

O'REILLY®

需要整本电子书，联系我QQ：2667271557

TURING

图灵程序设计丛书

第4版



C#经典实例

C# 6.0 Cookbook

针对C# 6.0和.NET Framework 4.6全面更新；涵盖C#开发的各类陷阱和问题

[美] Jay Hilyard Stephen Teilhet 著
徐敬德 译



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

需要整本电子书，联系我QQ：2667271557

需要整本电子书，联系我QQ：2667271557



图灵程序设计丛书

C#经典实例（第4版）

C# 6.0 Cookbook

[美] Jay Hilyard Stephen Teilhet 著

徐敬德 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc. 授权人民邮电出版社出版

人民邮电出版社

北 京

需要整本电子书，联系我QQ：2667271557

需要整本电子书，联系我QQ：2667271557

图书在版编目（C I P）数据

C#经典实例：第4版 / (美) 杰伊·希尔亚德
(Jay Hilyard), (美) 斯蒂芬·泰耶
(Stephen Teilhet) 著；徐敬德译. — 北京：人民邮
电出版社, 2016.10
(图灵程序设计丛书)
ISBN 978-7-115-43509-5

I. ①C… II. ①杰… ②斯… ③徐… III. ①C语言—
程序设计 IV. ①TP312.8

中国版本图书馆CIP数据核字(2016)第216681号

内 容 提 要

本书共分为13章，每一章侧重于特定主题的C#解决方案。具体内容包括：类和泛型，集合、枚举器和迭代器，数据类型，语言集成查询和lambda表达式，调试和异常处理，反射和动态编程，正则表达式，文件系统I/O，网络和Web，XML，安全，线程、同步和并发，工具箱。本书使用大量范例，帮助开发人员快速理解并解决现实中的问题。

本书面向所有级别的C#和.NET开发人员。

-
- ◆ 著 [美] Jay Hilyard Stephen Teilhet
译 徐敬德
责任编辑 朱巍
执行编辑 杨琳 赵瑞琳
责任印制 彭志环
 - ◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京 印刷
 - ◆ 开本：800×1000 1/16
印张：37
字数：874千字 2016年10月第1版
印数：1—3 500册 2016年10月北京第1次印刷
著作权合同登记号 图字：01-2016-4798号
-

定价：129.00元

读者服务热线：(010)51095186转600 印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第8052号

需要整本电子书，联系我QQ：2667271557

O'Reilly Media, Inc.介绍

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

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

业界评论

“O'Reilly Radar 博客有口皆碑。”

——Wired

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

——Business 2.0

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

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

——Linux Journal

目录

前言.....xi

第 1 章 类和泛型.....1

1.0 简介.....1

1.1 创建联合类型的结构.....3

1.2 使类型可排序.....5

1.3 使类型可查找.....9

1.4 从一个方法返回多个数据项.....12

1.5 解析命令行参数.....15

1.6 在运行时初始化常量字段.....25

1.7 构建可克隆的类.....28

1.8 确保对象的处置.....31

1.9 确定何时何处使用泛型.....33

1.10 理解泛型类型.....34

1.11 反转有序列表中的内容.....41

1.12 约束类型参数.....43

1.13 将泛型变量初始化为默认值.....46

1.14 向生成的实体中添加钩子.....48

1.15 控制如何触发多播委托中的一个委托.....50

1.16 在 C# 中使用闭包.....56

1.17 使用函数对象在列表中执行多种操作.....61

1.18 控制结构类型字段初始化.....64

1.19 以更简洁的方式检查 null 值.....68

第 2 章 集合、枚举器和迭代器	72
2.0 简介	72
2.1 寻找 List<T> 中的重复数据项	74
2.2 保持 List<T> 有序	78
2.3 对 Dictionary 的键和 / 或值排序	80
2.4 创建具有最小值和最大值边界的 Dictionary	82
2.5 在应用程序会话间持久化一个集合	84
2.6 测试 Array 或 List<T> 中的每个元素	86
2.7 创建自定义枚举器	88
2.8 处理 finally 语句块和迭代器	91
2.9 在类中实现嵌套的 foreach 功能	95
2.10 使用线程安全的字典进行并发访问，不手动加锁	99
第 3 章 数据类型	106
3.0 简介	106
3.1 把二进制数据编码为 base64 格式	108
3.2 解码 base64 编码的二进制数据	109
3.3 把作为 byte[] 返回的字符串转换为字符串	110
3.4 把字符串传递给只接受 byte[] 的方法	112
3.5 确定一个字符串是否为有效的数字	113
3.6 舍入浮点值	114
3.7 选择一种舍入算法	115
3.8 安全地执行窄化数据转换	116
3.9 测试有效的枚举值	118
3.10 在位掩码中使用枚举成员	120
3.11 确定是否设置了一个或多个枚举标志	122
第 4 章 语言集成查询和 lambda 表达式	126
4.0 简介	126
4.1 查询消息队列	128
4.2 对数据使用集合语义	132
4.3 利用 LINQ to SQL 重用参数化查询	136
4.4 以文化敏感的方式对结果排序	138
4.5 添加用于 LINQ 的函数式扩展	141
4.6 跨数据库执行查询和联接	144
4.7 利用 LINQ 查询配置文件	147
4.8 从数据库直接创建 XML 文件	150
4.9 有选择地输出查询结果	162
4.10 将 LINQ 用于不支持 IEnumerable<T> 的集合	165

4.11 执行高级接口查找	167
4.12 使用 lambda 表达式	168
4.13 在 lambda 表达式中使用不同的参数修饰符	173
4.14 用并行来加速 LINQ 操作	176
第 5 章 调试和异常处理	187
5.0 简介	187
5.1 知道何时捕获并重新引发异常	193
5.2 处理通过反射调用的方法引发的异常	194
5.3 创建新的异常类型	197
5.4 在首次异常上中断	204
5.5 处理从异步委托中引发的异常	209
5.6 利用 Exception.Data 为异常提供所需的额外信息	211
5.7 在 WinForms 应用程序中处理未经处理的异常	213
5.8 在 WPF 应用程序中处理未经处理的异常	214
5.9 确定一个进程是否停止了响应	217
5.10 在应用程序中使用事件日志	219
5.11 监视事件日志中的特定条目	229
5.12 实现一个简单的性能计数器	230
5.13 为类创建自定义的调试显示	233
5.14 跟踪异常从何而来	235
5.15 在异步情境下处理异常	237
5.16 有选择地处理异常	243
第 6 章 反射和动态编程	247
6.0 简介	247
6.1 列出引用的程序集	248
6.2 确定程序集中的类型特征	252
6.3 确定继承特征	256
6.4 使用反射调用成员	261
6.5 访问局部变量信息	264
6.6 创建一个泛型类型	267
6.7 使用 dynamic 与使用 object	268
6.8 动态构建对象	271
6.9 使对象可扩展	275
第 7 章 正则表达式	284
7.0 简介	284
7.1 从 MatchCollection 中提取组	285

7.2	验证正则表达式的语法	288
7.3	增强基本的字符串替换函数	289
7.4	实现一个更好的分词器	292
7.5	返回匹配所在的整行内容	293
7.6	找到特定次数的匹配	297
7.7	使用常见模式	299
第 8 章 文件系统 I/O		303
8.0	简介	303
8.1	使用通配符查找目录和文件	304
8.2	获取目录树	309
8.3	解析路径	313
8.4	启动并与控制台工具交互	314
8.5	锁定文件的一部分	316
8.6	等待文件系统中的动作发生	320
8.7	比较两个可执行模块的版本信息	322
8.8	查询系统上所有驱动器的信息	325
8.9	压缩和解压缩文件	327
第 9 章 网络和 Web		337
9.0	简介	337
9.1	处理 Web 服务器错误	338
9.2	与 Web 服务器通信	339
9.3	通过代理服务器	341
9.4	从一个 URL 获取 HTML	343
9.5	使用 Web 浏览器控件	344
9.6	以编程方式预构建一个 ASP.NET 网站	346
9.7	为 Web 应用对数据进行转义和取消转义	349
9.8	检查 Web 服务器的自定义错误页	351
9.9	编写一个 TCP 服务器	355
9.10	编写一个 TCP 客户端	362
9.11	模拟表单执行	370
9.12	通过 HTTP 传输数据	373
9.13	使用命名管道进行通信	377
9.14	以编程方式发送 ping	384
9.15	使用 SMTP 服务发送 SMTP 邮件	386
9.16	使用套接字扫描机器的端口	388
9.17	使用当前的互联网连接设置	392
9.18	使用 FTP 传输文件	398

第 10 章 XML	401
10.0 简介	401
10.1 以文档顺序读取和访问 XML 数据	401
10.2 查询 XML 文档的内容	405
10.3 验证 XML	409
10.4 检测对 XML 文档的修改	413
10.5 处理 XML 字符串中的无效字符	416
10.6 转换 XML	419
10.7 验证修改过的 XML 文档而无需重新加载	427
10.8 扩展转换	430
10.9 从现有 XML 文件批量获取架构	436
10.10 将参数传递给转换	438
第 11 章 安全	443
11.0 简介	443
11.1 加密和解密字符串	443
11.2 加密和解密文件	447
11.3 清理密码算法信息	452
11.4 避免字符串在传输或静止时被篡改	454
11.5 保证安全断言的安全	460
11.6 验证是否已授予程序集特定权限	462
11.7 最小化程序集的攻击面	463
11.8 获得安全和 / 或审计信息	464
11.9 授权或撤销对文件或注册表项的访问	469
11.10 使用安全字符串保护字符串数据	472
11.11 保护流数据	474
11.12 加密 web.config 信息	486
11.13 获得一个更安全的文件句柄	488
11.14 保存密码	489
第 12 章 线程、同步和并发	496
12.0 简介	496
12.1 创建每线程静态字段	497
12.2 对类成员提供线程安全的访问	499
12.3 避免沉默的线程终止	505
12.4 在异步委托完成时获得通知	507
12.5 私有化存储线程特定的数据	509
12.6 使用信号量允许资源的多重访问	512
12.7 使用互斥量同步多个进程	516

12.8	使用事件协调线程	525
12.9	在多线程间执行原子操作	527
12.10	优化以读为主的访问	528
12.11	使数据库请求更具扩展性	541
12.12	以一定顺序运行任务	543
第 13 章	工具箱	549
13.0	简介	549
13.1	处理操作系统关机、电源管理或用户会话变化	549
13.2	控制系统服务	554
13.3	列出加载一个程序集的进程	558
13.4	使用本地工作站上的消息队列	561
13.5	捕获标准输出流的输出	564
13.6	捕获一个进程的标准输出	566
13.7	在它自己的 AppDomain 中运行代码	568
13.8	确定当前操作系统的操作系统和 Service Pack 版本	570
关于作者		572
关于封面		572

前言

C# 是一门面向 Microsoft .NET 平台开发者的语言。Microsoft 将 C# 描述为一种用于 .NET 平台开发的富于创新的现代化语言，并且在 C# 6.0 中增添了用于支持动态编程、并行编程以及编写更少代码的新功能，进一步实现了这个目标。C# 不仅同时支持声明式编程和函数式编程，还包含了强大的面向对象特性。简而言之，用 C# 可以针对特定问题采取不同的编程风格。

我们基于自己最初学习 C# 时遇到的编程问题开始了本书的编写，并且根据 C# 语言中的新问题和新功能不断地扩展内容。在这一版中，我们重新编写了许多解决方案，以充分利用 C# 最近的创新，例如新的表达式级别功能（nameof、字符串插值、null 条件运算符、索引初始值设定项）、成员声明功能（自动属性初始值设定项、getter-only 自动属性、表达式-函数体成员）和语句级别功能（异常过滤器）。同时，我们在原有及新的范例中纳入了动态编程（C# 4.0）和异步编程（C# 5.0）的新应用，帮助读者了解如何应用这些语言特性。

无论是首次学习 C#，还是探索其新的能力，抑或是处理开发周期中较为罕见的问题，每个人都会遇到一些常见（和不那么常见的）陷阱和问题；希望我们的以上增补能帮助读者解决难题。此外，尽管 Microsoft 已经提供了大量的功能以避免人们“重复创造轮子”，但是我们仍然发现 .NET Framework 类库（Framework class library, FCL）中有一些缺少的内容，并将其纳入了本书的范例。一些解决方案你可能马上会用到，也有一些你可能永远都用不到，但是无论如何，我们都希望这本书能够帮助你尽可能充分地利用 C# 和 .NET Framework。

本书的内容按照一个 C# 程序员在学习过程中要解决的问题类型来进行组织。这些解决方案称为范例（recipe）；每个范例都包含一个问题、其解决方案、对解决方案的讨论和其他相关信息，最后是一个资源列表，包括在 FCL 的哪里可以找到相关类的更多信息、相关文章和其他范例。这种问答模式提供了问题的完整解决办法，使得本书易于阅读和使用。几乎每个范例都包含完整的、有文档的代码示例，向读者展示如何解决特定的问题，同时也讨论了底层技术如何运作，并且列出了替代技术、限制条件和应用时要考虑的其他事项。

读者对象

即使你不是经验丰富的 C# 或 .NET 开发人员，也可以使用本书——本书面向所有级别的读

者。本书既提供了开发人员日常问题的解决方案，也包含了一些出现频率较低的问题。书中范例针对的是现实开发人员需要立刻解决的问题，不需要首先学习大量理论知识。虽然参考书和教程类书籍可以讲述通用概念，但通常不提供读者实际问题所需的帮助。我们选择通过示例来教学，因为这是大多数人自然的学习方式。

本书中解决的大多数问题是 C# 开发人员会频繁面对的，但有一些更高级的问题需要结合多项技术的复杂方案。每个范例旨在帮助开发人员快速理解问题，学习如何解决，并找出任何潜在的权衡选择来帮助你快速、高效、轻松地解决问题。

为了免去读者手动输入解决方案的麻烦，我们在 O'Reilly 的网站上提供了本书的示例代码，以方便“编辑继承”模式的开发（复制和粘贴），同时有助于经验较少的开发人员看到优秀的编程实践。示例代码提供了利用到每个解决方案的可运行测试，不过本书也在每个解决方案中包含了足够的代码，读者不使用示例代码也能够实现解决方案。示例代码可以从本书的产品页面（https://github.com/oreillymedia/c_sharp_6_cookbook）上获取¹。

硬件和软件要求

要运行本书中的示例，你需要一台运行 Windows 7 或更高版本的计算机。一部分网络和 XML 解决方案需要 Microsoft IIS 7.5 或更高版本，第 9 章中 FTP 的范例需要一个本地配置好的 FTP 服务器。

打开和编译本书中的示例需要 Visual Studio 2015。如果你精通可下载的 Framework SDK 及其命令行编译器，也可以顺利地使用本书和示例代码。

平台说明

本书中的解决方案都是使用 Visual Studio 2015 开发的。C# 6.0 和 C# 3.0 之间的差异是非常显著的，本书的示例代码与第 3 版中有很大的不同，反映了其中的差异。

值得一提的是，尽管 C# 现在已经是 6.0 版，.NET Framework 则仍是 4.6 版。C# 随着 .NET Framework 的每次发布持续创新，现在的 C# 6.0 中包含许多功能，使得开发人员可以用最适合手头任务的风格进行编程。

内容结构

本书共分为 13 章，每一章侧重于特定主题的 C# 解决方案。下面总结了每一章的要点，概述了本书的内容。

- 第 1 章 类和泛型

这一章篇幅很长，包含处理类和结构数据类型的范例，以及泛型的使用。泛型能够让你的代码在不同类型的值上运行一致。这一章涵盖的范例范围很广，包括闭包、类转换、完善的命令行参数处理系统以及类设计的主题。有的范例增强了读者对泛型的整

注 1：示例代码也可以在图灵社区本书主页（<http://www.ituring.com.cn/book/1746>）下载。——编者注

体理解；有的范例涵盖了泛型何时适用，框架中提供了哪些支持，以及如何实现自定义集合。

- 第2章 集合、枚举器和迭代器

这一章范例考察了集合、枚举器和迭代器的用法。集合的范例使用了数组（单维、多维和锯齿状）、`List<T>`以及其他集合类，并且扩展了它们的功能。这一章讨论了泛型集合以及创建自定义强类型集合的多种方式。我们探讨了自定义枚举器的创建，向读者展示了如何为泛型和非泛型类型实现迭代器以及如何使用迭代器实现 `foreach` 功能，并涵盖了自定义迭代器实现。

- 第3章 数据类型

这一章包括字符串、数字和枚举。这些范例展示了如何完成诸如编码 / 解码字符串、执行数值转换，以及测试字符串以确定它们是否包含数字值的任务。我们还介绍了如何显示、转换和测试枚举类型，以及如何使用包含位标志的枚举。

- 第4章 语言集成查询和 `lambda` 表达式

这一章涵盖语言集成查询（language integrated query, LINQ）及其用法，包括并行 LINQ（parallel LINQ, PLINQ）的一个示例。有些范例使用了诸多标准查询运算符，也有些展示了如何使用功能强大但并不是语言关键字的查询运算符。这一章也对 `lambda` 表达式进行了探讨，通过范例展示如何用其代替旧风格的委托。

- 第5章 调试和异常处理

这一章介绍调试和异常处理。我们提供了使用 `System.Diagnostics` 命名空间下数据类型的范例，例如事件日志、进程、性能计数器和类型的自定义调试器显示。我们同时关注了在应用程序中实现异常处理的最佳方式。这一章还包括了避免未处理异常，读取和显示栈跟踪，引发和重新引发异常的范例。最后，我们提供的范例展示了如何克服某些棘手的情形，如后期绑定调用的方法中出现的异常和异步异常处理。

- 第6章 反射和动态编程

这一章展示了如何使用 .NET Framework 内置的程序集检视系统，确定一个程序集中实现了哪些类型、接口和方法，以及如何以后期绑定的方式访问它们。这一章也说明了如何使用 `dynamic`、`ExpandoObject` 和 `DynamicObject` 在应用程序中实现动态编程。

- 第7章 正则表达式

这一章介绍了一组有用的类，用于对字符串运行正则表达式。范例包括列举正则表达式匹配，将字符串解析为一组标记，查找 / 替换字符，以及验证正则表达式的语法。此外我们还加入了一个范例，其中包含许多常见的正则表达式模式。

- 第8章 文件系统 I/O

这一章涉及与文件系统交互的三种不同方式：典型的文件交互；基于目录或文件夹的交互；文件系统 I/O 的高级主题。

需要整本电子书，联系我QQ：2667271557

- 第9章 网络和 Web

这一章探讨了由 .NET Framework 提供的连接选项，以及如何以编程方式访问网络资源和 Web 上的内容。这一章中的范例包含了直接使用 TCP/IP，使用命名管道通信，构建自己的端口扫描程序，以编程方式确定网站配置，等等。

- 第10章 XML

如果你在使用 .NET，那么很可能需要在一定程度上处理 XML。在这一章中，我们将探讨 XML 的一些作用，以及如何使用 LINQ to XML、`XmlReader/XmlWriter` 类和 `XmlDocument` 类来进行 XML 编程。这一章包含了使用 XPath 和 XSLT 的示例，以及验证 XML、将 XML 转换到 HTML 等主题。

- 第11章 安全

编写不安全代码的方式有很多种，但仅有少数几个途径可以编写安全的代码。在这一章中，我们探讨了类型的访问控制、加密和解密、安全地存储数据、程式安全声明式安全等领域。

- 第12章 线程、同步和并发

这一章的主题是在 .NET 程序中使用多个执行线程，讨论的问题有：在应用实现多线程，避免资源的并行访问，允许安全的并行访问，存储每个线程的数据，顺序执行任务，在 .NET 中使用同步原语以编写线程安全的代码，等等。

- 第13章 工具箱

这一章包含的范例是开发人员会反复遇到的随机操作类型，例如确定系统资源的位置，发送电子邮件，以及使用服务。这一章还包括一些较少用到但非常有用的应用程序块，如消息队列，在单独的应用程序域中运行代码，以及在全局程序集缓存（global assembly cache, GAC）中查找应用程序集的版本。

有一些范例是相关联的；在这类范例中，参阅部分和讨论部分的文本中会注明这些关联关系。

未涉及的内容

这本书并不是 C# 的参考手册或入门书。O'Reilly 出版了一些优秀的入门书和参考手册，如 Joseph Albahari 和 Ben Albahari 的 *C# 6.0 in a Nutshell* (<http://shop.oreilly.com/product/0636920040323.do>) 和 *C# 6.0 Pocket Reference* (<http://shop.oreilly.com/product/0636920040675.do>)，以及 Stephen Cleary 的《C# 并发编程经典实例》(*Concurrency in C# Cookbook*, <http://shop.oreilly.com/product/0636920030171.do>)。MSDN 库也是极为有用的。它包含在 Visual Studio 2015 中，也可以在网站 <http://msdn.microsoft.com> 上在线查看。

排版约定

在本书中将会使用以下排版约定。

- 等宽字体 (*Constant width*)
用于程序清单和代码元素，如命令、选项、开关、变量、特性、键、函数、类型、类、命名空间、方法、模块、属性、参数、值、对象、事件、事件处理器、XML 标记、HTML 标记、宏、文件的内容和命令输出。
- 加粗等宽字体 (**Constant width bold**)
用于在程序清单中突出显示代码中的重要部分。
- 斜体等宽字体 (*Constant width italic*)
用于指示代码中可替换的部分。
- `//...`
C# 代码中的省略号表示为了段落清晰而省略掉的文本。
- `<!--.....-->`
在 XML 模式和文档的代码中的省略号表示为了段落清晰而省略掉的文本。



此标志指示提示、建议或一般注意事项。



此标志表示警告或警示。

关于代码

本书中几乎每个范例都包含一个或多个代码示例。这些示例包含在解决方案中，代码片段和完整项目都可以直接用于你的应用程序。大多数代码示例都写在一个类或结构中，使得它们更易于在应用程序中使用。除此之外，所有的 `using` 指令都包含在各个范例中，因此你不需要查明在你的代码中要包含哪个命名空间。

只有关键的部分才包括完整的错误处理，例如输入参数。这允许你轻松地查看什么是正确的输入，什么是错误的输入。许多范例省略了错误处理，关注重点概念会使得解决方案更易于理解。

使用代码示例

这本书的示例代码可从网页 https://github.com/oreillymedia/c_sharp_6_cookbook 上获取。

需要整本电子书，联系我QQ：2667271557

这本书是用来帮助你完成工作的。一般来说，你可以在自己的程序和文档中使用本书的代码。你没有必要联系我们来获得授权，除非想对代码做出大规模的重构。比如，使用本书中的若干程序来编写自己的代码是无需授权的，但是销售或分发 O'Reilly 出版书籍配套光盘中的代码是需要授权的；引用本书中的内容或示例代码来回答问题是无需授权的；而把本书中的大量代码合并到你的产品文档中就需要授权。

我们并不要求你注明引用内容的出处，但是非常感激你这么。一条引用说明通常包括书名、作者、出版商和 ISBN。例如，“*C# 6.0 Cookbook*, Fourth Edition, by Jay Hilyard and Stephen Teilhet. Copyright ©2015 Jay Hilyard and Stephen Teilhet, 978-1-4919-2146-3”。

如果你感觉你使用代码示例的方式不属于以上所述的任何方式，可以随时与我们联系，我们的电子邮箱地址为 permissions@oreilly.com。

Safari®在线图书



Safari 在线图书 (<http://safaribooksonline.com/>) 是一个基于用户需求，发行全球技术和商业领域顶级作者的优质图书和视频 (<https://www.safaribooksonline.com/explore/>) 的数字图书馆。

技术专家、软件开发人员、网页设计师以及商业和创意专家都选择将 Safari 在线图书作为研究、解决问题、学习和证书培训的首要资源。

Safari 在线图书为企业 (<https://www.safaribooksonline.com/enterprise/>)、政府 (<https://www.safaribooksonline.com/government/>)、教育机构 (<https://www.safaribooksonline.com/academic-public-library/>) 和个人提供了不同的产品组合和价格方案 (<https://www.safaribooksonline.com/pricing/>)。

用户可通过 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett 和 Course Technology 等数百个出版商的数据库 (<https://www.safaribooksonline.com/our-library/>) 搜寻上千种图书、培训视频和预出版的手稿。要了解有关 Safari 在线图书的更多信息，请访问我们的在线网站 (<http://safaribooksonline.com/>)。

联系我们

请把对本书的评价和问题发给出版社。

美国：

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

需要整本电子书，联系我QQ：2667271557

中国：

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

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

<http://shop.oreilly.com/product/0636920037347.do>

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

<http://www.oreilly.com>

我们在 Facebook 的地址如下：<http://facebook.com/oreilly>

请关注我们的 Twitter 动态：<http://twitter.com/oreillymedia>

我们的 YouTube 视频地址如下：<http://www.youtube.com/oreillymedia>

致谢

这本书始于我们最初接触 C#，并在我们多年来对该语言的探索和创新应用中演变成长。自本书的上一版出版之后，随着 C# 6.0 的发布以及 C# 4.0 和 C# 5.0 的新功能出现，我们决定复查前三个版本以确认如何改进原有的范例，并探索使用 C# 更好地完成编程任务的方法。通过对 C# 和 Framework 大量知识的不断学习，我们致力于在本书中为读者呈现 C# 是如何演变的，以及如何使用它更好地完成工作。

以下这些人对本书的完成具有不可或缺的作用，我们在此对他们给予的帮助表示感谢。

感谢 Brian MacDonald（我们的编辑）、Heather Scherer、Rachel Monaghan、Nick Adams 和 Sara Peyton。在你们的督促下本书才得以快速完成并出版。感谢你们的付出。

同时我们还要感谢我们的技术复审小组成员：Steve Munyan、Lee Coward 和 Nick Pinkham。感谢你们对完善本书的全身心投入和富有洞察力的见解。没有你们有价值的反馈，这本书就不可能存在，感谢你们。

Jay Hilyard 的致谢

感谢 Steve Teihet 的创意、幽默感以及愿意再次与我踏上写书的征途。我一直很享受与你共事，尽管都是在晚上和周末，并且几乎都是在网络上。

感谢我的妻子 Brooke。尽管明白写书意味着我们相处时间的减少，但你依然支持我、鼓励我、帮助我。这本书没有你就不可能诞生。谢谢你，我爱你！

感谢我的两个儿子，Owen 和 Drew。你们从新角度看待事物的能力每每让我惊讶，我为你们所完成的一切感到骄傲。我很高兴你们都对我为之奋斗一生的领域感兴趣，你们是我无可替代的珍宝。

需要整本电子书，联系我QQ：2667271557

感谢 Phil 和 Gail 对我不得不牺牲假期而工作的支持和理解，并肩负起祖父母的责任帮助我照顾孩子，同时还要感谢妈妈每个月的心灵鸡汤。

感谢我的一群好友：Seth、Katie Fiermonti、Tom Bebbington 和 Jenna Roberts。世上再无难事，只要有朋友和一杯好酒。

感谢 Scott Cronshaw、Bill Bolevic、Melissa Jurkoic、Mike Kennie、Alex Shore、Dave Flanders、Aaron Reddish、Rakshit Jain、Jason Phelps、Josh Clairmont、Bob Blais、Kim Serpa、Stu Savage、Gaurang Patel、Jesse Peters、Ken Jones、Mahesh Unnikrishnan、T Antonio、Mary Ellen Sawyer、Jon Godbout、Atul Kaul、Mark Miller、Rich Labenski、Lance Simpson、Tim Beaulieu 和 Lee Horgan。你们是最棒的团队，你们工作都非常努力，我感谢你们所做的一切。

最后，再次感谢我的家人和朋友。你们对一本自己并不了解的书表示了极大的关心，并为我感到骄傲。

Steve Teilhet的致谢

我很骄傲能与 Jay Hilyard 成为好友，他是一个出色的合作者，一个勤奋的合著者。不是每天都能找到一个既值得信任又能合作无间的好朋友。再次与你合著是我的荣幸。

感谢我的妻子 Kandis Teilhet 给了我坚持前行每一步的力量。我对你的爱已尽在不言中。感谢我的两个儿子 Patrick 和 Nicholas，是你们让艰难的时刻变得顺利。你们对我来说无可替代。现在你们都步入了人生的下一个阶段，我为能看到你们即将获取的成就感到激动，没准你们也会写一本书。

感谢我的妈妈、爸爸和兄弟，感谢他们一直以来的倾听和支持。

最后同样要感谢的是 IBM 团队、Larry Rose、Babita Sharma、Jessica Berliner、Jeff Turnham、John Peyton、Kris Duer、Robert Stanzel、Shu Wang、Bingzhou Zheng、Dave Steinberg、Dave Stewart、Jason Todd、Alexei Pivkine、Joshua Clark、William Frontiero、Matthew Murphy、Omer Trip、Marco Pistoia、Enrique Varillas、Guillermo Hurtado、Bao Lu、Mary Santo、Diane Redfearn、Urmi Chatterjee、Joshua Ho、Kenneth Cheung、Andrew Mak、Daniel Nguyen、Jennifer Calder、Tahseen Shabab、Srinivas Sripada、David Marshak、Larry Gerard、Douglas Wilson、Steve Hikida 以及其他许多人，你们的辛勤工作和才华一直激励着我。

电子书

扫描如下二维码，即可购买本书电子版。



第 1 章

类和泛型

1.0 简介

本章的范例涵盖了 C# 语言的基础，主题包括类和结构，如何使用它们，它们有哪些不同，何时使用类以及何时使用结构。在此基础上，我们将构建具有各种固有功能（如可排序、可搜索、可处理和可克隆）的类。此外，我们将深入讨论联合类型、字段初始化、lambda、局部方法、单路和多路广播委托、闭包、函数对象等主题。本章也包含了解析命令行参数的范例，这是开发人员一直喜爱的主题。

在开始展示这些范例之前，让我们回顾一下关于类、结构、泛型的面向对象能力的关键信息。类比结构灵活得多。结构可以跟类一样实现接口，但与类不同的是，它们不能继承自类或结构。这种限制使得你无法创建结构层次关系，而这用类可以做到。通过抽象基类实现的多态性也是在结构中无法使用的，因为除了装箱成 `Object`、`ValueType` 和 `Enum`，结构无法从另一个类派生。

结构与其他值类型一样，都是从 `System.ValueType` 隐式派生的。乍看之下，结构类似于类，但它们实际上有很大的差别。在设计应用程序时，知道何时使用结构优于使用类将对你有很大的帮助。不正确地使用结构可能会使代码性能低下、难以修改。

结构相对于引用类型有两个性能优势。首先，如果一个结构是在栈上分配的（即不包含在引用类型内），访问结构及其数据的速度要快于访问堆中引用类型的速度。

引用类型的对象必须要跟随它在堆上的引用以获取它们的数据。不过，这种性能优势相对于结构的第二个性能优势就相形见绌了：要清理在栈上为结构分配的内存，只需要在方法调用返回时修改栈指针所指向的地址即可。这个调用要远远快于垃圾回收器自动清理托管堆上分配的引用类型。然而垃圾回收器的成本是延后的，所以不会立刻被人注意到。

当以传值方式传入其他方法时，结构的性能就比不上类了。因为结构存在于栈上，当以传值方式传入一个方法时，结构及其数据必须复制到一个新的局部变量（方法用于接收结构的参数）中。这一复制过程相比将一个引用传入方法要花费更多的时间，除非结构的大小与机器的指针大小相同或更小一些；因此，在 32 位的机器上，传入一个 32 位大小的结构与传入一个引用（与指针大小相同）的成本是相同的。在类和结构之间选择时，要记得这一点。尽管创建、访问和销毁类对象可能需要更长时间，但并不能抵消将结构多次按值传入一个或多个方法产生的性能下降。保持较小的结构体可以减小按值传递时所产生的性能下降。

以下情况应使用类。

- 其同一性很重要。结构在按值传入方法时会被隐式复制。
- 有较大的内存占用。
- 其字段需要初始化。
- 需要从一个基类继承。
- 需要多态行为；也就是说，你需要实现一个抽象基类，并从此基类派生出多个相似的类。（注意，多态性也可以通过接口实现，但通常并不适合在一个值类型中实现接口。这是因为当结构转换为接口时，会因装箱操作而导致性能损失。）

以下情况应使用结构。

- 其行为方式类似于原语类型（int、long、byte 等）。
- 仅占用较小的内存。
- 调用一个需要将结构体以传值方式传入的 P/Invoke 方法。平台调用（Platform Invoke，P/Invoke）允许托管代码调用 DLL 内公开的非托管方法。许多时候，非托管 DLL 内的方法都需要传入一个结构参数。使用结构是执行此操作的一种高效方法，并且在需要按值传入时是唯一的途径。
- 需要降低垃圾回收对应用程序性能的影响。
- 其字段只需要被初始化为默认值。对于数值类型，这个值为 0；对于布尔类型，则为 false；对于引用类型，则为 null。注意在 C# 6.0 中，结构可以拥有默认构造函数并将字段初始化为非默认值。
- 不需要继承一个基类（除了 ValueType 之外，所有结构都继承它）。
- 不需要多态行为。

当把结构传递给需要一个对象参数的方法时，例如 Framework 类库（FCL）中的任何非泛型集合类型，它们也可能会引起性能降低。把一个结构（对此问题而言其实是任何简单类型）传入一个需要对象参数的方法中将会导致结构被装箱。装箱（boxing）是指将一个值类型包装在一个对象中。这种操作比较耗时，并且可能导致性能降低。

最后，将泛型功能加入进来就能够编写类型安全且高效的基于集合和模式的代码了。泛型提供相当强大的编程能力，但是要求你正确使用它。如果你考虑把 ArrayList、Queue、Stack 和 Hashtable 对象转换成其对应的泛型对象，可以阅读一下 1.9 节和 1.10 节中的范例。你将看到这种转换并非总是很简单，有一些原因可能导致你根本不想执行这种转换。

1.1 创建联合类型的结构

1.1.1 问题

你需要创建一种数据类型，其行为方式类似于 C++ 中的联合类型。联合类型主要用于互操作场景，其中非托管代码接受和 / 或返回一个联合类型；我们建议你不要在其他情况下使用它。

1.1.2 解决方案

使用一个结构，并用 `StructLayout` 特性标记它（在构造函数中指定 `LayoutKind.Explicit` 布局类型）。此外，利用 `FieldOffset` 特性标记结构中的每个字段。下面的结构定义了一个联合类型，其中可以存储一个带符号数值。

```
using System.Runtime.InteropServices;
[StructLayoutAttribute(LayoutKind.Explicit)]
struct SignedNumber
{
    [FieldOffsetAttribute(0)]
    public sbyte Num1;
    [FieldOffsetAttribute(0)]
    public short Num2;
    [FieldOffsetAttribute(0)]
    public int Num3;
    [FieldOffsetAttribute(0)]
    public long Num4;
    [FieldOffsetAttribute(0)]
    public float Num5;
    [FieldOffsetAttribute(0)]
    public double Num6;
}
```

下一个结构类似于 `SignedNumber` 结构，不同之处是除了带符号的数值之外，它还可以包含 `String` 类型。

```
[StructLayoutAttribute(LayoutKind.Explicit)]
struct SignedNumberWithText
{
    [FieldOffsetAttribute(0)]
    public sbyte Num1;
    [FieldOffsetAttribute(0)]
    public short Num2;
    [FieldOffsetAttribute(0)]
    public int Num3;
    [FieldOffsetAttribute(0)]
    public long Num4;
    [FieldOffsetAttribute(0)]
    public float Num5;
    [FieldOffsetAttribute(0)]
    public double Num6;
    [FieldOffsetAttribute(16)]
    public string Text;
}
```

```
    public string Text1;  
}
```

1.1.3 讨论

联合类型是一种在 C++ 代码中较为常见的结构类型；不过，有一种方式可以使用 C# 中的结构数据类型来复制其结构。联合（union）是一种结构，在内存中的特定位置为该结构接受多种类型。例如，SignedNumber 结构是使用 C# 结构创建的一个联合类型的结构。这种结构可以接受任何类型的带符号的数值类型（sbyte、int 和 long 等），但它只在结构中的同一个位置（同一偏移量）接受这种数字类型。



由于 StructLayoutAttribute 可以同时应用于结构和类，在创建联合数据类型时也可以使用类。

注意 FieldOffsetAttribute 将值 0 传递给它的构造函数。这表明这个字段距离结构开始处的偏移量为 0 字节。可以将这个特性与 StructLayoutAttribute 结合使用，手动强制指定结构中的字段开始于什么位置（即每个字段在内存中相对于这个结构开始处的偏移量）。FieldOffsetAttribute 只能与设置为 LayoutKind.Explicit 的 StructLayoutAttribute 一起使用。此外，它不能用于结构内的静态成员。

联合类型可能会带来一些问题，因为几种类型实质上是相互叠加在一起的。最大的问题是如何从联合类型结构中提取正确的数据类型。思考一下，如果你选择在 SignedNumber 结构中存储 long 数值类型的值 long.MaxValue，会发生什么情况。随后，你可能会偶然尝试从这个结构中提取一个 byte 数据类型值。这样操作，你将会只取回这个 long 值中的第一字节。

另一个问题是在正确的偏移位置开始字段。SignedNumberWithText 联合类型在偏移量为 0 的位置叠加了大量带符号的数值数据类型。这个结构中的最后一个字段位于内存中距离这个结构开始处偏移量为 16 字节的位置。如果你意外地把字符串字段 Text1 覆盖在任何其他带符号的数值数据类型之上，在运行时将得到一个异常。基本规则是：允许你把一种值类型叠加在另一种值类型之上，但是不能把一种引用类型叠加于一种值类型之上。如果用以下特性标记 Text1 字段：

```
[FieldOffsetAttribute(14)]
```

就会在运行时引发下面这个异常（注意，编译器不会捕获这个问题）。

```
System.TypeLoadException: Could not load type 'SignedNumberWithText' from  
assembly 'CSharpRecipes, Version=1.0.0.0, Culture=neutral,  
PublicKeyToken=fe85c3941fbcc4c5' because it contains an object field at  
offset 14 that is incorrectly aligned or overlapped by a non-object field.
```

在 C# 中使用复杂的联合类型时，必须保证正确的偏移量。

1.1.4 参考

MSDN 文档中的“StructLayoutAttribute 类”主题。

1.2 使类型可排序

1.2.1 问题

你有一种数据类型，它将存储为 `List<T>` 或 `SortedList<K,V>` 的元素。你想使用 `List<T>.Sort` 方法或者 `SortedList<K,V>` 的内部排序机制来自定义此数据类型在数组中的排序方式。此外，你可能需要在 `SortedList` 集合中使用这种类型。

1.2.2 解决方案

例 1-1 演示了如何实现 `IComparable<T>` 接口。例 1-1 中展示的 `Square` 类实现了这个接口，使得 `List<T>` 和 `SortedList<K,V>` 集合能够排序和查找这些 `Square` 对象。

例 1-1：通过实现 `IComparable<T>` 使类型可排序

```
public class Square : IComparable<Square>
{
    public Square(){}

    public Square(int height, int width)
    {
        this.Height = height;
        this.Width = width;
    }

    public int Height { get; set; }

    public int Width { get; set; }

    public int CompareTo(object obj)
    {
        Square square = obj as Square;
        if (square != null)
            return CompareTo(square);
        throw
            new ArgumentException(
                "Both objects being compared must be of type Square.");
    }

    public override string ToString()=>
        ($"Height: {this.Height}      Width: {this.Width}");

    public override bool Equals(object obj)
    {
        if (obj == null)
            return false;
    }
}
```

```
        Square square = obj as Square;
        if(square != null)
            return this.Height == square.Height;
        return false;
    }

    public override int GetHashCode()
    {
        return this.Height.GetHashCode() | this.Width.GetHashCode();
    }

    public static bool operator ==(Square x, Square y) => x.Equals(y);
    public static bool operator !=(Square x, Square y) => !(x == y);
    public static bool operator <(Square x, Square y) => (x.CompareTo(y) < 0);
    public static bool operator >(Square x, Square y) => (x.CompareTo(y) > 0);

    public int CompareTo(Square other)
    {
        long area1 = this.Height * this.Width;
        long area2 = other.Height * other.Width;

        if (area1 == area2)
            return 0;
        else if (area1 > area2)
            return 1;
        else if (area1 < area2)
            return -1;
        else
            return -1;
    }
}
```

1.2.3 讨论

通过在类（或结构）上实现 `IComparable<T>` 接口，就可以利用 `List<T>` 和 `SortedList<K,V>` 类的排序例程。排序算法内置在这些类中；你只需要通过在 `IComparable<T>.CompareTo` 方法中实现的代码告诉它们如何对你的类进行排序即可。

当调用 `List<Square>.Sort` 方法对 `Square` 对象的列表进行排序时，列表是通过 `Square` 对象的 `IComparable<Square>` 接口进行排序的。当把对象添加到 `SortedList<K,V>` 中时，`SortedList<K,V>` 类的 `Add` 方法使用这个接口对它们进行排序。

`IComparer<T>` 设计用于解决如下问题：允许基于不同环境中的不同标准对对象进行排序。这个接口还允许你对其他人编写的类型进行排序。如果你还想按高度对 `Square` 对象进行排序，就可以创建一个名为 `CompareHeight` 的新类，如例 1-2 中所示。它也实现了 `IComparer<Square>` 接口。

例 1-2：通过实现 `IComparer` 使类型可排序

```
public class CompareHeight : IComparer<Square>
{
    public int Compare(object firstSquare, object secondSquare)
```



```
{
    Square square1 = firstSquare as Square;
    Square square2 = secondSquare as Square;
    if (square1 == null || square2 == null)
        throw (new ArgumentException("Both parameters must be of type Square."));
    else
        return Compare(firstSquare,secondSquare);
}

#region IComparer<Square> Members

public int Compare(Square x, Square y)
{
    if (x.Height == y.Height)
        return 0;
    else if (x.Height > y.Height)
        return 1;
    else if (x.Height < y.Height)
        return -1;
    else
        return -1;
}

#endregion
}
```

然后将这个类传入 Sort 方法的 IComparer 参数。现在你可以指定以不同的方式对 Square 对象进行排序。比较器中实现的比较方法必须保持一致并应用全局排序，从而使得比较函数声明两个数据项相等时绝对正确，而不是以下情况的结果：一个数据项不大于另一个数据项或者一个数据项不小于另一个数据项。



为了获得最佳性能，需要保持 CompareTo 方法短小、高效，因为它将被 Sort 方法调用多次。例如，在对含有 4 个数据项的数组排序时，Compare 方法将被调用 10 次。

例 1-3 中展示的 TestSort 方法演示了如何对 List<Square> 和 SortedList<int,Square> 实例使用 Square 和 CompareHeight 类。

例 1-3: TestSort 方法

```
public static void TestSort()
{
    List<Square> listOfSquares = new List<Square>(){
        new Square(1,3),
        new Square(4,3),
        new Square(2,1),
        new Square(6,1)};

    // 测试List<String>
    Console.WriteLine("List<String>");
    Console.WriteLine("Original list");
    foreach (Square square in listOfSquares)
```

```
{
    Console.WriteLine(square.ToString());
}

Console.WriteLine();
IComparer<Square> heightCompare = new CompareHeight();
listOfSquares.Sort(heightCompare);
Console.WriteLine("Sorted list using IComparer<Square>=heightCompare");
foreach (Square square in listOfSquares)
{
    Console.WriteLine(square.ToString());
}

Console.WriteLine();
Console.WriteLine("Sorted list using IComparable<Square>");
listOfSquares.Sort();
foreach (Square square in listOfSquares)
{
    Console.WriteLine(square.ToString());
}

// 测试SortedList
var sortedListOfSquares = new SortedList<int, Square>(){
    { 0, new Square(1,3)},
    { 2, new Square(3,3)},
    { 1, new Square(2,1)},
    { 3, new Square(6,1)}};

Console.WriteLine();
Console.WriteLine();
Console.WriteLine("SortedList<Square>");
foreach (KeyValuePair<int, Square> kvp in sortedListOfSquares)
{
    Console.WriteLine($"{kvp.Key} : {kvp.Value}");
}
}
```

这些代码的输出如下所示。

```
List<String>
Original list
Height:1 Width:3
Height:4 Width:3
Height:2 Width:1
Height:6 Width:1

Sorted list using IComparer<Square>=heightCompare
Height:1 Width:3
Height:2 Width:1
Height:4 Width:3
Height:6 Width:1

Sorted list using IComparable<Square>
Height:2 Width:1
Height:1 Width:3
```

```
Height:6 Width:1
Height:4 Width:3
```

```
SortedList<Square>
0 : Height:1 Width:3
1 : Height:2 Width:1
2 : Height:3 Width:3
3 : Height:6 Width:1
```

1.2.4 参考

范例 1.3（即 1.3 节）；MSDN 文档中的“`IComparable<T>` 接口”主题。

1.3 使类型可查找

1.3.1 问题

你有一种数据类型，它将存储为 `List<T>` 中的元素。你想使用 `BinarySearch` 方法，自定义你的数据类型在列表中的查找方式。

1.3.2 解决方案

使用 `IComparable<T>` 和 `IComparer<T>` 接口。范例 1.2（即 1.2 节）中的 `Square` 类实现了 `IComparable<T>` 接口，使得 `List<T>` 和 `SortedList<K,V>` 集合可以排序和查找 `Square` 对象的数组和集合。

1.3.3 讨论

通过在类（或结构）上实现 `IComparable<T>` 接口，就可以利用 `List<T>` 和 `SortedList<K,V>` 类的排序例程。排序算法内置在这些类中；你只需要通过在 `IComparable<T>.CompareTo` 方法中实现的代码告诉它们如何对你的类进行排序即可。

要实现 `CompareTo` 方法，请参考范例 1.2（即 1.2 节）。

`List<T>` 类提供了一个 `BinarySearch` 方法来查找该列表中的元素。列表中的元素会与传递给对象参数中的 `BinarySearch` 方法的某个对象进行比较。`SortedList` 类没有 `BinarySearch` 方法；作为替代，它拥有 `ContainsKey` 方法，用于对列表中包含的键值执行二分查找。`SortedList` 类的 `ContainsValue` 方法在查找值时执行线性查找。这种线性查找使用 `SortedList` 集合中的元素的 `Equals` 方法来执行其工作。`Compare` 和 `CompareTo` 方法对于 `SortedList` 类中执行的线性查找不起任何作用，但是它们确实会影响二分查找。



为了使用 `List<T>` 类的 `BinarySearch` 方法执行准确的查找，首先必须使用 `List<T>` 的 `Sort` 方法对其进行排序。此外，如果把一个 `IComparer<T>` 接口传入给 `BinarySearch` 方法，还必须把相同的接口传递给 `Sort` 方法。否则，`BinarySearch` 方法也许无法找到你正在寻找的对象。

例 1-4 中的 TestSort 方法演示了如何对 List<Square> 和 SortedList<int,Square> 集合实例使用 Square 和 CompareHeight 类。

例 1-4：使类型可查找

```
public static void TestSearch()
{
    List<Square> listOfSquares = new List<Square> {new Square(1,3),
                                                    new Square(4,3),
                                                    new Square(2,1),
                                                    new Square(6,1)};

    IComparer<Square> heightCompare = new CompareHeight();

    // 测试List<Square>
    Console.WriteLine("List<Square>");
    Console.WriteLine("Original list");
    foreach (Square square in listOfSquares)
    {
        Console.WriteLine(square.ToString());
    }

    Console.WriteLine();
    Console.WriteLine("Sorted list using IComparer<Square>=heightCompare");
    listOfSquares.Sort(heightCompare);
    foreach (Square square in listOfSquares)
    {
        Console.WriteLine(square.ToString());
    }

    Console.WriteLine();
    Console.WriteLine("Search using IComparer<Square>=heightCompare");
    int found = listOfSquares.BinarySearch(new Square(1,3), heightCompare);
    Console.WriteLine($"Found (1,3): {found}");

    Console.WriteLine();
    Console.WriteLine("Sorted list using IComparable<Square>");
    listOfSquares.Sort();
    foreach (Square square in listOfSquares)
    {
        Console.WriteLine(square.ToString());
    }

    Console.WriteLine();
    Console.WriteLine("Search using IComparable<Square>");
    found = listOfSquares.BinarySearch(new Square(6,1)); // 使用 IComparable
    Console.WriteLine($"Found (6,1): {found}");

    // 测试SortedList<Square>
    var sortedListOfSquares = new SortedList<int,Square>(){
        {0, new Square(1,3)},
        {2, new Square(4,3)},
        {1, new Square(2,1)},
        {4, new Square(6,1)}};

    Console.WriteLine();
```

```
Console.WriteLine("SortedList<Square>");
foreach (KeyValuePair<int, Square> kvp in sortedListOfSquares)
{
    Console.WriteLine($"{kvp.Key} : {kvp.Value}");
}

Console.WriteLine();
bool foundItem = sortedListOfSquares.ContainsKey(2);
Console.WriteLine($"sortedListOfSquares.ContainsKey(2): {foundItem}");

// 不要使用IComparer和IComparable
// -- 使用未重写过的Equals方法实现线性查找
Console.WriteLine();
Square value = new Square(6,1);
foundItem = sortedListOfSquares.ContainsValue(value);
Console.WriteLine("sortedListOfSquares.ContainsValue " +
    $"{(new Square(6,1)): {foundItem}}");
}
```

这段代码显示的结果如下所示。

```
List<Square>
Original list
Height:1 Width:3
Height:4 Width:3
Height:2 Width:1
Height:6 Width:1

Sorted list using IComparer<Square>=heightCompare
Height:1 Width:3
Height:2 Width:1
Height:4 Width:3
Height:6 Width:1

Search using IComparer<Square>=heightCompare
Found (1,3): 0

Sorted list using IComparable<Square>
Height:2 Width:1
Height:1 Width:3
Height:6 Width:1
Height:4 Width:3

Search using IComparable<Square>
Found (6,1): 2

SortedList<Square>
0 : Height:1 Width:3
1 : Height:2 Width:1
2 : Height:4 Width:3
4 : Height:6 Width:1

sortedListOfSquares.ContainsKey(2): True
sortedListOfSquares.ContainsValue(new Square(6,1)): True
```

1.3.4 参考

范例 1.2（即 1.2 节）；MSDN 文档中的“`Comparable<T>` 接口”和“`Comparer<T>` 接口”主题。

1.4 从一个方法返回多个数据项

1.4.1 问题

在许多情况下，从一个方法返回一个值是不够的。你需要一种方式来从一个方法返回不止一个数据项。

1.4.2 解决方案

对充当返回参数的参数使用关键字 `out`。下面的方法接受一个 `inputShape` 参数，并通过该值计算 `height`、`width` 和 `depth`。

```
public void ReturnDimensions(int inputShape,
                             out int height,
                             out int width,
                             out int depth)
{
    height = 0;
    width = 0;
    depth = 0;

    // 通过inputShape值计算height,width和depth
}
```

这个方法以如下方式进行调用：

```
// 声明输出参数
int height;
int width;
int depth;

// 调用方法并返回height,width和depth
Obj.ReturnDimensions(1, out height, out width, out depth);
```

另一个方法将返回一个包含所有返回值的类或结构。修改前一个方法，使其返回一个结构，而不是使用 `out` 参数：

```
public Dimensions ReturnDimensions(int inputShape)
{
    // 默认构造函数自动将结构的成员初始化为0
    Dimensions objDim = new Dimensions();

    // 通过inputShape的值计算objDim.Height,objDim.Width,objDim.Depth……

    return objDim;
}
```

其中 `Dimensions` 的定义如下所示。

```
public struct Dimensions
{
    public int Height;
    public int Width;
    public int Depth;
}
```

现在以如下方式调用这个方法。

```
// 调用方法并且返回height、width和depth
Dimensions objDim = obj.ReturnDimensions(1);
```

除了从此方法返回一个用户定义类或结构，也可以用一个 `Tuple` 对象包含所有的返回值。修改前一个方法，使其返回一个 `Tuple`。

```
public Tuple<int, int, int> ReturnDimensionsAsTuple(int inputShape)
{
    // 通过inputShape值计算objDim.Height、objDim.Width、objDim.Depth
    // 例如{5, 10, 15}

    // 创建一个包含计算出的值的Tuple
    var objDim = Tuple.Create<int, int, int>(5, 10, 15);

    return (objDim);
}
```

现在以如下方式调用这个方法。

```
// 调用方法并且返回height、width和depth
Tuple<int, int, int> objDim = obj.ReturnDimensions(1);
```

1.4.3 讨论

在方法签名中使用 `out` 关键字创建一个参数，指示这个参数将由该方法初始化并返回。当需要方法返回多个值时，这个技巧就很有用。一个方法最多只能有一个返回值，但是通过使用 `out` 关键字，可以把多个参数标记为一个返回值。

要设置一个 `out` 参数，需要用 `out` 关键字标记方法签名中的参数，如下所示。

```
public void ReturnDimensions(int inputShape,
                             out int height,
                             out int width,
                             out int depth)
{
    ...
}
```

要调用这个方法，还必须用 `out` 关键字标记调用方法的参数，如下所示。

```
obj.ReturnDimensions(1, out height, out width, out depth);
```

这个方法中的 `out` 参数不必初始化；只需声明它们并传入 `ReturnDimensions` 方法中即可。

不管在调用方法之前是否初始化过它们，在 `ReturnDimensions` 方法内使用它们之前都必须初始化。即使不通过 `ReturnDimensions` 方法内的每条路径使用它们，仍然必须初始化它们。这就是这个方法以如下三行代码开始的原因。

```
height = 0;
width = 0;
depth = 0;
```

你可能想知道为什么不能使用 `ref` 参数代替 `out` 参数，鉴于它们都允许一个方法改变像这样标记的参数的值。答案是，`out` 参数使代码有些自文档化。当遇到一个 `out` 参数时，你知道这个参数充当一个返回值。此外，在把 `out` 参数传入方法中之前，不需要做额外的工作来初始化它；而 `ref` 参数则需要这样做。



在调用方法时不需要对 `out` 参数进行封送；相反，在方法把数据返回给调用者时对其封送一次。任何其他调用类型（按值调用或者使用 `ref` 关键字按引用调用）都要求在两个方向上对值进行封送。在封送场合下使用 `out` 关键字可以改进远程调用性能。

在仅有少量值需要返回时，`out` 参数是非常有用的；但是当你遇到需要返回 4 个、5 个、6 个甚至更多的值时，它就变得笨重了。另外一个返回多个值的选项是创建并返回用户定义类或结构，或者使用 `Tuple` 打包需要由某个方法返回的所有值。

使用类或结构返回多个值的第一个选项非常直接。只需要像下面这样创建类型（在本例中是该类型是一个结构）即可。

```
public struct Dimensions
{
    public int Height;
    public int Width;
    public int Depth;
}
```

如 1.4.2 节所展示的，将需要的数据填充到这个数据结构的每个字段中，并且从方法中返回它。

与使用用户定义的对象相比，使用 `Tuple` 的第二个选项更加简洁。可以创建一个 `Tuple`，用于包含不同类型的任意数量的值。此外，`Tuple` 中保存的数据是不可变的；一旦通过构造函数或者静态的 `Create` 方法将数据添加到 `Tuple` 中，就无法再修改这些数据了。

`Tuple` 可以接受并包含 8 个独立的值。如里你需要 8 个以上的值，那么需要使用这个特别的 `Tuple` 类。

```
Tuple<T1, T2, T3, T4, T5, T6, T7, TRest> Class
```

当创建一个包含超过 8 个值的 `Tuple` 时，你无法使用静态的 `Create` 方法，而是必须使用 `Tuple` 类的构造函数。下面的代码展示了如何创建一个包含 10 个整数值的 `Tuple`。

```
var values = new Tuple<int, int, int, int, int, int, int, int, Tuple<int, int, int>> (
    1, 2, 3, 4, 5, 6, 7, new Tuple<int, int, int> (8, 9, 10));
```


当然，你可以继续将更多的 Tuple 添加到每个内嵌的 Tuple 类的最后，以创建你需要的任何大小的 Tuple。

1.4.4 参考

MSDN 文档中的“Tuple 类”和“Tuple<T1, T2, T3, T4, T5, T6, T7, TRest> 类”主题。

1.5 解析命令行参数

1.5.1 问题

你需要应用程序以标准格式（在 1.5.3 节中介绍）接受一个或多个命令行参数。你需要访问和解析传递给应用程序的完整命令行。

1.5.2 解决方案

在例 1-5 中，结合使用以下类来帮你解析命令行参数：Argument、ArgumentDefinition 和 ArgumentSemanticAnalyzer。

例 1-5: Argument 类

```
using System;
using System.Diagnostics;
using System.Linq;
using System.Collections.ObjectModel;

public sealed class Argument
{
    public string Original { get; }
    public string Switch { get; private set; }
    public ReadOnlyCollection<string> SubArguments { get; }
    private List<string> subArguments;
    public Argument(string original)
    {
        Original = original;
        Switch = string.Empty;
        subArguments = new List<string>();
        SubArguments = new ReadOnlyCollection<string>(subArguments);
        Parse();
    }

    private void Parse()
    {
        if (string.IsNullOrEmpty(Original))
        {
            return;
        }
        char[] switchChars = { '/', '-' };
        if (!switchChars.Contains(Original[0]))
        {

```

```
        return;
    }

    string switchString = Original.Substring(1);
    string subArgsString = string.Empty;
    int colon = switchString.IndexOf(':');
    if (colon >= 0)
    {
        subArgsString = switchString.Substring(colon + 1);
        switchString = switchString.Substring(0, colon);
    }
    Switch = switchString;
    if (!string.IsNullOrEmpty(subArgsString))
        subArguments.AddRange(subArgsString.Split(';'));
}

// 一组谓词,提供关于参数的有用信息
// 使用lambda表达式实现
public bool IsSimple => SubArguments.Count == 0;
public bool IsSimpleSwitch =>
    !string.IsNullOrEmpty(Switch) && SubArguments.Count == 0;
public bool IsCompoundSwitch =>
    !string.IsNullOrEmpty(Switch) && SubArguments.Count == 1;
public bool IsComplexSwitch =>
    !string.IsNullOrEmpty(Switch) && SubArguments.Count > 0;
}

public sealed class ArgumentDefinition
{
    public string ArgumentSwitch { get; }
    public string Syntax { get; }
    public string Description { get; }
    public Func<Argument, bool> Verifier { get; }

    public ArgumentDefinition(string argumentSwitch,
                              string syntax,
                              string description,
                              Func<Argument, bool> verifier)
    {
        ArgumentSwitch = argumentSwitch.ToUpper();
        Syntax = syntax;
        Description = description;
        Verifier = verifier;
    }

    public bool Verify(Argument arg) => Verifier(arg);
}

public sealed class ArgumentSemanticAnalyzer
{
    private List<ArgumentDefinition> argumentDefinitions =
        new List<ArgumentDefinition>();
    private Dictionary<string, Action<Argument>>> argumentActions =
        new Dictionary<string, Action<Argument>>>();
}
```

```
public ReadOnlyCollection<Argument> UnrecognizedArguments { get; private set; }
public ReadOnlyCollection<Argument> MalformedArguments { get; private set; }
public ReadOnlyCollection<Argument> RepeatedArguments { get; private set; }

public ReadOnlyCollection<ArgumentDefinition> ArgumentDefinitions =>
    new ReadOnlyCollection<ArgumentDefinition>(argumentDefinitions);

public IEnumerable<string> DefinedSwitches =>
    from argumentDefinition in argumentDefinitions
    select argumentDefinition.ArgumentSwitch;

public void AddArgumentVerifier(ArgumentDefinition verifier) =>
    argumentDefinitions.Add(verifier);

public void RemoveArgumentVerifier(ArgumentDefinition verifier)
{
    var verifiersToRemove = from v in argumentDefinitions
                            where v.ArgumentSwitch == verifier.ArgumentSwitch
                            select v;
    foreach (var v in verifiersToRemove)
        argumentDefinitions.Remove(v);
}

public void AddArgumentAction(string argumentSwitch, Action<Argument> action) =>
    argumentActions.Add(argumentSwitch, action);

public void RemoveArgumentAction(string argumentSwitch)
{
    if (argumentActions.Keys.Contains(argumentSwitch))
        argumentActions.Remove(argumentSwitch);
}

public bool VerifyArguments(IEnumerable<Argument> arguments)
{
    // 没有任何参数进行验证,失败
    if (!argumentDefinitions.Any())

        return false;

    // 确认是否存在任一未定义的参数
    this.UnrecognizedArguments =
        ( from argument in arguments
          where !DefinedSwitches.Contains(argument.Switch.ToUpper())
          select argument).ToList().AsReadOnly();

    if (this.UnrecognizedArguments.Any())
        return false;

    //检查开关与某个已知开关匹配但是检查格式是否正确
    //的谓词为false的所有参数
    this.MalformedArguments = ( from argument in arguments
                                join argumentDefinition in argumentDefinitions
                                on argument.Switch.ToUpper() equals
                                argumentDefinition.ArgumentSwitch
```

```
        where !argumentDefinition.Verify(argument)
        select argument).ToList().AsReadOnly();

    if (this.MalformedArguments.Any())
        return false;

    //将所有参数按照开关进行分组,统计每个组的数量,
    //并选出包含超过一个元素的所有组,
    //然后我们获得一个包含这些数据项的只读列表
    this.RepeatedArguments =
        (from argumentGroup in
            from argument in arguments
            where !argument.IsSimple
            group argument by argument.Switch.ToUpper()
            where argumentGroup.Count() > 1
            select argumentGroup).SelectMany(ag => ag).ToList().AsReadOnly();

    if (this.RepeatedArguments.Any())
        return false;

    return true;
}

public void EvaluateArguments(IEnumerable<Argument> arguments)
{
    //此时只需应用每个动作:
    foreach (Argument argument in arguments)
        argumentActions[argument.Switch.ToUpper()](argument);
}

public string InvalidArgumentsDisplay()
{
    StringBuilder builder = new StringBuilder();
    builder.AppendFormat($"Invalid arguments: {Environment.NewLine}");
    // 添加未识别的参数

    FormatInvalidArguments(builder, this.UnrecognizedArguments,
        "Unrecognized argument: {0}{1}");

    // 添加格式不正式的参数
    FormatInvalidArguments(builder, this.MalformedArguments,
        "Malformed argument: {0}{1}");

    // 对于重复的参数,我们想要将其分组以用于显示,
    // 因此通过开关分组并且将其添加到正在构建的字符串
    var argumentGroups = from argument in this.RepeatedArguments
        group argument by argument.Switch.ToUpper() into ag
        select new { Switch = ag.Key, Instances = ag};

    foreach (var argumentGroup in argumentGroups)
    {
        builder.AppendFormat($"Repeated argument:
                                {argumentGroup.Switch}{Environment.NewLine}");
        FormatInvalidArguments(builder, argumentGroup.Instances.ToList(),
            "\t{0}{1}");
    }
}
```

```
    }
    return builder.ToString();
}

private void FormatInvalidArguments(StringBuilder builder,
    IEnumerable<Argument> invalidArguments, string errorFormat)
{
    if (invalidArguments != null)
    {
        foreach (Argument argument in invalidArguments)
        {
            builder.AppendFormat(errorFormat,
                argument.Original, Environment.NewLine);
        }
    }
}
```

如何使用这些类为应用程序处理命令行？方法如下所示。

```
public static void Main(string[] argumentStrings)
{
    var arguments = (from argument in argumentStrings
        select new Argument(argument)).ToArray();

    Console.Write("Command line: ");
    foreach (Argument a in arguments)
    {
        Console.Write($"{a.Original} ");
    }
    Console.WriteLine("");

    ArgumentSemanticAnalyzer analyzer = new ArgumentSemanticAnalyzer();
    analyzer.AddArgumentVerifier(
        new ArgumentDefinition("output",
            "/output:[path to output]",
            "Specifies the location of the output file.",
            x => x.IsCompoundSwitch));
    analyzer.AddArgumentVerifier(
        new ArgumentDefinition("trialMode",
            "/trialmode",
            "If this is specified it places the product into trial mode",
            x => x.IsSimpleSwitch));
    analyzer.AddArgumentVerifier(
        new ArgumentDefinition("DEBUGOUTPUT",
            "/debugoutput:[value1];[value2];[value3]",
            "A listing of the files the debug output " +
            "information will be written to",
            x => x.IsComplexSwitch));
    analyzer.AddArgumentVerifier(
        new ArgumentDefinition("",
            "[literal value]",
            "A literal value",
            x => x.IsSimple));
}
```

```
if (!analyzer.VerifyArguments(arguments))
{
    string invalidArguments = analyzer.InvalidArgumentsDisplay();
    Console.WriteLine(invalidArguments);
    ShowUsage(analyzer);
    return;
}

// 设置命令行解析结果的容器
string output = string.Empty;
bool trialmode = false;
IEnumerable<string> debugOutput = null;
List<string> literals = new List<string>();

//我们想对每一个解析出的参数应用一个动作，
//因此将它们添加到分析器
analyzer.AddArgumentAction("OUTPUT", x => { output = x.SubArguments[0]; });
analyzer.AddArgumentAction("TRIALMODE", x => { trialmode = true; });
analyzer.AddArgumentAction("DEBUGOUTPUT", x =>
{ debugOutput = x.SubArguments;
});

analyzer.AddArgumentAction("", x=>{literals.Add(x.Original);});

// 检查参数并运行动作
analyzer.EvaluateArguments(arguments);

// 显示结果
Console.WriteLine("");
Console.WriteLine($"OUTPUT: {output}");
Console.WriteLine($"TRIALMODE: {trialmode}");
if (debugOutput != null)
{
    foreach (string item in debugOutput)
    {
        Console.WriteLine($"DEBUGOUTPUT: {item}");
    }
}
foreach (string literal in literals)
{
    Console.WriteLine($"LITERAL: {literal}");
}
}

public static void ShowUsage(ArgumentSemanticAnalyzer analyzer)
{
    Console.WriteLine("Program.exe allows the following arguments:");
    foreach (ArgumentDefinition definition in analyzer.ArgumentDefinitions)
    {
        Console.WriteLine($"{definition.ArgumentSwitch}:
({definition.Description}){Environment.NewLine}
\tSyntax: {definition.Syntax}");
    }
}
}
```