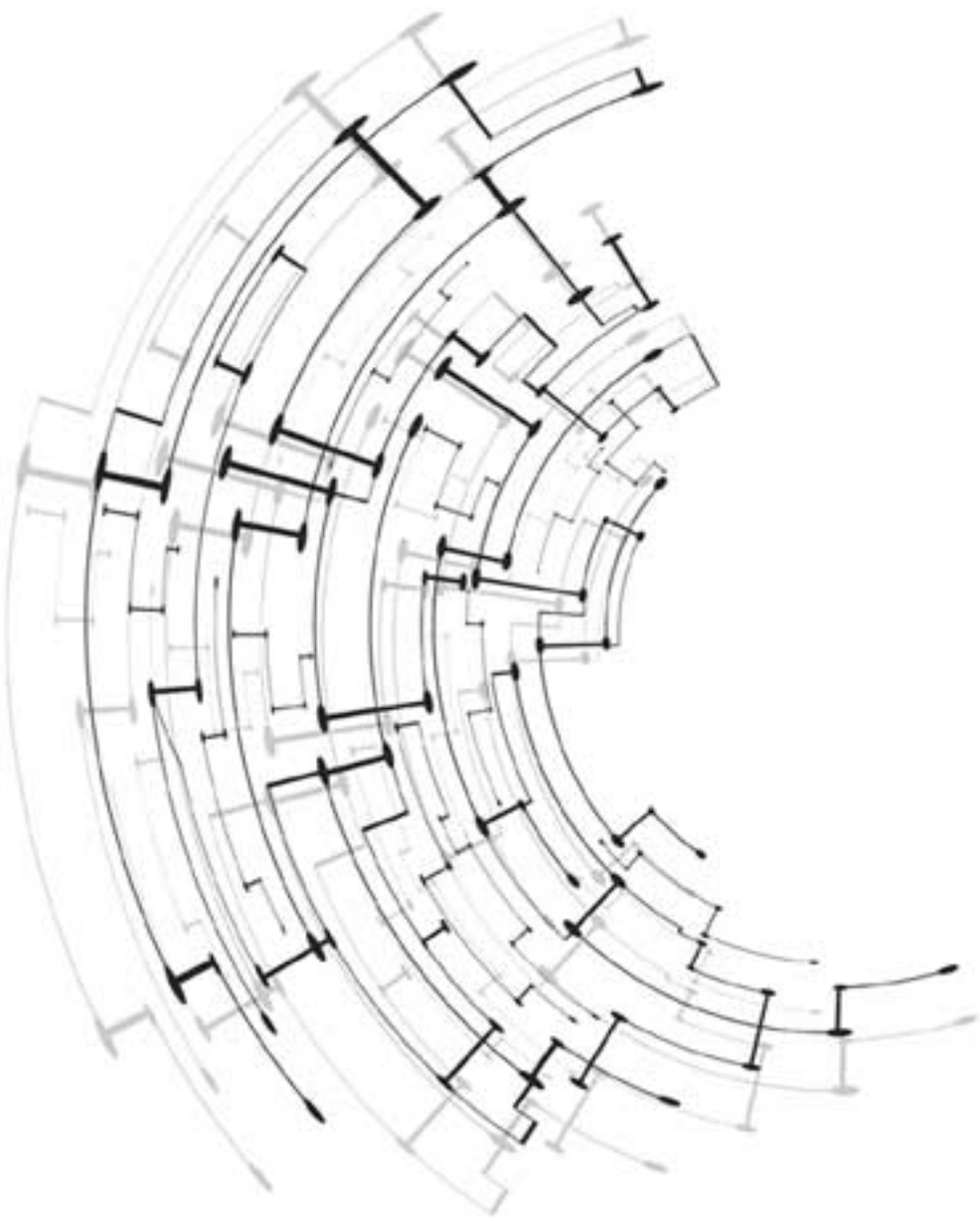
FUNCTIONAL PROGRAMMING IN C#, HOW TO WRITE BETTER C# CODE

C#函数式编程

编写更优质的C#代码

[美] 恩里科·博南诺(Enrico Buonanno) 著
张久修 译



MANNING

清华大学出版社

C#函数式编程

编写更优质的C#代码

[美]恩里科·博南诺(Enrico Buonanno) 著
张久修 译

清华大学出版社

北 京

Enrico Buonanno
Functional Programming in C#, How to Write Better C# Code
EISBN: 978-1-61729-395-5

北京市版权局著作权合同登记号　图字：01-2017-8973

图书在版编目(CIP)数据

C#函数式编程 编写更优质的 C#代码 / (美)恩里科·博南诺(Enrico Buonanno) 著；张久修译. —北京：清华大学出版社，2019
书名原文：Functional Programming in C#, How to Write Better C# Code
ISBN 978-7-302-51055-0

Ⅰ. ①C…　Ⅱ. ①恩…　②张…　Ⅲ. ①C 语言－程序设计　Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字(2018)第 192028 号

责任编辑：王　军　韩宏志
封面设计：周晓亮
版式设计：思创景点
责任校对：牛艳敏
责任印制：丛怀宇

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：三河市龙大印装有限公司
经　　销：全国新华书店
开　　本：170mm×240mm　　**印　　张：**24　　**字　　数：**484 千字
版　　次：2019 年 1 月第 1 版　　**印　　次：**2019 年 1 月第 1 次印刷
定　　价：98.00 元

产品编号：078166-01

译 者 序

历经数月，终于完成了本书的翻译工作。没有过多的轻松和兴奋之情，反而更多的是一份疼惜，一份不舍。从事技术书籍的翻译工作，既是一个付出的过程，也是一个收获的过程，更是对自己技术的一种沉淀。每当一本书的翻译工作结束后，我便会无所适从，仿佛少了些什么似的。每天都要花费几个小时的翻译工作，可让我的内心真正沉静下来，只专注于字里行间。那种状态、那种感觉真的很好，也正是如此，内心便充满依依不舍之情。

生活中充满了未知以及不确定性，谁也不能保证你的内心每时每刻都是清静的。在翻译本书的这段时间内，我也经历了或多或少会影响心境的一些琐事。可见，想要专心地做一件需要长期坚持且耗费精力的事情是多么艰难。因此，虽未曾写过书，但即使是翻译一本书，也能深深体会到作者付出的心血，可以想象到写书人夜以继日工作的情景。同时，对于能成为本书的译者，我感到特别荣幸。

正所谓“工欲善其事，必先利其器”。函数式编程作为主流编程的重要组成部分，必将是高级程序员手中的一大利器。而想要掌握并使用这件利器，首先需要了解和培养函数式思维，以不同的视角来看待代码。我们首先需要从宏观上了解函数式编程，了解它是什么，及其定义、特性、优点分别是什么，甚至其所存在的缺点或顾虑又有哪些。

顾名思义，函数式编程是一种特定的编程方式，将计算机运算视为函数的计算。简单来说它是一种“编程范式”，也就是如何编写程序的方法论。同时，它又属于“结构化编程”的一种，主要思想是将运算过程尽量写成一系列嵌套的函数调用。与指令式编程相比，函数式编程强调函数的计算比指令的执行重要；与过程化编程相比，函数式编程里函数的计算可随时调用。而函数编程语言最重要的基础是λ演算(lambda calculus)，而且λ演算的函数可接受函数当作输入(参数)和输出(返回值)。

函数式编程具有三个特性：闭包和高阶函数，惰性计算，以及递归。并有五个鲜明特点：函数是“一等公民”(first class)、只用“表达式”而非“语句”、没有“副作用”(side effect)、不修改状态、具有引用透明性。函数式编程的优点为：代码简洁且开发快速、接近自然语言而且易于理解、方便了代码管理、易于“并发编程”，且利于代码的热升级。

凡事有利必有弊。在良好的特性和优点的背后，必然也存在顾虑。函数式编程常被认为严重耗费 CPU 和存储器资源，主因有二：其一是，早期的函数式编程语言实现时并未考虑过效率问题；第二点是，有些非函数式编程语言为提升速度，不提供自动边界检查或自动垃圾回收等功能。同时，惰性求值亦为语言(如 Haskell)增加了额外的管理工作。

以上只是对函数式编程的一个宏观介绍；而通过本书，你可以更加系统、详细地了解并掌握函数式编程。本书由浅入深地讲解函数式编程的基本原理和技术，并通过现实中的各种示例场景带领你完成非函数式编程代码到函数式编程代码的不断重构，以逐渐让你树立函数式编程的思想，引领你从一个全新的视角看待代码。本书必将让你受益匪浅。

作为本书的译者，我本着“诚惶诚恐”的态度投入工作，为避免误导读者，文中的一词一句皆反复斟酌。但是鉴于译者水平有限，错误和失误在所难免，如有任何意见和建议，请不吝指正，感激不尽！

关于本书

如今，函数式编程(Functional Programming，FP)已成为主流编程的一个重要且令人兴奋的组成部分。近十年来创建的大多数语言和框架都是函数式的，这导致人们纷纷预测编程的未来也将是函数式的。与此同时，诸如 C#和 Java 的面向对象主流语言，在其每个新版本中都会引入更多函数式特性，从而实现多范式编程风格。

然而，C#社区关于函数式编程的推行却十分缓慢。为何会这样呢？我认为，其中一个原因是缺乏优秀的文献：

- 大多数 FP 文献都是用函数式语言(尤其是 Haskell)编写的。而对于具有 OOP(Oriented Object Programming，面向对象编程)背景的开发人员来说，要学习其概念就必须跨越编程语言的障碍。尽管许多概念适用于诸如 C#的多范式语言，但同时学习一种新范式和新语言是一项艰巨的任务。
- 更重要的是，文献中的大部分书籍倾向于用数学或计算机科学领域的例子来阐明函数式编程的技术和概念。对于大部分终日从事业务(LOB)应用开发的程序员来说，这会产生一个领域差异，并使得他们难以知悉这些技术与实际应用间的相关性。

这些缺陷是我学习 FP 的主要绊脚石。有些书籍试图解释什么是“柯里化”，通过利用数字 3，来创建一个可将 3 添加到任意数字的函数，以展示 add 函数是如何被柯里化的(在你能想到的所有应用中，它有一点点实际用处吗？)。放弃此类书籍后，我决定追寻一条属于自己的研究之路。这包括学习 6 种函数式语言(最优秀的几种语言)，并探究 FP 中的哪些概念可在 C#中有效地应用，以及研究由大量开发人员有偿撰写的该类型的应用，并且最终撰写了本书。

本书展示如何利用 C#语言的函数式技术来弥补 C#开发人员的语言差异。还展示如何将这些技术应用于典型的业务场景来弥补领域差异。我采取了一种务实的方法，并且涵盖了函数式技术，使其在典型的 LOB 应用场景中非常有用，并省略了 FP 背后的大部分理论。

最终，你应该关注 FP，因其赋予了以下优势：

- **高效率**——这意味着可用更少的代码完成更多工作。FP 提高了抽象层级，使你可编写高级代码，同时将你从那些只是增加复杂性，却没有任何价值

的低级技术层面解放出来。

- **安全性**——在处理并发性时尤其如此。一个用命令风格编写的程序可能在单线程实现中运行良好，但当并发发生时会导致各种错误。函数式代码在并发场景中提供了更好的保障，因此，在多核处理器时代，我们很自然会看到开发人员对 FP 的兴趣激增。
- **清晰性**——相对于编写新代码，我们会花费更多时间来维护和使用现有的代码，所以让我们的代码清晰明了并且意义明确是非常重要的。当你转向函数式思维时，达到这种清晰性将水到渠成。

如果你已经用面向对象的风格进行了一段时间的编程，在本书的概念实现之前，可能需要做出一些努力和意愿去尝试。为确保学习 FP 是一个愉快而有益的过程，我有两个建议：

- **耐心**——你可能需要多次重复阅读一些章节。你可能会把这本书放下几个星期，当你再次拿起这本书时，突然间发现有些模糊的东西开始变得有意义了。
- **用代码进行实验**——实践出真知。本书提供了许多示例和练习，许多代码片段可在 REPL 中进行测试。

你的同事可能比你更不愿意去探索新东西。预料到他们可能会抗议你采用这种新风格，并对你的代码感到困惑，然后发问“为什么不只是做 x？”(其中 x 是枯燥的、过时的，并且通常是有害的)。不必过多地讨论。坐下来，看着他们最终转身，并用你的技术来解决他们屡次遇到的问题。

致　　谢

感谢 Paul Louth，他不但通过自己编写的 LanguageExt 库赋予我灵感(我从中借鉴了很多很棒的想法)，而且亲力亲为地在各个阶段对本书进行了审阅。

Manning 出版社的详尽编辑过程确保了本书的质量。为此，我要感谢与本书合作的团队，包括 Mike Stephens、开发编辑 Marina Michaels、技术编辑 Joel Kotarski 技术校对员 Jürgen Hoetzel，以及版权编辑 Andy Carroll。

特别感谢 Daniel Marbach 和 Tamir Dresher 所给予的技术见解，以及所有参与同行评审的人，包括 Alex Basile、Aurélien Gounot、Blair Leduc、Chris Frank、Daniel Marbach、Devon Burriss、Gonzalo Barba López、Guy Smith、Kofi Sarfo、Pauli Sutelainen、Russell Day、Tate Antrim 和 Wayne Mather。

感谢 Scott Wlaschin 在 http://fsharpforfunandprofit.com 上分享的文章，感谢所有通过文章、博客和开源代码分享自己的知识和热情的其他 FP 社区成员。

前　言

本书旨在展示如何利用 C#中的函数式技术编写简洁、优雅、健壮和可维护的代码。

本书读者对象

本书是为那些具有雄心壮志的开发人员所编写的。你需要了解 C#语言和.NET 框架。你需要具备开发实际应用的经验，熟悉 OOP 的概念、模式和最佳实践。并且，你正在寻求通过学习函数式技术来扩展编程技能，以便可以充分利用 C#的多范式语言特性。如果你正在尝试或正在计划学习一门函数式语言，那么本书也将是非常有价值的，因为你将学习如何在一门你所熟悉的语言上进行函数式思考。改变自己的思考方式是很难的；而一旦做到，那么学习任何特定语言的语法将变得相对容易。

本书的组织结构

全书共 15 章，分为 3 个部分：

- 第Ⅰ部分介绍函数式编程的基本技术和原理。我们将初窥函数式编程是什么，以及 C#是如何支持函数式编程风格的。然后，将研究高阶函数的功能、纯函数及其与可测性的关系、类型和函数签名的设计，以及如何将简单的函数组合到复杂的程序中。在第Ⅰ部分的最后，你将很好地感受到一个用函数式风格所编写的程序是什么样的，以及这种风格所带来的好处。
- 第Ⅱ部分将加快速度，转向更广泛的关注点，例如函数式的错误处理、模块化和组合应用，以及理解状态和表示变化的函数式方法。到第Ⅱ部分结束时，你将掌握一系列工具的用法，将能利用函数式方法来有效地完成许多编程任务。
- 第Ⅲ部分将讨论更高级的主题，包括惰性求值、有状态计算、异步、数据流和并发性。第Ⅲ部分的每章都介绍一些重要技术，它们可能彻底改变你编写软件的方式和思考方式。

你会在每章中找到更详细的主题分类，并在阅读任何特定章节之前，都能从本书的内封了解到需要预先阅读哪些章节。

为实际应用编码

本书旨在让实际场景保持真实。为此，很多例子都涉及实际任务，例如读取配置、连接数据库、验证 HTTP 请求；对于这些事情，你可能已经知道如何做了，但你将用函数式思维的新视角来重新看待它们。

在本书中，我使用了一个长期运行的例子来说明在编写 LOB 应用时，FP 是如何提供帮助的。为此，我选择了一个在线银行应用，它是虚拟的 Codeland 银行(BOC) ——我知道这或许有些生搬硬套了，但至少它有了必需的三个字母的缩写。由于大多数人都可访问在线银行设施，因此很容易想象其所需的功能，并且清楚地看到所讨论的问题是如何与实际应用关联的。

我也使用了场景来说明如何解决函数式风格中典型的编程问题。在实际的例子和 FP 概念之间的不断反复，将帮助我们弥合理论与实践之间的差异。

利用函数式库

诸如 C#的语言具有函数式特性，但为了充分利用这些特性，你将经常使用便于实现常见任务的库。Microsoft 已经提供了几个库，以便进行函数式风格的编程，包括：

- **System.Linq**——这是一个功能库。我假定你是熟悉它的，因为它是.NET 的一个重要组成部分。
- **System.Collections.Immutable**——这是一个不可变集合的库，第 9 章将开始使用它。
- **System.Reactive**——这是.NET 的 Reactive Extensions 的实现，允许你使用数据流，第 14 章将讨论这些数据流。

当然还有其他许多重要的类型和功能未列举，这些都是 FP 的主要部分。因此，一些独立的开发人员已经编写了一些开源的代码库来填补这些空白。到目前为止，其中最完整的是 LanguageExt，这是由 Paul Louth 编写的一个库，用于在进行函数式编码时改进 C#开发人员的体验。[1]

本书并没有直接使用 LanguageExt；相反，将向你展示如何开发自己的函数式实用工具库，且将其命名为 LaYumba.Functional，尽管它与 LanguageExt 在很大程

1 LanguageExt 是开源的，可在 GitHub 和 NuGet 上找到：https://github.com/louthy/language-ext。

度上是重叠的，但这在教学方面会更有用，原因有如下几点：

- 在本书出版后，将保持代码的稳定。
- 你可以透过现象看本质，将看到看似简单实则强大的函数式构造。
- 你可以专注于基本要素：我将以最纯粹的形式向你展示这些构造，这样你就不会被一个完整的库所处理的细节和边缘情况分散注意力。

代码约定和下载

代码示例使用了 C# 7，大部分与 C# 6 兼容。C# 7 中专门介绍的语言特性仅用于第 10 章及之后章节(另外，1.2 节的几个示例中明确地展示了 C# 7)。可在 REPL 中执行许多较短的代码片段，从而获得动手练习的实时反馈。更多的扩展示例可通过 https://github.com/la-yumba/functional-csharp-code 下载，其中还配有练习的设置和解决方案。

本书中的代码清单重点讨论了正在讨论的主题，因此可能会省略命名空间(namespace)、using 语句、简单的构造函数，或先前代码清单中出现的并保持不变的代码段。如果你想查看代码清单的完整编译版本，可在代码存储库中找到它：https://github.com/la-yumba/functional-csharp-code。

另外，读者也可扫描封底的二维码下载相关资料。

图书论坛

购买本书后，可免费访问由 Manning 出版社运行的私人网络论坛，你可在这里提交有关本书的评论，询问技术问题，并获得作者和其他用户的帮助。可通过 https://forums.manning.com/forums/functional-programming-in-c-sharp 访问该论坛。你也可通过 https://forums.manning.com/forums/about 了解更多关于 Manning 论坛及论坛行为准则的信息。

Manning 出版社为读者提供一个场所，在这里，读者之间以及读者和作者之间可以进行有意义的对话。但不承诺作者的任何具体参与度，作者对论坛的贡献是自愿的(并且是无偿的)。我们建议你尝试向作者提出一些具有挑战性的问题，以免他的兴趣流失！只要本书还在市场上销售，论坛和之前所讨论的内容存档将可从出版商的网站上直接访问。

目　录

第 I 部分　核心概念

第Ⅰ部分

核心概念

在这一部分，我们将介绍函数式编程的基本技术和原理。

第 1 章首先了解函数式编程是什么，以及 C#如何支持函数式风格编程。然后深入研究 FP 的基本技术——高阶函数。

第 2 章解释纯函数是什么，为什么纯洁性对函数的可测试性有重要影响，以及为什么纯函数适用于并行化和其他优化。

第 3 章涉及设计类型和函数签名的原则——这些内容你原本就了解，但从函数式角度看，却是新的内容。

第 4 章介绍 FP 的一些核心函数：Map、Bind、ForEach 和 Where (过滤器)。这些函数提供了在 FP 中与最常见的数据结构交互的基本工具。

第 5 章介绍如何将函数链接到捕捉程序工作流的管道中。然后，将扩大作用域并以函数式风格开发整个用例。

第Ⅰ部分结束时，你会对用函数式风格所编写的程序拥有良好的感觉，并会理解这种风格所带来的好处。

第1章

介绍函数式编程

本章涵盖的主要内容：

- 函数式编程的优点和原理
- C#语言的函数式特性
- C#中的函数表示
- 高阶函数

函数式编程是一种编程范式，是指对于程序的一种不同思考方式，而不是你可能习惯的主流命令式范式。出于这个原因，函数式思维的学习是具有挑战性的，但也是丰富多彩的。我的愿望是，在阅读本书后，你永远不会再用以前那样的视角来看待代码！

学习本书可能是一段颠簸的旅程。你可能会因某些概念而感到挫败，这些概念似乎晦涩或无用，而当你茅塞顿开时却又不亦乐乎，并且你能够用几行优雅的函数式代码代替命令式代码。

本章将介绍你在开始这段旅程时可能遇到的一些问题：函数式编程究竟是什么？我为什么要在乎它？我可在 C#中进行函数式编码吗？这值得努力吗？

我们将首先概述函数式编程(FP)是什么以及C#语言如何以函数式风格支持编程。然后讨论函数以及其在 C#中的表示方式。最后，将介绍高阶函数，并用一个实例对其进行说明。

1.1 什么是函数式编程

函数式编程究竟是什么？在一个非常高的层面上，这是一种强调函数的同时避免状态突变的编程风格。这个定义是双重的，因为其包含两个基本概念：

- 函数作为第一类值
- 避免状态突变

下面介绍这些概念的含义。

1.1.1 函数作为第一类值

在函数是第一类值的语言中，可将它们用作其他函数的输入或输出，可将它们赋值给变量，也可将它们存储在集合中。换句话说，可使用函数完成你可对任何其他类型的值进行的所有操作。

例如，在 REPL 中输入以下内容：[1]

```
Func<int, int> triple = x => x * 3;
var range = Enumerable.Range(1, 3);
var triples = range.Select(triple);

triples // => [3, 6, 9]
```

在这个例子中，首先声明了一个函数，该函数返回给定整数乘以 3 的结果，并将其赋给变量 triple。然后使用 Range 创建一个 IEnumerable<int>，其值为[1, 2, 3]。接着调用 Select(IEnumerable 上的一个扩展方法)，将 range 和 triple 函数作为其参数；这将创建一个新的 IEnumerable，它包含通过将 triple 函数应用于输入的 range 中的每个元素而获得的元素。

这个简短的代码片段演示了 C#中的函数确实是第一类值，因为你可将乘以 3 的函数赋给变量 triple，并将其作为参数提供给 Select。在整本书中你会看到，将函数当作值处理可让你编写一些非常强大和简洁的代码。

1.1.2 避免状态突变

如果我们要遵循函数式范式，就应该完全避免状态突变：一旦创建一个对象，便永远不会改变，且变量永远不会被重新赋值。术语“突变(mutation)”表示某个值就地更改——更新存储器中某处存储的值。例如，下面的代码创建并填充一个

1 REPL 是一个命令行界面，允许你通过输入语句并获得即时反馈来测试该语言。如果你使用的是 Visual Studio，则可通过 View> Other Windows> C# Interactive 来启动 REPL。在 Mono 上，你可使用 csharp 命令。还有其他几个实用工具可让你以交互方式运行 C#代码片段，有些甚至可在浏览器中运行。

数组，然后更新了数组中的一个值：

```
int[] nums = { 1, 2, 3 };
nums[0] = 7;

nums // => [7, 2, 3]
```

这种更新也称为破坏性更新，因为更新前所存储的值遭到破坏。在函数式编码时应始终避免这些(纯粹的函数式语言根本不允许就地更新)。

遵循这一原则，对列表进行排序或过滤时不应该修改列表，而应该新建一个列表，在不影响原列表的情况下适宜地过滤或排序列表。在 REPL 中输入以下内容以查看使用 LINQ 的 Where 和 OrderBy 函数对列表进行排序或过滤时会发生什么。

代码清单1.1　函数式方法：Where和OrderBy不影响原始列表

```
Func<int, bool> isOdd = x => x % 2 == 1;
int[] original = { 7, 6, 1 };

var sorted = original.OrderBy(x => x);
var filtered = original.Where(isOdd);

original // => [7, 6, 1]
sorted   // => [1, 6, 7]
filtered // => [7, 1]
```


如你所见，原始列表不受排序或过滤操作的影响，而会产生新的 IEnumerable。

下面分析一个反例。如果你有一个 List<T>，可通过调用其 Sort 方法对其进行就地排序。

代码清单1.2　非函数式方法：List<T>.Sort对列表进行就地排序

```
var original = new List<int> { 5, 7, 1 };
original.Sort();

original // => [1, 5, 7]
```

在本示例中，排序后，原始排序遭到破坏。你马上就会明白为什么这是有问题的。

注意：你在该框架中同时看到函数式和非函数式方法，这是有历史原因的：List<T>.Sort 在日期上早于 LINQ，而这标志着在函数式方向上的决定性转向。

1.1.3　编写具有强力保证的程序

在刚才讨论的两个概念中，作为第一类值的函数最初显得更令人兴奋，本

章后半部分将集中讨论它。但在继续之前，我想简要说明为什么避免状态突变也是非常有益的，因为它消除了由可变状态引起的许多复杂性。

下面来看一个例子(后面将更详细地介绍这些主题，所以即使你目前尚未清楚理解所有事项，也不要担心)。将以下代码输入 REPL 中。

代码清单1.3 来自并发进程的状态突变会产生不可预知的结果

```
using static System.Linq.Enumerable;
using static System.Console;

var nums = Range(-10000, 10000).Reverse().ToList();
// => [10000, 9999, ... , -9999, -10000]

Action task1 = () => WriteLine(nums.Sum());
Action task2 = () => { nums.Sort(); WriteLine(nums.Sum()); };

Parallel.Invoke(task1, task2);
// prints: 92332970
//         0
```

这允许你在没有完全限定的情况下调用 Range 和 WriteLine

并行执行这两项任务

这里将 nums 定义为 10 000 到−10 000 之间的所有整数的列表；它们的总和显然应该为 0。然后创建两个任务：task1 计算并打印出总和；task2 首先对列表进行排序，然后计算并打印总和。如果独立运行，每项任务都将正确计算总和。然而，当你同时运行两个任务时，task1 会产生一个不正确且不可预知的结果。

很容易看出原因：当 task1 读取列表中的数字以计算总和时，task2 将重新排序该列表。这有点像试图在其他人翻页的同时阅读一本书：你会阅读一些残缺不全的句子！图 1.1 描述了这种情形。

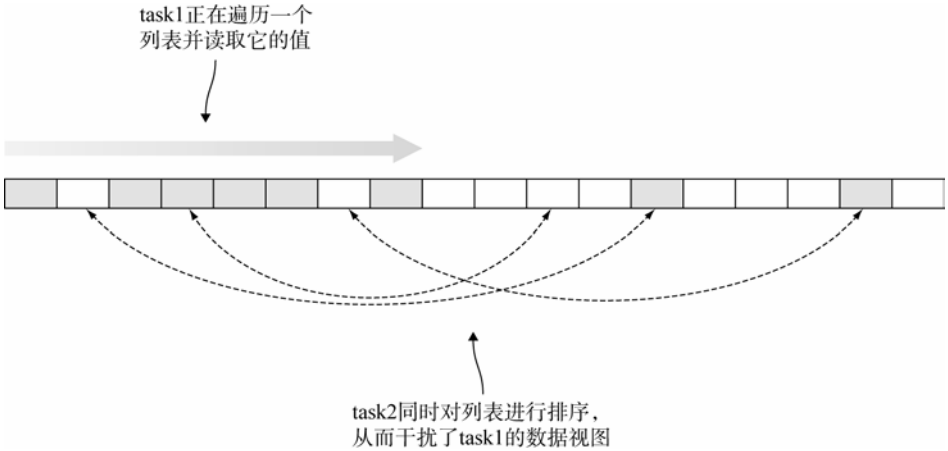


图 1.1 就地修改数据会带给并发线程一个不正确的数据视图

如果我们使用 LINQ 的 OrderBy 方法，而不是就地排序列表呢？

```
Action task3 = () => WriteLine(nums.OrderBy(x => x).Sum());

Parallel.Invoke(task1, task3);
// prints: 0
//         0
```

如你所见，即使你并行执行任务，使用 LINQ 的函数式实现依然能提供可预测的结果。这是因为 task3 没有修改原始列表，而是创建了一个全新的数据视图，这是已排序的—— task1 和 task3 同时从原始列表中读取，但并发读取不会导致任何不一致，如图 1.2 所示。

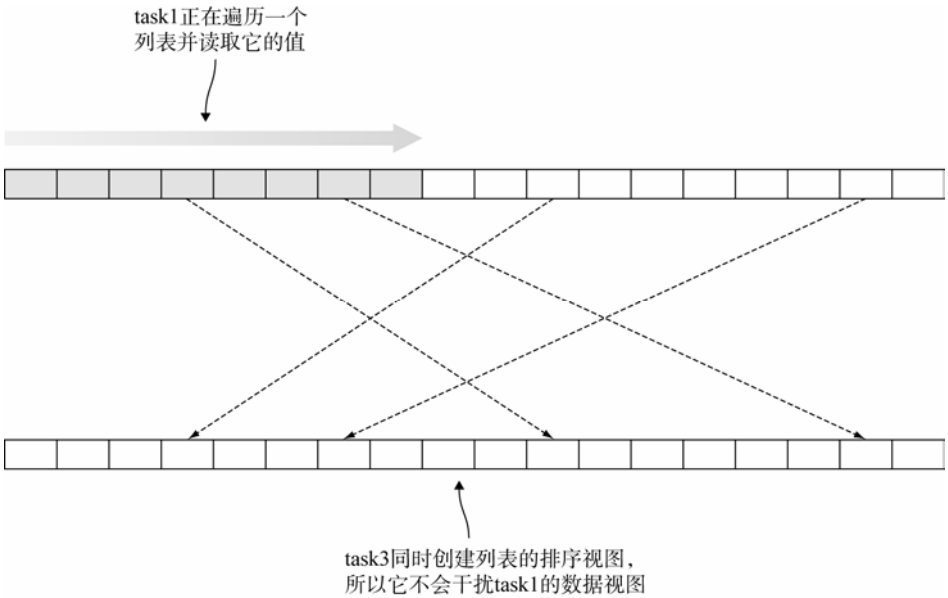


图 1.2　函数式方法：创建原始结构的新修改版本

这个简单示例说明了一个更广泛的事实：当开发人员用命令式风格编写应用程序(显式地使程序状态突变)并在后来引入并发(由于新的需求或需要提高性能)时，不可避免地会面临大量的工作和一些潜在的棘手 bug。从一开始就以函数式风格编写程序，通常可自由添加并发机制，或耗费更少的工作量。第 2 章和第 9 章将更详细地讨论状态突变和并发性。现在，让我们回到对 FP 的概述。

虽然大多数人会认同将函数视为第一类值并避免状态突变是 FP 的基本原则，但它们的应用催生了一系列实践和技术，所以在这样一本书中考虑哪些技术应该被认为是必要且可被收录的是值得商榷的。

我鼓励你对这个问题采取务实的态度，并尝试将 FP 理解为一组工具，以使用它来解决编程任务。当学习这些技术时，你将开始从不同的视角来看待问题：将开启函数式思维。

现在我们已经有了 FP 的工作定义，下面分析 C#语言本身，以及它对 FP 技术的支持。

“函数式”与“面向对象”

我经常被要求将FP与面向对象编程(OOP)进行比较。这并不简单，主要是因为对于OOP应该是什么样的，有许多不正确的臆测。

从理论上讲，OOP(封装、数据抽象等)的基本原理与FP的原理是正交的，所以没理由将这两种范式组合在一起。

然而，在实践中，大多数面向对象(OO)开发人员在其方法实现中严重依赖于命令式风格，使状态就地突变并使用显式控制流：他们在大规模的命令式编程中使用了OO设计。所以真正的问题是命令式与函数式编程，本章最后将总结FP的益处。

经常出现的另一个问题是，FP如何在构建一个大型、复杂的应用程序方面与OOP有所不同。构建一个复杂应用程序的难点在于需要遵循以下几个原则：

- 模块化(将软件划分为可复用组件)
- 关注分离(每个组件只应做一件事)
- 分层(高层组件可依赖于低层组件，但反之则不然)
- 松耦合(对组件的更改不应影响依赖它的组件)

这些原则是通用的，无论所讨论的组件是函数、类还是应用程序。

它们并非特定于OOP，因此可用相同的原则来构造用函数式风格所编写的应用程序——不同之处在于组件是什么，以及其所暴露的API。

在实践中，函数式所强调的纯函数(将在第2章中讨论)和可组合性(将在第5章中讨论)使得实现某些设计目标更加容易。[2]

1.2 C#的函数式语言

在前面的代码清单中，函数确实是 C#中的第一类值。实际上，从语言的最早版本到 Delegate 类型，C#都支持函数作为第一类值，随后的 lambda 表达式的引入使语法支持变得更好了——下一节将回顾这些语言特性。

有一些怪癖和限制，例如类型推断；我们将在第 8 章讨论这些内容。但总的来说，对函数作为第一类值的支持是相当不错的。

至于支持避免就地更新的编程模型，这方面的基本要求是语言具有垃圾回收功能。由于你创建了修改的版本，而不是就地更新现有值，因此你希望根据需要

2 关于为什么命令式风格的 OOP 使程序更复杂的更全面讨论，请参阅由 Ben Moseley 和 Peter Marks 撰写的 *Out of the Tar Pit*，2006(https://github.com/papers-we-love/papers-we-love/raw/master/design/out-of- the-tar-pit.pdf)。

对旧版本进行垃圾回收。同样，C#满足这个要求。

理想情况下，该语言还应该阻止就地更新。这是 C#最大的缺点：默认情况下一切都是可变的，程序员必须投入大量精力才能实现不可变。字段和变量必须显式地被标记为 readonly 以防止突变(将其与 F#进行比较，默认情况下 F#的变量是不可变的，并且必须显式地被标记为 mutable 以允许突变)。

关于类型呢？在框架中有几个不可变的类型，比如 string 和 DateTime，但对于用户所定义的不可变类型的语言支持却很差(不过，在 C# 6 中有所改进，并可能进一步改进未来版本，如下所述)。最后，框架中的集合是可变的，但一个可靠的不变集合库是可用的。

总之，C#很好地支持某些函数式技术，但对其他函数式技术的支持则不是很好。在迭代过程中，已有所改善，并将继续改进对函数式技术的支持。在本书中，你将了解哪些特性可被利用，以及如何消除其缺点。

接下来将回顾与 FP 相关的 C#的过去、现在和未来版本中的一些语言特性。

1.2.1　LINQ 的函数式性质

当 C# 3 与.NET Framework 3.5 版本一起发布时，包含许多受函数式语言所启发的特性，包括 LINQ 库(System.Linq)和一些新的语言特性，这些特性使你能增强用 LINQ 所做的事情，如扩展方法和表达式树。

LINQ 确实是一个函数式库——我之前使用 LINQ 来说明 FP 的两个原则——随着你进一步阅读本书，LINQ 的函数式性质将变得更明显。

LINQ 为列表上的许多常见操作提供了实现(或更笼统地讲，在“序列”中，作为 IEnumerable 的实例)，其中最常见的操作是映射、排序和过滤(请参见补充段落“对于序列的常见操作”)。这是一个结合所有三个例子的示例：

```
Enumerable.Range(1, 100).
  Where(i => i % 20 == 0).
  OrderBy(i => -i).
  Select(i => $"{i}%")
// => ["100%", "80%", "60%", "40%", "20%"]
```

注意 Where、OrderBy 和 Select 都接受函数作为参数，并且不会使给定的 IEnumerable 突变，而是返回一个新的 IEnumerable。这体现了前面介绍的两个 FP 原则。

LINQ 不仅可查询内存中的对象(LINQ 到 Objects)，还可查询其他各种数据源，如 SQL 表和 XML 数据。C#程序员已将 LINQ 作为处理列表和关系数据的标准工具集(与此相关的典型代码库数量众多)。另一方面，这意味着你已对函数式库的 API 有了基本印象。

另一方面，当使用其他类型时，C#程序员通常坚持使用流控制语句的命

令式风格来表达程序的预期行为。因此，我见过的大多数 C#代码库都是函数式风格(使用 IEnumerables 和 IQueryables 时)和命令式风格(其他所有内容)的拼合物。

这意味着虽然 C#程序员已经意识到使用诸如 LINQ 之类的函数式库的好处，但还不能完全揭示 LINQ 背后的设计原则，以便在设计中利用这些技术。这是本书旨在解决的问题。

对于序列的常见操作

LINQ库包含许多用于对序列执行常见操作的方法，如下所示：

- **映射**——给定一个序列和一个函数，映射生成一个新序列，其元素是通过将给定函数应用于给定序列中的每个元素(在 LINQ 中，这是通过 Select 方法完成的)而获得的。

```
Enumerable.Range(1, 3).Select(i => i * 3) // => [3, 6, 9]
```

- **过滤**——给定一个序列和一个谓词，过滤生成一个新序列，它由给定序列中传递谓词(在 LINQ 中为 Where)的元素组成。

```
Enumerable.Range(1, 10).Where(i => i % 3 == 0) // => [3, 6, 9]
```

- **排序**——给定一个序列和一个键选择器函数，排序生成一个按键(在 LINQ 中，为 OrderBy 和 OrderByDescending)排序的新序列。

```
Enumerable.Range(1, 5).OrderBy(i => -i)  // => [5, 4, 3, 2, 1]
```

1.2.2 C# 6 和 C# 7 中的函数式特性

C# 6 和 C# 7 没有 C# 3 那么具有革命性，但它们包含许多更小的语言特性，这些特性共同提供了更好的体验和用于函数式编码的更符合习惯的语法。

注意：C# 6 和 C# 7 中引入的大多数特性都提供了更好的语法，而不是新功能。如果你使用的是旧版本的 C#，则仍可应用本书中展示的所有技术(只需要额外输入一些代码)。但这些新特性显著提高了可读性，使函数式风格的编程更具吸引力。

你可在代码清单 1.4 中看到这些特性。

代码清单1.4 与FP相关的C# 6和C# 7特性

```
using static System.Math;

public class Circle
```

using static 可对 System.Math 的静态成员(如 PI 和 Pow)进行非限制访问

```
{
    public Circle(double radius)
      => Radius = radius;

    public double Radius { get; }

    public double Circumference
       => PI * 2 * Radius;

    public double Area
    {
       get
       {
          double Square(double d) => Pow(d, 2);
          return PI * Square(Radius);
       }
    }

    public (double Circumference, double Area) Stats
       => (Circumference, Area);
}
```

一个只读的自动属性只能在构造函数中设置

一个具有表达式体式的属性

局部函数是在另一个方法中所声明的方法

具有命名元素的 C# 7 元组语法

使用 using static 导入静态成员

C# 6 中的 using static 语句允许你导入类的静态成员(在本例中为 System.Math 类)。因此，在本例中，你可调用 Math 的 PI 和 Pow 成员，而不需要进一步限定条件：

```
using static System.Math;

public double Circumference
   => PI * 2 * Radius;
```

为什么这很重要？在 FP 中，我们更喜欢行为仅依赖于输入参数的函数，因为我们可独立推理和测试这些函数(与实例方法相比，其实现通常会与实例变量进行交互)。这些函数在 C#中用静态方法实现，因此 C#中的函数式库主要由静态方法组成。

using static 使你可更轻松地使用这些库，尽管过度使用可能导致命名空间污染，但合理使用会产生干净可读的代码。

具有只读的自动属性的更简易不可变类型

当声明一个只读的自动属性，如 Radius 时，编译器会隐式声明一个 readonly 支持字段。因此，这些属性只能在构造函数或内联中被赋值：

```
public class Circle
{
    public Circle(double radius)
       => Radius = radius;

    public double Radius { get; }
}
```

只读的自动属性有助于定义不可变类型，第 9 章将进行详细介绍。Circle 类

表明：它只有一个字段(Radius 的支持字段)，且是只读的，所以一旦创建，Circle 将永远不会改变。

具有表达式体式成员的更简洁函数

Circumference 属性是用带有 => 的表达式体所声明的，而不是使用{}的寻常语句体：

```
public double Circumference
   => PI * 2 * Radius;
```

注意，与 Area 属性相比，这更简洁明了！

在 FP 中，我们倾向于编写大量简单的函数，其中许多是单行的，然后将它们组合成更复杂的工作流程。表达式体式方法允许你用最小的语法噪音做到这一点。

当想要编写返回一个函数的函数时，这一点尤其明显——在本书中你会做很多事情。

表达式体式语法是在 C# 6 中为方法和属性引入的，在 C# 7 中被广泛应用于构造函数、析构函数、getter 和 setter 中。

局部函数

编写大量简单函数意味着许多函数只能从一个位置被调用。C# 7 允许你通过在方法作用域内声明方法来明确这一点；例如，在 Area 的 getter 的作用域内声明 Square 方法：

```
get
{
   double Square(double d) => Pow(d, 2);
   return PI * Square(Radius);
}
```

更佳的元组语法

更佳的元组语法是 C# 7 中最重要的特性。允许你轻松创建和使用元组，并为其元素指定有意义的名称。例如，Stats 属性返回一个类型为 (double, double)的元组，并指定可访问其元素的有意义名称：

```
public (double Circumference, double Area) Stats
   => (Circumference, Area);
```

元组在 FP 中很重要，因为它倾向于将任务分解为非常小的函数。你最终可能收到一个数据类型，其唯一目的是捕获一个函数返回的信息，并且这个数据类型应该是另一个函数的输入。为这种结构定义专用类型是不切合实际的，这种结构不符合有意义的域抽象，这就是元组的意义所在。

1.2.3　未来的 C#将更趋函数化

撰写本章的第一稿是在 2016 年初，当时 C# 7 的发展还处于初级阶段，而有趣的是，该语言团队具有“极强兴趣”的所有特性通常都与函数式语言相关联。包括以下内容：

- 记录类型(不含样板代码的不可变类型)
- 代数数据类型(对类型系统的强大补充)
- 模式匹配(类似作用于数据形态的 switch 语句，如类型，而不仅是值)
- 更佳的元组语法

一方面，令人失望的是只有最后一项可被交付。C# 7 也包含模式匹配的有限实现，但与函数式语言中可用的模式匹配种类相差甚远，而且在函数式编程时，我们所使用的模式匹配方式通常是不适当的(参见第 10.2.4 节)。

另一方面，这些特性仍处于未来的版本中，并已完成相应的提案工作。这意味着我们很可能在未来的 C#版本中看到记录类型和更完整的模式匹配实现。因此 C#已经准备好在其发展过程中继续作为一种具有越来越强大的函数式组件的多范式语言。

本书将为你奠定良好基础，以跟上语言和行业的发展步伐。还会让你更好地理解语言的未来版本背后的概念和动机。

1.3　函数思维

本节将阐明函数是什么。我将从这个词的数学用法开始，逐渐讨论 C#所提供的用于表示函数的各种语言结构。

1.3.1　映射函数

在数学中，函数是两个集合之间的映射，分别称为定义域和值域。即，给定一个来自其定义域的元素，函数从其值域产生一个元素。这就是所有过程；无论映射基于某个公式还是完全任意的都无关紧要。

从这个意义上来说，函数是一个完全抽象的数学对象，函数产生的值完全由其输入决定。但你会发现编程中的函数并不总是这样的。

例如，想象一个将小写字母映射到对应的大写字母的函数，如图 1.3 所示。在本示例中，

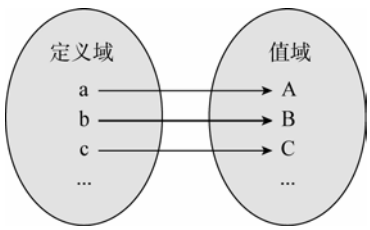


图 1.3　数学函数是两个集合的元素之间的映射

定义域是集合{a, b, c, ...}，值域是集合{A, B, C, ...}。当然，有一些函数的定义域和值域是相同的集合，你能想到一个例子吗？

这与编程的函数有什么关系呢？在 C#这样的静态类型语言中，集合(定义域和值域)是用类型表示的。例如，如果你对上面的函数进行编码，则可使用 char 来表示定义域和值域。函数类型可写成：

```
char → char
```

也就是说，该函数将 char 映射到 char，或等价于给出一个 char，其会生成一个 char。定义域和值域的类型构成一个函数的接口，也称为类型或签名。你可将此看成一个契约；一个函数签名声明：给定一个来自定义域的元素，将从值域生成一个元素。[3]这已经说得很明白了，但你会在第 3 章中了解到的是，在现实中，违反签名合同的情况比比皆是。

接下来分析如何对函数本身进行编码。

1.3.2 在 C#中表示函数

在 C#中有几种可用于表示函数的语言结构：

- 方法(method)
- 委托(delegate)
- lambda 表达式
- 字典(dictionary)

下面简单复习一下这些结构。如果你精通这些内容，请直接跳至下一节。

方法

方法是 C#中最常见和惯用的函数表示。例如，System.Math 类包含表示许多常用数学函数的方法。方法可表示函数，但它们也适用于面向对象的范式——可用来实现接口，可被重载等。

真正使你能以函数式风格进行编程的构造是委托和 lambda 表达式。

委托

委托是类型安全的函数指针。这里的类型安全意味着委托是强类型：函数的输入和输出值的类型在编译时是已知的，统一由编译器强制执行。

创建委托是一个两步过程：首先声明委托类型，然后提供一个实现(这类似于编写接口，然后实例化实现该接口的类)。

第一步通过使用委托关键字并为委托提供签名来完成。例如，.NET 包含以下

3 OO 意义上的接口是该想法的扩展：一组带有各自输入和输出类型的函数，或更确切地说，本质上是函数的方法，将当前实例作为一个隐式参数。

定义的 Comparison<T>委托。

代码清单1.5　声明一个委托

```
namespace System {
   public delegate int Comparison<in T>(T x, T y);
}
```

如你所见，一个 Comparison<T>委托可被赋予两个 T 类型的值，并会生成一个指示哪一个更大的 int。

一旦有了委托类型，即可通过提供一个实现来实例化它，如下面的代码清单所示。

代码清单1.6　实例化和使用委托

```
var list = Enumerable.Range(1, 10).Select(i => i * 3).ToList();
list // => [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]

Comparison<int> alphabetically = (l, r)
   => l.ToString().CompareTo(r.ToString());

list.Sort(alphabetically);
list // => [12, 15, 18, 21, 24, 27, 3, 30, 6, 9]
```

提供 Comparison 的实现

将 Comparison 委托用作 Sort 的参数

如你所见，在技术层面上，一个委托只是一个表示操作的对象，在本示例中，是一个比较操作。就像任何其他对象一样，你可将委托用作另一个方法的参数，如代码清单 1.6 所示，因此委托是使 C#中的函数具有第一类值的语言特性。

Func 和 Action 委托

.NET 框架包含几个可表示几乎任何函数类型的委托“家族”：

- Func<R> 表示一个不接受参数并返回一个 R 类型结果的函数。
- Func<T1,R> 表示一个接受一个 T1 类型的参数并返回一个 R 类型结果的函数。
- Func<T1,T2,R>表示一个接受一个 T1 类型的参数和一个 T2 类型的参数并返回一个 R 类型结果的函数。

委托可表示各种“元数(arity)”的函数(请参阅补充说明“函数元数”)。

自引入 Func 以来，已罕有自定义委托的使用。例如，不应按如下方式声明自定义委托：

```
delegate Greeting Greeter(Person p);
```

你可以使用类型：

```
Func<Person, Greeting>
```

上例中的 Greeter 类型与 Func<Person,Greeting>等效或“兼容”。这两种情况下，它都是一个接受 Person 并返回 Greeting 的函数。

有一个类似的委托家族可表示动作(action)——没有返回值的函数，比如 void 方法：

- Action 表示一个没有输入参数的动作。
- Action<T1>表示一个输入参数类型为 T1 的动作。
- Action<T1,T2>等表示一个具有多个输入参数的动作。

.NET 的发展已经远离了自定义委托，支持更通用的 Func 和 Action 委托。例如，对于谓词的表示：[4]

- .NET 2 中引入一个 Predicate<T>委托，例如，在 FindAll 方法中用于过滤一个 List<T>。
- 在.NET 3 中，Where 方法也用于过滤，但在更通用的 IEnumerable<T>中定义，不接受 Predicate<T>，只接受一个 Func<T,bool>。

两种函数是等效的。建议使用 Func 来避免表示相同函数签名的委托类型的泛滥，但仍然需要说明对自定义委托的可表达性支持：Predicate<T>比 Func< T,bool>能更清楚地传达意图，并更接近口语。

函数元数

元数(arity)是一个有趣的词语，指的是函数所接受的参数数量：

- 零元函数不接受任何参数。
- 一元函数接受一个参数。
- 二元函数接受两个参数。
- 三元函数接受三个参数。

其他函数以此类推。实际上，可将所有函数都看成一元的，因为传递n个参数相当于传递一个n元组作为唯一参数。例如，加法(就像其他任何二元算术运算一样)是一个函数，其定义域是所有数字对的集合。

lambda 表达式

lambda 表达式简称为 lambda，用于声明一个函数内联。例如，按照字母顺序排列数字列表，便可以使用 lambda 来完成。

代码清单1.7 用一个lambda声明一个函数内联

```
var list = Enumerable.Range(1, 10).Select(i => i * 3).ToList();
list // => [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]

list.Sort((l, r) => l.ToString().CompareTo(r.ToString()));
list // => [12, 15, 18, 21, 24, 27, 3, 30, 6, 9]
```

4 谓词是一个函数；给出一个值(如一个整数)，它表明是否满足某种条件(比如，是否为偶数)。

如果你的函数很简短，且不需要在其他地方重复使用，那么 lambda 提供了最优雅的表示法。另外注意，在上例中，编译器不仅会推断出 x 和 y 的类型为 int，还会将 lambda 转换为 Sort 方法所期望的委托类型 Comparison<int>，前提是所提供的 lambda 与该类型兼容。

就像方法一样，委托和 lambda 可以访问其作用域内声明的变量。这在利用 lambda 表达式中的闭包时特别有用。[5]以下是一个例子。

代码清单1.8　lambda可访问封闭作用域内的变量

```
var days = Enum.GetValues(typeof(DayOfWeek)).Cast<DayOfWeek>();
// => [Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday]

IEnumerable<DayOfWeek> daysStartingWith(string pattern)
   => days.Where(d => d.ToString().StartsWith(pattern));
                                                    pattern 变量在 lambda 内被引
daysStartingWith("S") // => [Sunday, Saturday]        用，因此在闭包中被捕获
```

在这个例子中，Where 期望一个接受 DayOfWeek 并返回 bool 的函数。实际上，由 lambda 表达式所表达的函数也使用在闭包中被捕获的 pattern 值来计算结果。

这很有趣。如果你用更数学化的眼光来看待由 lambda 表达式所表达的函数，你可能会说其实际上是一个二元函数，它接受一个 DayOfWeek 和一个 string(即 pattern)作为输入，并生成一个 bool。但作为程序员，我们通常主要关注函数签名，因此你更可能将其视为一个从 DayOfWeek 到 bool 的一元函数。这两种观点都是有根据的：函数必须符合其一元签名，但其依赖于两个值来完成工作。

字典

字典也被称为映射(map)或哈希表(hashtable)；它们是数据结构，提供了一个非常直接的函数表示。它们实际上包含键(定义域中的元素)与值(来自值域的相应元素)的关联。

我们通常将字典视为数据，因此，在某一时刻改变观点并将其视为函数是可行的。字典适用于表示完全任意的函数，其中映射无法计算，但必须详尽存储。例如，要将 Boolean 的值映射到其法语名称，你可编写以下内容。

代码清单1.9　一个可用字典来详尽表示的函数

```
var frenchFor = new Dictionary<bool, string>
{
    [true] = "Vrai",        C# 6 中字典的
    [false] = "Faux",       初始化器语法
```

5　闭包是 lambda 表达式本身与声明 lambda 的上下文(即 lambda 所处的作用域中所有可用的变量)的组合。

```
};

frenchFor[true]
// => "Vrai"
```

通过查找执行函数应用程序

函数可用字典表示的事实，也使得通过将计算结果存储在字典中而不是每次重新计算它们来优化计算昂贵的函数成为可能。

为方便起见，本书其余部分将使用术语 function 来表示函数的 C#表示法。请记住，这不完全符合术语的数学定义。你将在第 2 章中了解数学和编程函数之间的更多差异。

1.4 高阶函数

现在你已经了解到 FP 是什么，已回顾了该语言的函数式特性，现在是时候开始探索一些实际的函数式技术了。我们将以函数作为第一类值的最重要优点开始：它使你能定义高阶函数(Higher-Order Function，HOF)。

HOF 是接受其他函数作为输入或返回一个函数作为输出的函数，或两者兼而有之。假设你已经在某种程度上用过 HOF，比如 LINQ。本书将使用 HOF，所以本节应该作为一个复习，并介绍一些你可能不太熟悉的 HOF 使用案例。HOF 很有趣，本节中的大多数示例都可在 REPL 中运行。请确保你亲自尝试过一些变化。

1.4.1 依赖于其他函数的函数

有些 HOF 接受其他函数作为参数并调用它们以完成工作，有点像公司可能将其工作分包给另一家公司。本章前面已经看到了一些这样的 HOF 例子：Sort(List 上的实例方法)和 Where(IEnumerable 上的扩展方法)。

当用一个 Comparison 委托来调用 List.Sort 时，List.Sort 便是一个方法，表示：“好吧，我会对自己排序，只要告诉我该如何比较所包含的任意两个元素。”Sort 的工作就是排序，但调用者可决定使用什么样的逻辑进行比较。

同样，Where 的工作是过滤，调用者可决定以什么样的逻辑来确定是否应该包括一个元素。可以图形化方式表示 Where 的类型，如图 1.4 所示。

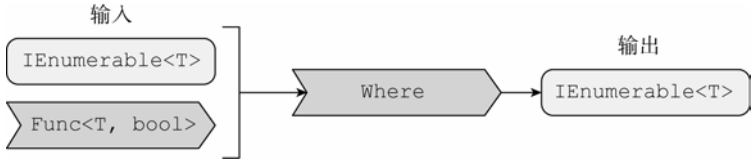


图 1.4 Where 接受一个谓词函数作为输入

下面看看 Where 的理想化实现。[6]

代码清单1.10　Where：一个迭代应用给定谓词的典型HOF

迭代列表的任务是 Where 的一个实现细节

```
public static IEnumerable<T> Where<T>
    (this IEnumerable<T> ts, Func<T, bool> predicate)
{
    foreach (T t in ts)
        if (predicate(t))
            yield return t;
}
```

要包含哪些项目的准则由调用者确定

Where 方法负责排序逻辑，调用者提供谓词，这是基于该条件过滤 IEnumerable 的准则。

如你所见，HOF 有助于在不能轻易分开逻辑的情况下关注分离。Where 和 Sort 是迭代应用程序的示例——HOF 将为集合中的每个元素重复应用给定的函数。

一个非常粗略的观察方法是，你传递一个函数作为参数，其代码最终将在 HOF 的循环体内执行——仅通过静态数据无法做到这一点，总体方案如图 1.5 所示。

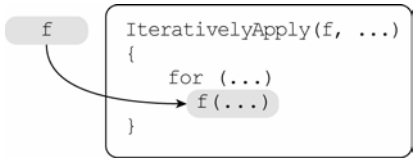


图 1.5　HOF 迭代地应用给定函数作为参数

可选执行是 HOF 的另一个很棒的选择。如果只想在特定条件下调用给定函数，这将很有用，如图 1.6 所示。

例如，设想一种从缓存中查找元素的方法。提供一个委托，并在缓存未命中时调用委托。

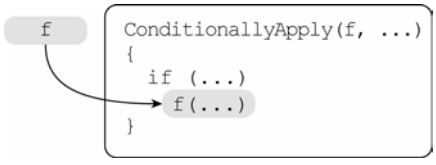


图 1.6　HOF 有条件地应用给定函数作为参数

代码清单1.11　HOF可选地调用给定函数

```
class Cache<T> where T : class
{
    public T Get(Guid id) => //…

    public T Get(Guid id, Func<T> onMiss)
        => Get(id) ?? onMiss();
}
```

onMiss 中的逻辑可能涉及昂贵的操作，如数据库调用，所以你不希望其被不必要地执行。

前面的例子阐明了 HOF，HOF 接受一个函数作为输入(通常称为回调或后续传递)，并用它来执行任务或计算值。[7]这或许是 HOF 最常见的模式，有时被称为

6　这个实现在函数式上是正确的，但缺少 LINQ 实现中的错误检查和优化。

7　这或许是 HOF 最常见的模式，有时被称为控制倒转：HOF 的调用者通过提供一个函数来决定做什么，函数通过调用给定的函数来决定何时执行该操作。

控制的倒转：HOF 的调用者通过提供一个函数来决定做什么，被调用者通过调用给定的函数来决定何时执行该操作。

下面分析可将 HOF 派上用场的其他场景。

1.4.2 适配器函数

有些 HOF 根本不应用所给定的函数，而是返回一个新函数，以某种方式与给定的作为参数的函数相关。例如，假设你有一个执行整数除法的函数：

```
Func<int, int, int> divide = (x, y) => x / y;
divide(10, 2) // => 5
```

你想要更改参数的顺序，以便除数首先出现。这可以看成一个更普遍问题的特例：改变参数的顺序。

可编写一个泛化的 HOF，通过交换任何要修改的二元函数的参数顺序对该二元函数进行修改：

```
static Func<T2, T1, R> SwapArgs<T1, T2, R>(this Func<T1, T2, R> f)
    => (t2, t1) => f(t1, t2);
```

从技术层面讲，可更准确地认为 SwapArgs 会返回一个新函数，该函数会以相反的参数顺序调用给定函数。但从直观层面上讲，我觉得更容易想到的是我正在取回原始函数的一个修改版本。

你现在可通过应用 SwapArgs 来修改原始的除法函数：

```
var divideBy = divide.SwapArgs();
divideBy(2, 10) // => 5
```

使用这种类型的 HOF 会导致一个有趣的想法，即函数并非是一成不变的：如果你不喜欢函数的接口，可通过另一个函数来调用它，以提供更符合自己需要的接口。这就是将其称为适配器函数的原因。[8]

1.4.3 创建其他函数的函数

有时你会编写主要用于创建其他函数的函数——可将其视为函数工厂。下例使用 lambda 来过滤数字序列，只保留可被 2 整除的数字：

```
var range = Enumerable.Range(1, 20);

range.Where(i => i % 2 == 0)
// => [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

8 OOP 中众所周知的适配器模式可被看成将适配器函数的思想应用到一个对象的接口上。

如果你想要更通用的，比如能够过滤可被任何数字整除的数字 n，该怎么办呢？你可以定义一个函数，其接受 n 并生成一个合适的谓词，该谓词将计算任何给定的数是否可被 n 整除：

```
Func<int, bool> isMod(int n) => i => i % n == 0;
```

我们之前还没有像这样研究过 HOF：它接受一些静态数据并返回一个函数。下面分析如何使用它：

```
using static System.Linq.Enumerable;

Range(1, 20).Where(isMod(2)) // => [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
Range(1, 20).Where(isMod(3)) // => [3, 6, 9, 12, 15, 18]
```

请注意这使你不仅获得了通用性，还获得了可读性！在这个例子中，你使用名为 isMod 的 HOF 生成一个函数，然后将其作为输入提供给另一个 HOF，如图 1.7 所示。

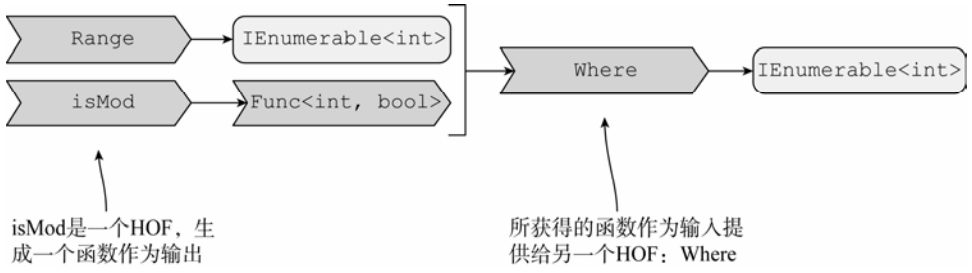


图 1.7　一个 HOF 生成一个函数，作为另一个 HOF 的输入

本书中你将看到使用 HOF 的更多情形。最终你将它们视为常规函数，而忘记它们是高阶函数。现在分析如何在日常开发中使用它们。

1.5　使用 HOF 避免重复

HOF 的另一个常见用例是封装安装和拆卸的操作。例如，与数据库进行交互需要一些设置来获取和打开连接，并在交互后进行一些清理以关闭连接并将其返回给底层连接池，代码如下所示。

代码清单1.12　连接到数据库需要一些安装和拆卸

```
string connString = "myDatabase";

var conn = new SqlConnection(connString));
conn.Open();                                  安装：获取并打开一个连接

// interact with the database…
```

```
conn.Close();
conn.Dispose();
```
拆卸：关闭并释放连接

无论你是正在读取数据库、写入数据库还是执行一个或多个动作，安装和拆卸都是一致的。前面的代码通常用一个 using 块编写，如下所示：

```
using (var conn = new SqlConnection(connString))
{
   conn.Open();
   // interact with the database...
}
```

这更简短、更好，[9]但本质上并无区别。研究下面这个简单的 DbLogger 类的例子，该类具有两个与数据库交互的方法：Log 插入给定的日志消息，GetLogs 从给定的日期开始检索所有日志。

代码清单1.13 安装/拆卸逻辑的重复

```
using Dapper;                                   ← 将 Execute 和 Query 公开为连接上的扩展方法
// ...

public class DbLogger
{
    string connString;                          ← 假设这是在构造函数中安装的

    public void Log(LogMessage msg)
    {
        using (var conn = new SqlConnection(connString)) ← 安装
        {
            int affectedRows = conn.Execute("sp_create_log"
                , msg, commandType: CommandType.StoredProcedure);
        }                                       ← 将 LogMessage 存入数据库
    }                                           ← 拆卸作为 Dispose 的一部分执行

    public IEnumerable<LogMessage> GetLogs(DateTime since)
    {
        var sqlGetLogs = "SELECT * FROM [Logs] WHERE [Timestamp] > @since";
        using (var conn = new SqlConnection(connString)) ← 安装
        {
            return conn.Query<LogMessage>(sqlGetLogs
                , new {since = since});
        }                                       ← 拆卸
    }                                           ← 查询数据库并对结果进行反序列化
}
```

注意，这两种方法有一些重复，即安装和拆卸的逻辑的重复。我们能否摆脱

9 它更简短，是因为将在你退出 using 块时调用 Dispose，并依次调用 Close；它更好，是因为交互将被封装在 try/finally 中，所以即使在 using 块的主体中抛出异常，也会丢弃连接。

这种重复呢？

与数据库交互的细节与本次讨论无关，但如果你感兴趣，代码将使用 Dapper 库(整理在 GitHub 上：https://github.com/StackExchange/dapper-dot-net)，它是 ADO.NET 顶层的一个薄层，允许你通过一个非常简单的 API 与数据库进行交互：

- Query 查询数据库并返回反序列化的 LogMessage。
- Execute 运行存储过程并返回受影响的行数。

这两种方法都被定义为连接上的扩展方法。更重要的是，注意这两种情况下，数据库交互取决于所获取的连接以及所返回的一些数据。这将允许你将 IDbConnection 中的数据库交互作为一个函数来表示“某事”。

在现实世界中，我建议你总是异步执行 I/O 操作(所以在这个例子中，GetLogs 应该真的调用 QueryAsync 并返回一个 Task<IEnumerable<LogMessage>>)。但是，异步增加了一些复杂性，在你尝试学习已经具有挑战性的 FP 时，将无助于你。第 13 章将进一步讨论异步。

如你所见，Dapper 公开了一个舒适的 API，并且如有必要，它甚至会打开连接。但你仍需要创建连接，并且一旦完成，应该尽快处理它。因此，数据库调用的结果最终被夹在执行安装和拆卸的相同代码段之间。下面分析如何通过将安装和拆卸逻辑提取到 HOF 中来避免这种重复。

1.5.1　将安装和拆卸封装到 HOF 中

你希望编写一个函数来执行安装和拆卸，并将其间的操作参数化。对于 HOF 来说，这是一个完美场景，因为你可以用一个函数来表示它们之间的逻辑关系。[10]如图 1.8 所示。

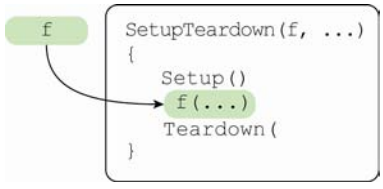


图 1.8　一个在安装和拆卸的逻辑之间包装给定函数的 HOF

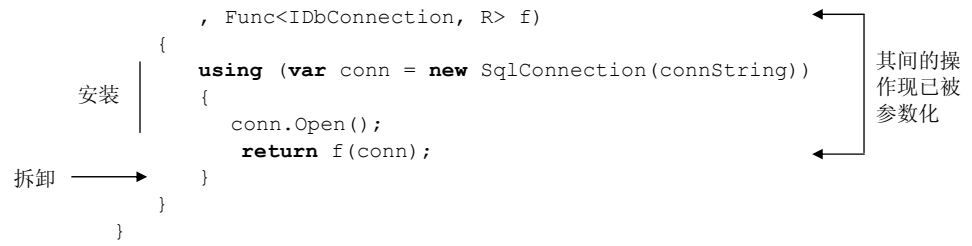
因为连接的安装和拆卸比 DbLogger 更普遍，所以可将它们提取到一个新的 ConnectionHelper 类。

代码清单1.14　将数据库连接的安装和拆卸封装到HOF中

```
using System;
using System.Data;
using System.Data.SqlClient;

public static class ConnectionHelper
{
   public static R Connect<R>(string connString
```

10　出于这个原因，你可能会听到这种模式被称为“中间洞”。



Connect 函数执行了安装和拆卸，并根据期间应发生的操作对其进行参数化。主体的签名很有趣，它接受一个 IDbConnection(通过它与数据库交互)，并返回一个泛型对象 R。在我们所见到的用例中，如在查询的情况下，R 将是 IEnumerable<LogMessage>，如在插入的情况下，R 将是 int。你现在可使用 DbLogger 中的 Connect 函数，如下所示：

```
using Dapper;
using static ConnectionHelper;

public class DbLogger
{
    string connString;

    public void Log(LogMessage message)
        => Connect(connString, c => c.Execute("sp_create_log"
            , message, commandType: CommandType.StoredProcedure));

    public IEnumerable<LogMessage> GetLogs(DateTime since)
        => Connect(connString, c => c.Query<LogMessage>(@"SELECT *
            FROM [Logs] WHERE [Timestamp] > @since", new {since = since}));
}
```

你摆脱了 DbLogger 中的重复逻辑，并且 DbLogger 不再知道有关创建、打开或处理连接的详细信息。

1.5.2 将 using 语句转换为 HOF

前面的结果虽然令人满意。但为了进一步理解 HOF，下面更激进些。试问，using 语句本身不是安装/拆卸的例子吗？毕竟，一个 using 块总会做到以下几点：

- **安装**——通过计算给定的声明或表达式来获取 IDisposable 资源。
- **主体**——执行块中的内容。
- **拆卸**——退出该块，致使在安装中所获取的对象上调用 Dispose。

是的，它是！至少有一个是的。安装并不总是相同的，所以也需要被参数化。然后，我们可编写一个更加泛化的安装/拆卸 HOF 来执行 using。

这是一种广泛复用的函数，属于库。本书将向你展示已在我的 LaYumba.Functional 库中存在的此类可复用构造，从而在函数式编码时实现更好的体验。

代码清单1.15　一个可用来代替using语句的HOF

```
using System;

namespace LaYumba.Functional
{
   public static class F
   {
      public static R Using<TDisp, R>(TDisp disposable
         , Func<TDisp, R> f) where TDisp : IDisposable
      {
         using (disposable) return f(disposable);
      }
   }
}
```

上面的代码清单定义了一个名为 F 的类，将包含函数式库的核心函数。思想是这些函数应该在没有被 using static 所限定的情况下可用，如下一个代码示例所示。

这个 using 函数接受两个参数：第一个是一次性资源，第二个是在资源被处理之前所执行的函数。这样，你可更简洁地重写 Connect 函数：

```
using static LaYumba.Functional.F;

public static class ConnectionHelper
{
   public static R Connect<R>(string connStr, Func<IDbConnection, R> f)
      => Using(new SqlConnection(connStr)
         , conn => { conn.Open(); return f(conn); });
}
```

第一行的 using static 使你可调用 using 函数来作为 using 语句的一种全局替换。请注意，与 using 语句不同的是，调用 using 函数的是一个表达式。[11]这有如下两个好处。

- 允许你使用更紧凑的表达式体式的方法语法。
- 一个表达式会有一个值，所以 Using 函数可与其他函数进行组合。

第 5.5.1 节将深入探讨组合以及语句与表达式的思想。

1.5.3　HOF 的权衡

下面分析通过比较 DbLogger 中的某个方法的初始版本和重构版本后所取得的成果：

```
// initial implementation
public void Log(LogMessage msg)
{
   using (var conn = new SqlConnection(connString))
   {
      int affectedRows = conn.Execute("sp_create_log"
```

11　这里简要回顾一下差异：表达式会返回一个值；而语句则不会。

```
            , msg, commandType: CommandType.StoredProcedure);
    }
}

// refactored implementation
public void Log(LogMessage message)
    => Connect(connString, c => c.Execute("sp_create_log"
        , message, commandType: CommandType.StoredProcedure));
```

这很好地说明了接受一个函数作为参数的 HOF 带来的好处：

- **简洁**——新版本显然更简洁。一般而言，安装/拆卸越复杂，使用越广泛，通过将其提取到 HOF 中获得的好处就越多。
- **避免重复**——整个安装/拆卸的逻辑，目前在一个地方执行。
- **关注分离**——你已经设法将连接管理隔离到 ConnectionHelper 类中，所以 DbLogger 本身只需要关注特定于日志的逻辑。

下面分析调用堆栈是如何改变的。在原始实现中，对 Execute 的调用发生在 Log 的堆栈帧上，在新实现中它们相距四个堆栈帧(见图 1.9)。

```
class DbLogger
{
   public void Log(LogMessage message)
      => Connect(connString, c => c.Execute("sp_create_log",
         , message, commandType: CommandType.StoredProcedure));
}

public static class ConnectionHelper
{
   public static R Connect<R>(string connStr, Func<IDbConnection, R> f)
      => Using(new SqlConnection(connStr)
         , conn => { conn.Open(); return f(conn); });
}

public static class F
{
   public static R Using<TDisp, R>(TDisp disposable
      , Func<TDisp, R> f) where TDisp : IDisposable
   {
      using (var disp = disposable) return f(disp);
   }
}
```

图 1.9 HOF 回调到调用函数中

当 Log 执行时，代码调用 Connect，并在连接准备就绪时传递回调函数来调用。Connect 依次将该回调重新打包为新回调，并将其传递给 Using。

所以，HOF 也有一些缺点：

- 增加了堆栈的使用。所以性能受到影响，但其可以忽略不计。
- 由于回调，调试应用程序会稍微复杂一些。

总体而言，DbLogger 的改进使其成为一个值得考虑的折中方案。你现在可能认同 HOF 是非常强大的工具，尽管过度使用可能会使得代码难以理解。适时可使用 HOF，但要注意可读性：使用简短的 lambda 表达式、清晰的命名以及有意义的缩进。

1.6　函数式编程的好处

上一节介绍了如何使用 HOF 来避免重复并更好地实现关注分离。确实，FP 的优点之一在于它的简洁性：用较少的代码行就可以获得相同的结果。典型的应用程序中拥有成千上万行代码，所以简洁性也会对应用程序的可维护性产生积极影响。

通过应用你在本书中学习的函数式技术，可获得更多好处，这些好处大致分为三类：

- **更干净的代码**——除了前面提到的简洁性外，FP 导致了更具表现力、更易读、更易于测试的代码。干净的代码不仅是一个开发人员的愉悦所在，而且通过降低维护成本，还为业务带来巨大的经济效益。
- **更好地支持并发**——从多核 CPU 到分布式系统的多个因素为你的应用程序带来了高度的并发性。并发在传统上与诸如死锁、丢失更新等难题相关联；FP 提供了防止发生这些问题的技术。本书第 2 章将介绍一个简单示例，本书最后将列举更高级示例。
- **一个多范式方法**——人们常说，如果你拥有的唯一工具是锤子，那么每个问题看起来都像钉子。相反，越是从更多的角度观察一个给定的问题，就越有可能会找到一个最佳的解决方案。如果你已经熟练掌握面向对象技术，学习像 FP 这样的不同范式将不可避免地给予你一个更丰富的视角。遇到问题时，你可考虑多种方法并选择最有效的方法。

练习

我建议你花点时间完成这些练习，并在练习中提出自己的一些想法。GitHub (https://github.com/la-yumba/functional-csharp-code)上的代码示例存储库包含了占位符，因此你可以使用最少的安装工作来编写、编译和运行代码。其中还包括可以检查结果的解决方案：

1. 浏览 System.Linq.Enumerable(https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable)的方法。哪些是 HOF？你认为哪一个隐藏着给定函数的迭代应用程序？

2. 编写一个可否定所给定谓词的函数：只要给定谓词的计算结果为 true，则结果函数的计算结果为 false，反之亦然。

3. 编写一个使用快速排序对 List<int>进行排序的方法(返回一个新列表，而不是就地排序)。

4. 泛化前面的实现以接受一个 List<T>，另外有一个 Comparison<T>委托。

5. 在本章中，你已经见到一个 Using 函数，它接受一个 IDisposable 函数和一个类型为 Func<TDisp,R>的函数。编写接受一个 Func<IDisposable>作为第一个参数的重载，而不是接受 IDisposable（这可用来避免由某些代码分析工具所提出的有关“实例化一个 IDisposable 而不处理它”的警告）。

小结

- FP 是一个强大的范式，可促使代码更简洁、更易于维护、表达性强、可测试且易于实现并发。
- FP 与 OOP 有所不同，其侧重于函数而不是对象，并且关注数据转换而不是状态突变。
- FP 可以被看成基于两个基本原则的一系列技术：
  - 函数是第一类值。
  - 应避免就地更新。
- 可用方法、委托和 lambda 来表示 C#中的函数。
- FP 利用高阶函数(接受其他函数作为输入或输出的函数)；因此语言必须具有作为第一类值的函数。