TURING 图灵程序设计丛书

Apress®

Accelerated C# 2010

C# 4.0捷径教程

[美] Trey Nash 著
刘新军 译

- 好评如潮的C# 4.0实战图书
- 汲取.NET技术精髓的捷径
- 专章讲述习惯用法与设计模式

人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

Accelerated C# 2010

C# 4.0捷径教程

人民邮电出版社
北京

图书在版编目（CIP）数据

C# 4.0捷径教程 / （美）纳什（Nash, T.）著 ; 刘新军译. -- 北京 : 人民邮电出版社, 2011.1
（图灵程序设计丛书）
书名原文: Accelerated C# 2010
ISBN 978-7-115-24342-3

Ⅰ. ①C… Ⅱ. ①纳… ②刘… Ⅲ. ①C语言－程序设计－教材 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2010)第223096号

内 容 提 要

本书是经典教程的全面升级，通过许多精彩的示例介绍了C#语言的每个新特性，深入浅出地讲解了C#语言的核心概念，以及如何聪明地应用C#的习惯用法和面向对象的设计模式来挖掘C#和CLR的能力。这一版还介绍了C# 4.0中新加入的动态类型，它简化了与包括COM Automation对象在内的动态.NET语言的集成。联合使用动态类型和ExpandoObject这样的DLR类型，你可以在C#里创建并实现真正的动态类型，本书所探讨的技术也适用于任何针对.NET运行时的语言。

本书适合有一定编程经验的程序员阅读。

图灵程序设计丛书

C# 4.0 捷径教程

◆ 著　[美] Trey Nash
译　刘新军
责任编辑　傅志红
执行编辑　李　瑛

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京艺辉印刷有限公司印刷

◆ 开本：800×1000　1/16
印张：32
字数：902千字　2011年1月第1版
印数：1－3 000册　2011年1月北京第1次印刷

著作权合同登记号　图字：01-2010-1731号

ISBN 978-7-115-24342-3

定价：79.00元

读者服务热线：(010)51095186　印装质量热线：(010)67129223
反盗版热线：(010)67171154

版 权 声 明

致　　谢

写书是一个漫长而艰辛的过程，在这个过程中我得到了朋友和家人的巨大支持，我为之深深感动。如果没有他们的支持，这个过程会更加困难，而且不会取得这样的成效。

我想特别感谢下面这些人，他们对本书的前两版做出了巨大贡献，他们是（排名不分先后）：David Weller、Stephen Toub、Rex Jaeschke、Vladimir Levin、Jerry Maresca、Chris Pels、Christopher T. McNabb、Brad Wilson、Peter Partch、Paul Stubbs、Rufus Littlefield、Tomas Restrepo、John Lambert、Joan Murray、Sheri Cain、Jessica D'Amico、Karen Gettman、Jim Huddleston、Richard Dal Porto、Gary Cornell、Brad Abrams、Ellie Fountain、Nicole Abramowitz，还有 Apress 的全体员工以及 Shelley Nash、Michael Pulk、Shawn Wildermuth、Sofia Marchant、Jim Compton、Dominic Shakeshaft、Wes Dyer、Kelly Winquist 和 Laura Cheu。

在第三版的写作过程中，我得到了以下这些人的帮助和支持，他们是（排名不分先后）：Jonathan Hassell、Mary Tobin、Damien Foggon、Maite Cervera。

如果我还遗漏了谁，请原谅我的失误，这绝不是故意的。如果没有你们的支持，我绝对不可能完成本书。

谢谢你们大家！

前　言

对熟悉其他面向对象语言的人来说，Visual C# .NET（C#）学习起来相对容易。熟悉 Visual Basic 6.0 的人想学一门面向对象语言，也会发现 C#很容易上手。然而，尽管 C#和.NET 框架为创建简单应用提供了一条捷径，但为了开发复杂、健壮和容错的 C#应用，你还是需要掌握很多的信息并理解怎样正确地使用它们。本书将教给你需要掌握的知识，并解释如何最好地运用这些知识来快速掌握真正的 C#专业技能。

学会习惯用法和设计模式对培养和应用专业技能有不可估量的作用，本书将展示怎样使用它们来创建高效、健壮、容错和异常安全（exception-safe）的应用程序。虽然 Java 和 C++程序员对于其中的许多模式都比较熟悉，但有一些是.NET 和公共语言运行库（CLR）独有的。本书后面的章节会展示如何应用这些必不可少的习惯用法和设计模式，将 C#应用程序与.NET 运行库无缝整合起来，重点将放在 C# 3.0 的新功能上。

设计模式记录的是许多程序员在应用程序设计中反复采用的最佳实践。事实上，.NET 框架本身就实现了许多众所周知的设计模式。同样，在过去的.NET 框架的三个版本和 C#的两个版本中，许多新的习惯用法和最佳实践也已经广为人知，你会看到本书对这些实践的详细描述。另外，值得注意的是，重要的技术工具库也在不断革新。

随着C# 3.0 的到来，可以使用lambda表达式、扩展方法和语言集成查询（Language Integrated Query，LINQ）方便地进行函数式编程。lambda表达式可以方便地在某个点声明和实例化函数委托（function delegate）。另外，有了lambda表达式，创建functional就是小菜一碟。functional是以函数作为参数并返回另一个函数的函数。即使你之前可以在C#里面实现函数式编程（虽然还是有点困难），但C# 3.0 里面的新语言特性提供了一个新的环境，在这里函数式编程和典型的命令式编程可以和谐共存。LINQ 允许使用这种语言的语法来表示数据查询操作（这本质上也是functional）。一旦知道了LINQ的工作原理，你就会意识到你能做的远远不止简单的数据查询，还可以用它来实现复杂的函数式编程。

.NET 和 CLR 提供了一个独特和稳定的跨平台执行环境。C#只是针对这一有效运行时的语言之一，但是你会发现本书探讨的技术也适用于任何针对.NET 运行时的语言。

对于那些有丰富 C++经验，熟悉 C++规范形式（canonical form）、异常安全、资源获得即初始化（Resource Acquisition Is Initialization，RAII）、const 正确性（const correctness）等概念的读者，本书说明了如何把这些概念应用于 C#；对于那些有多年技术积累的 Java 或 Visual Basic 程序员，本书也会探讨如何把这些技术有效地应用于 C#。

总之，要成为一名C#专家，并不需要花几年的功夫去摸着石头过河，只需要学习正确的知识并以正确的方式来使用它们。这正是我写作本书的动机。

关于本书

本书假设你已经具备了一些面向对象编程语言的基本知识，比如用过C++、Java或Visual Basic .NET。因为C#语法起源于C++和Java，所以我不会花很多时间来介绍C#语法，重点讨论的是与C++或Java有显著区别的地方。如果你已经对C#有所了解，那么可以略读甚至跳过第1~3章。

第1章大致给出了一个简单的C#应用，并描述了C#编程环境与C++编程环境之间的基本区别。

第2章是第1章的具体展开，快速说明了C#应用程序运行的托管环境。接着介绍了程序集(assembly)，它是C#代码文件编译成的基本构建块。还讲解了元数据是如何使程序集成为自我描述的。

第3章深入讲解了C#的语法。这里介绍了CLR中的两种基本类型：值类型和引用类型。还介绍了命名空间，以及如何用它来从逻辑上划分应用中的类型和功能。

第4~13章深入介绍了如何在C#程序和设计里运用习惯用法、设计模式和最佳实践。这本书尽量以逻辑顺序来安排这些章节，但还是会在其中某一章不可避免地引用后面一章涉及的技术或主题。

第4章详细讲解了如何在C#里定义类型。这一章讲述了关于CLR里值类型和引用类型的更多知识，还简单涉及了CLR和C#对本地接口的支持。你会发现C#里类型继承的工作原理，也会看到每个对象如何派生自 `System.Object`。这一章还包括了相当多关于托管环境以及如何在其中定义有用类型所必需的信息。这章介绍了其中很多的主题，并将在随后的几章中更详细地讨论它们。

第5章详细介绍了接口及其在C#语言中扮演的角色。接口定义了类型可以选择实现的功能性契约。你可掌握类型实现接口的各种方法，以及接口方法被调用的时候，运行时如何选择调用哪个方法。

第6章详细介绍了将内建的C#操作符应用于自定义的类型时如何赋予其定制的功能。你将了解如何可靠地重载操作符，因为不是所有的为CLR编译代码的托管语言都能使用重载的操作符。

第7章展示了C#和CLR的异常处理功能。虽然与C++的语法相似，但创建异常安全和异常中立的代码还是颇有难度的，甚至比在本地C++中创建异常安全代码还要难一点。你会看到创建容错的、异常安全的代码根本不需要 `try`、`catch` 或 `finally` 指令。这一章还描述了.NET 2.0运行时中新增的一些功能，允许你创建比用.NET 1.1创建的代码容错性更好的代码。

第8章描述了字符串如何作为CLR的头等（first-class）类型及如何在C#中有效地使用它们。这一章的大部分篇幅集中在.NET框架中各种类型的字符串格式化功能，以及如何让自己定义的类型通过实现 `IFormattable` 接口来具有类似的行为。另外，还介绍了.NET框架的全球化功能以及如何为.NET框架目前还不知道的文化和地区创建定制的 `CultureInfo`。

第9章介绍了C#中的各种数组和集合类型。你可以创建两种多维数组，可以使用内置的集合类，也可以创建自定义集合类。你会看到如何使用C# 2.0引入的新迭代器语法定义前向、反向和双向迭代器，这样你自己的集合类型就能与 `foreach` 语句配合工作了。

第10章展示C#里的回调机制。历史上，所有可行的框架都提供了某种机制来实现回调。C#更进了一步，把回调包装到了称为委托的可调用对象中。C# 2.0提供了一种简短的语法来创建委托，这就是*匿名方法*。匿名方法与函数式编程里的 `lambda` 函数类似。另外，这一章还会讲到如何基于委托提供发布/订阅事件通知机制，以便实现事件来源与事件处理代码的分离。

第11章介绍C# 2.0和CLR中加入的也许是最激动人心的特性。那些熟悉C++模板的读者会发现泛型似曾相识，虽然二者存在一些本质的区别。使用泛型可以提供一个功能的壳，可以在运行时在这个壳内定义更具体的类型。泛型用于容器类型的时候最有用，它与之前.NET版本里的容器相比效率更高。从C# 4.0开始泛型应用由于协变性和反变性的支持而变得更加直观。在创造直观的类型识别时，从一种泛型向另一种泛型的过渡现在已成为可能，这就减少了此前所需的转换方法上的混乱。

第 12 章介绍在 C#托管虚拟执行环境中创建多线程应用需要做的工作。如果你熟悉本地 Win32 环境的线程，就会注意到二者的显著区别。此外，托管环境提供了更多的基础设施使多线程更加容易使用。可以看到委托如何通过“我欠你（I Owe You，IOU）”模式提供一个优秀的网关通往进程的线程池。可以说，同步是使多线程并发运行的最重要的概念。这一章介绍了可以在应用中使用的各种同步工具。在当今世界，并行是一个方向。因为与花费大量的时间和财力去制造更快的处理器相比，人们更愿意去研制多核处理器。此外，还介绍了.NET 4.0 中的并行扩展和任务并行库（TPL）。

第 13 章讨论定义新类型的最佳设计实践，以及如何实现这些类型，从而可以自然地使用它们，也避免类型的使用者不经意地误用它们。其中有些主题在其他章也有所涉及，而这一章作了详细论述。这一章最后给出了一个使用 C#定义新类型时应该考虑的问题检查单。

第 14 章介绍 C# 3.0 引入的一个新特性：扩展方法。由于可以像实例方法那样在它们所扩展的类型上调用扩展方法，因此扩展方法看起来像类型契约的扩展。但扩展方法的作用不止于此，这一章向你展示 C#中扩展方法将如何打开函数式编程之门。

第 15 章介绍 C# 3.0 里引入的另一个新特性：lambda 表达式。你可以用 lambda 表达式以一种简洁和描述性的语法来声明并实例化委托。虽然前面提到的匿名方法也能达到同样的目的，但匿名方法更加啰嗦而且语法上没有这么清新流畅。而在 C# 3.0 及后来的版本中，可以把 lambda 表达式转换成表达式树。也就是说，C#语言具有内建的功能来把代码转换成数据结构。这个功能本身就很有用，但如果与 LINQ 结合起来使用功效更大。lambda 表达式与扩展方法的结合把函数式编程完整地引入了 C#。

第 16 章介绍 C# 3.0 引入的最强大的新特性 LINQ。通过面向 LINQ 的关键字来使用 LINQ 表达式，可以把数据查询无缝地集成到代码中。LINQ 在通常的 C#命令式编程与数据查询的函数式编程之间架起了一座桥梁。LINQ 表达式可以用来操作正常的对象，也可以用来操作来自 SQL 数据库、数据集和 XML 等多种来源的数据。

第 17 章介绍了 C# 4.0 里加入的动态类型。动态类型简化了与包括 COM Automation 对象在内的动态.NET 语言的集成。仅仅为了与这些组件集成而编写不自然而且难以阅读的代码的日子已经一去不复返了，因为动态类型为你处理了所有的琐碎工作。动态类型的实现利用了动态语言运行时（Dynamic Language Runtime，DLR）。DLR 也是 IronRuby 和 IronPython 之类的动态语言的基础。联合使用动态类型和 ExpandoObject 这样的 DLR 类型，你可以在 C#里创建并实现真正的动态类型。

目　录

第 1 章　C#预览 …… 1
1.1　C#和 C++的区别 …… 1
1.1.1　C# …… 1
1.1.2　C++ …… 2
1.1.3　CLR 垃圾回收 …… 2
1.2　C#程序示例 …… 3
1.3　C# 2.0 新特性概览 …… 4
1.4　C# 3.0 新特性概览 …… 5
1.5　C# 4.0 新特性概览 …… 6
1.6　小结 …… 6

第 2 章　C# 和 CLR …… 8
2.1　CLR 中的 JIT 编译器 …… 8
2.2　程序集及程序集加载器 …… 10
2.2.1　程序工作集最小化 …… 10
2.2.2　给程序集命名 …… 11
2.2.3　加载程序集 …… 11
2.3　元数据 …… 11
2.4　交义语言的兼容性 …… 12
2.5　小结 …… 13

第 3 章　C#语法概述 …… 14
3.1　C#是一门强类型的语言 …… 14
3.2　表达式 …… 15
3.3　语句和表达式 …… 16
3.4　类型和变量 …… 16
3.4.1　值类型 …… 18
3.4.2　引用类型 …… 20
3.4.3　默认变量初始化 …… 21
3.4.4　隐式类型化局部变量 …… 22
3.4.5　类型转换 …… 23
3.4.6　as 和 is 操作符 …… 25
3.4.7　泛型 …… 27
3.5　命名空间 …… 28
3.5.1　定义命名空间 …… 28
3.5.2　使用命名空间 …… 29
3.6　控制流 …… 30
3.6.1　if-else、while、do-while 和 for …… 31
3.6.2　switch …… 31
3.6.3　foreach …… 31
3.6.4　break、continue、goto、return 和 throw …… 32
3.7　小结 …… 32

第 4 章　类、结构和对象 …… 33
4.1　类定义 …… 34
4.1.1　字段 …… 35
4.1.2　构造函数 …… 37
4.1.3　方法 …… 38
4.1.4　属性 …… 39
4.1.5　封装 …… 43
4.1.6　可访问性 …… 46
4.1.7　接口 …… 47
4.1.8　继承 …… 48
4.1.9　密封类 …… 54
4.1.10　抽象类 …… 55
4.1.11　嵌套类 …… 56
4.1.12　索引器 …… 58
4.1.13　分部类 …… 60

4.1.14 分部方法 …… 61
4.1.15 静态类 …… 62
4.1.16 保留的成员名字 …… 64
4.2 值类型定义 …… 64
4.2.1 构造函数 …… 65
4.2.2 this 的含义 …… 66
4.2.3 终结器 …… 69
4.2.4 接口 …… 69
4.3 匿名类型 …… 69
4.4 对象初始化器 …… 72
4.5 装箱和拆箱 …… 74
4.5.1 什么时候发生装箱 …… 78
4.5.2 效率和混淆 …… 79
4.6 System.Object …… 80
4.6.1 等同性及其意义 …… 81
4.6.2 IComparable 接口 …… 81
4.7 创建对象 …… 82
4.7.1 new 关键字 …… 82
4.7.2 字段初始化 …… 83
4.7.3 静态（类）构造函数 …… 84
4.7.4 实例构造函数和创建顺序 …… 86
4.8 销毁对象 …… 90
4.8.1 终结器 …… 90
4.8.2 确定性的析构 …… 91
4.8.3 异常处理 …… 92
4.9 可清除对象 …… 92
4.9.1 IDisposable 接口 …… 92
4.9.2 using 关键字 …… 94
4.10 方法参数类型 …… 95
4.10.1 值参数 …… 96
4.10.2 ref 参数 …… 96
4.10.3 out 参数 …… 98
4.10.4 参数数组 …… 98
4.10.5 方法重载 …… 99
4.10.6 可选参数 …… 99
4.10.7 命名参数 …… 100
4.11 继承和虚方法 …… 103
4.11.1 虚方法和抽象方法 …… 103
4.11.2 override 和 new 方法 …… 103
4.11.3 密封方法 …… 105
4.11.4 关于 C#虚方法再啰嗦几句 …… 105
4.12 继承、包含和委托 …… 106
4.12.1 接口继承和类继承的选择 …… 106
4.12.2 委托和组合与继承 …… 107
4.13 小结 …… 109

第 5 章 接口和契约 …… 110
5.1 接口定义类型 …… 110
5.2 定义接口 …… 111
5.2.1 接口中可以有什么 …… 112
5.2.2 接口继承与成员隐藏 …… 113
5.3 实现接口 …… 114
5.3.1 隐式接口实现 …… 115
5.3.2 显式接口实现 …… 115
5.3.3 派生类中覆盖接口实现 …… 117
5.3.4 小心值类型实现接口的副作用 …… 120
5.4 接口成员匹配规则 …… 121
5.5 值类型的显式接口实现 …… 124
5.6 版本考虑 …… 126
5.7 契约 …… 127
5.7.1 类实现契约 …… 127
5.7.2 接口契约 …… 128
5.8 在接口和类之间选择 …… 129
5.9 小结 …… 132

第 6 章 重载操作符 …… 133
6.1 可以并不意味着应该 …… 133
6.2 重载操作符的类型和格式 …… 133
6.3 操作符不应该改变其操作数 …… 134
6.4 参数顺序有影响么 …… 135
6.5 重载加法运算符 …… 135
6.6 可重载的操作符 …… 136
6.6.1 比较操作符 …… 137
6.6.2 转换操作符 …… 139
6.6.3 布尔操作符 …… 142
6.7 小结 …… 144

第 7 章 异常处理和异常安全 …… 145
7.1 CLR 如何对待异常 …… 145
7.2 C#里的异常处理机制 …… 145

7.2.1 抛出异常 …… 145
7.2.2 从.NET 2.0 开始的未处理异常的变化 …… 146
7.2.3 try, catch 和 finally 语句语法预览 …… 147
7.2.4 重新抛出异常和转译异常 …… 149
7.2.5 finally 代码块抛出的异常 …… 151
7.2.6 终结器抛出的异常 …… 152
7.2.7 静态构造函数抛出的异常 …… 153
7.3 谁应该处理异常 …… 154
7.4 避免使用异常来控制流程 …… 154
7.5 取得异常中立 …… 155
7.5.1 异常中立代码的基本结构 …… 155
7.5.2 受限执行区域 …… 160
7.5.3 临界终结器和 SafeHandle …… 162
7.6 创建定制的异常类 …… 165
7.7 使用分配的资源和异常 …… 167
7.8 提供回滚行为 …… 170
7.9 小结 …… 173

第 8 章 使用字符串 …… 174

8.1 字符串概述 …… 174
8.2 字符串字面量 …… 175
8.3 格式指定和全球化 …… 176
8.3.1 Object.ToString、IFormattable 和 CultureInfo …… 176
8.3.2 创建和注册自定义 CultureInfo 类型 …… 177
8.3.3 格式化字符串 …… 179
8.3.4 Console.WriteLine 和 String.Format …… 180
8.3.5 自定义类型的字符串格式化举例 …… 181
8.3.6 ICustomFormatter …… 182
8.3.7 字符串比较 …… 184
8.4 处理来自外部的字符串 …… 185
8.5 StringBuilder …… 187
8.6 使用正则表达式搜索字符串 …… 188
8.6.1 使用正则表达式搜索 …… 189
8.6.2 搜索和分组 …… 190
8.6.3 使用正则表达式替换文本 …… 194
8.6.4 正则表达式创建选项 …… 196
8.7 小结 …… 197

第 9 章 数组、集合类型和迭代器 …… 198

9.1 数组介绍 …… 198
9.1.1 隐式类型化数组 …… 199
9.1.2 类型的转换和协方差 …… 201
9.1.3 排序和搜索 …… 202
9.1.4 同步 …… 202
9.1.5 向量与数组 …… 203
9.2 多维矩形数组 …… 204
9.3 多维锯齿数组 …… 206
9.4 集合类型 …… 207
9.4.1 比较 ICollection<T>和 ICollection …… 207
9.4.2 集合同步 …… 209
9.4.3 列表 …… 209
9.4.4 字典 …… 210
9.4.5 集合 …… 211
9.4.6 System.Collections.ObjectModel …… 211
9.4.7 效率 …… 213
9.5 IEnumerable<T>、IEnumerator<T>、IEnumerable 和 IEnumerator …… 214
9.6 迭代器 …… 218
9.7 集合初始化器 …… 226
9.8 小结 …… 227

第 10 章 委托、匿名方法和事件 …… 228

10.1 委托概览 …… 228
10.2 委托的创建和使用 …… 229
10.2.1 单委托 …… 229
10.2.2 委托链 …… 231
10.2.3 迭代委托链 …… 232
10.2.4 非绑定(公开实例)的委托 …… 233
10.3 事件 …… 236
10.4 匿名方法 …… 239
10.4.1 捕获变量与闭包 …… 241
10.4.2 当心捕获变量的意外 …… 243
10.4.3 匿名方法作为委托参数绑定器 …… 246

10.5 Strategy 模式 …… 248
10.6 小结 …… 250

第 11 章 泛型 …… 251
11.1 泛型和 C++模板之间的区别 …… 252
11.2 泛型的效率和类型安全 …… 253
11.3 泛型的类型定义和构造类型 …… 254
11.3.1 泛型类和结构 …… 255
11.3.2 泛型接口 …… 257
11.3.3 泛型方法 …… 257
11.3.4 泛型委托 …… 259
11.3.5 泛型转换 …… 262
11.3.6 默认值表达式 …… 263
11.3.7 Nullable 类型 …… 264
11.3.8 构造类型访问权限控制 …… 266
11.3.9 泛型和继承 …… 266
11.4 约束 …… 267
11.5 协变与逆变 …… 272
11.5.1 协变 …… 274
11.5.2 逆变 …… 276
11.5.3 不变性 …… 278
11.5.4 方差与委托 …… 279
11.6 泛型系统集合 …… 282
11.7 泛型系统接口 …… 283
11.8 精选的问题和解决方案 …… 284
11.8.1 泛型类型中的转化和操作符 … 285
11.8.2 动态地创建构造类型 …… 293
11.9 小结 …… 294

第 12 章 C#中的线程 …… 295
12.1 C#和.NET 中的线程 …… 295
12.1.1 开始线程编程 …… 296
12.1.2 IOU 模式和异步方法调用 …… 299
12.1.3 线程状态 …… 299
12.1.4 终止线程 …… 301
12.1.5 停止和唤醒休眠线程 …… 303
12.1.6 等待线程退出 …… 304
12.1.7 前台和后台线程 …… 304
12.1.8 线程本地存储 …… 305
12.1.9 非托管线程和 COM 套件如何适应 …… 308
12.2 线程间同步工作 …… 309
12.2.1 用 Interlocked 类实现轻量级的同步 …… 311
12.2.2 SpinLock 类 …… 316
12.2.3 Monitor 类 …… 317
12.2.4 锁对象 …… 325
12.2.5 信号量 …… 329
12.2.6 事件 …… 331
12.2.7 Win32 的同步对象和 WaitHandle …… 332
12.3 使用线程池 …… 334
12.3.1 异步方法调用 …… 335
12.3.2 定时器 …… 341
12.4 并发编程 …… 343
12.4.1 Task 类 …… 343
12.4.2 Parallel 类 …… 345
12.4.3 线程池的简单入口 …… 349
12.4.4 线程安全集合类 …… 350
12.5 小结 …… 350

第 13 章 C#规范形式探索 …… 351
13.1 引用类型的规范形式 …… 351
13.1.1 类默认是密封的 …… 352
13.1.2 使用非虚拟接口 (NVI) 模式 …… 353
13.1.3 对象是否可克隆 …… 355
13.1.4 对象是否可清除 …… 360
13.1.5 对象需要终结器吗 …… 362
13.1.6 对象相等意味着什么 …… 368
13.1.7 如果重写了 Equals 方法，那么也应该重写 GetHashCode 方法 …… 374
13.1.8 对象支持排序吗 …… 377
13.1.9 对象需要形式化显示吗 …… 379
13.1.10 对象可以被转换吗 …… 382
13.1.11 在所有情况下都保证类型安全 …… 384
13.1.12 使用非可变的引用类型 …… 387
13.2 值类型的规范形式 …… 389
13.2.1 为了获得更好的性能而重写 Equals 方法 …… 390

13.2.2 值类型需要支持接口吗 …… 394
13.2.3 将接口成员和派生方法实现为类型安全的形式 …… 394
13.3 小结 …… 397
13.3.1 引用类型的检查单 …… 397
13.3.2 值类型的检查单 …… 398

第 14 章 扩展方法 …… 399
14.1 扩展方法介绍 …… 399
14.1.1 编译器如何发现扩展方法 …… 400
14.1.2 探究内部实现 …… 403
14.1.3 代码易读性与代码易懂性 …… 403
14.2 使用建议 …… 404
14.2.1 考虑扩展方法优先于继承 …… 404
14.2.2 分离的命名空间中的隔离扩展方法 …… 405
14.2.3 修改一个类型的契约可能会打破扩展方法 …… 406
14.3 转换 …… 406
14.4 链式操作 …… 410
14.5 自定义迭代器 …… 411
14.6 访问者模式 …… 417
14.7 小结 …… 420

第 15 章 lambda 表达式 …… 422
15.1 lambda 表达式介绍 …… 422
15.1.1 lambda 表达式与闭包 …… 423
15.1.2 lambda 语句 …… 427
15.2 表达式树 …… 428
15.2.1 对表达式的操作 …… 430
15.2.2 函数的数据表现 …… 431
15.3 lambda 表达式的有益应用 …… 431
15.3.1 迭代器和生成器重访问 …… 432
15.3.2 再谈闭包（变量捕获）和缓存 …… 435
15.3.3 currying …… 439
15.3.4 匿名递归 …… 441
15.4 小结 …… 442

第 16 章 LINQ：语言集成查询 …… 443
16.1 连接数据的桥梁 …… 443
16.1.1 查询表达式 …… 444
16.1.2 再谈扩展方法和 lambda 表达式 …… 446
16.2 标准查询操作符 …… 446
16.3 C#查询关键字 …… 448
16.3.1 from 子句和范围变量 …… 448
16.3.2 join 子句 …… 449
16.3.3 where 子句和过滤器 …… 451
16.3.4 orderby 子句 …… 451
16.3.5 select 子句和投影 …… 452
16.3.6 let 子句 …… 454
16.3.7 group 子句 …… 455
16.3.8 into 子句和持续性 …… 458
16.4 偷懒的好处 …… 459
16.4.1 C#迭代器鼓励懒惰 …… 459
16.4.2 不能偷懒 …… 460
16.4.3 立即执行查询 …… 462
16.4.4 再谈表达式树 …… 462
16.5 函数式编程中的技术 …… 462
16.5.1 自定义标准查询操作符和延迟求值 …… 463
16.5.2 替换 foreach 语句 …… 469
16.6 小结 …… 471

第 17 章 Dynamic 类型 …… 472
17.1 dynamic 意味着什么？ …… 472
17.2 dynamic 如何工作？ …… 474
17.2.1 大统一 …… 476
17.2.2 调用站 …… 476
17.2.3 具有自定义动态行为的对象 …… 478
17.2.4 效率 …… 480
17.2.5 Dynamic 装箱 …… 482
17.3 Dynamic 转换 …… 482
17.4 动态重载解析 …… 485
17.5 Dynamic 继承 …… 486
17.5.1 不能派生自 dynamic …… 486
17.5.2 不能实现动态接口 …… 487
17.5.3 可以派生自 Dynamic 基类 …… 489
17.6 C#里的推断类型 …… 490
17.7 dynamic 类型的限制 …… 493
17.8 ExpandoObject：动态地创建对象 …… 493
17.9 小结 …… 497

第1章 C# 预 览

因为本书的目标读者是有经验的面向对象的开发人员，因此我假定你已经对.NET 运行时有了一些了解。Don Box 的《.NET 本质论，第一卷：公共语言运行库》[①]是关于.NET 运行时的一本好书，建议读者看看，如果想进一步了解.NET 运行时，可以参阅此书。另外，了解 C#和 C++之间的异同也很重要，因此本章从比较二者的区别开始。之后我们会用一个基本的“Hello World”例子来加以说明。如果你有.NET 应用程序的开发经验，可以跳过本章。但是，你可能还是应该看看 1.5 节介绍 C# 4.0 的内容。

1.1 C#和 C++的区别

C#是一种强类型的面向对象语言，其代码看起来和 C++以及 Java 类似。C#语言设计者的这个决定使 C++开发者可在已有的知识结构基础上更高效地应用 C#语言。从语法上来讲，C#和 C++有些差异，但它们之间主要的差别是语义和行为上的，这是由执行它们的运行时环境的不同带来的。

1.1.1 C#

C#源代码编译成托管代码。我们知道，托管代码是一种中间语言，它介于高级语言（C#）和最低级的语言（汇编语言或机器码）之间。运行的时候，公共语言运行库（Common Language Runtime，CLR）用即时（Just In Time，JIT）编译来动态编译托管代码。和其他工程产物一样，这种技术有利也有弊。一个显著的弊端就是运行时编译的效率不高。这个过程不同于 Perl、JScript 等脚本语言所用的解释（interpreting）方式。JIT 编译器并不会在每个函数或方法每次调用的时候都编译它们，而只是在第一次调用的时候把托管代码编译成运行平台的本地机器码。因为中间语言的内存占用量比较少，JIT 编译的一个显著优势就是应用程序的工作集（working set）减少了。在应用程序执行过程中，只有必要的代码才用 JIT 编译。例如，如果应用程序里面有打印的代码，但用户不打印文档，这段代码就是不需要的，因而 JIT 编译器不会编译它。另外，CLR 在运行时可以动态地优化程序的执行。例如，CLR 可以通过重新调整内存中编译后的代码，用某种方式来减少内存管理中的页错误，这都是在运行时做的。考虑到所有这些优势，你就会发现对大部分应用程序来说，这种技术是利大于弊的。

注解 事实上，可以选择用原始的中间语言（IL）来编写程序，然后在IL汇编程序（ILASM）中调试。但是，这样做很浪费时间。高级语言几乎总是可以提供通过IL代码获得的全部功能。

① 该书曾由中国电力出版社翻译出版。——编者注

1.1.2 C++

与C#不同，C++代码总是编译成本地代码。本地代码是针对编译程序的处理器的机器码。为了方便，我们约定这里讨论的是本地编译的C++代码，而不是通过C++/CLI实现的托管C++。如果希望本地C++应用程序运行在不同的平台，比如32位平台和64位平台，那么必须分别单独编译。通常，本地的二进制输出不是跨平台兼容的。

而CLR构建的基础——CLI（Common Language Infrastructure，公共语言基础设施）是一个国际标准[①]，所以中间语言是跨平台兼容的。这个标准正被快速推动实施，也正在Microsoft Windows之外的平台上实现。

注解 建议你看看Mono团队所取得的成果，他们正在其他平台创建开源的虚拟执行系统（VES）。[②]

CLI定义了托管代码的可移植的执行（Portable Executable，PE）文件格式。因此可以在Windows平台上实际编译一个C#程序，其输出在Windows和Linux平台都可以执行而不需要重新编译，因为甚至文件格式都是标准的[③]。这个级别的可移植性非常便利，这是COM/DCOM设计者以前梦寐以求的，但由于种种原因，它没有在这个层次上取得跨越异构平台的成功[④]。失败的主要原因之一就是，COM对描述类型及其依赖关系缺乏有足够表现力的、可扩展的机制。而CLI通过引入元数据轻而易举地解决了这个问题。我们将在第2章介绍元数据。

1.1.3 CLR垃圾回收

CLR中有个关键工具是垃圾回收器（Garbage Collector，GC）。GC让你从分配和释放内存的负担中解脱出来，而这些内存管理工作是很多软件错误发生的根源。然而GC并没有为你解除所有的资源处理负担，从第4章可以看到这一点。例如，文件句柄作为一个资源必须在使用后释放，就像内存一样。GC只直接管理内存资源，其他的比如数据库连接和文件句柄，可以用一个终结器（finalizer，将在第13章介绍）在GC通知你对象将要被摧毁的时候来释放。但是，一个更好的办法是用Disposable模式来完成这个任务，我们将在第4章和第13章介绍。

注解 CLR间接引用所有引用类型的对象，和C++里面的指针和引用类似，但C#没有指针的语法定义。在C#中定义一个引用类型变量的时候，事实上是预留了一个与类型关联的存储位置，这个位置在堆或者栈上保存着对象的引用。当把一个变量的对象引用复制到另一个变量时，会得到引用到同一个对象的两个变量。所有的引用类型实例都位于托管堆上。CLR管理这些对象的位置，如果需要移动，它会更新那些指向被移动对象的引用去指向新的位置。CLR里面也有值类型，它们的实例存活在栈上或作为托管堆上对象的一个域。它们的用法有很多限制和细微差别。通

① 可以在www.ecma-international.org网站上找到CLI标准文档Ecma-335。另外，Ecma-334是C#语言的标准文档。

② 可以在www.mono-project.com网站上找到关于Mono项目的内容。

③ 当然，目标平台上要安装好程序所需要的所有库文件。有了.NET标准库，这很快就要成为现实了。例如，浏览一下www.go-mono.com/docs/，就会看到Mono项目的库文件覆盖有多广。

④ 要了解这段惨痛的历史，我推荐阅读Don Box和Chris Sell的《.NET本质论，第一卷：公共语言运行库》——这个标题让我们相信第二卷随时会出版，我们希望它别像Mel Brooks的《帝国时代：第一部》那样。（Mel Brooks是美国著名导演，《帝国时代》只出了第一部，没有第二部。作者此言不幸而言中，《.NET本质论》一直未出第二卷。）——编者注

常在需要轻量级的结构来管理相关数据的时候需要用到它们。值类型在对非可变（immutable）的数据块建模的时候也有用，第4章将更详细地讨论这个问题。

用C#可以快速开发应用，而不必处理如C++环境里面那么繁琐的细节。同时，C#是一种让C++或Java开发者感到熟悉的语言。

1.2 C#程序示例

我们来进一步看看一个简单的C#程序。首先看一下大家都喜闻乐见的“Hello World!”程序。用C#编写的控制台版本是这个样子的：

```
class EntryPoint {
    static void Main() {
        System.Console.WriteLine( "Hello World!" );
    }
}
```

注意这个C#程序的结构。它声明了一个类型（名为EntryPoint的类）和类的一个成员（名为Main的方法）。这和C++不同，C++是在头文件里面定义类型，然后在一个单独的编译单元里定义它，通常采用一个.cpp之类的文件名。另外，元数据（元数据描述了模块里面的所有类型，并由C#编译器透明生成）使C++里面的前置声明和包含（inclusion）不再需要。实际上，C#中甚至不存在前置声明。

C++程序员会发现静态的Main方法比较熟悉，只是名字的首字母变成了大写。所有的程序都需要一个入口点，对C#而言就是静态Main方法。当然还有更多的区别，例如，Main方法在类里面声明（在这个例子中，是名为Entry Point的类）。在C#里，所有的方法都必须在类型定义里面声明，而没有C++里面的静态、自由函数。Main方法的返回值可以是int或void，这个取决于你的需要。在这个例子中，Main没有参数，但如果需要访问命令行参数，Main方法可以声明一个参数（一个字符串数组）来访问它们。

注解 如果应用程序包含多个含静态Main方法的类型，则可以通过/main编译开关来选择用哪一个。

你可能注意到WriteLine调用看起来比较繁琐。必须用类名Console来限定这个方法的名字，还必须指定Console类所在的命名空间（这里是System）。.NET（C#也一样）支持命名空间来避免巨大的全局命名空间内名字的冲突。当然，全限定名（fully qualified name），包括命名空间，是不必每次输入的，C#提供了using指令来解决这个问题。using与Java里的import和C++里的using namespace类似。因此可以对上面的程序稍加修改，得到代码清单1-1。

代码清单1-1 hello_world.cs

```
using System;

class EntryPoint {
    static void Main() {
        Console.WriteLine( "Hello World!" );
    }
}
```

有了using System;指令，可以在调用Console.WriteLine的时候省略System命名空间。

可以在一个Windows命令行窗口执行下面的命令来编译这个例子：

```
csc.exe /r:mscorlib.dll /target:exe hello_world.cs
```

我们来仔细看看这个命令行做了什么。

- csc.exe是Microsoft的C#编译器。
- /r选项指定了这个程序的程序集依赖。程序集在概念上和本地代码中的DLL类似。mscorlib.dll里面定义了System.Console对象。现实中，不必引用mscorlib，因为编译器会自动引用它，除非用了/nostdlib选项。
- /target:exe选项告诉编译器你在编译一个命令行应用程序，这是不指定时的默认值。其他的选项包括用来编译Windows GUI应用的/target:winexe，用来生成带.dll后缀的DLL程序集的/target:library，用来生成带.netmodule后缀的DLL的/target:module。/target:module生成的模块不包含任何程序集清单（manifest），因此之后必须用程序集链接器al.exe把它引入一个程序集，这为创建多文件的程序集提供了一种方法。
- hello_world.cs是正在编译的C#程序。如果项目中有多个C#文件，可以在命令行的末尾把它们全部列上。

执行这个命令行后，产生了hello_world.exe，可以通过命令行来执行它，并查看预期的结果。如果愿意，还可以用/debug选项来重新编译这份代码，这样你可以在调试器里面单步跟踪执行过程。为了体验C#平台独立的特性，如果正好有一个安装有Mono VES的Linux操作系统，你可以把hello_world.exe以二进制方式直接复制到上面，如果Linux一切设置正确，将可以看到程序按预期的方式运行。

1.3　C# 2.0 新特性概览

自 2000 年末最初发布以来，C#取得了长足的发展，其发展历程因得到广泛的采用而进一步加速。随着 Visual Studio 2005 和.NET Framework 2.0 的发布，C#编译器开始支持 C# 2.0 的增强功能。这是一个好消息，因为 C# 2.0 包括了很多方便的特性，这些特性带来更舒适的编程体验，也提高了效率。本节简要叙述这些新特性，并指出本书的哪些章节包括这方面的详细信息。

毫无疑问，C# 2.0 最好的扩展特性是对泛型（generic）的支持。它的语法类似 C++的模板，但二者之间的主要区别在于从.NET 泛型创建的构造类型（constructed type）本质上是动态的，也就是说，它们是在运行时被绑定和构造的。这与 C++从模板创造具体类型有区别，C++的这个过程是静态的，从某种程度上说是编译时绑定和构造的[①]。当与向量、列表、散列表等集合类型结合使用时，泛型是最有用的，它们带来的成果是最高效的。泛型可以依具体类型而不是使用所有对象的基类型 System.object 来处理其所包含的类型。第 11 章讲解泛型，第 9 章将讲解集合（collection）。

C# 2.0 支持匿名方法。匿名方法有时也被称为 lambda 函数，这是函数式编程的术语。C#的匿名方法对委托和事件特别有用。委托和事件是用来注册回调函数的结构，这些回调方法在委托或事件触发的时候被调用。通常，可以用一个在其他地方定义好的方法。但有了匿名方法，你可以在委托或事件被构造的时候内联地定义它们的代码。这在委托只需要做有限的工作的时候特别方便，因为这种情况下定义一个完整的方法就有点小题大做了。更好的是，匿名方法体里面可以访问它的定义点所在范围内的所有变量[②]。匿名方法将在第 10 章讨论。C# 3.0 新引入的 lambda 表达式取代了匿名方法，使代码的可读性更好。

① 用 C++/CLI，可以同时使用泛型和模板。C++/CLI 是在 Ecma-372 中标准化，并由 Visual Studio 2008 首先实现的。

② 这被称为闭包（closure）或变量捕获（variable capture）。