：

```
public static
void TypeDisplay( object o )
{
    Type t = o.GetType();

    // ... OK: now we have metadata access
}
```

```
// example of TypeDisplay() invocation
```

```
TextQuery tq = new TextQuery();
TypeDisplay( tq );
```

我们可以通过型别名称来获取相应之 Type object，也就是将内含型别名称的字符串传给 Type 的 static GetType()：

```
public static void TypeDisplay( string t_name )
{
    Type t = Type.GetType( t_name );

    if ( t == null )
        // couldn't find it ...
        return;

    // ... OK: now we have metadata access

    // an unqualified name ... note that it won't be found
    // if the type is defined within a namespace
    TypeDisplay( "Math" );

    // a fully qualified name
    TypeDisplay( "System.Math" );

    // a fully qualified name and assembly
    TypeDisplay( "System.Math, mscorlib" );
}
```

我们可以从某个 `Assembly` object 身上获得其中定义的所有 `Type` 实体(s)或某一特定 `Type` 实体。上一节我们已经见过这样的例子。

一旦拿到 `Type` object, 下一步是取得相应的 `Info` class objects. 举个例子, 假设我们对 `string` class 的 `Length` property 的型别及 `get/set` 访问器 (accessor) 感兴趣, 该如何获得这些信息呢?

首先我们应该获取某个 `string` 实体的 `Type` object:

```
string s = "a simple string";
Type t = s.GetType();
```

接下来调用 `Type` class 的 `GetProperty()`, 并指明 `Length` property 的名称:

```
PropertyInfo pi = t.GetProperty( "Length" );
```

拿到 `Length` property 对应的 `PropertyInfo` object 之后, 我们就可以进行深度探索了。例如下面输出结果:

```
Length is of type Int32
Can Read? True
Can Write? False
Actual value is 15
```


是由以下代码生成的:

```
Console.WriteLine( "{0} is of type {1}",  
    pi.Name, pi.PropertyType );  
  
Console.WriteLine( "Can Read? {0}\nCan Write? {1}",  
    pi.CanRead, pi.CanWrite );  
  
Console.WriteLine( "Actual value is {0}",  
    pi.GetValue( s, null ) );
```

如果我们对 `string` class 所包含的 `properties` 一无所知, 又该如何? 如何获得这些信息? `Type` class 提供了一些成对的 `Get` 函数 (译注: 例如 `GetEvent / GetEvents`、`GetField / GetFields`), 其中之一返回与某一指定成员相应的 `Info` object, 就像先前 `GetProperty()` 的作为那样; 另一个 `Get` 函数在缺省情况下返回某型别的所有 `public` 成员——通过一个由 `Info` objects 组成的 `array`. 如果未能取回成员, 则返回一个空数组 (注意, 不是 `null`). 因此, 如果要获得所有的 `properties` 信息, 首先使用 `GetProperties()`:

```
PropertyInfo [] parray = t.GetProperties();  
Console.WriteLine( "{0} has {1} properties",  
    t.FullName, parray.Length );
```

然后将取回的这个 `array` 遍历一遍, 依次检视每个元素:

```
foreach ( PropertyInfo pinfo in parray )  
{  
    // same Console.WriteLine as above ...  
    Console.WriteLine( "Actual value is {0}",  
        pinfo.GetValue(s, null));  
}
```

不幸的是以上调用 `GetValue()` 时会引发异常 (exception). 原因是 `GetProperties()` 不仅取回 `properties` (属性) 还取回了 `indexers` (索引器). 对 `property` 而言, 传给 `GetValue()` 的第二引数应该是 `null`, 对 `indexer` 而言, 第二引数应该是个 `array` (译注: 型别为 `object[]`), 其所持有的元素个数应该和 `indexer` 所支持的维度相等 (译注: 也就是说 `array` 的大小等于调用此 `indexer` 时的引数个数). `array` 内的元素值(s)依次与调用 `indexer` 所用的引数值对应. 因此, 为了让代码不失一般性, 我们必须做以下工作.

```
Can Read? True
Can Write? False
First index value is A // 译注: 应为小写 a
```

```
Length is of type Int32
Can Read? True
Can Write? False
The Property value is 15
```

但愿以上讨论至少能够阐明本章所言的运行期编程 (runtime programming) 和传统的编译期编程 (compile-time programming) 是何等不同。运行期编程 (runtime programming) 要求程序员更关心型别系统 (type system) 的实现细节。它带来了巨大灵活性, 但同时也因为它与运行期环境 (runtime environment) 的交互作用, 使得性能明显下降。

8.3 通过 BindingFlags 修改拣取策略 (Retrieval)

string class 有 5 个 properties, 其中只有两个是 public。缺省情况下, Get() 只拣取 public instance 成员, 包括继承而来的成员和 class 自己声明的成员。我们可利用 BindingFlags enum object 改变上述缺省策略, 例如我们可以要求拣取 static 成员和 nonpublic 成员, 或者不拣取继承而来的任何成员。

BindingFlags 枚举元 (enumerators) 用来告诉“拣取算法”应考虑哪些成员。缺省的 BindingFlags 枚举元包含 Public 和 Instance (译注: 正如上一段所言, 在缺省情况下)。如果想要拣取 nonpublic 成员, 你必须将 NonPublic 枚举元传给 GetProperties()²²。然而下面的写法无法奏效:

```
PropertyInfo[] parray = t.GetProperties(BindingFlags.NonPublic);
```

以上动作总是返回 null, 表示未能取回任何东西。要想让它有效运作, 你还必须告诉“拣取算法”拣选何种成员。欲找出所有 nonpublic instance 成员, 你应该以 bitwise OR (按位或) 操作符连接两个枚举元 (enumerators):

```
BindingFlags f = BindingFlags.NonPublic | BindingFlags.Instance;
PropertyInfo[] parray = t.GetProperties(f);
```

²² 前提是 ReflectionPermission security code 允许访问 nonpublic 成员。如果禁止访问, 会抛出 SecurityException 异常。

如果要找出所有 `static` 和 `instance` properties, 包括 `public` 和 `nonpublic`, 可以这么写:

```
// OK: retrieve all static and instance members  
// that are public and nonpublic
```

```
BindingFlags bitmap = BindingFlags.Public |  
                        BindingFlags.NonPublic;
```

```
// consider all static members  
bitmap |= BindingFlags.Static;
```

```
// consider all instance members  
bitmap |= BindingFlags.Instance;
```

```
PropertyInfo [] parray = t.GetProperties( bitmap );
```

如果要拣取所有 `public`、`nonpublic`、`static`、`instance` 成员, 可使用 `LookupAll` 枚举元。以下调用与上面所写的完全等价:

```
// OK: this is a shorthand notation:  
// it also retrieves all public  
// and nonpublic static and instance members
```

```
PropertyInfo[] parray = t.GetProperties(BindingFlags.LookupAll);
```

一般来说, 只有当我们需要在“缺省策略(也就是仅拣取 `public instance` 成员)”和“以 `LookupAll` 拣取所有成员”之间做细微调整时, 才需要对一个个枚举元 (enumerators) 施以 bitwise OR 运算。

`NonPublic` 和 `LookupAll` 取回的是 `base class` 的 `protected` 成员, 但不取回 `base class` 的 `private` 成员。

以下, 让我们分别以“使用 `BindingFlags`”和“不使用 `BindingFlags`”两种情况来查询 `String class` 的数据成员。

`FieldInfo class` 负责持有某个数据成员的信息。缺省情况下, `GetFields()` 可能不会取回大部分 `class` 成员, 因为通常我们将数据成员定义为 `private`。以下是对某个字符串 (`string`) 调用 `GetFields()` 的结果:

```
number of public string fields: 1
```

很令人惊讶吧。我们原以为 `String class` 拥有不只一个数据成员。当我改以 `LookupAll` 为引数再次调用 `GetFields()`:

```
FieldInfo[] fi = t.GetFields(BindingFlags.LookupAll);
Console.WriteLine("total number of string fields: {0}", fi.Length);
```

这次取回的结果就好多了:

```
total number of string fields: 13 // 译注: 我的实际运行结果是 8
```

FieldInfo class 的 properties 有 Name、IsPublic、IsPrivate、IsStatic 等等。出于某种理由, 目前尚未有“与 protected 访问级别相应”的 property。以下的 foreach 循环用来读取拣选回来的各个成员:

```
foreach ( FieldInfo f in fi )
{
    Console.Write( "\t{0} :: ", f.Name );
    Console.Write( "{0} ",
        f.IsPublic
            ? "public"
            : f.IsPrivate
            ? "private" : "property?" );
    // 译注: 我想作者在上一行应该是想写成 ? "private" : "protected"
    Console.Write( "{0}\n", f.IsStatic ? "static" : "" );
}
```

这个循环将生成以下输出:

```
The 13 fields are as follows:
m_arrayLength :: private
m_stringLength :: private
m_firstChar :: private
Empty :: public static
WhitespaceChars :: private static
MASK_LENGTH :: private static
MASK_CHARS :: private static
HAS_NO_HIGH_CHARS :: private static
HAS_HIGH_CHARS :: private static
HIGH_CHARS_UNDETERMINED :: private static
TrimHead :: private static
TrimTail :: private static
TrimBoth :: private static
```

译注: 我的实际运行结果是:

```
total number of string fields: 8
```

```
The 8 fields are as follows:
```

```
m_arrayLength :: private
m_stringLength :: private
m_firstChar :: private
Empty :: public static
WhitespaceChars :: protected static
```



```
TrimHead :: private static
TrimTail :: private static
TrimBoth :: private static
```

BindingFlags 总共定义了 20 个枚举元 (译注: 我的统计结果是 18 个)。此处再提两个, 一个是 DeclaredOnly, 只拣取定义于 class 中的成员 (不含继承而来者), 另一个是 IgnoreCase, 当我们通过名称来拣取成员时可能用得上。每个 Get 函数都有另一份重载实体, 以 BindingFlags 作为第二引数。(译注: 这里应是指取回单个成员的那些 Get 函数, 因为对于取回多个成员的 Get 函数而言, BindingFlags 常被作为第一引数来传递)

GetConstructors() 只取回 public instance 构造函数。如果要同时取回 static 构造函数, 必须显式要求拣取 static, 像这样:

```
// retrieves public constructors and
// the static constructor, if present
BindingFlags f = BindingFlags.Instance;
f |= BindingFlags.Static;
f |= BindingFlags.Public;
```

```
ConstructorInfo[] ci = t.GetConstructors(f);
```

如果要同时取回 nonpublic 构造函数, 可以使用 Private 枚举元搭配 bitwise OR 操作符, 或使用 LookupAll 枚举元。由于 base-class 构造函数不会被继承, 所以不会被拣取。

我们如何选取某个构造函数? 不, 不能只用名称。因为所有构造函数共用相同名称。构造函数的区分倚赖的是其 signature (标记式), 这正是我们必须传入的参数。如果想要拣取某个构造函数, 我们必须创建一个由 Type objects 组成的 array 用以表示构造函数的每一个参数, 并将它传给 GetConstructor()。以下代码展示两个例子, 第一个例子取回不带引数的构造函数, 如果不存在这样的构造函数就返回 null。第二个例子取回“以 string 为第一引数、int 为第二引数”的构造函数:

```
// first we define the two arrays
static Type [] nullSignature = new Type[0];

static Type [] stringIntSignature = new Type[] {
    Type.GetType("System.String", true ),
    Type.GetType("System.Int32" )
};
```

```

ConstructorInfo ctor = t.GetConstructor( nullSignature );
if ( ctor != null ){ ... }

ctor = t.GetConstructor( stringIntSignature );
if ( ctor != null ){ ... }

```

以上概念对所有重载函数都成立。我们不能仅凭名称就区分名为 `Print` 的两份重载实体。我们必须传入第二引数，因而确认想拣取的是哪一份重载函数实体。这里的第二引数是指由 `Type objects` 组成的一个 `array`，其元素依次代表函数的各个参数。例如：

```
MethodInfo mi = t.GetMethod( "Print", stringIntSignature );
```

如果要取回某个 `class` 的所有成员函数（不含构造函数），请用 `GetMethods()`，它会返回一个 `array of MethodInfo objects`。如果该 `class` 没有任何成员函数，则返回一个空空如也的 `array`。

截至目前我们取回的成员函数均为同一类型，或许是 `properties`（属性）或许是 `constructors`（构造函数）。如果要取回所有不同类型的成员，可用 `GetMembers()`。缺省情况下这个函数只返回 `public static` 成员和 `public instance` 成员。当然，你可以传给它 `BindingFlags` 引数，改变它的缺省行为。

另有一个 `GetMember(string)`。如果的确找到了名称吻合的成员，`GetMember()` 返回一个 `MemberInfo array`。注意，不是返回单个 `object`，因为成员函数的名称可能被重载。如果没有找到名称相符的成员就返回 `null`（注意，不是返回空的 `array`）。

8.4 在运行期 (runtime) 调用某个成员函数

`MethodInfo` 和 `ConstructorInfo` 两个 `classes` 都重载了 `Invoke()`，我们可通过它来运行 `MethodInfo object` 或 `ConstructorInfo object` 所反射 (*reflected*，所代表) 的成员（译注：这里的成员指的是 `methods` 和 `constructors`）。首先应该构造一个 `object array`，代表“传递给该成员函数”的参数。如果该成员函数是个 `instance`（而非 `static`）函数，为了调用它我们还必须提供一个相应型别的 `object`。现在来看一个例子。

假设我所设计的函数有着如下的标记式 (*signature*) 和返回型别：

```

static public bool
invokeMethod( string type, string method, object[] args ){ }

```


首先, 我要拣取某个指定型别的 `Type object`. 如果找不到这个型别就返回 `false`:

```
Type t = Type.GetType( type );  
if ( t == null )  
    return false;
```

然后我要拣取我希望调用的成员函数。如果找不到, 还是返回 `false` (我们很容易就想到, 可以采用“抛出异常”的办法来取代上述“返回 `false`”的做法, 本节末尾将考虑采用哪种办法):

```
MethodInfo mi = t.GetMethod( method );  
if ( mi == null )  
    return false;
```

如果这个成员函数是 `static`, 我们可以直接调用, 因为调用一个 `static` 成员函数并不需要借助 `class object`:

```
if ( mi.IsStatic )  
    mi.Invoke( null, args );
```

但如果这个成员函数是 `nonstatic`, 就得借助 `class object` 才能调用。欲获得这样一个 `class object` 很简单:

```
if ( mi.IsStatic == false ){  
    object o = Activator.CreateInstance( t );  
    mi.Invoke( o, args );  
}
```

其中用到的 `Activator class` 定义于 `System` 命名空间内, 提供一组函数, 用来根据特定 `Type object` 创建一个 `local object` 或一个 `remote object`. 缺省情况下它会调用相应型别的“无引数构造函数” (如果有的话) 以进行初始化. 此外我们可以创建一个由引数值组成的 `object array`, 传递给吻合的构造函数。

`Invoke()` 返回一个 `object`, 其中持有该函数的返回值; 如果该函数的返回值型别为 `void`, 则返回 `null`. 目前我们的实现忽略了该返回值, 这可能会招致用户的抱怨. `invokeMethod()` 的一个更加适当的实现方法是, 返回 `Invoke()` 所返回的那个值. 我们以“抛出异常” (而非返回 `false` 值) 的方式来表现运行失败:

```

static public object
invokeMethod( string type, string method, object[] args )
{
    Type t = Type.GetType( type );
    if ( t == null )
        throw new Exception("Unable to find type "+type); //原少 'new'

    MethodInfo mi = t.GetMethod( method );
    if ( mi == null )
        throw new Exception("Unable to find method "+method);

    if ( mi.IsStatic )
        return mi.Invoke( null, args );

    object o = Activator.CreateInstance( t );
    return mi.Invoke( o, args );
}

```

除了 `MethodInfo` class 的 `Invoke()`, `FieldInfo` class 也有一对 `GetValue()` 和 `SetValue()`, 可用来读写反射出来的数据成员 (fields)。

如果想要调用某个 `property` 的 `get` 或 `set` 访问器 (accessor), 可分两个步骤进行:

1. 通过 `PropertyInfo` 的 `GetMethod()` 取回 `get` 访问器所对应的 `MethodInfo` object. 如果是针对 `set` 访问器, 则应使用 `SetMethod()`。
2. 面对取回的 `MethodInfo` object, 调用其 `Invoke()` 函数。

8.5 将测试委托 (Delegating) 给 Reflection

2.12 节介绍 `delegate` 型别时, 曾经大致讨论了一个 `testHarness` class 的实现. `testHarness` 拥有一个 `static delegate` 成员, 别的 classes 可以将想要运行的测试函数注册到这个 `delegate` 成员身上. 这个 class 看起来像这样 (译注: 同 p.88):

```

public delegate void Action();
public class testHarness
{
    static private Action theAction;
    static public Action Tester
    { get{ return theAction; }
      set{ theAction = value; } }
}

```



```

static private void reset() { theAction = null; }
static public int count()
{ return theAction != null
    ? theAction.GetInvocationList().Length : 0; }

// ...
}

```

按约定, 某个 class 应该在其 static 构造函数内将自己的“某个”或“数个”成员函数注册到 Tester 身上, 例如:

```

public class testHashtable
{
    public void test0(){ ... }
    public void test1(){ ... }

    static testHashtable()
    {
        testHarness.Tester += new testHarness.Action( test0 );
        testHarness.Tester += new testHarness.Action( test1 );

        // 译注: 以上两处似应修改如下。见 2.12 节 (p.90)
        // testHarness.Tester += new Action( test0 );
        // testHarness.Tester += new Action( test1 );
    }
    // ...
}

```

并在程序入口处调用 testHarness 的 static run().run() 先测试这个 delegate 是否真的指向某函数; 如果是, 就运行这些注册过的函数, 并重置 (reset) delegate:

```

public class EntryPoint
{
    public static void Main(){ testHarness.run(); }
    // ...
}

public class testHarness
{
    public static void run()
    {
        if ( Tester != null )
            { Tester(); reset(); }
    }

    // ...
}

```

然而这样的设计有一丝考虑不周。我们的策略取决于“想注册 test 函数”的每一个 class 的 static 构造函数的调用——这些构造函数必须在 run() 运行之前先被调用。由于 run() 是程序入口 Main() 的第一条语句（见上页），我们没有多少地方可以开展这项工作。因此我们面临两个问题：

1. 我们应该在何处调用各个 class 的 static 构造函数，从而确保这项任务在调用 run() 之前完成（亦即运行 Main() 的第一条语句之前完成）？
2. 到底要做什么工作，才能确保调用相应的 static 构造函数？

要完成以上工作，只有一个地方是绝对保险的，我们得到的保证是 testHarness 的 static 构造函数一定会在“Main() 内的 run() 函数被执行前”先被调用。如果这样，那么 testHarness 的 static 构造函数便是我们完成工作所需的时间点和地点。

到底要做些什么工作呢？首先我们要找出应用程序中的所有型别，因为它们都有可能想要注册测试函数。

接下来我们需要引发该型别的 static 构造函数的执行。这可以通过“为该型别创建一份实体”而办到。为求优化起见，我们先检测该型别是否为预定义之 .NET Framework 的一部分，如果是就不必创建这份实体。

当然，这会耗去一些时间，但因为我们处于测试阶段，所以时间不是太大问题。此外，将测试过程自动化，可以节约劳动时间。

要想找出某个可执行文件（executable）中出现的所有型别，首先必须取回程序相应的所有装配件(s)，例如（译注：请参考 p.350）：

```
AppDomain appdomain = AppDomain.CurrentDomain;  
Assembly [] assemblies = appdomain.GetAssemblies();
```

GetAssemblies() 返回一个由 Assembly class objects 组成的 array，用以表示加载于当前应用程序中的装配件。接下来滤除所有 System 装配件：

```
foreach ( Assembly a in assemblies )  
{  
    if ( isSystemAssembly( a ) )  
        // 译注：见配套源码 chapter8\tester\tester.cs 的具体做法  
        continue;  
}
```


余下的装配件有可能包含“想要注册测试函数”的 classes。因此，对每个装配件，我们调用 `GetTypes()` 取回一个 `Type objects array` (持有装配件中定义的所有型别)。然后遍历这个 `array`，并为其中记录的每个型别创建一份实体：

```
Type [] t = a.GetTypes();

foreach ( Type tt in t )
    if ( tt.isClass )
        Activator.CreateInstance( tt );
```

以上调用 `CreateInstance()` 将触发相应 class 的 `static` 构造函数 (如果有的话)，这些构造函数将依次向 `test.Tester delegate` 中添加东西，这一切都在执行 `test.run()` 之前发生 (译注: `test` 是个 `testHarness object`)。

当然，还有其他设计策略可用。例如令 class 注册其名称，或传入其型别，或以某种方式让自己被 `testHarness` 知晓。但我希望这些 classes 被动些，我希望它们只需知道它们的实体将被神奇地创建出来，而后测试动作会自动被执行。

8.6 Attributes (特征属性)

Attributes 被视为“元声明信息” (*metadeclarative information*)。C# 支持若干预定义的 attributes (又称 *intrinsic attributes*，固有属性、内置属性)。程序员也能定义新的 attribute 型别，并可在运行期通过 *type reflection* (型别反射) 取得或查询这些 attributes。“固有型 attributes”和“用户自定义型 attributes”都是 classes，尽管它们的语法看起来是基于文本 (*text based*)。在学习“自定义 attribute”之前，我们先大致研究一下“固有型 attributes”。

8.6.1 固有型 Attribute: Conditional

Conditional attribute 使我们能够定义一些 class 成员函数，这些函数将根据某字符串是否被定义而决定是否被调用 (然而我们不能将 Conditional attribute 置于数据成员或 *properties* 身上)。attribute 被置于一对方括弧 ([]) 之间，位于它所修饰的成员函数前。例如 `open_debug_output()` 和 `display()` 就附有一个 Conditional attribute:

```

using System.Diagnostics;
public class string_length : IComparer
{
    [Conditional("DEBUG")]
    private void open_debug_output()
    {
        FileStream fout =
            new FileStream("debug.txt", FileMode.Create );
        of = new StreamWriter( fout );
    }

    [Conditional("DEBUG")]
    public void display( string xs, string ys, int ret_val )
    {
        of.WriteLine("inside conditional function display()!");
        of.WriteLine("word #1: {0} : {1} ", xs, xs.Length);
        of.WriteLine("word #2: {0} : {1} ", ys, ys.Length);
        of.WriteLine("return value: {0} ", ret_val);
    }

    // not allowed: conditional data member
    // [Conditional("DEBUG")]
    private StreamWriter of;
}

```

方括弧内的字符串总是与 #define 语句相关。如果要想上述的 Conditional attribute 的值为 true，你必须在程序中写下：

```
#define DEBUG
```

你也可以使用 #undef 取消某个定义，例如：

```
#undef DEBUG
```

由于预处理器 (preprocessor) 命令必须出现在 C# 语句之前，所以代码中不能嵌套使用 #undef 命令。调用上述成员函数并不需要什么条件，虽然它们有可能根本不会被调用。例如：

```

public class string_length : IComparer
{
    public string length() {
        // if DEBUG is defined, this executes
        open_debug_output();
    }
}

```



```

public int Compare( object x, object y )
{
    if (( ! ( x is string )) || ( ! ( y is string )))
        throw new ArgumentException("both must be strings");

    string xs = (string) x, ys = (string) y;
    int ret_val = 1;

    // calculate result in ret_val

    // if DEBUG is defined, this executes
    display( xs, ys, ret_val );

    return ret_val;
}

```

如果 Conditional 内的字符串没有被定义出来，则执行期会忽略代码之中那些 Conditional 成员函数的调用行为。

8.6.2 固有型 Attribute: Serializable

Serializable attribute 表示某个 class 可被序列化 (serialized)。serialization 的意思是让某个 object 超越可执行文件之运行寿命而持续存在 (persisting)，并保持 object 的当时状态，以便日后恢复。我们可以将 objects 序列化至某个存储设备，例如硬盘或远程 (remote) 计算机。此外也可以指明某些数据成员 (fields) 为 NonSerialized (不可序列化)。以下是定义为 Serializable 的 Matrix class，它的 out_stream 成员被标示为 NonSerialized：

```

[Serializable]
class Matrix
{
    [NonSerialized]
    private string out_stream;

    float [,] mat;
}

```

既然将这个 class 标记为 Serializable，那么该如何进行序列化呢？以下样例将某个 class object 保存到磁盘文件中，再读取回来（没有做任何错误检查）。留给我们的工作少得惊人。

首先将 object 序列化 (serialize) 至磁盘, 这会用到定义于 Binary 命名空间中的 BinaryFormatter class。序列化行为 (serialization) 是通过 BinaryFormatter 的 Serialize() 完成的:

```
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;
public void ToDisk()
{
    Stream s = File.Open(out_stream, FileMode.Create);
    // 译注: 原书上一行有误, 已改正。
    BinaryFormatter bfm = new BinaryFormatter();
    bfm.Serialize(s, this);
    s.Close();
}
```

涂写动作的反向操作 (反序列化) —— 将写至磁盘的 objects 恢复原样 —— 是通过 BinaryFormatter 的 Deserialize() 完成的:

```
public void FromDisk( string filename )
{
    Stream s = File.Open(filename, FileMode.Open);
    BinaryFormatter bfm = new BinaryFormatter();
    Matrix m = (Matrix) bfm.Deserialize(s);
    s.Close();
    mat = m.mat;
    out_stream = filename;
    // 译注: out_stream 是 NonSerialized, 所以必须由我们自行恢复
}
```

为了能够对所有型别起作用, Deserialize() 必须返回一个 object object。因此程序中必须将它显式向下转型为 Matrix。

8.6.3 固有型 Attribute: DllImport

DllImport attribute 可让我们调用一个“不是由 .NET 产生的函数”。举个例子, 我们想要实现一个简单的“起/停计时器”Timer class, 用以测量某个例程 (routine) 的运行耗时²³。这里所用的底层例程是 unmanaged (非受控) Win32 API。Timer class 的两个函数声明如下:

²³ 当 Microsoft 从 .NET Framework beta 版本中拿掉了 Counter class 后, *A Programmer's Introduction to C#* (APress, 2000) 的作者 Eric Gunnerson 慷慨地与大家共享了他的 Counter class, 这里的 Timer 就是从 Eric 版本简化来的。


```

public class Timer
{
    private long m_elapsedCount;
    private long m_startCount;
    private string m_context;

    [System.Runtime.InteropServices.DllImport("KERNEL32.dll")]
    private static extern bool
        QueryPerformanceCounter(ref long cnt);

    [System.Runtime.InteropServices.DllImport("KERNEL32.dll")]
    private static extern bool
        QueryPerformanceFrequency(ref long freq);
}

```

这两个函数被声明为 `extern`，因为它们是外部定义的——我们只声明它们，并不为它们提供定义。它们可以是 `private`、`public` 或 `protected`，我们就当它们是普通成员函数似地调用它们，例如：

```

public class Timer
{
    public void start() {
        m_startCount = 0;
        QueryPerformanceCounter(ref m_startCount);
    }

    public void stop() {
        long stopCount = 0;
        QueryPerformanceCounter(ref stopCount);
        m_elapsedCount = (stopCount - m_startCount);
    }

    public override string ToString()
    {
        long freq = 0;
        QueryPerformanceFrequency(ref freq);
        float seconds = (float) m_elapsedCount / (float) freq;
        //译注：原书上一行有误，已改正
        return m_context + " : " +
            seconds.ToString() + " secs.";
    }
    // ...
}

```

在第一章的 WordCount 程序中，我曾经这样使用 Timer class:

```
private void writeWords()
{
    Timer tt = null;24
    if ( m_spy )
    {
        tt = new Timer();
        tt.context = "Time to write file ";
        tt.start();
    }

    // ... the actual writing goes here ...

    if ( m_spy )
    {
        tt.stop();
        m_times.Add( tt.ToString() );
    }
}
```

这里的 `m_spy` 是一个由用户于命令行设置的选项；`m_times` 是个 `ArrayList` object——我们将待测例程的计时字符串（内含所花费的时间）插入其中；`context` 是 `Timer` class 的 `public property`，封装了 `m_context` 字符串。

8.7 实现我们自己的 Attribute class

`Attribute` 实际上是个 `class` 实体。我们可以引入自定义的 `attributes`，前提是代表 `attribute` 的那个 `class` 必须直接或间接继承自 `System.Attribute`。此外，如果它是个非抽象基类（`nonabstract classes`），至少要有一个 `public` 构造函数，而且这个 `class` 的访问级别必须是 `public`。以下是个简单的 `attribute class`（来自 Microsoft 文档），它定义了一个 `Author attribute`，允许程序员将其代码标注上“实现者姓名、代码版本、注解”等所谓的 `metadata attributes`：

²⁴ 尽管从程序的逻辑来看，这里并不需要显式初始化 `tt`，但如果不初始化它，编译器就会认为以下第二个 `if` 语句使用了未初始化的对象。见 1.8 节对编译器所使用的静态流程分析的讨论。


```
// 译注：以下第一行应改为
// [AttributeUsage( AttributeTargets.Method | AttributeTargets.Class,
// [AttributeUsage( AttributeTargets.ClassMembers,
// AllowMultiple = true )]

public class AuthorAttribute: Attribute
{
    public AuthorAttribute( string nm )
    { name = nm; version = 1.0; }

    public string name // 译注：这是对应于 m_name 的一个 property
    {
        get{ return m_name; }
        set{ m_name = value; }
    }

    public double version // 译注：这是对应于 m_version 的一个
property
    {
        get{ return m_version; }
        set{ m_version = value; }
    }

    public string comment // 译注：这是对应于 m_comment 的一个
property
    {
        get{ return m_comment; }
        set{ m_comment = value; }
    }

    private string m_name;
    private string m_comment;
    private double m_version;
}
```

这里的 AttributeUsage attribute:

```
[AttributeUsage( AttributeTargets.ClassMembers,
                AllowMultiple = true )]
```

// 译注：应修改如本页最上之译注说明。另可参考 p.376

规定了 Author attribute 可以出现的地方。ClassMembers 枚举元表明这一 attribute 能够用在 class、struct 或 enum 的任何成员上，也可用在型别自身（译注：新版 .NET Framework 已经修改，只能用于 Class、Method、Enum 等枚举元）。

缺省情况下，一个物体（entity，程序元素）只能关联某个 attribute 的一份实体。但因为代码可能有多位作者，势必需要允许多份 attribute 实体。这就是参数

AllowMultiple 所起的作用。

下面这个 class 同时用到了 Author attribute 和固有的 Serializable attribute:

```
[Author("Anna Lippman", comment = "new hire; first project")]
[Serializable]
class testAttributes
{
    // 译注: 以下是两个 attributes 修饰同一个函数 doit()。
    [Author("Kenny Meyer", version=2.0,
           comment = "added threading support")]

    [Author("Danny Lippman", version=1.1)]
    static public bool doit(){ return true; }

    [Author("Kenny Meyer", version=2.0,
           comment = "extensibility for user")]
    public virtual void display() { /* ... */ }
}
```

你也许会问, 为什么我不写 attribute 的全名:

```
// unnecessary ...
[AuthorAttribute("Anna Lippman")]
```

按规定, attribute 的名称会自动加上后缀 Attribute, 因此如果写下 Author, 编译器会理解为 AuthorAttribute。这是为了少打几个字而设立的方便语法。

如果没有一种适当的机制能够在运行期取回这些 attributes, 那么定制 attributes 就实在没有多大用处。当然啦, 的确存在这样的机制, 例如以下是从 testAttributes 取回的 Author attributes 的输出样例:

```
There are 1 attributes associated with testAttributes
Author Attribute: Anna Lippman :: version 1.00
    new hire; first project
```

```
There are 7 members associated with testAttributes
There are 1 attributes associated with display
Author Attribute: Kenny Meyer :: version 2.00
    extensibility for user
```

```
There are 0 attributes associated with GetHashCode
```



```
There are 0 attributes associated with Equals
There are 0 attributes associated with ToString
There are 2 attributes associated with doit
Author Attribute: Danny Lippman :: version 1.10
Author Attribute: Kenny Meyer :: version 2.00
    added threading support
There are 0 attributes associated with GetType
There are 0 attributes associated with .ctor
```

请注意，同一个成员所附带的多份 attributes 是按“自底而上”的顺序取出，这是基于以下前提：离成员愈近的 attribute 愈老旧。

试看我们能否在以下数小节中弄懂这些概念。

8.7.1 位置 (Positional) 参数与具名 (Named) 参数

attribute 声明式中的位置参数 (positional parameters) 是 attribute 构造函数(s)必要的值。本例中只有作者姓名是位置参数，它在 Author attribute 的每一份实体中都必须出现。

具名参数 (named parameters) 代表 Attribute class 的 public property (或 public 数据成员)。它们在 attribute 实体中可有可无，格式类似“向具名的成员赋予某值”，例如：

```
name = expression
```

本例有两个具名参数：version 和 comment。在 Author attribute 中这两个具名参数可以出现也可以不出现。它们如果出现，必须跟在位置参数之后，且以逗号隔开。

具名参数可以任意顺序列出，只要保证在位置参数之后即可，例如：

```
[Author( "Kenny Meyer", version = 2.0,
        comment = "added threading support" ) ]
```

由于具名参数的排列顺序无关紧要，所以下面的 attribute 完全等价于上面那个：

```
[Author( "Kenny Meyer", comment = "added threading support",
        version = 2.0 ) ]
```

8.7.2 AttributeUsage

我们可以使用 `AttributeUsage` 来确定“用户得以安放 `attribute` 实体”的位置。缺省情况下，`attribute` 可安放于任何位置。

位置的选择定义于 `AttributeTargets` enum 中，包括 `Parameter`（指明 `attribute` 可用以修饰某个成员函数的参数）、`Return`（用以修饰某个成员函数的返回值）、`Constructor`、`Method`、`Property`、`Field`、`Event`、`Delegate` 等等。此外还可以设置为 `Class`、`Struct`、`Enum` 或 `Interface`。ALL 则表示允许 `attribute` 置于任何位置。

我们可以使用 bitwise OR 运算将上述多个枚举元合并使用，藉以指明 `attribute` 得以安放的多个位置：

```
[AttributeUsage( AttributeTargets.Class |  
                 AttributeTargets.Struct )]
```

以上表示 `attribute` 只被允许用于 `class` 或 `struct` 身上。如果以此 `attribute` 修饰某个 `method` 或 `field`，会被编译器视为错误。

`AttributeUsage` 有一个具名参数 `AllowMultiple`。如果你将它设为 `true`，便可在同一位置安放该 `attribute` 的多份实体。

8.8 利用 Reflection 在运行期获取 Attributes

既然我们的 `class` 已经用上了“用户定制型 `attributes`”，自然会想要在程序运行期间访问这些 `attributes`。有若干途径可以办到，例如以 `Type` object 为引数调用 `Attribute` class 的 `static GetCustomAttributes()`：

```
static public void retrieveClass1( object obj )  
{  
    Type tp = obj.GetType();  
  
    Attribute [] attrs =  
        Attribute.GetCustomAttributes( tp );
```

返回的是一个 `Attribute` array，持有各个“定制型 `attribute`”的实体。如果没有定制型 `attribute`，则返回一个空 array。注意，这里并没有拣取“固有型 `attributes`”，例如我们的 `class` 所采用的固有型 `Serializable` `attribute` 就没有被取回来。

另一个办法是使用 `Type` class 的 `GetCustomAttributes()`。这个函数返回一个更一般化的 `object` array:

```
static public void retrieveClass2( object obj )
{
    Type tp = obj.GetType();
    object [] attrs = tp.GetCustomAttributes();
```

上述两个 `GetCustomAttributes()` 都返回“与某型别相关”之所有定制型 attributes。如果想找出其中某一特定型别，例如 `Author`，我们可以使用操作符 `is` 或运行期操作符 `as`:

```
Attribute [] attrs = Attribute.GetCustomAttributes( tp );

foreach( Attribute attr in attrs )
    if ( attr is AuthorAttribute )
    {
        AuthorAttribute auth = (AuthorAttribute) attr;
        Console.WriteLine(
            "Author Attribute: {0} :: version {1:F}",
            auth.name, auth.version );

        if ( auth.comment != null )
            Console.WriteLine( "\t{0}", auth.comment );
    }
```

由于我们只关心 `Author` attribute，所以最好能有简单的办法可以只取回这个 attribute 的实体。是的，我们可以将我们感兴趣的 attribute 的 `Type` object 传给 `GetCustomAttributes()` 而达到目的。这将返回一个 `object` array，其中的元素都是符合要求的实体（如果有的话）。例如：

```
static public void retrieveMembers( object obj )
{
    Type tp = obj.GetType();
    MemberInfo [] mi = tp.GetMembers();

    // prepare to select retrieval type
    Type attrType = Type.GetType( "AuthorAttribute" );

    foreach( MemberInfo m in mi )
    {
        // retrieve only AuthorAttribute instances
        object [] attrs = m.GetCustomAttributes( attrType );
```

```
if ( attrs.Length == 0 )
    continue;

foreach( object o in attrs )
{
    string msg = "Author Attribute: {0} :: version {1:F}";
    AuthorAttribute auth = (AuthorAttribute) o;
    Console.WriteLine(msg, auth.name, auth.version);
    if ( auth.comment != null )
        Console.WriteLine( "\t{0}", auth.comment );
}
}
```

目前尚无办法可以取回任何固有型 attributes。

8.9 中间语言 (Intermediate Language)

C# 程序经过编译后，生成的代码并不能直接运行。编译后生成的并不是具体机器的汇编语言 (assembly language)，而是与机器不相干的所谓中间语言 (intermediate language, IL) ²⁵。

所有 .NET 语言都被编译为 IL。这意味着在 IL 层级，所有语言的代码看起来都很相像。将所有 .NET 语言编译为通用的 (common) 中间语言，好处主要有两点，首先是实现语言之间的互操作性 (interoperability) ——不仅在二进制代码 (binary) 层级，而且在源代码层级，我们不仅可以将不同的 .NET 语言所编写的模块 (modules) 结合在一起，而且能够运用 (或继承) 其他 .NET 语言定义的 classes。

第二个好处是，一旦拥有语言中立的代码，各种工具便可以以这种中立代码为操作对象。这样的工具便能够通过一个公共接口 (common interface) 与各种 .NET 语言协同工作。

为防止误解，我必须申明：并非所有语言都能支持 IL 的全部特性。例如 C# 支持不带正负号 (unsigned) 型别，但 VB.NET (Visual Basic) 不支持它们。C# 的标识符 (identifiers) 有大小写之分，VB.NET 没有。为了提高 (增强) 语言的互操作性，.NET 定义了 CLS (Common Language Specification, 共通语言规范)。CLS 代

²⁵ 中间语言的具体细节，超过本节讨论范围，请参考 *The IL Assembly Language Programmers' Reference* 和 *The MSIL Instruction Set Specification*，位于目录 Program Files\Microsoft.NET\FrameworkSDK\Tool Developers Guide\docs 中。（译注：自从 CLI 提交给 ECMA 标准化组织后，这两份文档已更名，位于 Program Files\Microsoft Visual Studio .NET\FrameworkSDK\Tool Developers Guide\docs\）

表大多数语言共有的基本功能。为确认代码与 CLS 兼容, 你可以把 `CLSCompliant` attribute 设为 `true`:

```
[CLSCompliant( true )]
```

这么一来, 一旦出现 CLS 不兼容代码, 就会触发编译错误²⁶。

8.9.1 检视中间语言

我们已经讨论过, `value` 型别和 `reference` 型别的主要区别在于, `value` 型别将数据直接存储于 `object` 之内, 而 `reference` 型别分为一对 `handle/object`, 其中 `handle` 在本地分配, `object` 则在受控堆 (managed heap) 上分配。为了介绍中间语言, 我们来看一看 `struct value` 型别与 `class reference` 型别各生成了些什么样的代码 (译注: 以下讨论, 读者最好具备 `stack machine` 的基本知识):

```
structDef sd = new structDef(1, 2, 3);  
classDef cd = new classDef( 1, 2, 3);
```

中间语言的 `load` 和 `store` 操作用到了一个 `evaluation stack` (求值栈)。每个成员函数都维护一个 `local stack` (本地栈) —— 一开始进入成员函数时此 `stack` 为空。

`Load` 指令 (`ld*`) 负责将值从内存复制到 `evaluation stack` (求值栈), `Store` 指令 (`st*`) 负责将值从 `stack` 复制回内存。传递给成员函数的各个引数 (`s`) 被压入 (`push`) `evaluation stack` 的顶端, 函数返回值则是从 `evaluation stack` 的顶端弹出 (`pop`)。

每个成员函数都维护一个 `array`, 名为 `.locals`, 负责持有 `local objects`。上述的 `struct value` 型别 `sd`, 以及上述的 `reference` 型别 `cd` 的 `handle` 部分, 都置于这个 `array` 中:

```
.locals {  
    [0] value class structDef sd,  
    [1] class classDef cd,  
    ...  
}
```

²⁶ 请见 *.NET Framework Developer's Guide* 中的 *Cross-Language Interoperability* 这一节中有关于 *Common Language Specification* 以及“如何编写 CLS 兼容代码”的讨论。(译注: 目前位于 <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconcommonlanguagespecification.asp>, 或查阅你的本地文档)

以下是初始化 `struct value` 型别 `df` 的代码序列。动作十分简单：将 `sd` 装入 (load) `stack` 顶端，然后装入三个常量，然后调用构造函数。因为指令不容易阅读，所以我在源代码之间加上注解，以三斜线 (`///`) 标明：

```
/// source line: structDef sd = new structDef( 1, 2, 3 );

/// the ldloca instruction pushes the address of the local
/// object onto the stack

ldloca.s sd

/// the ldc instruction pushes a constant number onto the
/// stack; i4 represents the type — in this case, a 4-byte
/// integer; the last digit represents the literal value
///
/// so the next three instructions push 1, 2, 3 onto the
/// stack; these represent the arguments to the constructor

ldc.i4.1
ldc.i4.2
ldc.i4.3

/// an invocation of the three-argument constructor;
/// it pulls the three arguments and the object to
/// initialize from the stack

call instance void structDef::.ctor(int32,int32,int32)
```

`reference` 型别的初始化要复杂得多，因为它必须在 `heap` 之上分配空间。`newobj` 指令负责在 `heap` 上分配空间，然后调用 `class` 构造函数，并返回一个 `reference`，指向构造出来的这个位于 `heap` 之中的 `object`。返回的 `reference` 将被存储到 `local handle` 中：

```
/// source line: classDef cd = new classDef(1,2,3);

/// load the constant literals on the stack
ldc.i4.1
ldc.i4.2
ldc.i4.3

/// the constructor to be called is specified as part of
```



```
/// the newobj instruction; the parameters accepted by the  
/// constructor are pulled from the stack  
newobj instance void classDef::ctor(int32, int32, int32)  
  
/// store the heap object reference at index 1 of .locals  
stloc.1 (译注: 请注意, newobj 的返回值放在 stack 顶端)
```

这应该使你对中间语言 IL 有了一些认识。探究中间语言, 从而搞清楚各种构件 (constructs) 如何被实现出来, 是一件非常有趣的事。为了帮助你研究, Visual Studio.NET 附带了中间语言反汇编器 (disassembler): ildasm。

不要忘了, 中间语言所表现的是代码存储机制。当程序运行起来时, 并非逐条解释 (interpreted) 这些指令, 事实上在运行之前, IL 会被及时 (Just in time, JIT) 编译为目标机器上的机器码。真正运行的是机器码。

8.9.2 ildasm (IL 反汇编) 工具

有若干工具以中间语言 IL 为工作对象, 其中之一就是“中间语言反汇编器” ildasm²⁷。作为一种学习工具, ildasm 使我们能够反汇编并查看 IL 代码, 并使我们得以观察装配件 (assembly) 相关的 metadata (元数据)。图 8.1 所示的是 ildasm 树状视图 (tree view) 的一部分, 观察对象是系统核心库装配件 mscorlib.dll。请参照图 8.2, 理解其中几何图标的意义。

此处的树状视图 (tree view) 提供了一个阶层视图 (hierarchical view), 将所有型别按命名空间分门别类, 一览无遗。命名空间之下列出了其所定义的所有型别和嵌套命名空间。型别之下列出该型别的所有成员。如果双击一个终端节点 (atomic) 图标, 例如 ICollection 的 property Count 的红三角型图标, 会弹出一个窗口, 提供附加信息。图 8.3 所示的便是双击 Count 后显示的窗口, 从中可以看出 Count 的型别为 int, 只提供一个 nonstatic get 访问器 (accessor)。如果双击某个成员函数, 弹出的窗口中会显示 IL 指令。我把这个留给读者自行练习。

²⁷ ildasm.exe 和其他许多 .NET 工具都位于这个目录下:

%windir%\Microsoft.NET\Framework\v1.0.xxxx, 这里的 xxxx 代表你使用的 .NET Framework 的 build number。 (译注: 事实上 ildasm.exe 目前并不位于这个目录内)

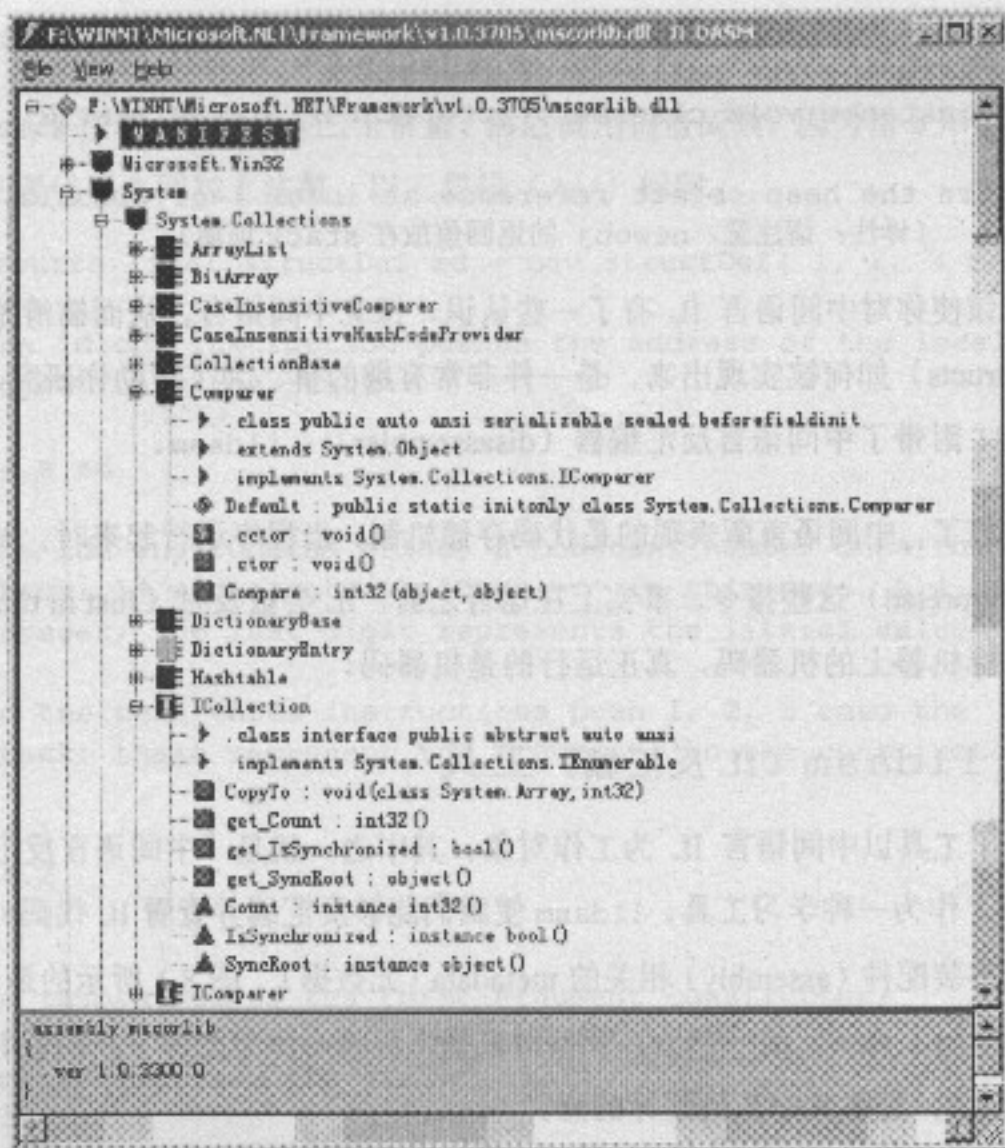


图 8.1 mscorlib.dll 的树状视图 (tree view)

ildasm 可运行缺省 (default) 模式和高级 (advanced) 模式。高级模式能访问更多的装配件 (assembly) 的 metadata 信息。

以下命令以缺省模式启动 ildasm:

```
C:\WIN\Microsoft.NET\Framework\v1.0.2914\ildasm.exe
// 译注: 如果按本书 p.xix “C#环境设置” 的安装提示, 这里便是:
//      D:\VS.NET\FrameworkSDK\Bin\ildasm.exe
```

如果在 ildasm.exe 之后加上 /adv, 便表示要以高级模式启动:

```
C:\WIN\Microsoft.NET\Framework\v1.0.2914\ildasm.exe /adv
```

以上两条命令既可在 **Command Prompt** 窗口中运行, 也可在 **Run** 对话框中运行。

(译注: 建议使用“开始菜单/程序/Microsoft Visual Studio .NET/Visual Studio .NET tools/Visual Studio .NET Command Prompt”来启动 Command Prompt 窗口, 这样就不必输入一长串路径)

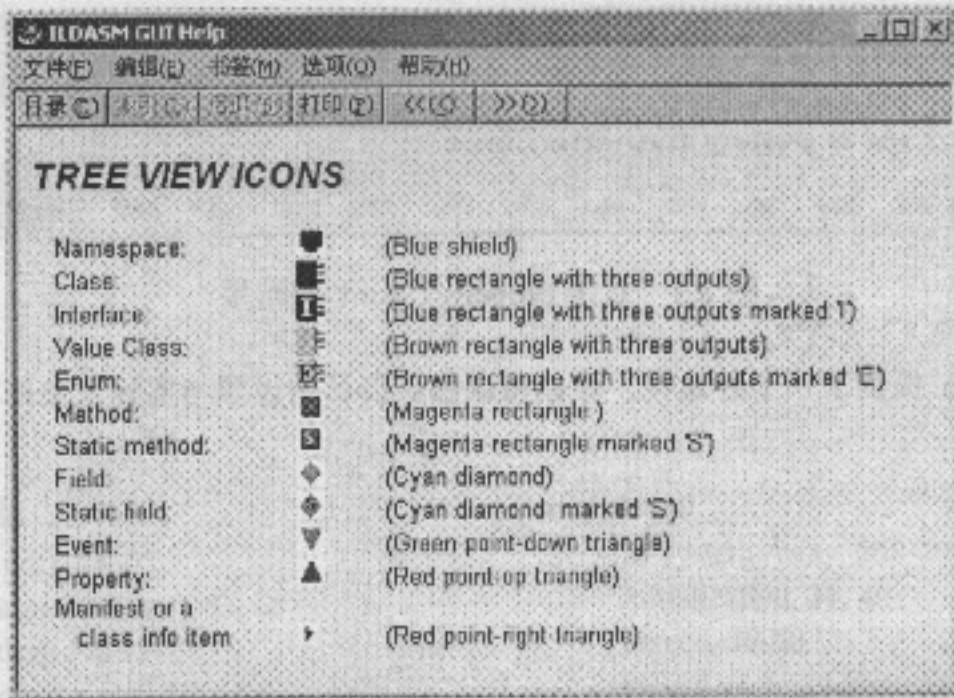


图 8.2 树状视图 (tree view) 的图标说明

我们可以使用 `ildasm` 的 **View** 菜单来设置各种显示属性，例如：

- *Sort by Name*: 将树状视图 (tree view) 中的条目按名称排序。
- *Show Public*: 显示访问级别为 `public` 的条目。
- *Show Private*: 显示访问级别为 `private` 的条目。
- *Show Assembly*: 显示访问级别为 `assembly` 的条目。
- *Show Source Lines*: 在显示 IL 的同时，列出源代码。
- *Show Statistics* (限高级模式): 显示文件的统计信息。
- *Show MetaInfo* (限高级模式): 在反汇编窗口中显示 `metadata` (元数据)。

`ildasm` 是一个重要的学习工具。可以说，它给了我们钻进魔术师帐篷内一窥堂奥的机会。当我学习某项 .NET 语言特性并希望证实自己的理解时，我常钻研中间语言。

举个例子，当我为了理解“将 `reference` 型别或 `value` 型别赋值给 `object` 型别时所发生的不同行为”，以及为了理解“通过 `new` 表达式来初始化 `reference` 型别和 `value` 型别时所发生的不同行为”时，看一眼中间语言的实现，就像尝一口蛋糕上的糖衣那般令我畅快。

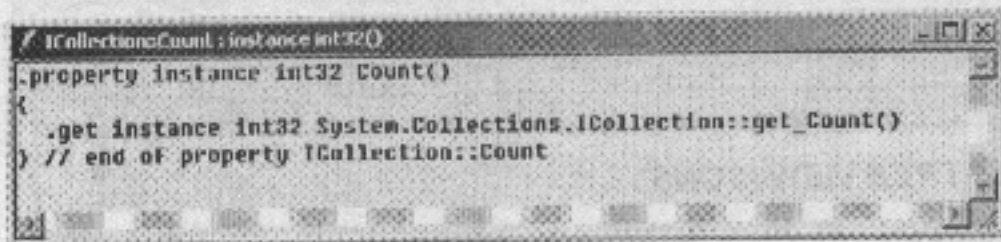


图 8.3 双击 property Count 的结果

ildasm 提供了一个好途径，让你得以确认自己的设想与实际发生的相同！

索引

Symbols

! (logical not), 53, 55
 != (inequality operator), 53, 55
 % (remainder), 52, 55
 %= (compound remainder), 54
 & (bitwise and), 53, 55
 && (logical and), 53, 55
 --(decrement), 52, 54
 -(subtraction), 52, 55
 * (multiplication), 52, 55
 *= (compound multiply), 54
 + (addition), 52, 55
 ++ (increment), 52, 54
 += (compound add), 54
 . (scope), 4
 / (division), 52, 55
 // (comment), 2
 /= (compound divide), 54
 <= (less than equal), 53, 55
 = (assignment), 54, 55
 -= (compound minus), 54
 == (equality operator), 53, 55
 > (greater than), 53, 55
 >= (greater than equal), 53, 55
 ?: (conditional operator), 53
 @ (prefix), 17, 48
 | (bitwise or), 53, 55
 || (logical or), 53, 55

A

arithmetic operators,
 see expressions

array, 29-30

- as System.Array, 200-203
- initialization, 31, 40
- jagged, 39-40
- jagged vs. multidimensional, 39
- multidimensional, 29

ArrayList, see System.Collections

as operator, 38

ASP.NET, 315-348

Controls

- Calendar, 344
- CheckBoxList, 338-340
- DataGrid, 321
- Document, 318-319
- DropDownList, 335, 336
- HyperLink, 321
- Image, 345
- ImageButton, 329
- Label, 319
- ListBox, 329
- programming, 345-348
- RadioButtonList, 337
- TextBox, 328

database connection, 325-326

event handling, 323

- change events, 323

- click events, 323

HttpApplication class, 333

Page

- AutoPostBack, 323

- event life cycle, 323-325

- IsPostBack, 324

- linking to, 321
- Page_Load(), 324
- post back, 323
- round-trip, 323
- Project
 - adding a page, 320
 - Design view, 317
 - FlowLayout, 319
 - GridLayout, 319
 - HTML view, 317
 - opening a project, 316-319
 - Properties window, 316, 318
 - Solution Explorer, 318
 - Toolbox window, 316, 319
- state management, 326-328
 - Application Object, 333-334
 - instance members, 331-332
 - Session Object, 332-333
- Validator controls, 334, 340-343
 - CompareValidator, 340
 - CustomValidator, 341
 - RangeValidator, 340
 - RegularExpressionValidator, 341
 - RequiredFieldValidator, 340
 - ValidationSummary, 341
- assemblies, 349-353, 366
 - AppDomain, 350
 - Assembly, 350
- Attributes, 367-378
 - AttributeUsage, 373, 376
 - custom Attributes, 372-376
 - GetCustomAttributes(), 377
 - intrinsic, 367-372
 - CLSCompliant, 379
 - Conditional, 367-369
 - DllImport, 370-372
 - NonSerialized, 369-370
 - Serializable, 369-370
 - multiple Attributes, 375
 - named parameters, 375
 - position parameters, 375
 - runtime discovery, 376-378
 - See also reflection

B

- base keyword
 - see class constructor
 - see object-oriented
- bool, see types
- boxing 36-37
 - See also unboxing
- break, see statement

C

- capacity, 34
- cast, see conversion
- catch, see exception handling
- class, 59-116
 - access level, 62
 - constructor, 73-76
 - access level, 76
 - base, 143
 - base vs. this, 144
 - new expression, 74
 - static, 80, 81, 366
 - this keyword, 75
 - when to use, 73
 - conversion operators, 110-112, 196
 - explicit, 111
 - implicit, 110
 - copy semantics
 - see System.ICloneable
 - data member, 66-67
 - access level, 67, 68
 - const, 81-83
 - readonly, 81-83
 - readonly vs. const, 83
 - declaration order, 62
 - destructor, 113
 - deterministic finalization
 - see System.IDisposable
 - domain abstraction, 2
 - independent abstraction, 59-63
 - indexer, 69-72
 - initialization, 72-76
 - default, 72
 - explicit, 72
 - three strategies, 76
 - See also constructor

- member function, 60-62
 - access level, 61
 - arguments vs. parameters, 93
 - overload resolution, 100-103
 - overloading, 99-103
 - parameter list, 61, 92-99
 - pass by reference
 - out, 97
 - ref, 96
 - pass by value, 94-96
 - return type, 61
 - signature, 61
 - variable-length parameter list, 103-106
- operator overloading, 107-110
 - binary operators, 110
 - invocation, 107
 - compound assignment, 108
 - unary operators, 109
- properties, 67-69
 - get read access, 69
 - set write access, 69
- public interface, 59
- reference type, 29
- sealing, 153
- static member, 79-81
 - access usage, 80
 - static vs. instance, 79
- this
 - in constructor, 75
 - as reference, 76-78
- See also object-oriented
- command-line arguments, 11
- comment, 2
- Common Language
 - Runtime (CLR), 349
 - Specification (CLS), 378
- const, see class data member
- continue, see statement
- conversion
 - cast, 37
 - derived to base class, 127
 - explicit, 37, 42, 49

- foreach vs. for loop, 39
- implicit, 101
- operators, see class
- overload resolution, 102
- smaller than int, 49
- standard implicit, 101
- conversion operators, see class
- copy constructor
 - see System.ICloneable

D

- data member, see class
- database, see System.Data
- decimal, see types
- declaration space, see scope
- deep copy, 28, 186
- definite assignment, 25
- delegate type, 86-92, 364-367
 - declaration, 87
 - Delegate interface, 92
 - invocation, 90
 - object reference, 91
 - reference type, 88
 - set to instance method, 88
 - set to multiple methods, 90
 - set to static method, 88
 - single vs. multiple methods, 87
- destructor, see class
- deterministic finalization
 - see System.IDisposable
- dictionary, see
 - System.Collections.Hashtable
- double, see types
- do-while, see statement
- dynamic binding, 120, 128

E

- entry point, see Main()
- enum type, 83-86
 - enumerator values, 84
 - relation to int, 84
 - underlying type, 85
- environment variables
 - see System.Environment

exception handling, 44-47
 catch clause, 44-45
 catch resolution, 157
 defining our own exceptions, 156
 Exception class hierarchy, 45, 154-158
 exception safety, 189
 finally clause, 47
 handling the exception, 46
 inner exception, 155
 non-resumption, 46
 throw expression, 44, 45
 try block, 45

expressions
 arithmetic, 51
 checked, 52
 conditional, 51
 exceptions, 52
 integral promotion, 50
 overflow, 172
 relational, 51
 unchecked, 52

F

Fantasia 2000, 228
 files, see `System.IO`
 finally, see exception handling
 float, see types
 for, see statement
 foreach, see statement
 Forms Designer, see Windows
 freeing unmanaged resources
 see `System.IDisposable`
 fully qualified name, 5
 See also namespaces
 function overloading
 see class member function
 function, see class

G

garbage collection, 32-33
 resource cleanup,
 see `System.IDisposable`
 See also class destructor

H

HashTable, see
 `System.Collections.Hashtable`

I

identifier, see name
 if, see statement
 ildasm tool, 381-383
 indexer, see class
 information hiding, see class 68
 inheritance
 see interface inheritance
 see object-oriented
 initialization, 24-25, 27
 array, 31
 input/output, see `System.IO`
 int, see types
 integral promotion, 50
 interface inheritance, 159-197
 access existing interface, 163-166
 allowed member types, 167
 contrast to abstract base class, 159
 defining an interface, 166-180
 determining the exceptions, 173-174
 explicit interface member, 178-180
 implement all members, 168
 implement interface, 160-163
 inheritance and visibility, 180-185
 inheriting from an interface, 168-174
 integration with framework, 174
 master copy semantics, 185-187
 master finalize semantics, 187-190
 resolving ambiguity, 184
 simplest definition, 167
 virtual methods, 181-183
 See also object-oriented
 intermediate language, 378-383

interoperability, 370
 is operator, 38
 iterator, 175
 See also `System.IEnumerator`

K

keywords, 47

L

lifetime, 36

See also garbage collection

literals, see types

local object, see scope

lock, 241, See also System.Threading

long, see types

M

Main(), 2, 10-11

command-line arguments, 11

program entry point, 2

program exit status, 11

return value, 11

managed heap, 28, 32

See also garbage collection

map, see

System.Collections.Hashtable

member function, see class

metadata, see reflection

N

name

fully qualified, 5, 8

inheritance resolution, 148

local resolution, 26

name collision, 6

naming rules, 6, 48

resolution based on visibility, 150

visibility of name, 6

namespaces, 3, 6-10

definition, 8

name collision, 7

naming conventions, 10

visibility rules, 8

native method call, 370

See also Attributes,DllImport

new expression, 28, 30-32, 36

new specifier, 149, 181

numeric types, see types

O

object, 35-38

heterogeneous parameter list, 105

universal assignment, 130

See also System.Object

See also universal type system

object-oriented, 117-158

abstract base class, 118, 132-139

abstract keyword, 135

abstract virtual function, 134

definition, 133

hybrid design, 138-139

abstract derived class, 135

composition, 194

copy semantics

see System.ICloneable

derived class, 118, 143-152

abstract, 143

access hidden member, 135

base constructor, 144

constructor, 143

member access, 147-152

member resolution, 148

new specifier, 149

virtual function, 145-146

deterministic finalization

see System.IDisposable

hybrid base class, 141-142

implementation inheritance, 142

implicit conversion, 127

inheritance hierarchy, 118, 126

member resolution, 150

methods hidden by signature, 184

polymorphism, 118-120

refactoring, 136

single inheritance, 140-141

static member, 137-138

type/subtype, 126

virtual function, 128, 134

abstract, 134-135

covariant return type, 145

override, 145

static invocation, 153

virtual indexers, 136

virtual properties, 136

See also interface inheritance

See also class

operator
 precedence, 18, 54-55
 overloading, see class

output
 formatted output, 20
 See also Console
 See also System.IO

overloading
 functions, see class
 operators, see class
 subscript, see class indexer

P

params keyword, see
 class member functions

pass by reference, see
 class member functions

pass by value, see
 class member functions

pointer to function, see delegate
 polymorphism, see object-oriented
 program entry point, see Main()
 properties, see class

R

readonly, see class data member
 reference counting, 32
 reference types, 28-29, 36

 pass by value, 94-96
 See also array
 See also class
 See also delegate

reflection, 353-367
 See also System.Reflection 353
 See also System.Type

regular expressions, see
 System.Text.RegularExpressions

relational operators, see expressions
 return, see statement

runtime object creation
 see System.Activator

runtime type discovery
 see System.Reflection

runtime type query

as operator, 38
 is operator, 38

S

scope
 global, 6
 local, 24-27
 lifetime, 36
 order dependent, 26
 uninitialized, 24

scope operator (.), 4

sealing, see class

serialization, 370

shallow copy, 28, 32, 36, 96, 186

SOAP, 281

sockets, see System.Net.Sockets

Sqrt(), see System.Math

statement, 55-57
 break, 20
 continue, 19
 do-while, 25, 56
 for, 22, 56
 foreach, 13, 39, 56
 if, 12, 56
 if-else, 13
 return, 12-13
 switch, 14-16, 57
 while, 56

statement block, 12

static binding, 119, 128

string, see types

struct, 113-116
 implicit default constructor, 114
 initialization, 115
 new expression, 114
 non-garbage collected, 114
 performance, 116
 value type, 29

switch, see statement

System, 199-281

 Activator, 363

 CreateInstance(), 363, 367

 Array, 200-203

 BinarySearch(), 203

- Clear(), 202
- Copy(), 202
- CopyTo(), 201
- GetLength(), 201
- IndexOf(), 202, 203
- Sort(), 206
- Attribute, see Attributes
- Console, 2
 - ReadLine(), 24
 - Write(), 2
 - WriteLine(), 2
- DateTime, 208, 226-227
- Drawing
 - Bitmap, 312
 - Graphics, 314
- Environment, 204-207
 - GetLogicalDrives(), 209
 - OSVersion, 205
- ICloneable, 185-287, 195
- IComparable, 160-163, 164-166
- IDisposable, 187-190
- IEnumerable, 174
- IEnumerator, 175-178
- Math, 225
- namespace introduction, 3
- OperatingSystem, 205
- PlatformID, 205
- TimeSpan, 208
- Windows.Forms, see Windows
- System.Collections
 - ArrayList, 33-35, 160
 - BitArray, 190-197
 - Hashtable, 41-43
 - ICollection, 194
 - IDictionary, 43
 - Queue, 223
 - Stack, 221-223
- System.Data, 249-259
 - connection string, 252, 253, 325
 - DataRelation, 251, 257-258
 - DataSet, 253
 - DataSet/XML interop, 260
 - DataTable, 254-257
 - Select(), 258-259
 - OleDbCommand, 253
 - OleDbConnection, 253
 - OleDbDataAdapter, 253
 - selection string, 325
 - SqlClient, 325-326
 - SqlCommand, 326
 - SqlConnection, 325
 - SqlDataAdapter, 326
- System.Diagnostics
 - Process, 207-208
 - GetCurrentProcess(), 207
 - GetProcesses(), 207
 - TraceListener, 223-225
- System.IO, 17-19, 209-221
 - Directory, 209, 210, 212-215
 - Exists(), 210
 - GetDirectories(), 213
 - DirectoryInfo, 209, 212-215
 - CreateFile(), 213
 - CreateSubdirectory(), 213
 - GetFiles(), 213
 - File, 44, 210, 215-221
 - Exists(), 210
 - OpenRead(), 218
 - OpenWrite(), 218
 - file read and write, 216-221
 - FileAccess, 219
 - FileInfo, 209, 215-221
 - AppendText(), 217
 - CreateText(), 217
 - OpenText(), 217
 - FileMode, 218
 - FileShare, 219
 - Path, 209, 210-212
 - ChangeExtension(), 211
 - DirectorySeparatorChar, 214
 - GetExtension(), 211
 - Stream, 218-221, 244
 - Read(), 218
 - Seek, 219-221
 - Write(), 218
 - StreamReader, 17-18, 216-217
 - StreamWriter, 17-18, 216-217
 - See also System.Console

- System.Net, 241-249
 - HttpRequest, 243
 - Sockets, 245-249
 - NetworkStream, 245, 248
 - Socket, 247
 - Connected, 247
 - Receive(), 247
 - Send(), 248
 - TcpClient, 245, 248-249
 - GetStream(), 248
 - TcpListener, 245, 246-248
 - AcceptSocket(), 247
 - Start(), 246
 - Uri, 242
 - UriBuilder, 243
 - WebRequest, 243
 - Create(), 243
 - GetResponse(), 244
 - WebResponse, 244
- System.Object, 130-132
 - Equals(), 130, 131, 147
 - GetType(), 131
 - implicit base class, 131
 - overriding methods, 146-147
 - ToString(), 130, 131, 146
- System.Reflection
 - Assembly, 350
 - BindingFlags enum, 358-362
 - ConstructorInfo, 354, 361
 - EventInfo, 354
 - FieldInfo, 354, 359, 360
 - Invoke(), 362
 - MemberInfo, 354
 - MethodInfo, 351, 354, 362-364
 - ParameterInfo, 354, 357
 - PropertyInfo, 354, 358, 359
 - runtime invocation, 362-364
- System.Runtime.Serialization.Formatters.Binary.BinaryFormatter, 370
- System.Text
 - RegularExpressions, 228-235
 - expression syntax, 228-230
 - Group, 233
 - Regex, 231-234
 - Split(), 233
 - Replace(), 234
 - StringBuilder, 23, 147, 195
- System.Threading, 235-241
 - Monitor, 240-241
 - Enter(), 240-241
 - Exit(), 240-241
 - TryEnter(), 241
 - Thread, 236-239
 - Abort(), 239
 - Join(), 239
 - Resume(), 238
 - Sleep(), 237
 - Start(), 238
 - Suspend(), 238
 - ThreadPool, 241
 - ThreadStart class, 238
- System.Type, 131, 351
 - access to Info classes, 355
 - gateway to reflection, 354-356
 - GetType(), 351
- System.Xml, 259-281
 - interop with DataSet, 259
 - Schema, 281
 - Serialization, 281
 - whitespace, 268
 - XmlDocument, 260-263, 272-277
 - XmlNode hierarchy, 274-276
 - XmlNodeType, 268-270
 - XmlSerializer, 263
 - XmlTextReader, 265-272
 - XmlTextWriter, 263
 - XPath, 279-281
 - XPathNavigator, 279
 - MoveToFirstChild(), 280
 - MoveToNext(), 280
 - Select(), 280
 - SelectChildren(), 280
 - XPathDocument, 279
 - Xsl, 277-279
 - Load(), 277
 - Transform(), 277
 - XslTransform, 277

T

throw, see exception handling
 ToString(), 36, 146
 try block, see exception handling
 type cast, see conversion
 types
 alias for System, 51, 199-200
 bool, 14
 byte, 35, 49
 char, 50
 conversion, 37, 49
 decimal, 50, 172
 double, 50
 float, 50
 int, 49
 long, 49
 numeric literals, 49
 numeric types, 49-51
 predefined, 5
 promotion, 50
 reference, 28, 36
 run-time query, 38
 string, 21-23
 Equals(), 14
 immutable, 23
 Split(), 21
 unified type system, 35-38
 unsigned, 49
 value, 28, 36
 verbatim string literal, 17
 void, 11

U

unboxing, 37-38
 unified type system, see types
 unmanaged method call, 370
 unmanaged resources
 see System.IDisposable
 unsigned, see types
 using
 directive, 2, 4, 8
 directive alias, 9
 directive vs. qualified name, 5
 statement, 190

V

value types, 28-29, 36
 See also enum
 See also struct
 variable-length parameter list,
 see params keyword
 see class member function
 virtual, see object-oriented
 visibility
 see name
 see namespaces
 Visual Studio
 add new item, 65
 build program, 65
 Class View window, 65
 compiler errors, 65
 execute program, 65
 opening a project, 63-65
 Properties window, 286, 287, 290
 renaming file, 65
 Solution Explorer, 64
 Toolbox window, 288
 void, as return type, 11

W

Web Forms designer, see ASP.NET
 Win32 API, 370
 See also Attributes,DllImport
 Windows, 283-314
 Bitmap class, 312
 Controls
 Button, 293-294, 304
 CheckBox, 306
 ContextMenu, 307
 DataGrid, 308-310
 FileDialog, 302-303
 Labels, 296-297
 ListBox, 299-302
 Menus, 306-307
 PictureBox, 310-312
 RadioButton, 305
 TextBox, 292, 292-293
 event handlers, 288-295
 Click, 293
 inspecting events, 295-296

C# Primer

A Practical Approach

Stanley B. Lippman

以他举世闻名的“primer”风格，畅销书作者 Stan Lippman 现为您呈上一份不容错过的C#指南。《C# Primer》是一本内容详实、实例丰富的入门读物，全面介绍了这门新的面向对象编程语言。

C# 是 Microsoft 新一代 .NET 平台的基石。它继承了 Java(tm) 和 C++ 的诸多特性，C# 是为创建高性能 Windows 与 Web 应用程序（及组件）——无论是基于 XML 的 Web 服务 (Web Services)，还是中间层业务对象、系统级应用等——而诞生的高级语言。

本书特色：

- 涵括诸如命名空间 (namespace)、异常处理、统一型别系统 (unified type system) 等等基础知识。
- 细致讲解 class (类) 继承与 interface (接口) 继承，并配以详尽的实例，还讨论了二者如何取舍。
- 大范围巡视【or 涉猎】.NET 类库，包括ADO.NET入门、建立数据库连接、正则表达式、多线程、网络套接口编程 (sockets)、XML 编程 (利用 firehose 以及 DOM 解析模式)，以及 XSLT 和 XPATH 等等。
- 详细讨论 ASP.NET Web Form 设计器，涉及页面生命周期 (page life cycle) 与缓存 (caching) 等内容，并提供大量样例。
- 介绍 .NET 共通语言运行层 (Common Language Runtime — CLR)。

学习掌握 C# 不仅可以增强您的 Web 编程功力，还能提升您的生产效率。C# Primer 为这一切提供了坚实的基础。

作者 Stanley B. Lippman 现为 Microsoft Visual C++ 开发组的系统架构师 (Architect)。在此之前，他曾是美国喷气推进实验所 (JPL) 的特别顾问。Stan 在 Bell 实验室呆了12年多，其间他与 Bjarne Stroustrup 共事，一同开发最初的 C++ 编译器，并参与 Foundation 研究项目。离开 Bell 实验室之后，他在迪斯尼电影动画公司 (Disney Feature Animation) 任职，先是首席软件工程师，然后是电影《幻想曲2000》的软件技术指导 (TD)。Stan 撰写了数本书籍：《C++ Primer》、《Essential C++》、《Inside the C++ Object Model》(均由 Addison-Wesley 出版)。他还编辑了《C++ Gems (SIGS Books)》一书。

译者 侯捷，IT教育工作者，也是畅销书《深入浅出 MFC》(华中科技大学出版社，2001) 和《STL 源码剖析》(华中科技大学出版社，2002) 的作者。通过他的演讲、研讨会、著作和顾问工作，帮助 IT 技术人员和在校学生学习计算机与编程技术。

个人网站：<http://www.jjhou.com>(繁体)；<http://jjhou.csdn.net> (简体)

译者 陈硕，北京师范大学电子信息与科学技术系2000级本科生，酷爱读书，常写程序，自得其乐。

个人网站：<http://www.chenshuo.com>

ISBN 7-5609-3006-9



9 787560 930060 >

定价：45.00元

