《深入 C#函数式编程》

[英] 西蒙·J·佩恩特（Simon J. Painter） 著

周子衿 译

周靖 校对&润色

中文试读版 1-4 章，翻译原稿，仅供参考，

京东购买>> 淘宝购买>> 当当购买>>

配套资源和试读下载：ys168 网盘>> 百度网盘>> GitHub 项目>>

访问中文版主页，获取最新资讯

清华大学出版社

北 京

本书由父女联袂奉献：女儿担纲翻译，父亲负责校对与润色。 女儿文风委婉准确，对新技术充满好奇；父亲治学严谨，尤重技术细节的精准呈现。在合作过程中，父亲不仅严格把关，更主动增补了大量译注，对原著内容进行了深入扩展与阐释（特别是对其中涉及的英式幽默与文化背景，提供了必要的解读）。同时，全书确保了C#语言特性与函数式编程概念在技术阐述上的精确无误。

——周靖，CHEF，Coder，Translator，Microsoft MVP

献给我的妻子，苏希玛·马哈迪克（Sushma Mahadik）。

也献给我的两个女儿，苏菲（Sophie）和凯蒂（Katie）。爸爸爱你们。

本书的溢美之辞：

西蒙（Simon）是我学习函数式编程时最喜欢的导师。这本书充分展现了他深厚的专业知识，为读者在这一领域的深入学习提供了专业的指导。

——皮特·加拉格尔（Pete Gallagher），微软技术聚会组织者，Avanade UK 开发部门经理

这本书不会空谈理论，而是会教你如何结合 C#已有的函数式特性和一些额外技巧，来构建实用的函数式编程解决方案。

——凯瑟琳·多拉德（Kathleen Dollard），微软.NET 首席项目经理

对于想要深入了解函数式编程并运用 C#的读者来说，本书将是一个清晰且实用的指南。西蒙没有滥用专业术语或过度侧重于抽象概念，而是将复杂的主题以浅显易懂的方式拆解开来，使读者能够轻松理解。生动的案例是这本书的一大亮点。西蒙精心挑选了一系列来自现实世界的案例，展现了函数式编程在 C#中的实际优势。书中探讨的内容绝非空中楼阁，而是可以立即应用到项目中的实用工具和技术。

——克里斯·艾尔斯（Chris Ayers），微软高级客户工程师

这本书是指导读者以函数式思维进行思考，并充分发挥 C#函数式特性，写出更简洁、更健壮的代码的绝佳指南。

——伊恩·拉塞尔（Ian Russell），*Essential F#*的作者

《C#函数式编程》是那些想提升编程技能的读者的绝佳选择。西蒙以一种既通俗易懂又充满趣味的风格，深入浅出地阐述了函数式编程的原理与应用。通过参考书中各种实用的示例，我的代码简洁性和可读性已经有了极大的提升。无论函数式编程的初学者，还是希望深入挖掘 C#潜能的资深开发者，都值得认真读一读这本书。

——亚历克斯·怀尔德（Alex Wild），高级软件工程师

这本书完美地将精巧的实例、深刻的理念和冷笑话融为了一体。

——马修·弗莱彻（Matthew Fletcher），Bravissimo 高级软件工程师

这本卓越的指南将函数式编程的精髓与 C#的强大能力巧妙地结合了起来，对想要提高代码清晰度和生产力的开发者来说，这本书非常值得一读。

——杰拉尔德·弗斯卢伊斯（Gerald Versluis），微软高级软件工程师

西蒙在我的.NET 用户组活动和播客节目中多次分享了他对函数式编程的深刻见解。鉴于西蒙在阐释问题和提供解决方案方面的卓越才能，我相信这本书将为读者带来许多启发，正如他在节目中的演讲一样。

——丹·克拉克（Dan Clarke），独立软件顾问和“Unhandled Exception Podcast”的主持人

这本书已经成为了我探索 C#函数式编程的首选资源，让我告别了对函数式编程的恐惧。

——马特·伊兰德（Matt Eland），微软 MVP 和 Leading EDJE 高级解决方案开发者

函数式编程（Functional Programming）？这是否意味着其他编程方式都有点功能失调（Dysfunctional）？

——理查德·坎贝尔（Richard Campbell），.NET Rocks! Podcast 的主持人

《C#函数式编程》将引领你轻松踏入函数式编程的广阔天地。西蒙以其独树一帜的风格清晰地解释了函数式编程的核心概念。对于想要了解函数式编程的读者，我强烈推荐这本书。

——普尔尼玛·奈尔（Poornima Nayar），自由职业.NET 开发者和微软 MVP

在《C#函数式编程》中，西蒙·佩恩特使得函数式编程易于被各阶层的 C#开发者所接受和运用，无论是初学者、资深开发者，还是像我这样固执己见的人。

——吉米·博加德（Jimmy Bogard），独立顾问

西蒙以简明扼要的方式讲解了函数式编程的益处。他通过代码示例说明了函数式编程的基础知识及更复杂的功能，并阐释了如何将这些知识应用于 C#编程实践中。

——艾萨克·亚伯拉罕（Isaac Abraham），*F# in Action* 的作者

作为一名初入开发领域的新手（尽管拥有超过 25 年的基础设施经验），一开始阅读《C#函数式编程》时，我感到有些吃力。然而，当西蒙使用“烘焙蛋糕”的比喻来解释概念时，一切都豁然开朗了。这本书以开发一个背景设定在 2147 年的“火星之旅”游戏作为结尾，不仅为我的学习之旅画上了完美的句号，更体现了书中知识的广度。

——彼得·德·坦德（Peter De Tender），微软技术培训师

目录

译者序

在翻开《深入 C# 函数式编程》这本书之前，我本以为它会像大多数技术书籍一样，枯燥难懂，让人昏昏欲睡。然而，作者西蒙 • J • 佩恩特（Simon J. Painter）那风趣幽默的文风瞬间颠覆了我的刻板印象。阅读这本书的过程，宛如与一位经验丰富的老朋友促膝长谈，他以轻松幽默的语言，将复杂难懂的函数式编程概念娓娓道来，使原本晦涩难懂的技术知识变得亲切易懂，充满趣味。

西蒙不愧是一位资深的软件开发者，他不仅精通 C# 语言，更对函数式编程有着深刻的理解和丰富的实践经验。在书中，他巧妙地将理论与实际案例相结合，用生动的比喻和形象的示例，为我们详细解读了 C#中的函数式编程特性。从基础的不变性、高阶函数，到模式匹配、递归等高级概念，他都讲解得深入浅出，让我们能够快速掌握函数式编程的精髓，并将其应用到实际的编程工作中

更难能可贵的是，西蒙在书中始终保持着积极乐观的态度，他用幽默的语言和轻松的笔触，化解了技术学习过程中的枯燥与压力。无论是面对复杂的代码，还是难以理解的理论，他总能以一种轻松的方式引导我们去思考和探索，让学习变成一种享受。这种独特的写作风格，不仅让读者在获取知识的同时，也收获了愉悦的阅读体验。

西蒙的幽默贯穿全书，让人忍俊不禁。例如，他在谈到“递归”这个概念时写道：“如果不知道递归是什么，请查看‘递归’一节。”另外，作为一个老派的英国人，西蒙特别喜欢《007》和《神秘博士》系列，所以书中用到了它们的大量“段子”。译者也贴心地添加了注释，方便读者理解。

但是，别以为这本书只有玩笑和调侃。西蒙的专业能力十分深厚，他把技术知识用幽默的方式呈现出来，讲解得清晰又不失严谨。书中逐一讲解了函数式编程的核心理念——不变性、高阶函数、引用透明性等，并通过一个个贴近实际的例子，把它们的魅力展现得淋漓尽致。

那么，函数式编程到底有哪些优势呢？简单来说，它是一种强调“简洁、可靠和可预测性”的编程范式。比如，函数式编程的“不变性”要求变量一旦赋值便不能改变，从而减少了因为状态变动导致的潜在问题；“引用透明性”则保证了函数在相同输入下始终返回相同的输出，这大大提高了代码的可预测性和可测试性。这些特性让函数式编程在处理多线程任务时表现得尤为出色。

《深入 C#函数式编程》最大的亮点在于，它非常贴近 C#开发者的实际需求。书中没有空泛的理论堆砌，而是通过实用的代码示例展示了如何将函数式编程的思想融入日常工作中。当然，作者也没有回避函数式编程的学习曲线的问题。对于习惯了面向对象思维的开发者来说，函数式编程的确需要一点时间适应。但是，为它付出一些时间是值得的。

近年来，函数式编程呈现出迅猛的发展态势，在软件开发领域占据了愈发重要的地位。随着多核处理器的普及以及大数据和云计算的兴起，传统编程范式在应对并发和分布式系统时面临诸多挑战，而函数式编程凭借其独特的特性，如不可变数据、纯函

数和高阶函数等，为这些难题提供了优雅的解决方案。这使得函数式编程在处理复杂业务逻辑和大规模数据处理时，展现出更高的效率和稳定性，吸引了许多开发者的目光。

总的来说，本书是一本“严谨而不失幽默”的技术指南。读者不仅可以在欢笑中理解函数式编程的核心思想，也可以学会如何在 C#中灵活应用这些工具。希望读者在阅读这本书时，既收获技术的提升，也感受到学习编程的乐趣。

最后，当涉及到 C#编程，不得不提一下清华大学出版社出版的“C#三剑客”。首先，《深入 CLR》第 4 版（原《CLR via C#》）为您提供了一个高瞻远瞩的视角，深入解析了运行时和框架——C#语言的根基。语言的每一项设计都是基于这个核心架构构建的。其次，《C# 12.0 本质论》，它如同一本全面且深入的语言百科全书，覆盖了 C#语言的方方面面。最后，《Visual C#从入门到精通》第 10 版虽然同样涵盖了语言的基础知识，但更多地是从 GUI 编程的角度帮助您理解 C#编程。注意，包括本书在内，所有这些书的代码均通过了 Visual Studio 2022 的测试，读者可以访问 https://bookzhou.com 来获取。

——周子衿

前言

我经常参加开发者大会，从这些会议中，我观察到函数式编程（Functional Programming，FP）的讨论热度似乎一年高过一年。许多会议都有一个专门讨论函数式编程的议程，并且其他演讲中也多少会提到这一主题。

函数式编程的重要性正在稳步上升。这背后的原因是什么呢？

原因在于，函数式编程是软件开发史上最伟大的创新之一。它不仅很酷，还充满乐趣。

随着诸如容器化和无服务器应用这样的概念的兴起，函数式编程不再只是开发者的业余爱好，也不是那种几年后就会被遗忘的短暂热潮，而是成为了能为利益相关者带来实质性好处的重要概念。

此外，在.NET 生态系统中，函数式编程的推广还得益于一些关键因素。例如，C#的首席设计师马德斯·托格森（Mads Torgerson）就是函数式编程的忠实拥趸，也是将函数式编程引入.NET 的主要推动者之一。同时，作为.NET 家族中的函数式编程语言，F#的影响也不容忽视。鉴于 F#与 C#共享相同的运行时环境，F#团队开发的许多函数式特性往往也会以某种形式被集成到 C#中。

然而，一个重要的问题仍然存在：函数式编程究竟是什么？我是不是需要学习一门全新的编程语言才能使用它？好消息是，如果你是一名.NET 开发者，那么不需要为了紧跟潮流而花费大量业余时间学习新技术，也不需要引入新的第三方库来增加应用程序的依赖：用开箱即用（out-of-the-box）的 C#代码即可实现函数式编程，做一些小小的调整即可。

本书将介绍函数式编程的基本概念，展现其优势，并说明如何在 C#中实现它们——学习这些知识不仅仅能满足你的个人编程爱好，更能为你的工作带来直接好处。

这些好处包括：

- 代码更加清晰、整洁且易于理解
- 代码库更容易维护
- 减少应用程序中未处理的异常，避免它们所带来的不可预测的后果
- 能更轻松地为代码库编写自动化单元测试

除此之外，还有其他很多好处。

谁应该阅读本书？

无论是专业开发人员、学生还是编程爱好者，只要已经掌握了 C#的基础知识，这本书就是为你准备的。虽然不需要达到专家水平，但需要熟悉基础知识，并且至少能够独立编写简单的 C#应用程序。

虽然书中涵盖了一些更高级的.NET 知识，但在谈到这些知识时，我会对他们进行详细说明。

本书是为以下几类人编写的：

- 如果已经掌握了 C#基础知识，但为了编写出更好、更健壮的代码，想通过学习更高级的技术来进一步提升自己，那么本书值得一读。
- 对于.NET 开发者，如果曾听说过函数式编程或甚至对它有一定的了解，并且想知道如何在 C#中采用这样的编码风格，本书将引领你启程。
- 如果是 F#开发者，希望继续使用熟悉的函数式编程工具，本书将展示如何做到这一点。
- 如果是从其他支持函数式编程的语言（如 Java）转向.NET 的开发者，本书将是一个宝贵的资源。
- 如果真心热爱编程——即使在办公室里写了一天代码后，回家还会出于兴趣而写一写自己的应用——那么本书很适合你。

写这本书的原因

我从小就对编程有浓厚的兴趣。小时候，我家里有一台 ZX Spectrum，这是一款由 Sinclair Research 在 80 年代初开发的家用电脑。对于熟悉 Commodore 64 的人来说，ZX Spectrum 或许会让他们感到似曾相识，但相比之下，它更加原始。ZX Spectrum 只支持 15 种颜色和 256×192 的屏幕分辨率。[1]我使用的是有 48K 内存的高级型号，而我父亲使用的是更老的 ZX81 型号，其内存仅有 1K，键盘则是橡胶材质。它甚至无法显示彩色的游戏角色，只有屏幕上的色块，因此游戏中的人物角色会根据背景的颜色而改变色彩。总而言之，它们真的很酷！

最让人兴奋的是，它的操作系统实际上是一个文本编程界面。我需要通过输入代码来加载游戏（使用 `LOAD ""`命令从磁带加载）。当时市面上有一些为孩子们准备的包含游戏代码的杂志和书籍，正是这些资料培育了我对计算机代码的热爱。特此向奥斯朋出版社（Usborne Publishing）表示感谢！

在我 14 岁左右的时候，学校电脑上的一个就业指导程序建议我考虑从事软件开发职业。那时我才第一次意识到，原来可以把这个不太正经的爱好转变成赚钱的手段！

[1] 它有 8 种基本颜色，每种颜色都有一个亮色版本。不过，其中一个颜色是黑色，而它并不存在亮色版本。所以，总共是 15 种颜色。

大学毕业后，我决定找一份正式的工作，那是我第一次接触到 C#。顺理成章地，我设定了下一目标：找到编写代码的恰当方式。这听起来很简单，对吧？但老实说，在差不多二十年之后，我仍在努力弄清楚如何做到这一点。

我编程生涯的关键转折点是挪威的一次开发者大会。在那次大会后，我终于理解了我闻名已久的函数式编程究竟是什么。函数式编程的代码非常优雅和简洁，而且易于阅读，这是其他编程风格难以企及的。虽然和其他类型的编程风格一样，采用函数式编程仍然有可能编写出结构不佳的代码，但函数式编程给我带来了一种前所未有的体验，让我感觉自己终于找到了一种“恰当”的编程方式。希望你在阅读这本书后，不仅会认同这个观点，还会被这个充满无限可能的编程世界深深吸引。

本书导航

本书的结构如下：

- 第 1 章是引言，介绍了如何立即开始使用 C#进行函数式编程，无需引入任何新的 NuGet 包、使用第三方库或者对 C#语言进行特殊处理。这一章的几乎所有示例都适用于 C# 3.0 及以后的版本。这一章是踏上函数式编程之旅的第一步，其中的所有代码都相对简单，为本书后续的内容奠定了基础。
- 第 I 部分，“我们已经在做的事”（第 2 章到第 4 章），探讨了如何将函数式编程的一些理念自然地融入日常 C#编程中，而不需要进行任何根本性的改变。在这一部分中，许多代码示例都是直接使用标准 C#实现的。如果你之前从未听说过函数式编程，想要循序渐进地入门，那么这一部分将是个很好的起点。打个比方，这就像是轻轻地将脚浸入水中，看看自己是否对下水游泳感兴趣。
- 第 II 部分，“深入函数式编程”（第 5 至第 10 章），标志着深入探讨的开始，这一部分会介绍一些“真正”的函数式编程概念。不过别担心，我会把这些概念拆解开来，由浅入深地讲解它们。
- 第 III 部分，“走出迷雾”（第 11 至第 14 章），旨在总结并巩固你所学到的知识，并介绍其他一些值得钻研的领域。

随意挑选一个你最感兴趣的部分开始阅读。这本书不是小说，[2]请按照对自己而言最合适的顺序阅读。

排版约定

本书使用的排版约定如下：

粗体

表示新术语、URL、电子邮件地址、文件名和文件扩展名。

`Constant width(等宽字体)`

[2] 但是，假如这是一本小说，那么我保证它会是一部谋杀悬疑小说，而且凶手肯定是管家！

表示程序代码和段落内的程序元素，如变量名、函数名、数据库、数据类型、环境变量、语句和关键字等。

 这个元素表示一般性的附注。

 这个元素表示警告或需要注意的事项。

使用代码示例

可以在以下网址下载本书的补充材料（包括代码示例、练习等）：
https://oreil.ly/functional-programming-with-csharp-code

如果在使用代码示例时遇到技术问题或有其他疑问，请通过以下电子邮件地址联系我们：bookquestions@oreilly.com。

这本书旨在帮助你学习。一般来说，除非计划大量复制或使用书中的代码，否则可以随意在自己的程序和文档中应用本书提供的示例代码，无需事先征得我们的同意。举例来说，编写一个包含书中多个代码片段的程序不需要特别许可。但是，若要出售或分发 O'Reilly 书籍中的示例代码，则必须获得我们的正式许可。引用本书和书中的示例代码来回答问题不需要特别许可，但是，如果在你的产品文档中大量引用了本书的示例代码，就需要获得正式许可。

虽然不是强制性要求，但我们总是欢迎并感激任何对书籍归属（Attribution）的标注。书籍归属应包含书名、作者、出版社和 ISBN 号。例如：“《C#函数式编程》作者：西蒙 • J • 佩恩特（O'Reilly 出版社），2023 年版权所有，西蒙 • J • 佩恩特，ISBN 978-1-492-09707-5。”

如果认为自己使用代码示例的方式可能超出了合理使用范围或上述许可范围，请随时通过以下邮箱联系我们：permissions@oreilly.com。

致谢

首先要感谢的是凯瑟琳 • 多拉德（Kathleen Dollard）。几年前，她在奥斯陆的挪威开发者大会（NDC）上发表了题为“C#的函数式技巧”的演讲。那是我第一次真正接触到函数式编程，它为我打开了新世界的大门（https://oreil.ly/nBpWu）。

在这段探索的旅程中，恩里科 • 布奥南诺（Enrico Buonanno）也是我的一位重要导师。通过阅读他所撰写的《C#函数式编程：编写更优质的 C#代码》（Functional

Programming in C#）一书，我得以理解了一些复杂的函数式概念的运作方式。如果你对本书感兴趣，强烈推荐阅读他的作品。

感谢各位阅读了本书的早期草稿并提供了宝贵反馈的人，包括伊恩·拉塞尔（Ian Russell）、马修·弗莱彻（Matthew Fletcher）、利亚姆·莱利（Liam Riley）、马克斯·迪特泽（Max Dietze）、史蒂夫·柯林斯（Steve Collins）、赫拉尔多·李斯（Gerardo Lijs）、马特·伊兰（Matt Eland）、拉胡尔·纳特（Rahul Nath）、西瓦·古迪瓦达（Siva Gudivada）、克里斯蒂安·霍斯达尔（Christian Horsdal）、马丁·福斯（Martin Fuß）、戴夫·麦克洛（Dave McCollough）、塞巴斯蒂安·罗宾斯（Sebastian Robbins）、大卫·谢弗（David Schaefer）、彼得·德·坦德（Peter De Tender）、马克·西曼（Mark Seeman）、杰拉尔德·弗斯鲁斯（Gerald Versluis）、亚历克斯·怀尔德（Alex Wild）、瓦拉迪斯·诺瓦科维茨（Valadis Novakovits）、莱克纳·莱因哈德（Lackner Reinhard）、埃里克·卢卡斯（Eric Lucas）、克里斯托弗·斯特拉滕（Christopher Straten）、凯瑟琳·多拉德（Kathleen Dollard）以及斯科特·瓦斯欣（Scott Wlaschin）。非常感谢大家！

还要特别感谢我的编辑，吉尔·伦纳德（Jill Leonard）。在过去的一年里，她始终对我有着无限的耐心。

第 1 章 引言

如果你以前学过编程，无论是 C#、Microsoft Visual Basic、Python 还是其他任何编程语言，你学到的知识很可能都基于目前最主流的编程范式——面向对象编程（Object-Oriented Programming，OOP）。面向对象编程已经存在很长一段时间了，虽然确切的时间存在争议，但它很可能诞生于 20 世纪 50 年代末或 60 年代初。

面向对象编程围绕一个核心理念展开：将数据片段（也就是属性）和功能封装到称为“类”的逻辑代码块中。这些类被用作实例化对象的模板。面向对象编程还涉及其他许多概念，比如继承、多态、虚拟方法和抽象方法等。

但本书的重点不是面向对象编程。实际上，如果对面向对象编程有深入的了解，最好暂时将这些知识抛到脑后，这样会从这本书中取得更多收获。

本书将介绍一种编程风格，它是面向对象编程的一种替代方案：**函数式编程**（Functional Programming，FP）。虽然函数式编程在近几年才逐渐进入主流视野，但它的历史可能比面向对象编程还要久远。函数式编程基于 19 世纪末到 20 世纪 50 年代之间发展的数学理论，并且自 60 年代起，就已经成为了一些编程语言的特性。我将向你展示如何在不学习一种全新的编程语言的前提下，在 C#中应用函数式编程。

在开始写代码之前，我想先谈谈函数式编程本身。它究竟是什么？为何值得关注？它最适合在哪些情况下使用？这些都是非常重要的问题。

函数式编程是什么？

函数式编程围绕一些基本概念展开，虽然这些概念的名称可能有些晦涩，但它们本身并不复杂。我会尽量以最简洁明了的方式来阐述它们。

函数式编程是编程语言、API，还是别的什么？

函数式编程既不是一门编程语言，也不是 NuGet 中的某个第三方库；它是一种编程范式。这意味着什么？虽然“范式”（paradigm）一词有着更为正式的定义，但我认为它可以被看作是一种编程“风格”（style）。就像吉他可以用来演奏风格迥异的音乐一样，某些编程语言也支持多种不同的编程风格。

函数式编程的历史可能比面向对象编程还要久远。我将在“函数式编程的起源”一节中进一步讨论它的起源，但现在只需要明白，函数式编程并不是什么新概念，它背后的理论不仅比面向对象编程更早诞生，甚至可能比计算机行业还要早。

值得一提的是，不同的编程范式可以混合使用，就像结合了两种音乐风格的摇滚爵士乐一样。不仅可以结合使用不同的编程范式，还可以利用每种范式的优点，创造出更好的结果。

尽管编程范式有很多种“口味”[3]，但为了简单起见，我接下来将重点介绍现代编程中两种最常见的范式。

命令式（Imperative）

这在很长一段时间内都是唯一一种编程范式。过程式编程和面向对象编程都属于这个类别。这种编程风格倾向于明确指导执行环境，详细说明需要执行的每一步，包括涉及的变量、中间步骤以及整个流程的执行顺序等。这种编程方式在学校和职场培训中极其常见。

声明式或宣告式（Declarative）

在这种编程范式中，我们不太关心如何具体实现目标。这种范式下的代码更像是对期望结果的高层次概述，而如何达到这些结果的具体步骤和顺序则交由执行环境决定。函数式编程就属于这个类别。结构化查询语言（Structured Query Language，SQL）也属于这个类别，所以从某些方面来看，函数式编程更类似于 SQL 而不是面向对象编程。在编写 SQL 语句时，不需要关心操作的顺序（并不是真的要先执行 `SELECT`，然后是 `WHERE`，接着是 `ORDER BY`[4]），也不需要关心数据转换的具体实现细节。相反，只需要编写一个脚本，有效地描述想要的输出即可。这种编程理念也是函数式 C#所追求的。因此，对那些使用过 Microsoft SQL Server 或其他关系型数据库的人来说，接下来介绍的一些概念会比较容易理解。

尽管还有其他许多编程范式，但它们并不在本书的讨论范围内。公平地说，其他许多范式都相当晦涩难懂，所以你不太可能在日常工作中遇到它们。

函数式编程的特性

接下来将分别探讨函数式编程的几个核心属性，以及它们对开发者的实际意义。

不变性

如果某个事物可以改变，那么就说它具有**变化**（mutate）的能力，就像忍者神龟[5]一样。换句话说，如果某物能发生变化，那么它就是**可变的**（mutable）。相反，如果某物完全无法改变，那么它就是**不可变的**（immutable）。

---

[3] 比如香草口味，还有我的最爱：香蕉口味。

[4] 译注：虽然 SQL 语句通常按照 `SELECT`，`FROM`，`WHERE`，`GROUP BY`，`HAVING`，`ORDER BY` 的顺序书写，但这并不代表数据库引擎会严格按照这个顺序执行操作。实际上，数据库引擎会根据优化器（optimizer）的分析，选择一个最优的执行计划，这个计划可能与语句的书写顺序不同。

[5] 在我小时候，大约是上世纪 90 年代，它们在英国被称为“英雄神龟”（Hero Turtles），我猜当时的电视节目制作人是想要规避“忍者”一词所隐含的暴力色彩。尽管如此，他们却完全没有避讳英雄们使用锋利的武器对抗坏蛋的场景。

在编程语境中，这意味着一旦变量被赋值，就再也不能被修改了。如需一个新值，那么必须新建一个变量，而不是更改现有变量的值。这是函数式编程中处理变量的标准做法。

这种方式与命令式编程稍有不同，但用它创建程序的过程更接近于我们在数学中逐步求解的过程。这种方法促进了结构的合理性和代码的可预测性，从而使代码更加**健壮**（robust）。

在.NET 中，`DateTime` 和 `String` 都是不可变数据结构。你可能*以为*自己修改了它们，但实际上，每次所谓的“修改”都会在栈上创建一个全新的对象。这就是为什么很多新手程序员经常听到的一个建议是“不要在 `for` 循环中直接拼接字符串”——这确实是一个绝对应该避免的做法。

高阶函数

高阶函数（Higher-order function）能以变量的形式传递——无论是作为局部变量、函数的参数还是作为函数的返回值。`Func<T, TResult>`或 `Action<T>`委托类型就是典型的例子。

下面简要介绍一下**委托**（Delegate）的工作原理。委托本质上是以变量形式存储的函数。它们接受一组泛型类型（Generic Type）作为参数和返回类型（如果有的话）。`Func` 和 `Action` 之间的区别在于，`Action` 不返回任何值——也就是说，它是一个不包含 `return` 关键字的 `void` 函数。`Func` 中列出的最后一个泛型类型就是它的返回类型[6]。

以下面这两个函数为例：

```
// 给定参数 10 和 20，这将输出以下字符串：
// "10 + 20 = 30"
public string ComposeMessage(int a, int b)
{
    return a + " + " + b + " = " + (a + b);
}

public void LogMessage(string a)
{
    this.Logger.LogInfo("message received: " + a);
}
```

它们可以重写为委托类型，如下所示：

```
Func<int, int, string> ComposeMessage =
    (a, b) => a + " + " + b + " = " + (a + b);

Action<string> LogMessage = a =>
    this.Logger.LogInfo($"message received: {a}");
```

[6] 译注：例如，`Func<string, int, bool>` 表示接受一个 `string` 和一个 `int` 参数，并返回一个 `bool` 值的函数。

这些委托类型可以像标准函数一样调用：

```
var message = ComposeMessage(10, 20);
LogMessage(message);
```

使用委托类型的一个主要优势是它们可以作为变量来存储，并在整个代码库中进行传递。这些委托不仅可以作为参数传递给其他函数，还可以作为返回类型从函数返回。如果使用得当，它们是 C#中非常强大的特性之一。

通过函数式编程的方法，我们可以将委托类型组合起来，从而将较小的函数构建块组合成更大、更复杂的函数——就像用乐高积木搭建千年隼号模型（或者你喜欢的任何模型）一样。这正是该范式被称为**函数式**（functional）编程的真正原因：因为我们用**函数**构建应用程序。当然，这里不是说用其他范式编写的代码就不起作用了（doesn't function）[7]。毕竟，要是那些范式真的不起作用，谁会使用它们呢？

实际上，这里有一个经验法则：对于任何问题，函数式编程的答案总是“函数，函数，再加上更多的函数”。

编程中存在两种类型的可调用代码单元：函数和方法。函数总是返回一个值，方法则不一定。在 C#中，函数会返回某种类型的数据，而方法的返回类型可能是`void`。由于方法几乎总是会带来副作用[8]，所以应该尽量避免在代码中使用它们——除非避无可避。例如，日志记录就是一个必须使用方法的例子，在编写高质量生产代码时，它是不可或缺的。

首选表达式而非语句

为了理解接下来的内容，我们需要先明确几个定义。这里所说的“表达式”指的是能计算出值的独立代码单元。具体是什么意思呢？

下面是一些最简单的示例表达式：

```
const int exp1 = 6;
const int exp2 = 6 * 10;
```

也可以通过输入值来构建表达式，如下所示：

```
public int addTen(int x) => x + 10;
```

虽然下例执行了一个操作——判断条件是否为真——但它最终只是简单地返回了一个布尔值，因此依然被视为一个表达式：

---

[7] 译注：作者在这里用 function 的不同意思（函数、起作用、功能等）开了一个玩笑。

[8] 译注：在计算机编程中，如果一个函数/方法或表达式除了生成一个值，还会造成对象状态的改变（例如，写入文件或者创建数据库连接等），就说它会造成副作用；或者说会执行一个副作用。

```
public bool IsTen(int x) => x == 10;
```

用三元操作符（`if-else` 语句的简写形式）来确定返回值时，也可以将其视为表达式：

```
var randomNumber = this._rnd.Generate();
var message = randomNumber == 10
    ? "等于 10 "
    : "不等于 10 ";
```

另一个简单的经验法则是，如果一行代码包含一个等号，那么这行代码很可能是表达式，因为它在进行赋值操作。但这个规则并非绝对，因为对其他函数的调用可能会带来一些不可预见的结果。尽管如此，这个经验法则仍然值得一记。

与之相对，语句是不会求值出一个数据结果的代码片段，以分号结尾。它们通常用于执行某些操作，比如通过 `if`、`where`、`for` 和 `foreach` 等关键字来指示执行环境改变执行顺序，或者调用不返回任何值的函数，而这些调用本质上是在执行某种操作。下面是一个例子：

```
this._thingDoer.GoDoSomething();
```

最后一个经验法则是，如果代码中不包含等号，那么它*肯定*是语句[9]。

基于表达式的编程

为了进一步理解，不妨回想一下小时候的数学课。在解题时，是不是需要逐步写出计算过程，以证明你是如何得出答案的？基于表达式的编程也是同理。

每一行都是一个完整的计算步骤，并且建立在之前一行或多行的基础上。通过编写基于表达式的代码，计算过程会在函数运行时被固定下来。这种方式的一个优势在于，它使代码更容易调试，因为可以回顾之前的所有值，并且确信这些值不会因为循环的上一次迭代或其他类似情况而发生改变。

采用这种方法可能看似不切实际，仿佛是要求在绑住双手的情况下编程。但实际上，这完全是可行的，甚至都不能说太难。在 C#中，所需的大部分工具在过去十年中都已经就绪，而且还有更多高效的结构可供选择。

下面用一个例子来说明我的观点：

```
public decimal CalculateHypotenuse(decimal b, decimal c)
{
    var bSquared = b * b;
    var cSquared = c * c;
    var aSquared = bSquared + cSquared;
    var a = Math.Sqrt(aSquared);
    return a;
}
```

[9] 特别感谢函数式编程专家马克·西曼（Mark Seemann）（https://blog.ploeh.dk），是他给出了这些实用的经验法则。

严格来说，可以在一行中写所有这些代码，但那样的话，代码就不那么容易阅读和理解了，对吧？还可以像下面这样编写代码，省去所有的中间变量：

```
public decimal CalculateHypotenuse(decimal b, decimal c)
{
    var returnValue = b * b;
    returnValue += c * c;
    returnValue = Math.Sqrt(returnValue);
    return returnValue;
}
```

这里的问题在于，缺少变量名称使得代码难以阅读，而且所有中间计算结果都丢失了。这意味着，如果出现了 bug，就必须逐步检查每个阶段的 `returnValue`。而在基于表达式的解决方案中，所有计算过程都会被清晰地保留在原地。

习惯了这种方式之后，可能会很不适应以前的编程方式，甚至会觉得它有些笨重和不方便。

引用透明性

引用透明性（Referential Transparency）一词看上去似乎颇为“高大上”，但它其实是一个很简单的概念。在函数式编程中，**纯函数**（Pure Function）具有以下特点：

- 不会改变函数外部的任何东西。这意味着它们没有“副作用”，不会更新状态，不会写入文件，不会创建数据库连接等等。
- 给定相同的参数值，总是返回相同的结果，无论系统的状态如何。
- 纯函数不会引发任何意外的副作用，包括抛出异常在内。

引用透明性这一概念基于这样一个理念：给定相同的输入，总是能得到相同的输出。因此，在进行计算时，理论上可以把对函数的调用替换成它对于给定输入的输出结果。请看以下示例：

```
var addTen = (int x) => x + 10;
var twenty = addTen(10);
```

调用 `addTen()`时，如果参数值为 `10`，那么其结果*总是* `20`，没有任何例外。这个函数非常简单，因此也不可能出现副作用。因此，从原则上讲，可以将 `addTen(10)`的调用替换为常量值 `20`，而不会引起任何副作用。这就是所谓的引用透明性。

下面列举了纯函数的几个例子：

```
public int Add(int a, int b) => a + b;  // 计算两个整数 a 和 b 之和

public string SayHello(string name) =>  // 生成问候语。
    "Hello " + // 问候语的开头
    (string.IsNullOrWhitespace(name) // 检查输入的 name 是否为空或仅包含空白字符
        // 如果 name 为空，那么返回一句友好的问候
        ? "I don't believe we've met. Would you like a Jelly Baby?"
        : name); // 如果 name 不为空，就在问候语中加入 name
```

这些函数不会引起任何副作用（我已经确保对字符串进行了空值检查），它们不会改变函数作用域之外的任何状态；它们只会生成并返回一个新的值。

以下是相同函数的非纯（impure）版本：

```
public void Add(int a) => this.total += a; // 改变了状态
public string SayHello() => "Hello " + this.Name;
// 从状态而不是参数中获取值
```

这两个示例都包含了对当前类的属性的引用，这超出了函数本身的作用域。Add()函数修改了状态属性，而 SayHello()函数也没有执行空值检查。由此可以看出，这些函数都不是纯函数，它们都会产生“副作用”。

那么，以下函数又如何呢？

```
public string SayHello() => "Hello " + this.GetName();

public string SayHello2(Customer c)
{
    c.SaidHelloTo = true;
    return "Hello " + (c?.Name ?? "Unknown Person");
}

public string SayHello3(string name) =>
    DateTime.Now + " - Hello " + (name ?? "Unknown Person");
```

这些函数很可能都不是纯函数。

SayHello()依赖于一个外部函数。我不清楚 GetName()具体执行了什么操作[10]，如果它只会返回一个常量，那么可以将 SayHello()视为纯函数。然而，如果它执行的是数据库表的查询操作，那么任何数据缺失或网络数据包丢失都可能导致抛出异常（这些都是意外产生副作用的例子）。如果*必须*用一个函数来检索姓名，我会考虑使用 Func<T, TResult>委托重写该函数，以安全地将这个功能注入 SayHello()函数中[11]。

SayHello2()修改了传入的对象——这是该函数一个明显的副作用。在面向对象编程中，以“传引用”的方式传递对象并进行修改是常见的做法，但在函数式编程中这是绝对禁止的。要把它变成纯函数，我可能会考虑将更新对象属性的操作和显示问候语的逻辑分离成两个不同的函数来实现。

SayHello3()使用了 DateTime.Now，这会导致每次调用都返回不同的值，完全背离了纯函数的原则。修正该问题的一个简单方法是为函数增加一个 DateTime 类型的参数，并在调用函数时传入具体的时间值。

引用透明性是提高函数式代码可测试性的重要特性之一。但这也意味着必须采取其他方式来跟踪状态，稍后的“函数式编程的起源”一节将进一步探讨这个话题。

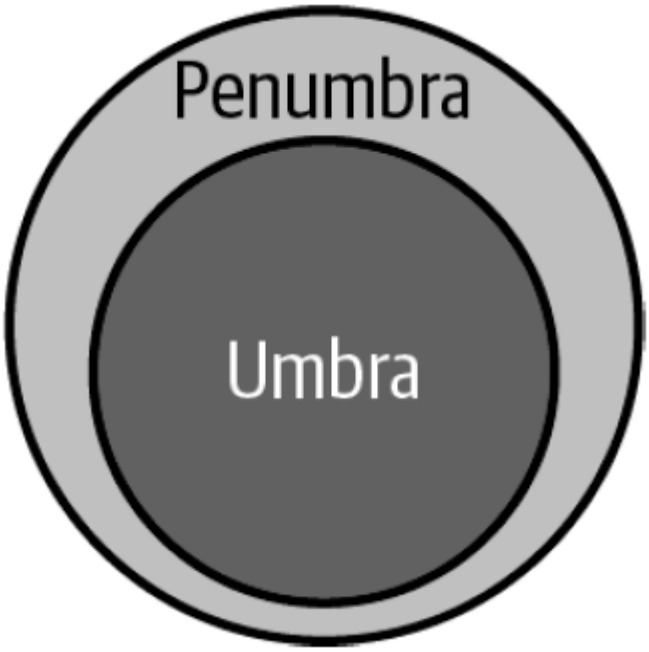
---

[10] 它是我为了这个例子而现编的。

[11] 译注：例如，可以像下面这样重构 SayHello 函数：public string SayHello(Func<string> getName) => "Hello " + getName();。这样一来，SayHello 现在只依赖于传入的 getName 函数，不再直接调用 this.GetName()。只要 getName 函数是纯函数，那么 SayHello 也将是纯函数。

此外，应用程序能实现的“纯度”（purity）是有限的，特别是在需要与外部世界、用户或不遵循函数式编程范式的第三方库交互时。在 C#中，我们不得不在某些方面做出一些妥协。

我喜欢通过一个比喻来解释这一点。如图 1.1 所示，阴影由两部分组成：本影（umbra）和半影（penumbra）。[12]其中，本影是实心的、深色的部分（它占据了阴影的大部分）。半影是外围一圈模糊的灰色部分，它是阴影与非阴影相交并逐渐过渡的区域。在 C#应用程序中，我会将代码库中的纯净区域视为本影，需要妥协的区域则视为半影。我的目标是尽可能扩大纯净区域，同时尽可能缩小非纯净区域。



（请如下替换图中英文：

Penumbra：半影

Umbra：本影）

图 1-1 本影和半影

如果想了解这种架构模式的正式定义，加里·伯恩哈特（Gary Bernhardt）曾在演讲中将其称为“函数式核心，命令式外壳”（https://oreil.ly/-ooC4）。

[12] 好了，大艺术家们，我知道阴影实际上大约由 12 个部分组成，但就这个比喻而言，两个部分就已经足够了。

递归

递归（Recursion）的历史几乎和编程本身一样久远。它是一种函数自我调用的机制，用来创建一个次数不定（但理想情况下并非无限）的循环。任何人只要写过代码来遍历文件夹结构或者执行高效排序算法，都应该不会对递归感到陌生。

递归函数通常由两部分组成：

- 条件：用来判断是需要再次调用函数，还是已经达到了结束状态（例如，已经计算出了目标值，或者没有更多的子文件夹需要遍历等）。
- 返回语句：根据是否已经达到结束状态，要么返回一个最终值，要么再次调用这个函数本身。

下面是一个非常简单的递归 `Add()`示例：

```
public int AddUntil(int startValue, int endValue)
{
    // 基本情况：如果起始值大于或等于结束值，直接返回起始值
    if (startValue >= endValue)
        return startValue;

    // 递归情况：否则，起始值递增 1，继续递归调用，直到满足基本情况
    else
        return AddUntil(startValue + 1, endValue);
}
```


不要将这个例子用于生产代码。为了方便解释，我对它进行了简化。

尽管这个例子看起来有点“傻”，但要注意，在这个过程中，我没有修改任何一个整型参数的值。在递归函数中，每一次函数调用时使用的参数，都是由这次调用接收到的参数决定的。这是不可变性的又一个实例：我没有修改变量中的值；而是根据接收到的值，通过一个表达式来进行函数调用[13]。

递归是函数式编程用来替代 `while` 和 `foreach` 这类语句的方法之一。但是，在 C#中使用递归时，可能会遇到一些性能问题。以不定循环为主题的第 9 章将会进一步探讨递归的使用。现在，你只需要谨慎使用递归，然后继续读下去就行。一切很快就会变得清晰起来的……

[13] 译注：每次递归调用时，`startValue` 的值都会增加 1，但这是通过创建一个新的 `startValue + 1` 表达式来实现的，而不是直接修改原始的 `startValue`。

模式匹配

在 C#中，**模式匹配**（Pattern Matching）基本上就是使用带有“疾速条纹”的 `switch` 语句[14]。不过，F#进一步扩展了这一概念。C#语言已经在几个版本前引入了模式匹配。C# 8.0 引入的 `switch` 表达式为 C#开发者提供了这一概念的原生实现，而微软的团队也在不断对其进行改进。

通过模式匹配，可以根据所检查的对象的类型及其属性来改变执行路径。这种方法可以用来简化复杂的嵌套 `if-else` 语句，如下所示：

```
public int NumberOfDays(int month, bool isLeapYear)
{
    if (month == 2)      // 如果是 2 月，
    {
        if (isLeapYear)  // 而且是闰年，
            return 29;   // 那么 2 月有 29 天，
        else
            return 28;   // 否则只有 28 天。
    }

    // 对于其他月份
    if (month == 1 || month == 3 || month == 5 || month == 7 ||
          month == 8 || month == 10 || month == 12)
        return 31; // 大月有 31 天
    else
        return 30; // 小月有 30 天
}
```

上述代码可以这样简化（要求 C# 8.0 或更高版本）：

```
public int NumberOfDays(int month, bool isLeapYear) =>
    (month, isLeapYear) switch
    {
        { month: 2, isLeapYear: true } => 29,
        { month: 2 } => 28,
        { month: 1 or 3 or 5 or 7 or 8 or 10 or 12 } => 31,
        _ => 30
    };
```

模式匹配是一个极其强大的特性，也是我的最爱之一[15]。

接下来的两章将展示这方面的更多示例，所以如果还想进一步了解模式匹配，可以跳到那些章。此外，即使仍在使用旧版本 C#（8.0 以前），也有办法实现模式匹配，我将在第 4 章的“C#旧版本中的模式匹配”一节分享一些技巧。

---

[14] 译注：“疾速条纹”（Go-Faster Stripes）指的是贴在赛车上的装饰性条纹。作者的意思是 C#的模式匹配在功能上类似于 `switch` 语句，但效率更高，写法更简洁。

[15] 不知为何，朱莉 • 安德鲁斯（电影《音乐之声》的女主角）没有回我的电话，讨论关于她那首著名歌曲的.NET 版本更新事宜。（译注：这里提到的歌是原电影中的“My Favorite Things”。作者说“也是我的最爱之一”，即 one of my favorite things，正好可以和歌曲名称对应。）

无状态

进行面向对象编程时，通常会有一系列状态对象，这些对象代表着某个实际或虚拟的进程。这些状态对象需要定期更新，以便与它们所代表的事物保持同步。例如，以下代码对我最喜爱的电视剧《神秘博士》的相关数据进行了建模。

```
public class DoctorWho // 神秘博士类，表示《神秘博士》电视剧的基本信息
{
    public int NumberOfStories { get; set; }       // 故事总数
    public int CurrentDoctor { get; set; }         // 当前博士的编号
    public string CurrentDoctorActor { get; set; } // 博士当前的演员
    public int SeasonNumber { get; set; }          // 当前季数
}

public class DoctorWhoRepository // 神秘博士数据仓库类，负责管理神秘博士的信息
{
    private DoctorWho State;     // 当前的神秘博士状态

    public DoctorWhoRepository(DoctorWho initialState) // 构造函数
                                                       // 初始化仓库的初始状态
    {
        this.State = initialState;
    }

    public void AddNewSeason(int storiesInSeason) // 添加新的一季，更新故事总数和季数
    {
        this.State.NumberOfStories += storiesInSeason; // 故事总数增加
        this.State.SeasonNumber++; // 季数增加
    }

    public void RegenerateDoctor(string newActorName) // 博士重生
                                                       // 更新博士编号和演员
    {
        this.State.CurrentDoctor++;                    // 递增博士编号
        this.State.CurrentDoctorActor = newActorName; // 更改演员
    }
}
```

如果你打算转向函数式编程，就不要像上面的示例代码那样做了。在函数式编程中，不存在中心状态对象或修改其属性的概念。

听起来是不是有点极端？严格地说，状态在函数式编程中确实存在，但它更多地被视为系统内部自然形成的一种属性。

任何使用过 React-Redux 的开发者都应该已经体会到了函数式编程处理状态的独特方式，这种方式实际上受到了函数式编程语言 Elm 的启发。在 Redux 中，应用程序的状态是一个不可变对象，它不会被直接更新。需要改变状态时，开发者会定义一个函数，这个函数接收当前状态、一个命令和必要的参数，然后基于当前状态返回一个新的状态对象。在 C# 9.0 中引入的记录类型（record types）使这个过程在 C#中变得无比简单，详情请参见第 3 章的“记录类型”一节。现在，先来看看一个简化版示例，它展示了如何重构某个仓库函数（repository function），使其按照函数式编程的方式工作：

```
public DoctorWho RegenerateDoctor(DoctorWho oldState, string newActorName)
{
    return new DoctorWho
    {
        NumberOfStories = oldState.NumberOfStories,
        CurrentDoctor = oldState.CurrentDoctor + 1,
        CurrentDoctorActor = newActorName,

         SeasonNumber = oldState.SeasonNumber
    };
}
```

可预测性是这种方法的一个重要优势。在 C#中，对象以“传引用”的方式传递的。这意味着如果在函数*内部*修改了对象，那么在函数外部，这个对象也会被修改。因此，即使没有显式地给一个对象赋值，但一旦把它作为参数传递给函数后，也不能保证它在函数执行后保持不变。

在函数式编程中，对象的任何修改都必须通过明确的赋值来进行，这样的要求消除了对象是否会被更改的不确定性。

函数式编程的特性就先说到这里，希望你已经对它的基本概念有了清晰的理解。接下来，我会从更广阔的视角来审视函数式编程，并简要探讨一下函数式编程的起源。

制作蛋糕

下面从一个更抽象的视角来探讨一下命令式（imperative）和声明式（declarative）编程范式的区别，看看如何使用这两种方式来制作纸杯蛋糕。[16]

命令式蛋糕

这不是真正的 C#代码，而是一种.NET 主题的伪代码，旨在展示这个问题的命令式解决方案。

```
// 将烤箱温度设置为摄氏 180 度
Oven.SetTemperatureInCentigrade(180);

// 打三个鸡蛋
for (int i = 0; i < 3; i++)
{
    bowl.AddEgg(); // 将鸡蛋加入碗中
    bool isEggBeaten = false; // 将鸡蛋是否打发状态初始化为否

    // 循环打发鸡蛋，直至打发完成
    while (!isEggBeaten)
    {
        Bowl.BeatContents(); // 打发碗中的内容
        isEggBeaten = Bowl.IsStirred(); // 检查是否打发完成
    }
}

// 将打好的鸡蛋液添加到 12 个纸杯蛋糕模具中
```

[16] 例子中包含一些自由发挥的成分。

```
for (int i = 0; i < 12; i++)
{
    OvenTray.Add(paperCase[i]); // 将纸杯蛋糕模具放入烤盘
    OvenTray.AddToCase(bowl.TakeSpoonfullOfContents()); // 将一勺鸡蛋液添加到模具中
}

Oven.Add(OvenTray); // 将烤盘放入烤箱
Thread.Sleep(TimeSpan.FromMinutes(25)); // 等待 25 分钟
Oven.ExtractAll();  // 从烤箱中取出所有物品
```

在我看来，这展示了典型的复杂命令式代码的特点：大量用于状态跟踪的临时变量。此外，这种代码非常注重操作的执行顺序，它就像是对一个毫不智能的机器人发号施令，需要把一切都提前讲得清清楚楚。

声明式蛋糕

下面是一个完全虚构的声明式代码示例，展示了如何以声明式的方法解决相同的问题：

```
// 将烤箱温度设置为摄氏 180 度
Oven.SetTemperatureInCentigrade(180);

// 从蛋盒中取出 3 个鸡蛋，进行一系列操作后得到蛋糕糊
var cakeBatter = EggBox.Take(3)
    .Each(e => Bowl.Add(e)   // 将每个鸡蛋添加到碗中
        .Then(b =>           // 然后对碗进行操作
            b.While(x => !x.IsStirred, x.BeatContents()) // 循环打发碗中的内容，直到完成
        )
    )
    .DivideInto(12);          // 将打发好的鸡蛋液分成 12 份蛋糕糊

// 将每份蛋糕糊添加到纸杯蛋糕模具中，并将模具放入烤盘
cakeBatter.Each(cb =>
    OvenTray.Add(PaperCaseBox.Take(1).Add(cb)) // 从纸盒中取出一个纸杯蛋糕模具，
                                               // 加入蛋糕糊，再放入烤盘。
);
```

如果还不熟悉函数式编程，这种方式可能初看起来有些陌生，甚至有点不合常理。但不用担心，我会循序渐进地阐述它的工作原理、优势，并指导你在 C#中亲自实践它。

不过，值得注意的是，函数式编程没有用于跟踪状态的变量，也没有 `if` 或 `while` 语句。我甚至无法确定具体的操作顺序是怎样的，但这并不重要，因为系统会确保在需要时自动完成所有必要的步骤。

这更像是向一个稍微智能一点的机器人下达指令，它能做一定程度的自主思考。你可能会给出这样的指令：“执行这一操作，直到达到某个特定状态”，而在过程式编程中，这通常需要通过结合使用 `while` 循环和状态跟踪代码来实现。

函数式编程的起源

首先，我想明确一点：尽管有些人可能不这么认为，但函数式编程的历史其实相当悠久（至少按照计算机行业的标准来看）。它不像那些“新奇”的 JavaScript 框架——今

年还在流行，明年就可能被淘汰了。函数式编程的诞生早于所有现代编程语言，甚至（在某种程度上）早于计算机本身。它的年龄比我们所有人都要大，而且等到我们退休之后，它很可能依然存在。

因此，我主要想强调的是，投入时间和精力去学习和理解函数式编程是非常值得的。哪怕将来不再使用 C#，其他大多数语言也都或多或少地支持函数式编程的概念（JavaScript 在这方面的领先程度是其他语言望尘莫及的），所以这些技能在你未来的职业生涯中总会有用武之地。

在继续之前，我要快速声明一下：我不是数学家。我热爱数学，它是我在整个学习生涯中最喜欢的科目之一，但一些理论式的高等数学仍然会让我感到头昏脑胀。话虽如此，我还是会尽我所能地简要介绍一下函数式编程的起源，也就是理论数学的世界。

在函数式编程的历史中，大多数人首先想到的人物可能是哈斯凯尔·布鲁克斯·柯里（Haskell Brooks Curry）（1900－1982），一位美国数学家，至少三种编程语言以及函数式编程中的**“柯里化”**（currying）概念都是用他的名字来命名的（第 8 章将会进一步讨论）[17]。他的主要研究领域是**组合逻辑**（Combinatory Logic），这种数学概念指的是使用 lambda（或箭头）表达式来编写函数，并通过组合它们来构建更加复杂的逻辑。这个概念也是函数式编程的核心基础。然而，柯里并不是第一个从事这项研究的人；他的研究建立在下面这些数学界先驱的论文和著作的基础上：

- 阿隆佐·丘奇（Alonzo Church）（1903－1955，美国人）

  创造了“lambda 表达式”这一术语，这一概念至今仍被 C#等多种编程语言广泛应用。

- 摩西·舍恩芬克尔（Moses Schönfinkel）（1888－1942，俄罗斯人）

  舍恩芬克尔撰写的关于组合逻辑的论文为柯里的研究奠定了基础。

- 弗里德里希·弗雷格（Friedrich Frege）（1848－1925，德国人）

  弗雷格可能是最早提出我们现在所熟知的“柯里化”概念的人。尽管把发现归功于正确的人非常重要，但将这一概念称为“Freging（弗雷格化）”似乎不那么顺耳。[18]

最早的函数式编程语言包括：

- 资讯处理语言（Information Processing Language，IPL）
  由艾伦·纽厄尔（Allen Newell）（1927－1992，美国人）、克利夫·肖（Cliff Shaw）（1922－1991，美国人）和赫伯特·西蒙（Herbert Simon）（1916－2001，美国人）于 1956 年开发。
- 表处理语言（LISt Processor，LISP）

[17] 译注：简单地说，“柯里化”是一种函数式编程技术，它将接受多个参数的一个函数转换为一系列接受单个参数的函数。每个函数都会返回一个新的函数，直到接收了所有参数并返回最终结果。

[18] 例如，如果说“这段 Freging（译注：与 freaking 的发音很相似）代码跑不起来！”（I can’t get this Freging code to work!），那么很容易产生歧义。

由约翰·麦卡锡（John McCarthy）（1927–2011，美国人）于 1958 年开发。我听说 LISP 至今仍然拥有一批忠实粉丝，并且一些企业仍在生产环境中使用它。不过，我未曾亲眼看到过这样的例子。

有趣的是，所有这些语言都不算是“纯”函数式语言。就像 C#、Java 以及其他许多语言那样，它们都采用了一种混合式方法，这和 Haskell 和 Elm 等现代“纯”函数式语言还是有所区别的。

这里不打算过多地谈论函数式编程的历史（尽管这段历史确实很吸引人），但从前文中，应该不难看出函数式编程拥有悠久而显赫的历史。

还有别人在使用函数式编程吗？

正如我之前提到的那样，函数式编程已经存在一段时间了，不只是.NET 开发者对此感兴趣。实际上，其他许多编程语言在.NET 之前就已经支持函数式编程范式很长时间了。

这里的“支持”是指编程语言提供了以函数式范式实现代码的能力。这大致分为两种类型：

- 纯函数式编程语言
  在这种语言中，开发者只能编写函数式代码。所有变量都是不可变的，并且内置了对柯里化和高阶函数的支持。它们可能在一定程度上支持面向对象的一些特性，但这通常不是开发团队的主要关注点。
- 混合或多范式编程语言
  这两个术语完全可以互换使用。这种编程语言提供了一系列特性，使得开发者能使用两种或更多编程范式编写代码——而且往往是同时支持多种范式。这些语言通常支持函数式和面向对象编程。它们可能不会为所有支持的范式都提供完美的实现。面向对象编程通常会得到最全面的支持，但函数式编程范式往往并非如此。

纯函数式语言

目前存在十几种纯函数式语言。这里简单介绍一下当今最受欢迎的几种。

- Haskell
  Haskell 在银行业中得到了广泛的应用，经常被推荐给那些想要深入了解函数式编程的人。虽然 Haskell 可能确实是学习函数式编程的好起点，但老实说，我并没有时间或精力去学习一门我在日常工作中用不到的编程语言。

如果想先成为函数式编程范式的专家，再学习 C#的函数式编程，那么研读 Haskell 的相关资料无疑是个好主意。米兰·利波瓦察（Miran Lipovača）所著的

《HASKELL 趣学指南》（Learn You a Haskell for Great Good!）[19]是一本经常被推荐的好书。虽然我自己没读过这本书，但我的一些朋友读过，并给出了高度评价。

- Elm
  Elm 最近似乎越来越受欢迎了，一个重要原因是它在 UI 更新方面的系统被 React 等多个其他项目采用了。
- Elixir
  Elixir 是一种基于 Erlang 虚拟机的通用编程语言。它在工业界很受欢迎，并且每年都会召开专门的会议。
- PureScript
  PureScript 是一种最终编译成 JavaScript 代码的编程语言，因此它既可以用来创建函数式前端代码，也可以用来在同构开发环境中编写服务器端代码和桌面应用程序。同构（isometric）开发环境指的是那些允许在客户端和服务器端使用相同编程语言的环境，比如 Node.js。

首先学习纯函数式语言是否值得？

目前，面向对象编程仍然在软件开发领域占据主导地位，函数式编程则被视为一种锦上添花的范式。虽然这种状况在未来有可能会发生改变，但至少现在的环境就是如此。

有些人认为，在从面向对象编程转向函数式编程时，最好先学习纯函数式编程，然后再将这些知识应用到 C#这样的“混合”语言中。如果真的想这么做，也完全没有问题，祝你学得开心。我相信这也会是一段有价值的旅程。

对我来说，这种观点让我想起了英国那些坚持让学生学习拉丁语的老师们，因为拉丁语是欧洲许多语言的根源，掌握了拉丁语就能更容易学习法语、意大利语、西班牙语等语言。

但我对这个观点持保留意见。[20]纯函数式编程语言没有拉丁语那么难学，尽管它们确实与面向对象开发大相径庭。实际上，与面向对象编程相比，函数式编程中需要学习的概念比较少。但是习惯了面向对象编程的人可能会比较难以适应函数式编程。

尽管如此，拉丁语和纯函数式编程语言在某种意义上是相似的，它们都代表了一种更纯粹、更原始的语言。除了少数专业领域，它们的价值都比较有限。

除非你对法律、古典文学、医学、历史等领域感兴趣，否则学习拉丁语几乎没有太大用处。相比之下，学习法语或意大利语这样的现代语言更有价值。这些语言学习起来容易得多。还可以在旅行时使用这些语言，与友好的当地人交流。而且，比利时还有一些很棒的法语漫画。真的很有意思，去看看吧，我等你。

---

[19] 《HASKELL 趣学指南》（Learn You a Haskell for Great Good!）一书可以在网上免费阅读（https://oreil.ly/jE4nv），告诉他们是我推荐你来的。

[20] 不过，我确实在学习拉丁语。Insipiens sum. Huiusmodi res est ioci facio.（我是笨蛋；开玩笑的。）

同样的，很少有公司会在生产环境中使用纯函数式语言。也就是说，在花大量时间彻底改变了自己的工作方式之后，你最后学到的可能是一种只能在个人业余项目中使用的语言。并最终学习一种你可能永远不会在自己的业余项目之外使用的语言。根据我多年的从业经验，从来没有一家公司在生产环境中使用比 C#更前沿的语言。

C#的一大优点是，它既支持面向对象编程也支持函数式编程。因此，开发者可以根据需求自由切换。可以随心所欲地使用这两种范式的任何特性，而不会遭受任何不良影响。这些范式可以在同一个代码库中和谐共存。所以，在这个环境中，可以完全按照自己的节奏，轻松地从纯面向对象过渡到函数式编程（反之亦然）。但是，在纯函数式编程语言中，像这样混合使用多种范式是不可能的。所以，即使 C#没有提供对函数式特性的全面支持，它也有一定的优势。

F#怎么样？是否有必要学习？

“那么 F#怎么样？”可能是我最常听到的问题。它不是一种纯函数式语言，但与 C#相比，它更接近于纯函数式范式的实现。F#内置了许多函数式特性，编码简单，并且能在生产环境中提供高性能的应用程序——那么，为什么不选择它呢？

在回答这个问题之前，我总是会先留意一下房间里的每个出口。F#有一个非常热情的用户群，他们很可能比我聪明许多[21]。但是……

并不是说 F#不容易学习。据我所见，它学起来比较容易。特别是对完全没有编程经验的新手来说，学习 F#可能比学习 C#还要容易。

并不是说 F#不能带来商业利益——我真心认为它好处多多。

也并不是说 F#做不到其他语言能做的事情。它肯定能做到，我也听过一些关于如何构建全栈 F#应用的精彩演讲。

是否学习 F#是一个职业上的选择。在我去过的所有国家里，都不难找到 C#开发者。如果把大型开发者会议的所有与会者的名字写在纸条上，然后把所有纸条都放到帽子里并随机抽取一张，很可能抽到的是一名能在工作中使用 C#的人。如果一个团队决定建立 C#代码库，他们能轻松地找到足够多的工程师来维护代码并保持业务的平稳运行。

相比之下，了解 F#的开发者相对较少。我认识的 F#开发者并不多。如果在代码库中采用了 F#，那么可能会对 F#开发者产生依赖，将来可能面临某些代码难以维护的风险，因为懂得如何维护 F#代码的人并不多。

需要注意的是，这种风险并不像引入一种全新技术（比如 Node.js）的风险那样高。F#仍然属于.NET 语言，并且编译成与 C#相同的中间语言（Intermediate Language，IL）。甚至可以在同一个解决方案中轻松地在 C#项目中引用 F#项目中的代码。然而，大多数.NET 开发者仍然会觉得 F#的语法相当陌生。

---

[21] 特别是 F#大师伊恩·拉塞尔（Ian Russell），他为本书中有关 F#的内容提供了很多帮助。谢谢你，伊恩！

我衷心希望随着时间的推移，这种情况会有所改变。我对 F#有很好的印象，并且很想做更多相关的工作。如果公司决定采用 F#，我绝对会是第一个表示欢迎的人！

但就目前而言，这种情况不太可能发生。至于未来会如何，谁也说不准。或许在本书未来的某个版本中，我需要进行大量修改，以迎合学习 F#的热潮，但目前我还没有看到这种征兆。

我的建议是先试着阅读这本书。如果你对所学到的内容感到兴趣，那么 F#可能是你的函数式编程之旅的下一步。

多范式语言

可以说，除了纯函数式语言之外，所有语言在某种程度上都是混合式的。换言之，至少函数式编程的某些特性是可以实现的。尽管如此，我只会简要介绍其中的几种语言。在这些语言中，函数式编程得到了完整或大部分的实现，并且很受开发团队的重视。

- JavaScript
  JavaScript 可以说是编程语言的“狂野西部”，用它可以做几乎任何事情。而且它在函数式编程方面的表现非常出色——有人甚至认为它在这方面做得比在面向对象编程方面更好。如果想了解如何正确并且有效地使用 JavaScript 进行函数式编程，推荐阅读道格拉斯·克罗克福特（Douglas Crockford）所著的《JavaScript 语言精粹》（JavaScript: The Good Parts，O'Reilly 出版社）和他的一些在线讲座（例如，“JavaScript: The Good Parts”：https://oreil.ly/rIDSN）。
- Python
  Python 在过去几年中迅速地成为了开源社区最喜爱的编程语言之一。令人惊讶的是，Python 从 80 年代末就已经存在了！Python 支持高阶函数，并提供了一些库，比如 `itertools` 和 `functools`，使得开发者可以实现更多的函数式编程特性。
- Java
  Java 平台对函数式特性的支持与.NET 相同。此外，Java 的一些衍生语言（例如，Scala、Clojure 和 Kotlin）提供了更多函数式编程特性。
- F#
  如前所述，F#是.NET 中更偏向纯函数式风格的语言。C#和 F#库之间存在一定的互操作性（Interoperability），使得项目能够结合利用这两者的最佳特性。
- C#
  微软几乎是从一开始就在慢慢增加对函数式编程的支持。2005 年在 C# 2.0 中引入的委托协变和匿名方法可以被视为支持函数式编程范式的第一项特性。不过，直到 2006 年的 C# 3.0，它对函数式编程的支持才真正得到了加强。C# 3.0 引入了我认为是有史以来最变革性的特性之一：LINQ。

LINQ（Language Integrated Query，语言集成查询）深植于函数式编程范式中，是开发者在 C#中开始编写函数式风格代码的最佳工具之一（第 2 章将进一步讨论）。实际上，C#团队致力于使每个新发布的 C#版本都比前一个版本更好地支持函数式编程。这

一策略背后有许多推动因素，其中包括 F#对.NET 运行时开发团队提出的新函数式特性请求，而 C#也从中受益。

函数式编程的好处

我希望你之所以选择这本书，是因为已经对函数式编程产生了兴趣，希望立刻开始学习。如果你和团队正在考虑是否要在工作中采用函数式编程，那么这一节或许能提供一些有价值的见解。

简洁

虽然这并不是函数式编程的固有特性，但在函数式编程众多益处中，最吸引我的就是它的简洁和优雅，这是面向对象或命令式编程无法企及的。

其他编程风格更加关注执行某项操作的底层细节，有时需要花大量时间研读代码，才能理解代码背后的真实意图。函数式编程更倾向于描述需要达成的目标。至于哪些变量需要更新、如何更新以及更新的时间点等具体实现细节，并不是我们关注的重点。

在与一些开发者讨论这个问题时，我发现他们对于无法深度参与数据处理的底层操作感到不满，但我个人其实乐于让执行环境来处理这些细节。这样，我就少了一件需要操心的事情。

这听起来可能是件小事，但相比命令式的替代方案，我真心喜欢函数式代码的简洁性。开发工作充满了挑战[22]，经常需要接手复杂的代码库并迅速理解它们。开发者在搞清楚某个函数的实际作用上花费的时间越多，企业损失就越大，因为企业实际上是在为这些理解工作而支付薪水，而不是为编写新代码。函数式代码通常以接近自然语言的方式表达它要完成的任务，这不仅提高了代码的可读性，也简化了调试过程，为企业节省了宝贵的时间和资源。

可测试性

函数式编程的可测试性是很多人钟爱的特性之一。函数式编程的可测试性确实相当高。如果代码库的可测试性不接近 100%，很可能意味着没有正确地实现这一编程范式。

测试驱动开发（Test-Driven Development，TDD）和行为驱动开发（Behavior-Driven Development，BDD）是软件开发中极为重要的专业实践。这些实践要求首先为生产代码编写自动单元测试，然后编写能通过测试的代码。通过这种方式创建的代码往往设计优良且更加健壮。函数式编程对这两种实践提供了很好的支持，而它们反过来又提高了代码的质量并减少了生产环境中的 bug。

[22] 至少，我们对领导是这么说的。

健壮性

除了可测试性以外，另一个让代码库更加健壮的原因是函数式编程的结构，这些结构会从源头上预防错误的产生。

另外，这些结构还能防止后续出现任何意料之外的行为，使得准确报告问题变得更加容易了。函数式编程中没有“null”的概念，这不仅消除了大量可能的错误，还减少了需要编写的自动化测试的数量。

可预测性

函数式编程的代码从代码块的开头一直执行到末尾——始终严格按照顺序执行。这是过程式编程所做不到的，因为它包含了循环和条件分支。函数式编程只有易于跟踪的单一代码流。

正确实现的函数式编程甚至不包含任何 `try/catch` 块，而这些结构往往是导致代码执行顺序无法预测的罪魁祸首。如果 `try` 块作用域广泛且与 `catch` 块的联系不够紧密，那么它就像是蒙着眼睛把一块石头抛向空中，谁知道它会落在哪里，会被什么人接住？在这样的执行流中断时，谁又能预测可能发生的意外行为呢？

我在职业生涯中观察到的许多生产环境中的意外行为都是由设计不当的 `try/catch` 块引起的，而这种问题在函数式编程中根本不存在。虽然函数式代码中也可能会出现错误处理不当的情况，但函数式编程天然倾向于避免这种情况。

更好地支持并发

在过去几年中，软件开发领域有两项技术取得了重大进展。

容器化（Containerization）

容器化是由 Docker、Kubernetes 等产品提供支持的一种技术，它的中心思想是，应用程序不再运行在传统服务器上——无论是虚拟服务器还是物理服务器——而是运行在类似于微型虚拟机的环境中。这种环境是在部署时通过脚本动态生成的。这种方式在技术实现上不同于传统虚拟机（因为不涉及到硬件仿真），但对用户来说，它们的结果大致相同。它解决了开发者们经常遇到的“在我的机器上明明可以”的问题。许多公司的软件基础设施都会采用容器化技术来部署同一应用的多个实例。这些容器处理相同的输入源，比如队列、用户请求等。托管这些容器的环境可以根据需求调整容器的数量。

无服务器（Serverless）

对于.NET 开发者来说，无服务器的概念可能并不陌生，他们可能已经通过 Azure Functions 或 Amazon Web Services（AWS） Lambda 接触过这种架构了。在这种架构下，代码不部署在传统的 Web 服务器上，比如 Internet Information Services（IIS），而是作为独立存在的单个函数部署在云托管环境中。这种方法可以实现与容器相同的自

动伸缩功能，也可以对细节进行优化，如此一来，就可以把更多资金花费在更为关键的功能上，对于那些生成输出比较耗时的功能，则可以减少资金投入。

这两项技术都广泛使用了并发处理（即同一功能的多个实例同时对同一个输入源进行处理）。 这与.NET 的异步（`async`）特性相似，但应用范围更广。

异步操作常常在处理共享资源时遇到问题，无论是内存状态还是实际共享的物理资源或基于软件的外部资源。函数式编程不使用状态，因此在线程、容器或无服务器函数之间没有状态可以共享。

若正确实现，函数式编程范式有助于实现这些广受欢迎的技术特性，还不会在生产环境中引起任何意料之外的行为。

降低代码噪音

音频处理领域有一个名为“信噪比”（Signal-to-Noise Ratio）的概念，指的是根据信号（你想要听到的声音）的音量与噪音（比如背景中的嘶嘶声、爆裂声或隆隆声）的比例来衡量录音的清晰度。

在编码中，信号代表着代码块的业务逻辑，也就是代码真正想要完成的目标。换言之，信号是代码的“目的”。

噪音则是指为了实现目标而需要编写的样板代码（Boilerplate Code），包括 `for` 循环的定义、`if` 语句等等。

与过程式编程的代码相比，函数式编程非常简洁，样板代码要少得多，因此具有更好的信噪比。这不仅对开发者有好处。健壮、易于维护的代码库也意味着企业在维护和扩展上的成本更低。

函数式编程的最佳应用场景

函数式编程能够实现其他编程范式所能实现的功能。在某些领域，它是最强大、最有益处的，而在其他一些领域则可能需要做出妥协，比如融入一些面向对象的特性，或略微放宽函数式范式的规则。在.NET 中，由于基础类库和扩展库大多遵循面向对象的范式，因此往往不得不做出一些妥协，但如果是在纯函数式语言中，则不需要做出任何妥协。

函数式编程特别适合需要高度可预测性的场景，例如数据处理模块或者将数据从一种形式转换为另一种形式的函数。另一个例子是业务逻辑类，这些类处理用户或数据库的数据，并将其传递到其他位置进行渲染。诸如此类。

函数式编程的无状态性质使它成为了并发系统的强大助力，这包括需要处理大量异步任务的代码库，或是多个处理器需要同时对同一输入队列进行监听的场景。在没有共享状态的情况下，几乎不可能出现资源争夺的问题。出于以上原因，如果你的团队正在考虑使用 Azure Functions 等无服务器应用程序，那么函数式编程将会非常有帮助。

相比于面向对象编程范式，函数式编程有助于减少错误并增强代码的健壮性，因此，对于业务关键型的系统而言，采用函数式编程是一个明智的选择。如果必须确保系统在面对未处理的异常或不正确的输入时也能稳定运行，那么函数式编程可能是最佳解决方案。

更适合使用其他范式的场景

当然，也不一定非要使用其他编程范式。函数式编程能做到任何事情，但在 C#环境中，探索其他编程范式在某些情况下可能是值得的。此外，C#是一种混合式语言，所以许多范式可以和谐共存，完全取决于开发者的需求。当然了，我也有自己的偏好！

一个需要考虑使用其他范式的情况是与外部实体的交互：例如，输入/输出（I/O）、用户输入、第三方应用程序和 Web API 等等。这些操作无法以纯函数（即没有副作用的函数）的形式实现，因此需要进行一定的妥协。从 NuGet 包导入的第三方模块也是如此。甚至有一些比较旧的 Microsoft 库也不支持函数式编程。在.NET Core 中也不例外。想要了解具体例子的话，可以看看.NET 中的 `SmtpClient` 或 `MailMessage` 类。

在 C#世界中，如果性能是项目中唯一的、凌驾于一切之上的关注点——甚至比可读性和模块性还要重要——那么遵循函数式编程范式可能不是最好的选择。函数式 C#代码的性能本身并不存在固有缺陷，但它也不一定是性能最高的解决方案。

不过我认为，函数式编程带来的好处远远大于它可能带来的轻微性能损失。今日，我们经常能轻松地为应用程序增加一些额外的硬件资源（虚拟或实体，视情况而定），这种做法的成本很可能远低于开发、测试、调试以及维护一个用命令式风格编写的代码库所需的成本。但是，如果要开发的是部署到移动设备上的代码，那么情况就不一样了。因为移动设备的内存资源有限且无法扩展，所以性能尤为关键。

能将函数式编程应用到何种程度？

令人遗憾的是，在 C#中完全实现函数式编程范式是不可能的。原因有很多，比如语言向后兼容的需求以及作为强类型语言的固有限制。

本书的目的不是展示如何完全实现函数式编程，而是展示在 C#中实现函数式编程的可能性和局限性之间的界限。我还将介绍一些切实可行的做法，特别是对那些负责维护生产环境代码库的开发者来说。本质上，这是一本实用、务实的函数式编码风格指南。

~~单子~~实际上，先不用担心这个

单子（Monad）通常被视为函数式编程的一大难题。如果查看它在维基百科上的定义，你会看到一堆奇怪的符号，包括 F、G、箭头……比图书馆书架下的符号还要多。

即使到现在，我仍然觉得这些官方定义宛如天书。说到底，我是工程师，而不是数学家。

道格拉斯·克罗克福特（Douglas Crockford）曾说过，单子的诅咒是，一旦理解了它，就失去了解释它的能力。所以我将不会解释它。不过，在这本书中，单子可能会在一些意想不到的时刻出现。

别担心；一切都会好起来的。我们会一起解决所有问题的，相信我。

小结

在《C#函数式编程》第一幕中，我们的主角——也就是你——勇敢地探索了函数式编程（FP）究竟是什么，以及它为什么值得学习。你初步了解了函数式编程范式的重要特性：

- 不变性
- 高阶函数
- 首选表达式而非语句
- 引用透明性
- 递归
- 模式匹配
- 无状态

此外，你还探索了函数式编程的应用场景、采用纯函数式编程的优势以及使用函数式范式编写应用程序的众多好处。

在下一个激动人心的章节中，你将学到一些可以立刻在 C#中应用的函数式编程技巧。不需要任何新的第三方库或 Microsoft Visual Studio 扩展——只需要原汁原味的 C#，再加上一点点创造力。

欲知后事如何，且听下回分解。同一.NET 时间。同一.NET 频道。[23]

[23] 严格来说，是“同一本书”。

第 I 部分 我们已经在做的事

信不信由你，如果你做过一段时间的.NET 编程，那么很有可能已经或多或少地在进行函数式编程了。

本书第 I 部分旨在向你展示，你的日常编程工作已经多大程度地采用了函数式编程，或是可以轻松地转换为函数式编程。本部分不会涉及除了微软提供的库之外的任何库，也不会涉及复杂的理论。

可以将这部分视为航向神秘、幽暗、迷雾重重的函数式编程之海的起点。此刻，你还站在陆地上，周围的一切都显得熟悉而陌生。

第 II 部分将更深入地探讨函数式编程的概念。如果觉得第 I 部分太过简单，那么可以随时跳到第 II 部分。

第 2 章 我们目前能做些什么

对于一部分读者来说，本章讨论的代码和概念可能比较基础，但请耐心一些。我不想过早引入太多内容。经验丰富的开发者或许可以直接跳到第 3 章，其中讨论了 C#在函数式编程方面的最新进展；也可以直接跳到第 4 章，其中将展示如何利用你熟悉的特性，以新颖的方法来进行函数式编程。

本章将介绍当今几乎每个 C#生产代码库中都可以实现的函数式编程特性。这里假设至少使用的是.NET Framework 3.5 版本，只需进行一些简单的调整，本章所有代码示例都可以在该环境下运行。如果使用的是版本更新的.NET，但还不太熟悉函数式编程，也建议阅读本章，它将为你采用函数式编程范式提供一个良好的起点。

如果已经对函数式代码有所了解，只是想看看.NET 最新版本都提供了哪些功能，可以直接跳到下一章。

开始

函数式编程真的很简单！与许多人的观念相反，学习函数式编程比学习面向对象编程（OOP）要容易，因为它涉及的新概念更少。

如果对此表示怀疑的话，去试着向不懂技术的家人解释什么是多态性吧。习惯了 OOP 的人可能已经忘记了最开始学习这些概念时有多么辛苦。

函数式编程其实不难理解，只是与平时习惯的编程方式不同。我遇见过很多刚从高校毕业、对函数式编程充满热情的学生。因此，如果他们能做到，相信你也可以。

一个普遍存在的误解是认为需要先学习大量知识才能开始函数式编程。但我想说的是，如果有过一段时间的 C#编程经验，那么很可能已经写过函数式代码了。下一节将具体说明这一点。

编写第一段函数式代码

开始编写函数式代码之前，先来看看非函数式编程的代码示例。你可能在 C#编程生涯的早期就已经学习过这种风格了。

非函数式的电影查询示例

在这个简单的例子中，我们要从一个假想的数据源获取包含所有电影的列表，并在此基础上创建一个新列表，但新列表中只包含动作类（Action）电影：[24]

```
public IEnumerable<Film> GetFilmsByGenre(string genre)
{
    var allFilms = GetAllFilms();
    var chosenFilms = new List<Film>();
```

[24] 顺带一提，我更喜欢科幻（SF 或 sci-fi）电影。

```
    foreach (var f in allFilms)
    {
        if (f.Genre == genre)
        {
            chosenFilms.Add((f));
        }
    }
    return chosenFilms;
}
var actionFilms = GetFilmsByGenre("Action");
```

这段代码存在什么问题？最明显的问题是，它不够优雅。对于一个相对简单的任务来说，这里编写的代码太多了。

还实例化了一个新对象，这个对象将在函数执行期间一直保留在作用域中。如果这就是整个函数的全部内容，那么还不算什么大问题。但如果这只是一个长函数中的小片段呢？那样的话，`allFilms` 和 `actionFilms` 变量会一直存在于作用域中并占用内存，即使它们没有被使用。

在复制数据时，是否需要在复制的对象中保留所有数据的副本取决于这个对象是类、结构还是其他类型。但只要这两个对象都在作用域内，就会在内存中保留一组重复的引用，不必要地占用额外的内存。

此外，这种方法还限定了操作顺序。我们指定了何时循环，何时添加——每一步应该在哪里以及何时执行。如果数据转换过程中需要执行任何中间步骤，那么还需要指定这些步骤，并将它们的结果存储在可能会长时间存在于内存中的变量里。

可以使用 `yield return` 来解决一些问题，如下所示：

```
public IEnumerable<Film> GetFilmsByGenre(string genre)
{
    var allFilms = GetAllFilms();
    foreach (var f in allFilms)
    {
        if (f.Genre == genre)
        {
            yield return f;
        }
    }
}

var actionFilms = GetFilmsByGenre("Action");
```

但这只是减少了几行代码而已，并没有从根本上解决问题。

如果设定的操作顺序不是最好的，该怎么办？如果后续代码导致我们实际上不需要返回 `actionFilms` 的内容，那之前的工作岂不是白费了？

这就是过程式代码的根本问题：一切都必须详细指示。函数式编程的一个主要目标就是摆脱这些麻烦事。不要具体到细枝末节上，放松一点，拥抱声明式代码。

函数式的电影查询示例

那么，如果按照函数式风格重写之前的代码示例，它会是什么样子呢？我希望你已经猜到了重写的方式：

```
public IEnumerable<Film> GetFilmsByGenre(
    IEnumerable<Film> source,
    string genre) =>
    source.Where(x => x.Genre == genre);

var allFilms = GetAllFilms();
var actionFilms = GetFilmsByGenre(allFilms, "Action");
```

这时你可能会说，“这不就是 LINQ 吗？”没错，这就是 LINQ。告诉你一个小秘密：LINQ 遵循了函数式编程范式。

考虑到可能还有人不清楚 LINQ 的神奇之处，这里简单介绍一下，LINQ 是自 C#早期版本以来就存在的一个库。它提供了一套丰富的函数，用于筛选、更改和扩展数据集合。`Select()`、`Where()`和 `All()`等函数都来自于 LINQ，并且在全世界范围内被广泛使用。

现在，回想一下函数式编程的特性，看看 LINQ 实现了其中的哪些：

- 高阶函数
  传递给 LINQ 函数的 lambda 表达式就是作为参数变量传递的函数。
- 不变性
  LINQ 不改变源数组；它基于源数组返回一个新的枚举。
- 首选表达式而非语句
  不使用 `foreach` 和 `if` 的使用。
- 引用透明性
  尽管没有强制要求，但前面 lambda 表达式确实符合引用透明性（即没有副作用）。虽然完全可以在 lambda 表达式中引用外部的字符串变量，但通过将源数据作为参数传入，我们不仅避免了这种情况，还简化了测试过程，因为这样就不必创建和设置代表外部数据源连接的模拟对象了[25]。函数需要的一切都由它自己的参数提供。

据我所知，迭代完全可以改为用递归来实现，但我不知道 `Where()`函数的源代码是什么样的。我决定继续认为它是通过递归实现的，除非有证据证明它不是。

[25] 译注：“代表外部数据源连接的模拟对象”指的是在软件开发过程中，特别是在单元测试时，不直接连接到实际的外部数据源（如数据库、文件系统或网络服务），而是使用一种技术来创建一个行为类似但可以控制的虚拟数据源。这种做法常见于测试环境，目的是为了验证代码的功能性和健壮性，同时避免在测试过程中对真实数据源产生影响或依赖。在单元测试中，模拟（Mocking）和存根（Stubbing）是常用的技术，它们使开发者能够模拟外部数据源的接口和行为。

从很多方面来说，这一行简短的代码是函数式编程的完美示例。我们通过传递函数来操作数据集合，基于原始集合创建一个新集合。采用函数式编程范式，最终得到的代码更加简洁、更易于阅读，因此也更容易维护。

以结果为导向的编程

函数式代码的一个关键特点是，它更重视最终结果，而不是获得这个结果的具体过程。如果完全采用过程式的方法来构建一个复杂对象，我们通常会在代码的开始处实例化一个空对象，然后逐步添加每个属性的值：

```
var sourceData = GetSourceData();
var obj = new ComplexCustomObject();

obj.PropertyA = sourceData.Something + sourceData.SomethingElse;
obj.PropertyB = sourceData.Ping * sourceData.Pong;

if(sourceData.AlternateTuesday)
{
    obj.PropertyC = sourceData.CaptainKirk;
    obj.PropertyD = sourceData.MrSpock;
}
else
{
    obj.PropertyC = sourceData.CaptainPicard;
    obj.PropertyD = sourceData.NumberOne;
}

return obj;
```

这种方法的问题在于，它非常容易被滥用。虽然这个虚构的代码块短小且易于维护，但实际的生产代码往往非常长，涉及多个需要预处理、合并以及再处理的数据源。这就可能导致大量的嵌套 `if` 语句，使代码的结构变得像族谱一样错综复杂。

每增加一层嵌套的 `if` 语句，代码的复杂度就会翻一番，特别是在代码库中到处都是 `return` 语句的时候。随着代码库变得越来越复杂，很容易在不经意间产生 `null` 值或遇到其他意料之外的情况。函数式编程不鼓励这样的结构，从而避免了过度的复杂性和意外的副作用。

之前的代码示例中，`PropertyC` 和 `PropertyD` 被定义了两次。虽然这段代码还算易于处理，但我曾遇到过同一个属性在多个类和子类中定义了五六次的情况[26]。不知道你是否有过处理这种代码的经历，反正我有过很多次。

随着时间的推移，这类庞大、难以管理的代码库会变得越来越难维护。每次添加新功能，开发人员的工作效率都会下降，业务领导者也可能会感到沮丧，因为他们无法理解为何如此“简单”的更新需要花费那么长的时间。

理想情况下，函数式代码应该被编写成小而简洁的代码块，完全聚焦于最终结果。它所偏好的表达式模仿了数学问题的求解步骤，因此在写函数式代码时，实际上是在写

[26] 有一次，一些定义甚至还分布在代码库之外的数据库存储过程中。

一系列小公式，每个公式都精确定义了一个结果及其所有相关变量。应该能轻而易举地在代码库中定位一个值的来源。

这里有一个例子：

```
function ComplexCustomObject MakeObject(SourceData source) =>
new ComplexCustomObject
{
    PropertyA = source.Something + source.SomethingElse,
    PropertyB = source.Ping * source.Pong,
    PropertyC = source.AlternateTuesday ? source.CaptainKirk : source.CaptainPicard,
    PropertyD = source.AlternateTuesday ? source.MrSpock : source.NumberOne
};
```

虽然这里重复使用了 `AlternateTuesday` 标志（flag），但是现在所有影响返回值的变量都集中在一处定义了。这简化了未来的代码维护工作。

如果一个属性非常复杂，以至于需要多行代码或一系列占据大量空间的 LINQ 操作，我会创建一个专门的函数来封装这些复杂的逻辑。不过，基于结果的返回仍然是一切的核心。

可枚举对象

有时，我觉得**可枚举对象**（Enumerable）是 C#中最容易被低估和误解的特性之一。可枚举对象是数据集合最抽象的表示形式——它本身不包含任何数据，只存储了一个关于如何获取数据的描述。在遍历完所有元素之前，可枚举对象连有多少元素都不知道——它仅知道当前元素的位置以及如何迭代到下一个元素。

这个过程被称为**惰性求值**（Lazy Evaluation）或**延迟执行**（Deferred Execution）。在开发中，懒惰是一件好事。不要让任何人否定这一点。[27]

事实上，还可以为可枚举对象编写自定义行为。底层有一个称作**枚举器**（Enumerator）的对象。通过与之交互，可以获取当前项或迭代到下一项。注意，不能使用可枚举对象或枚举器来确定列表的长度，且迭代只能朝着单一方向进行。

请看以下代码示例。首先，一组简单的日志记录函数会将信息添加到字符串列表中。

```
private IList<string> c = new List<string>();

public int DoSomethingOne(int x)
{
    c.Add(DateTime.Now + " - DoSomethingOne (" + x + ")");
    return x;
}

public int DoSomethingTwo(int x)
{
    c.Add(DateTime.Now + " - DoSomethingTwo (" + x + ")");
    return x;
}
```

[27] 除了你的老板，毕竟他们会给你发工资，如果他们人很好，或许每年还会送你一张生日贺卡。

```
public int DoSomethingThree(int x)
{
    c.Add(DateTime.Now + " - DoSomethingThree (" + x + ")");
    return x;
}
```

然后，一段代码轮流用不同的数据调用每个 DoSomething()函数：

```
var input = new[]
{
    75,
    22,
    36
};
var output = input.Select(x => DoSomethingOne(x))
                  .Select(x => DoSomethingTwo(x))
                  .Select(x => DoSomethingThree(x))
                  .ToArray();
```

你认为操作的顺序会是怎样的？你可能以为，在运行时，系统会先取原始输入数组，对其中每个元素执行 DoSomethingOne()函数来生成一个新的数组，再对这些元素执行 DoSomethingTwo()函数，以此类推。

如果检查那个字符串列表的内容，可以看到下面这样的情况：

```
18/08/1982 11:24:00 - DoSomethingOne(75)
18/08/1982 11:24:01 - DoSomethingTwo(75)
18/08/1982 11:24:02 - DoSomethingThree(75)
18/08/1982 11:24:03 - DoSomethingOne(22)
18/08/1982 11:24:04 - DoSomethingTwo(22)
18/08/1982 11:24:05 - DoSomethingThree(22)
18/08/1982 11:24:06 - DoSomethingOne(36)
18/08/1982 11:24:07 - DoSomethingTwo(36)
18/08/1982 11:24:08 - DoSomethingThree(36)
```

这几乎和通过 for 或 foreach 循环得到的结果完全相同，但我们实际已经把执行顺序的控制权交给了运行时环境。我们不必操心临时持有的变量的细节，也不用操心数据应当如何以及何时被处理。相反，只需描述想执行的操作，并等着得到一个最终结果。

最终生成的字符串列表可能不会和上述代码完全相同；这具体取决于与可枚举对象交互的代码（通过 LINQ 或 foreach）是怎么写的。但有一点始终不变：可枚举对象实际只会在数据被请求的那一刻才生成数据。它们在哪里定义的并不重要；重要的是它们在什么时候*使用*。

通过选择使用可枚举对象而不是固定大小的数组，我们设法实现了一些编写声明式代码所需要的行为。

令人惊讶的是，如果像下面这样重写代码，之前提到的日志文件还是会保持不变。

```
var input = new[]
{
    1,
    2,
```

```
        3
    };
    var temp1 = input.Select(x => DoSomethingOne(x));
    var temp2 = input.Select(x => DoSomethingTwo(x));
    var finalAnswer = input.Select(x => DoSomethingThree(x));
```

temp1、temp2 和 finalAnswer 都是可枚举对象，它们在进行迭代之前都不会包含任何数据。

请试着做个实验，参照以上示例编写一段代码，但不要原样照搬，也许可以尝试一些更简单的操作，比如通过一系列的 select 以某种方式修改一个整数值。在 Visual Studio 中设置一个断点，继续执行程序直到最终结果计算完毕，然后将鼠标悬停在 finalAnswer 上。你可能会发现，尽管代码执行到了这里，它仍然无法展示任何数据。这是因为可枚举对象还没有执行任何操作。

如果像下面这样做，情况就会有所改变：

```
var input = new[]
{
    1,
    2,
    3
};
var temp1 = input.Select(x => DoSomethingOne(x)).ToArray();
var temp2 = input.Select(x => DoSomethingTwo(x)).ToArray();
var finalAnswer = input.Select(x => DoSomethingThree(x)).ToArray();
```

因为现在专门调用了 ToArray()来强制对每个中间步骤进行枚举，因此我们会对输入列表中的每一项执行 DoSomethingOne()，再进入下一个环节。

现在，日志文件看起来会是这样的：

```
18/08/1982 11:24:00 - DoSomethingOne(75)
18/08/1982 11:24:01 - DoSomethingOne(22)
18/08/1982 11:24:02 - DoSomethingOne(36)
18/08/1982 11:24:03 - DoSomethingTwo(75)
18/08/1982 11:24:04 - DoSomethingTwo(22)
18/08/1982 11:24:05 - DoSomethingTwo(36)
18/08/1982 11:24:06 - DoSomethingThree(75)
18/08/1982 11:24:07 - DoSomethingThree(22)
18/08/1982 11:24:08 - DoSomethingThree(36)
```

出于这个原因，我通常会主张尽可能晚地调用 ToArray()或 ToList()[28]，以最大限度地推迟操作的执行。而且，如果后续逻辑阻止了枚举的发生，这些操作可能根本就不会被执行。

在需要提高性能或避免多次迭代的情况下，会有一些例外。当可枚举对象还未被枚举时，它虽然不包含任何数据，但这些待执行的操作仍会占用内存。如果堆积了太多可枚举对象——特别是进行递归操作时——可能会占用过多内存，导致性能下降，甚至可能出现栈溢出（stack overflow）的情况。

[28] 作为函数式编程的实践者，同时也是高度抽象接口的提倡者，我从不使用 ToList()，即使它在速度上略占优势。我始终选择使用 ToArray()。

首选表达式而非语句

本章后续部分将提供更多示例，展示如何通过更有效地使用 LINQ 来避免使用 `if`、`where` 和 `for` 这样的语句，或者避免改变状态（即更改变量的值）。在某些情况下，可能无法用现成的 C#功能完全替代这些语句，但本书的其余部分会探讨如何应对这种情况。

低调的 `Select`

读到这里，你很可能已经对 `Select()`和它的用法有所了解了。但我注意到，许多人都似乎不太了解它的一些特性，而这些特性可以使代码更具函数式风格。

第一个特性在上一节展示过了，即可以链式调用它们。可以创建一系列 `Select()`函数调用（一个接一个地排列，或者写成一行代码），也可以将每个 `Select()`的结果存储在不同的局部变量中。从功能上讲，这两种方法是一样的。即使在每个 `Select()`之后调用 `ToArray()`也没有关系。只要不修改生成的任何数组或者它们包含的对象，就不违背函数式编程范式。

需要避免的是命令式编程的典型做法——定义一个列表，用 `foreach` 遍历源对象，然后将每个新项加入列表中。这种做法不仅繁琐、可读性低，而且相当枯燥。

何必给自己找麻烦呢？一个简洁明了的 `Select()`语句不是更好吗？

迭代器的值是必需的

如果要将一个可枚举对象通过 `Select` 进行转换，并且转换过程中需要使用迭代器，那么该怎么办？假设遇到以下情况：

```
var films = GetAllFilmsForDirector("Jean-Pierre Jeunet")
    .OrderByDescending(x => x.BoxOfficeRevenue);

var i = 1;

Console.WriteLine("The films of visionary French director");
Console.WriteLine("Jean-Pierre Jeunet in descending order");
Console.WriteLine("of financial success are as follows:");

foreach (var f in films)
{
    Console.WriteLine($"{i} - {f.Title}");
    i++;
}

Console.WriteLine("But his best by far is Amelie");
```

为此，可以利用 `Select()`语句的一个鲜为人知的特性：它们拥有一个允许在 `Select()`操作中访问迭代器的重载版本。只需提供一个带有两个参数的 lambda 表达式，其第二个参数是一个整数，代表当前元素的索引位置。

函数式版本的代码如下所示：

```
var films = GetAllFilmsForDirector("Jean-Pierre Jeunet")
.OrderByDescending(x => x.BoxOfficeRevenue);
Console.WriteLine("The films of visionary French director");
Console.WriteLine("Jean-Pierre Jeunet in descending order");
Console.WriteLine("of financial success are as follows:");
var formattedFilms = films.Select((x, i) => $"{i} - {x.Title}");
Console.WriteLine(string.Join(Environment.NewLine, formattedFilms));
Console.WriteLine("But his best by far is Amelie");
```

掌握这些技巧后，就几乎没有必须使用 `foreach` 循环和列表的情况了。得益于 C#对函数式编程范式的支持，问题几乎总能用声明式方法来解决。

这两种获取索引位置变量 `i` 的方法很好地展示了命令式代码与声明式代码的区别。命令式的、面向对象的方法要求开发人员手动创建一个变量来存储 `i` 的值，并且还需要明确指定在哪里对这个变量进行递增。相反，声明式代码不关注变量在哪里定义或每个索引值是如何确定的。

注意这里是如何使用 `string.Join` 来拼接字符串的。这不仅是 C#语言中的另一枚沧海遗珠，而且是聚合（aggregation）的一个示例——也就是将一系列数据项聚合成一个。这正是接下来几节要讨论的内容。

没有初始数组怎么办?

如果一开始就有一个数组或其他类型的集合，那么用上面的技巧获取每次迭代中的 i 值非常有效。但是，如果没有现成的数组怎么办？如果需要按照设定的次数进行迭代呢？

对于这些相对罕见的情况，我们需要的是 `for` 循环而不是 `foreach` 循环。那么，如何凭空创建一个数组呢？在这种情况下，就轮到 `Enumerable.Range` 和 `Enumerable.Repeat` 这两个静态方法登场了。

`Range` 从一个起始整数值开始创建数组，要求我们指定数组中元素的数量，然后据此创建一个整数数组。下面是一个示例：

```
var a = Enumerable.Range(8, 5);
var s = string.Join(", ", a);
// s = "8, 9, 10, 11, 12"
// 包含 5 个元素，每个元素比前一个元素大 1，从 8 开始。
```

创建数组后，就可以应用 LINQ 操作来获取最终结果了。例如，假设我要为我女儿准备一份九九乘法表（只包含乘以 9 的那一部分），如下所示：[29]

```
var nineTimesTable = Enumerable.Range(1,10)
    .Select(x => x + " times 9 is " + (x * 9));
```

[29] 不，苏菲，光数手指头是不行的。

```
var message = string.Join("\r\n", nineTimesTable);
```

下面是另一个例子：假设需要从一个网格（grid）中获取所有值，这些值是通过各个单元格的 x 和 y 坐标来确定的，而我们可以访问一个网格存储库来获取这些值。

假设网格是 5×5 的，我们可以这样获取每个值：

```
var coords = Enumerable.Range(1, 5)
    .SelectMany(x => Enumerable.Range(1, 5)
        .Select(y => (X: x, Y: y))
);

var values = coords.Select(x => this.gridRepo.GetVal(x.Item1,x.Item2);
```

第一行代码生成一个包含值[1, 2, 3, 4, 5]的整数数组。接着，我们使用 Select() 方法，为这个数组中的每一个整数调用 Enumerable.Range，将每个整数转换成另一个包含五个整数的数组。此时，我们得到了一个包含五个元素的数组，每个元素自身也是一个包含五个整数的数组。通过对这个嵌套数组使用 Select()，我们将这些子元素转换成一个元组，元组中包含父数组中的一个值（x）和子数组中的一个值（y）。接着，我们使用 SelectMany()将所有内容展平（flatten）为一个简单的列表，这个列表包含了所有可能的坐标组合，即(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (2, 1), (2, 2)……。

为了获取值，可以对坐标数组进行 Select 操作，将其转换成一系列对仓库的 GetVal()函数的调用，传入之前创建的坐标元组中的 X 和 Y 的值。

在另一些情况下，我们可能每次都需要相同的起始值，但根据其在数组中的位置，我们需要以不同的方式对其进行转换。这正是 Enumerable.Repeat 的用武之地。Enumerable.Repeat 会创建一个指定大小的数组（在这里被视为可枚举对象），其中每个元素都是相同的、由用户提供的值。

Enumerable.Range 无法进行倒数计数。假设需要反向执行前面的例子，从(5,5)执行到(1,1)，那么可以通过以下代码实现：

```
var gridCoords = Enumerable.Repeat(5, 5).Select((x, i) => x - i)
.SelectMany(x => Enumerable.Repeat(5, 5)
.Select((y, i) => (x, y - i))
);

var values = gridCoords.Select(x => this.gridRepo.GetVal(x.Item1,x.Item2);
```

这段代码看似复杂了很多，但其实并非如此。这里只是将 Enumerable.Range 调用换成了两步操作。

首先，通过调用 Enumerable.Repeat，整数 5 被重复了五次，得到一个这样的数组：[5, 5, 5, 5, 5]。

接着，使用包含 i 值的重载版本的 Select()，并从数组中的当前值中减去 i 值。因此，在第一次迭代中，返回值是数组中的当前值（5）减去 i 的值（在这种情况下，第一次迭代的 i 值为 0），得到的结果是 5。在下一次迭代中，i 的值为 1，5 减 1 等于 4，所以结果是 4。以此类推。

最后，我们得到了一个类似于这样的数组：(5, 5), (5, 4), (5, 3), (5, 2), (5, 1), (4, 5), (4, 4)……。

尽管这个概念还可以进一步拓展，但本章探讨的是一些相对简单的情况，仅涉及 C#的现成功能，而不需要对它进行任何特殊处理。

合而为一：聚合的艺术

前面已经探讨了使用循环将一种事物转换成另一种事物的情况，即输入 X 个项 → 输出 X 个新项。现在，我想谈谈循环的另一个应用场景：将多个项合并为一个项。

这可以是统计总数；计算平均值、均值或其他统计数据；或者是其他更复杂的聚合操作。在传统的编程中，我们会设置一个循环和一个用于跟踪状态的变量，并在循环内部根据数组中的每个元素不断更新状态。下面是一个简单的示例：

```
var total = 0;
foreach(var x in listOfIntegers)
{
    total += x;
}
```

LINQ 提供了一个内置的方法来实现这一点：

```
var total = listOfIntegers.Sum();
```

其实不应该用如此繁琐的方式执行这类操作。即使要计算的是一个对象数组中的某个属性的总和，LINQ 也能提供相应的支持：

```
var films = GetAllFilmsForDirector("Alfred Hitchcock"); // 获取所有希区柯克电影数据
var totalRevenue = films.Sum(x => x.BoxOfficeRevenue);  // 只统计总票房
```

另一个以同样的方式计算平均数的函数是 Average()。据我所知，目前还没有能够直接计算中位数的函数。

不过，可以用一小段函数式风格的代码来计算中位数，如下所示：

```
var numbers = new []
{
    83,
    27,
    11,
    98
};
bool IsEvenNumber(int number) => number % 2 == 0;

var sortedList = numbers.OrderBy(x => x).ToArray();
var sortedListCount = sortedList.Count();

var median = IsEvenNumber(sortedList.Count())
    ? sortedList.Skip((sortedListCount/2)-1).Take(2).Average()
    : sortedList.Skip((sortedListCount) / 2).First();

// median = 55.
```

有时需要进行更复杂的聚合操作。例如，假设需要从一个包含复杂对象的可枚举对象中计算两个属性的总和，传统的过程式代码可能会这样写：

```
var films = GetAllFilmsForDirector("Christopher Nolan"); // 诺兰的电影数据
var totalBudget = 0.0M;   // 总预算
var totalRevenue = 0.0M;  // 总票房
foreach (var f in films)
{
    totalBudget += f.Budget;
    totalRevenue += f.BoxOfficeRevenue;
}
```

虽然可以调用两次 `Sum()`函数，但这意味着要对可枚举对象进行两次遍历，这绝不是最高效的获取信息的方式。更好的办法是利用 LINQ 另一个鲜为人知的特性：`Aggregate()`函数。它由以下几部分组成：

- **种子**（Seed）：用于计算最终值的初始值。
- **聚合器函数**（Aggregator Function）：它有两个参数：正在聚合的可枚举对象的当前项，以及当前的累加总和（running total）。

其中，种子不一定是 C#语言的某种基元类型（primitive type），比如 `int`；它也可以是一个复杂的对象。不过，若想以函数式编程风格重写之前的代码示例，只需要一个简单的元组，如下所示：

```
var films = GetAllFilmsForDirector("Christopher Nolan");
var (totalBudget, totalRevenue) = films.Aggregate(
    (Budget: 0.0M, Revenue: 0.0M),
    (runningTotals, x) => (
        runningTotals.Budget + x.Budget,
        runningTotals.Revenue + x.BoxOfficeRevenue
    )
);
```

在合适的场景下，`Aggregate()`是 C#中非常强大的特性，值得投入时间去深入探索和理解。它还体现了函数式编程中一个重要概念：递归（Recursion）。

自定义迭代行为

递归是许多函数式迭代方法的基础。简单来说，递归是一种编程技术，其中函数会重复调用自己，直至满足某个条件为止。

递归是一种强大的技术，但在 C#中使用时需要留意一些限制：

- 如果编写不当，递归代码可能导致不定循环[30]，直至要求用户手动终止程序或者因为栈空间彻底耗尽而造成程序崩溃。就像英国地下城探险真人秀《Knightmare》的地牢大师 Treguard 所说的那样：“哦，糟糕。”[31]
- 与其他形式的迭代相比，C#中的递归所占用的内存往往比较多。虽然有办法解决这个问题，但那是另一章的话题了。

对于递归，我稍后还有很多话要说。考虑到本章的目的，这里只会给出最简单的示例。

假设需要遍历一个枚举对象，但不确定要遍历多长时间。我们有一个列表，其中列出了整数的差值（delta value，即每次增减的数值），并想计算从某个起始值（无论它是多少）开始，需要经过多少次操作才能使值变为 0。

虽然可以使用 `Aggregate()`函数轻松得出最终数值，但我们并不关心这个最终数值。我们关心的是过程中的每一个中间值，并希望在迭代过程中的某一处提前终止。虽然这只是个简单的算术运算，但在处理复杂对象时，这种提前终止的能力可能会显著提升程序的性能。

传统的过程式代码可能是这样的：

```
var deltas = GetDeltas().ToArray();
var startingValue = 10;
var currentValue = startingValue;
var i = -1;

foreach(var d in deltas)
{
    if(currentValue == 0)
    {
        break;
    }
    i++;
    currentValue = startingValue + d;
}

return i;
```

在这个示例中，如果起始值就是要寻找的目标，那么返回`-1`，否则返回数组中使结果达到 `0` 的那个元素的索引。

以下代码展示了如何通过递归来实现：

```
var deltas = GetDeltas().ToArray();

int GetFirstPositionWithValueZero(int currentValue, int i = -1) =>
```

[30] 译注：之前说过，不定（indefinite）循环不一定是无限（infinite）循环。

[31] 译注：Treguard 的形象是一个身穿长袍、戴着兜帽的神秘人物，他主持着游戏，引导参赛者穿越充满挑战和危险的地下城。他以其独特的嗓音、戏剧化的风格和幽默的语言而闻名，是节目的标志性人物之一。Treguard 的角色形象深受观众喜爱，他的口头禅“Oooh, nasty”（哦，糟糕）和“Enter, stranger”（进来吧，陌生人）成为了节目的经典台词。

```
        currentValue == 0
        ? i
        : GetFirstPositionWithValueZero(currentValue + deltas[i + 1], i + 1);

    return GetFirstPositionWithValueZero(10);
```

虽然这是函数式代码，但它不太理想。嵌套函数在某些情况下确实很有用，但就个人而言，我认为这段代码的可读性不够高。代码虽然展现了递归的魅力，但还有提升的空间。

另一个主要问题是，如果差值列表很长，这个方法的效率会大打折扣。让我用例子说话。

假设差值列表里只有三个值：`2`，`-12` 和 `9`。我们期望得到的答案是 `1`，因为数组的第二个元素（索引为 1）的操作使得结果变为了 `0`（10 + 2 – 12）。此外，我们预期列表中的数字 `9` 将不会用于计算。这就是我们想通过代码来实现的效率提升。

不过，递归代码实际上是如何工作的呢？

首先，程序调用了 `GetFirstPositionWithValueZero()`，并传入了当前值（即起始值）`10`，而 `i` 被默认设置为`-1`。

函数体是一个三元 `if` 语句。如果结果达到 `0`，函数就会返回 `i`；否则，就用更新后的 `currentValue` 和 `i` 值再次调用自己。后者发生在第一个差值上（即 `i = 0`, `currentValue = 2`），因此会再次调用 `GetFirstPositionWithValueZero()`，结果是 `currentValue` 更新为 `12`，`i` 为 `0`。

新的值不为 `0`，所以 `GetFirstPositionWithValueZero()`的第二次调用将再次调用它自己，这次用的是更新后的当前值（加上了 `deltas[1]`）并且 `i` 的值增加到了 `1`。因为 `deltas[1]`为`-12`，所以第三次调用的结果为 `0`，这意味着可以直接返回 `i` 了。

不过，这里有个问题。尽管第三次调用得到了答案，但前两次调用仍在内存中，占用着调用栈。第三次调用返回的 `1` 被逐级向上传递给 `GetFirstPositionWithValueZero()`的第二次调用，该调用现在也返回 `1`，以此类推，直到最后 `GetFirstPositionWithValueZero()`的第一次调用也返回 `1`。

为了更直观地理解这个过程，可以像下面这样来想象它：

```
GetFirstPositionWithValueZero(10, -1)
    GetFirstPositionWithValueZero(12, 0)
        GetFirstPositionWithValueZero(0, 1)
        return 1;
    return 1;
return 1;
```

如果数组中只有三个元素，这样做不会有太大的问题，但如果有数百个呢？就像我之前说的那样，递归是一个强大的工具，但它在 C#中有一定的限制。更纯粹的函数式编程语言（例如 F#）提供了名为**尾调用优化递归**（Tail Call Optimized Recursion）的特性，使得递归可以在不占用过多内存的情况下使用。

尾递归是一个重要概念，将在第 9 章中深入探讨，因此本节不会过多地展开。就目前而言，尽管尾递归在.NET 的公共语言运行库（Common Language Runtime，CLR）中是可用的，但标准 C#还不支持。我们可以利用一些技巧实现尾递归，但这对于本章来说过于复杂了，所以我把这些技巧留到第 9 章讨论（不见不散喔）。现在，请按照这里所描述的那样理解递归，并记住在使用它的时候要谨慎。

使代码不可变

在 C#的函数式编程世界里，LINQ 只是冰山一角。我想探讨的另一个重要特性是不可变性（也就是说，一旦变量被声明，它的值就不能再被改变）。那么，C#中能在多大程度上实现代码的不可变性呢？

首先，C# 8.0 及以上版本引入了一些关于不可变性的新特性。这方面的详细内容将在第 3 章讨论，而在本章中，我将重点讨论适用于几乎所有.NET 版本的通用概念。

先来看看一个简单的 C#代码片段：

```
public class ClassA
{
    public string PropA { get; set; }
    public int PropB { get; set; }
    public DateTime PropC { get; set; }
    public IEnumerable<double> PropD { get; set; }
    public IList<string> PropE { get; set; }
}
```

这段代码是不可变的吗？显然不是。这些属性都可以通过 setter 方法（称为“赋值方法”）被重新赋值。`IList` 还提供了一系列函数，允许添加或删除其底层数组中的元素。

可以将所有 setter 设为私有，这样一来，就必须通过一个详细的构造函数来实例化这个类了，如下所示：

```
public class ClassA
{
    public string PropA { get; private set; }
    public int PropB { get; private set; }
    public DateTime PropC { get; private set; }
    public IEnumerable<double> PropD { get; private set; }
    public IList<string> PropE { get; private set; }

    public ClassA(
        string propA,
        int propB, DateTime propC,
        IEnumerable<double> propD,
        IList<string> propE)
    {
        this.PropA = propA;
        this.PropB = propB;
        this.PropC = propC;
        this.PropD = propD;
        this.PropE = propE;
    }
```

```
    }
```

现在，这个类是不可变的吗？答案仍然是否定的。我们不能在 `ClassA` 外部将任何属性替换成新对象，这一点很好。属性可以在类的内部被替换，但开发人员可以确保不添加这样的代码。我们应该通过某种代码审查机制来确保这一点。

`PropA` 和 `PropC` 没什么问题；字符串和 `DateTime` 在 C#中都是不可变的。`PropB` 的整数值也没问题，因为整数类型是不可变的，除非为变量重新赋值。但是，还有其他一些问题。

`PropE` 是一个列表，虽然不能直接替换这个列表对象本身，但列表中的元素仍然可以被添加、删除和替换。如果不需要保留 `PropE` 的可变副本，可以简单地将它替换为 `IEnumerable` 或 `IReadOnlyList`。

`PropD` 属性的 `IEnumerable<double>`类型乍一看似乎没问题，但如果它是作为 `List<double>`传递给构造函数的，而且外部代码仍然在以 `List<double>`这一类型引用它呢？那样的话，外部代码还是有可能改变它的内容。

还有一种情况需要注意：

```
public class ClassA
{
    ...
    public SubClassB PropF { get; private set; }
    ...
    public ClassA(
        ...
        SubClassB propF)
    {
        ...
        this.PropF = propF
    }
}
```

`PropF` 的所有属性也可能是可变的，除非它们也将 setter 设为私有。

那么，来自代码库外部的类呢？比如微软提供的类或者第三方 NuGet 包中的类？我们无法强制这些外部类保持不可变性。

遗憾的是，C#没有提供任何方式来强制实现全面的不可变性，即使在最新版本中也是如此。能够拥有一个 C#原生方法来默认支持不可变性固然很好，但由于需要保持向后兼容性，所以这样的方法目前并不存在，甚至未来也不太可能出现。我个人的应对策略是，在编程时，我始终假设项目中的一切都是不可变的，并且从不更改任何对象。C#没有任何机制可以强制实现不可变性，因此，你需要自己做出判断，或者和团队一起商讨是否要做出这种假设。

汇总：完整的函数式流程

之前讨论了很多可以立即用来使代码更加函数式的简单技术。本节要展示一个完整（虽然规模很小）的应用程序，以演示端到端的函数式过程。

接下来，我们打算编写一个简单的 CSV 解析器。本例的目标是读取一个 CSV 文件的全部内容，其中包含《神秘博士》[32]前几季的相关数据。读取数据后，我们会将其解析为一个 POCO（Plain Old C# Object，即仅包含数据而不包含任何逻辑的类），接着将这些数据整合成一个报告，统计每季的剧集数和缺失剧集的数量。[33]在本例中，我对 CSV 解析进行了简化；不用担心字符串字段两侧的引号、字段值中的逗号或者任何需要额外解析的值。对于这些情况，有许多第三方库可供使用，而这里只是想通过简单的例子来阐明一个观点。

这个完整的过程代表了一个非常典型的函数式流程：从一个单一项目开始，将其分解为列表，对列表进行操作，然后再将其聚合成单一的值。

表 2-1 展示了 CSV 文件的结构。

表 2-1 CSV 文件结构

索引	字段名称	描述
[0]	季数	整数值，介于 1 到 39 之间。虽然指明这一点可能会让这本书在未来显得有些过时，但到目前为止，《神秘博士》确实只有 39 季。
[1]	剧集名称	一个我不太关心的字符串字段。
[2]	编剧	同上。
[3]	导演	同上。
[4]	剧集数	直到 1989 年，每一季《神秘博士》都由 14 个剧集组成。
[5]	缺失剧集数	目前缺失的剧集数。任何大于 0 的数字对我而言都太多了，但这就是生活。

我们的目标是创建一个报告，其中只包括以下几个字段：

- 季数
- 剧集总数
- 缺失剧集数
- 缺失剧集百分比

现在开始编写代码。

```
var text = File.ReadAllText(filePath);
// 将文件的全部内容分割成一个数组，每一行（即每条记录）是数组中的一个元素
var splitLines = text.Split(Environment.NewLine);
```

[32] 这是一部从 1963 年开始播出的英国科幻电视剧。在我心目中，《神秘博士》是有史以来最棒的电视剧。不接受反驳。

[33] 遗憾的是，BBC 在 20 世纪 70 年代销毁了许多《神秘博士》的剧集。如果你恰好保留了这些剧集的存档，请务必归还。

```
// 将每一行再分割为字段数组，以逗号(',')为分隔符。为方便每次访问，将结果转换为数组形式
var splitLinesAndFields = splitLines.Select(x => x.Split(",").ToArray());

// 将字符串数组的每个字段转换为数据类
// 将非字符串字段解析成正确的类型
// 严格来说，考虑到之后的数据聚合操作，这一步不是必需的，但我倾向于编写易于扩展的代码
var parsedData = splitLinesAndFields.Select(x => new Story
{
    SeasonNumber = int.Parse(x[0]),
    StoryName = x[1],
    Writer = x[2],
    Director = x[3],
    NumberOfEpisodes = int.Parse(x[4]),
    NumberOfMissingEpisodes = int.Parse(x[5])
});

// 按 SeasonNumber 进行分组，得到每一季的 Story 对象数组
var groupedBySeason = parsedData.GroupBy(x => x.SeasonNumber);

// 使用包含 3 个字段元组作为聚合状态：
// S (int) = 季数。虽然对聚合操作来说不是必需的，但我们需要一种方法将每组聚合后的总数与特定的季数关联起来
// NumEps (int) = 该季中所有剧集的总数
// NumMisEps (int) = 该季中缺失剧集的总数
var aggregatedReportLines = groupedBySeason.Select(x =>
    x.Aggregate((S: x.Key, NumEps: 0, NumMisEps: 0),
        (acc, val) => (acc.S,
        acc.NumEps + val.NumberOfEpisodes,
        acc.NumMisEps + val.NumberOfMissingEpisodes)
    )
);

// 将基于元组的结果集转换为适当的对象，并添加计算字段 PercentageMissing
// 严格来说，这一步不是必需的，但它提高了代码的可读性和可扩展性
var report = aggregatedReportLines.Select(x => new ReportLine
{
    SeasonNumber = x.S,
    NumberOfEpisodes = x.NumEps,
    NumberOfMissingEpisodes = x.NumMisEps,
    PercentageMissing = (double)x.NumMisEps / x.NumEps * 100
});

// 将报告行格式化为字符串列表
var reportTextLines = report.Select(x =>
    $"{x.SeasonNumber}\t{x.NumberOfEpisodes}\t" +
    $"{x.NumberOfMissingEpisodes}\t{x.PercentageMissing:F2}");

// 将行联接成一个大字符串，每行之间用换行符分隔
var reportBody = string.Join(Environment.NewLine, reportTextLines);
var reportHeader = "Season\tNo Episodes\tNo MissingEps\tPercentage Missing";

// 最终报告由标题、一个换行符，和 reportbody 组成
var finalReport = $"{reportHeader}{Environment.NewLine}{reportBody}";
```

最终得到的报告大概是这样的（`\t` 代表制表符，它使输出更易于阅读）：

```
Season  No Episodes  No Missing Eps  Percentage Missing
```

```
1       42          9           21.4
2       39          2           5.1
3       45          28          62.2
4       43          33          76.7
5       40          18          45.0
6       44          8           18.2
7       25          0           0.0
8       25          0           0.0
9       26          0           0.0
...
```

这段示例代码可以写得更加紧凑，通过一个连续而流畅的长表达式把几乎所有内容组合在一起，如下所示：

```
var reportTextLines = File.ReadAllText(filePath)
    .Split(Environment.NewLine)
    .Select(x => x.Split(",").ToArray())
    .GroupBy(x => x[0])
    .Select(x =>
        x.Aggregate((S: x.Key, NumEps: 0, NumMisEps: 0),
            (acc, val) => (acc.S,
                           acc.NumEps + int.Parse(val[4]),
                           acc.NumMisEps + int.Parse(val[5])
            )
        )
    )
    .Select(x => $"{x.S}, {x.NumEps}, {x.NumMisEps},
        {(x.NumMisEps/x.NumEps) * 100}");

var reportBody = string.Join(Environment.NewLine, reportTextLines);
var reportHeader = "Season, No Episodes, No MissingEps, Percentage Missing";

var finalReport = $"{reportHeader}{Environment.NewLine}{reportBody}";
```

采用这种方法并没有什么问题，但我喜欢把它拆分成单独的代码行，原因如下：

- 变量名有助于解释代码的意图。这其实算是半强制了一种代码注释形式。
- 可以检查中间变量，以了解它们在每一步中包含了什么信息。这降低了调试的难度，正如前一章所说的那样——这就好比回顾数学题的解题步骤，看看自己是在哪一步出错了。

这两种方法在功能上并没有实质性的差别，终端用户也不会察觉到任何不同，所以采用哪种风格完全取决于个人偏好。按照你认为最合适的方式来编写代码，但尽量保持代码易于阅读和理解。

更进一步：精进函数式编程技能

这里有一个挑战等着你。如果本章介绍的部分或全部技术对你来说很陌生，那就去实践一番，享受编程带来的乐趣吧。试着挑战自己，按照以下规则编写代码：

- 将所有变量视为不可变的：一旦变量被赋值，就不再更改它的值。换言之，将一切当作常量来对待。
- 不使用以下语句：`if`、`for`、`foreach`、`while`。只有在三元表达式中才使用 `if` 语句——即 `someBoolean ? valueOne : valueTwo` 这样的单行表达式。
- 尽可能将函数写成短小精悍的箭头函数（即 lambda 表达式）。

可以在自己的项目代码中尝试这些技巧，也可以找一个线编程挑战网站来练练手。例如，“Advent of Code”（https://oreil.ly/_yysc）或“Project Euler”（https://oreil.ly/k8tLX）都是不错的选择。

如果觉得在 Visual Studio 中为这些练习构建完整的解决方案过于繁琐，那么 LINQPad 也是一个不错的选择，它提供了一种简单快速地编写 C#代码的方式。

对这些技术有了充分的理解后，就可以进入下一个学习阶段了。希望你在这段旅途上一直保持愉快的心情！

小结

本章介绍了一系列基于 LINQ 的简单技术，通过这些技术，可以在任何采用了.NET Framework 3.5 及以上版本的 C#代码库中立即编写函数式风格的代码。这些特性是常驻的，自.NET Framework 3.5 以来，它们就存在于.NET 的每一个新版本中，无需进行任何更新或替换。

此外，本章还讨论了 `Select()`语句的高级特性、LINQ 的一些鲜为人知的功能以及进行数据聚合和递归的方法。

下一章将聚焦于 C#的一些最新进展，它们为构建和维护更为先进的代码库提供了强有力的支持。

第 3 章 C# 7.0 及后续版本的函数式编程

我不确定 C#具体是在什么时候决定成为一种同时支持面向对象和函数式编程的混合语言的。C# 3.0 版本为此做了一些准备工作，这一版引入了 lambda 表达式和匿名类型等语言特性，它们后来成了.NET 3.5 中 LINQ 的核心组成部分。

但是，在那之后的很长一段时间里，函数式编程的特性一直都没什么进展。实际上，直到 2017 年 C# 7.0 发布，函数式编程似乎才重新回到了 C#开发团队的视野中。从 C# 7.0 开始，C#的每个新版本都包含了一些令人兴奋的新特性，为函数式编程风格的提供更多支持，而且这一趋势目前还没有任何减缓的迹象！

第 2 章介绍了在几乎任何现有的 C#代码库中都可以实现的函数式特性。本章将抛弃这一假设，探讨假如代码库可以使用语言的最新特性（或者至少自 C# 7 以来发布的那些），那么可以使用其中的哪些。

元组

元组自 C# 7.0 引入，不过一些 NuGet 包也能帮助我们在旧版本 C#中使用元组。元组提供了一种便捷的方式，能临时将多个属性打包到一起，而无需再创建和维护一个类。

如果有几个属性需要临时保存一小会儿，随后便可以舍弃，那么元组就非常合适。

另外，如果需要在 `Select()`方法之间传递多个对象，或者需要在某个方法的输入和输出中传递多个数据项，那么元组也能派上用场。

下面是使用了元组的一个例子：

```
var filmIds = new[]
{
    4665,
    6718,
    7101
};

// 通过 GetFilm 和 GetCastList 函数来获取与 filmIds 数组中的
// 每个电影 ID 对应的电影信息和演员名单，并将其作为单独的属性来
// 组合成一个元组
var filmsWithCast = filmIds.Select(x => (
    film: GetFilm(x),
    castList: GetCastList(x)
));

// 这里的'x'是一个元组，现在将其转换为字符串
var renderedFilmDetails = filmsWithCast.Select(x =>
    "Title: " + x.film.Title + // 电影名称
    "Director: " + x.film.Director + // 导演
    "Cast: " + string.Join(", ", x.castList)); // 演员表
```

在本例中，针对每个电影 ID，我们都利用一个元组来结合两个查询函数所返回的数据。这样一来，就可以通过后续的 `Select()`调用将这两个对象简化为单一的返回值。

模式匹配

`switch` 语句有着悠久的历史，可能比许多在职开发者的职业生涯还要长。尽管 `switch` 语句有其用武之地，但其功能相对有限。函数式编程把这个概念提升到了一个新高度，即所谓的“模式匹配”。

C# 7.0 首次引入了模式匹配特性。随着后续版本的更新，模式匹配得到了多次增强，而且未来很可能还会进一步增强。

模式匹配是一种能显著减少工作量的强大技术。为了更直观地说明这一点，下面首先展示一段过程式代码，然后展示如何在 C#的不同版本中通过模式匹配来实现相同的功能。

银行账户的过程式解决方案

让我们通过一个典型的面向对象编程案例来理解：银行账户。下面将设计多种银行账户类型，它们根据不同的规则来计算利息。这些规则并非出自于真实的银行业务。相反，它们都是我虚构的。

我设定了以下规则：

- 标准银行账户（`StandardBankAccount`）的利息计算方式是将账户余额与利率相乘。
- 对于余额不超过 10 000 美元的高级银行账户（`PremiumBankAccount`），其计息方式与标准账户相同。
- 对于余额超过 10 000 美元的高级银行账户（`PremiumBankAccount`），则将享有额外的奖励利息。
- 百万富翁银行账户（`MillionairesBankAccount`）中的钱并不是只允许一百万。相反，账户中的钱比一个十进制数能容纳的最大值（一个非常、非常大的数字，大约是8×10^{28}）还要大。所以，他们一定非常富有。使用一个溢出余额属性（`OverflowBalance`）来容纳从最大十进制值溢出的钱。百万富翁的利息是基于这两个余额来计算的。
- 大富翁玩家银行账户（`MonopolyPlayersBankAccount`）在经过“起点”（Go）时会额外获得 200 美元。我不打算在程序中实现“直接入狱”（Go Directly to Jail）逻辑，因为我们一天的时间有限。[34]

基于上述描述，我们定义了所需的各种类。

```
public class StandardBankAccount
{
    public decimal Balance { get; set; }
    public decimal InterestRate { get; set; }
```

[34] 译注：前面说过，作者虚构了这些账户糊弄。最后一种账户来自“大富翁”游戏。

```
}

public class PremiumBankAccount : StandardBankAccount
{
    public decimal BonusInterestRate { get; set; }
}

public class MillionairesBankAccount : StandardBankAccount
{
    public decimal OverflowBalance { get; set; }
}

public class MonopolyPlayersBankAccount : StandardBankAccount
{
    public decimal PassingGoBonus { get; set; }
}
```

下面用过程式方法（或者说繁琐方法）为银行账户实现 CalculateNewBalance()函数，它计算加上利息之后的新余额。

```
public decimal CalculateNewBalance(StandardBankAccount sba)
{
    // 如果对象的实际类型是 PremiumBankAccount
    if (sba.GetType() == typeof(PremiumBankAccount))
    {
        // 强制转换为正确的类型，以便加上奖息
        var pba = (PremiumBankAccount)sba;
        if (pba.Balance > 10000)
        {
            return pba.Balance * (pba.InterestRate + pba.BonusInterestRate);
        }
    }

    // 如果对象的实际类型是百万富翁银行账户
    if(sba.GetType() == typeof(MillionairesBankAccount))
    {
        // 强制转换为正确的类型，以便加上溢出的部分
        var mba = (MillionairesBankAccount)sba;
        return (mba.Balance * mba.InterestRate) +
               (mba.OverflowBalance * mba.InterestRate);
    }

    // 如果对象的实际类型是大富翁玩家银行账户
    if(sba.GetType() == typeof(MonopolyPlayersBankAccount))
    {
        // 强制转换为正确的类型，以便在每次经过起点时获得奖金
        var mba = (MonopolyPlayersBankAccount)sba;
        return (mba.Balance * mba.InterestRate) +
               mba.PassingGoBonus;
    }

    // 没有适用的特殊规则
    return sba.Balance * sba.InterestRate;
}
```

和典型的过程式代码一样，这段代码不够简洁，并且代码意图也不太好理解。如果在系统发布后继续增加更多的新规则，这种代码将非常容易出问题。

面向对象的方法则可能会使用接口或多态性，即创建一个抽象基类，在其中为`CalculateNewBalance()`函数创建一个虚方法。但这种做法会导致逻辑分散于多处，而不是集中在一个清晰、易于理解的函数中。接下来的小节将展示 C#的各个新版本是如何处理这个问题的。

C# 7.0 中的模式匹配

C# 7.0 提供了两种解决这个问题的方法。第一种方法是这个版本新增的 `is` 操作符，它比之前的类型检查方法要好用得多。另外，`is` 操作符还能自动将源变量转换为正确的类型。

下面是更新后的源代码：

```
public decimal CalculateNewBalance(StandardBankAccount sba)
{
    // 如果对象的实际类型是 PremiumBankAccount
    if (sba is PremiumBankAccount pba)
    {
        if (pba.Balance > 10000)
        {
            return pba.Balance * (pba.InterestRate + pba.BonusInterestRate);
        }
    }

    // 如果对象的实际类型是百万富翁银行账户
    if(sba is MillionairesBankAccount mba)
    {
        return (mba.Balance * mba.InterestRate) +
               (mba.OverflowBalance * mba.InterestRate);
    }

    // 如果对象的实际类型是大富翁玩家银行账户
    if(sba is MonopolyPlayersBankAccount mba)
    {
        return (mba.Balance * mba.InterestRate) +
               mba.PassingGoBonus;
    }

    // 没有适用的特殊规则
    return sba.Balance * sba.InterestRate;
}
```

注意，在上述代码示例中，`is` 操作符还自动将源变量包装成了一个新的、正确类型的局部变量。代码更优雅了，并且省去了一些冗余的代码行。但是，我们还可以做得更好。现在，让我们有请 C# 7.0 的另一个特性：**类型切换**（Type Switching）。

```
public decimal CalculateNewBalance(StandardBankAccount sba)
{
    switch (sba)
    {
        case PremiumBankAccount pba when pba.Balance > 10000:
```

```
                return pba.Balance * (pba.InterestRate + pba.BonusInterestRate);
            case MillionairesBankAccount mba:
                return (mba.Balance * mba.InterestRate) +
                       (mba.OverflowBalance & mba.InterestRate);
            case MonopolyPlayersBankAccount mba:
                return (mba.Balance * mba.InterestRate) + PassingGoBonus;
            default:
                return sba.Balance * sba.InterestRate;
        }
    }
```

挺酷的，对吧？模式匹配可以说是近年来 C#发展得最快的特性之一，C#7.0 之后的每个大版本都在不断地拓展这一特性，接下来的几个小节将具体演示。

C# 8.0 中的模式匹配

在 C# 8.0 中，模式匹配再次得到了增强，它基本保留了之前的概念，但引入了一种新的匹配语法，这种语法更类似于 JavaScript 对象表示法（JSON）或 C#对象初始化器表达式。现在，对于所检查对象的属性或子属性，可以在大括号内添加任意数量的子句，默认情况则通过弃元符号（_）来表示。

```
public decimal CalculateNewBalance(StandardBankAccount sba) =>
    sba switch
    {
        PremiumBankAccount { Balance: > 10000 } pba => pba.Balance *
            (pba.InterestRate + pba.BonusInterestRate),
        MillionairesBankAccount mba => (mba.Balance * mba.InterestRate) +
            (mba.OverflowBalance & mba.InterestRate),
        MonopolyPlayersBankAccount mba =>
            (mba.Balance * mba.InterestRate) + PassingGoBonus,
        _ => sba.Balance * sba.InterestRate
    };
}
```

另外，`switch` 现在*也*可以作为一个表达式使用，可以将其用作一个小型的、单一用途函数的主体，而且它的功能出人意料地丰富。这意味着它还可以存储在 `Func` 委托中，以便作为高阶函数传递。

下例与童年的一个经典游戏有关：剪刀、石头、布（Scissors、Stone、Paper，这个小游戏在日本被称为 Janken）。针对这个游戏，我创建了一个 `Func` 委托，并设定了几个简单的规则：

- 如果两位玩家出了相同的手势，则平局。
- `Scissors` 赢了 `Paper`。
- `Paper` 赢了 `Stone`。
- `Stone` 赢了 `Scissors`。

这个函数将从我的视角判定我与虚拟对手之间的游戏结果，因此，举例来说，如果我出剪刀战胜了对手的布，结果被计为 `Win`，因为我赢了，尽管在对手看来它输了。

```
public enum SPS
{
```

```
    Scissor,
    Paper,
    Stone
}

public enum GameResult
{
    Win,  // 赢了
    Lose, // 输了
    Draw  // 平局
}

var calculateMatchResult = (SPS myMove, SPS theirMove) =>
    (myMove, theirMove) switch
    {
        _ when myMove == theirMove => GameResult.Draw,
        ( SPS.Scissor, SPS.Paper)  => GameResult.Win,
        ( SPS.Paper, SPS.Stone )   => GameResult.Win,
        ( SPS.Stone, SPS.Scissor ) => GameResult.Win,
        _ => GameResult.Lose
    };
```

通过将判断胜者的逻辑封装在一个类型为 `Func<SPS, SPS>`的变量中，可以将这个逻辑方便地传递到程序中任何需要它的地方。

该逻辑可以作为函数的参数传入，以便在运行时动态地注入功能性，如下所示：

```
public string formatGames(
    IEnumerable<(SPS,SPS)> game,
    Func<SPS,SPS,Result) calc) =>

string.Join("\r\n",
    game.Select((x, i) => "Game " + i + ": " +
        calc(x.Item1,x.Item2).ToString()));
```

测试 `formatGames` 函数时，如果暂时不想使用它依赖的实际业务逻辑（即 `calculateMatchResult` 函数），那么可以简单地注入一个自定义的 `Func`（例如，让它总是返回平局），这样就不必关心实际的计算逻辑是什么了——实际的逻辑可以在其他某个专门的地方验证。这是让结构变得更加实用的一个小技巧。

C# 9.0 中的模式匹配

C# 9.0 虽然没有带来重大更新，但引入了一些实用的小改进。现在，可以在 `switch` 表达式的大括号内使用原本只能在 `is` 表达式中使用的 `and` 和 `not` 关键字。另外，在不需要使用强制转换类型的属性时，不再需要创建局部变量。

这些改进虽然不是革命性的，但确实进一步减少了样板代码的数量，并为我们提供了更具表现力的语法。

在接下来的示例中，我利用这些新特性为银行账户增加了一些规则。现在有两种 PremiumBankAccount，它们具有不同等级的特殊利率[35]，还有一个表示账户已关闭的类型，这种账户不产生任何利息。

```
public decimal CalculateNewBalance(StandardBankAccount sba) =>
    sba switch
    {
        PremiumBankAccount { Balance: > 10000 and <= 20000 } pba => pba.Balance *
            (pba.InterestRate + pba.BonusInterestRate),
        PremiumBankAccount { Balance: > 20000 } pba => pba.Balance *
            (pba.InterestRate + pba.BonusInterestRate * 1.25M),
        MillionairesBankAccount mba => (mba.Balance * mba.InterestRate) +
            (mba.OverflowBalance + mba.InterestRate),
        MonopolyPlayersBankAccount {CurrSquare: not "InJail" } mba =>
            (mba.Balance * mba.InterestRate) + mba.PassingGoBonus;
        ClosedBankAccount => 0,
        _ => sba.Balance * sba.InterestRate
    };
```

还不错，对吧？

C# 10.0 中的模式匹配

和 C# 9.0 一样，C# 10.0 也新增了一个能节省时间和避免大量样板代码的特性。下面是一种简单的语法，用于对正在进行类型检查的子对象的属性进行比较。

```
public decimal CalculateNewBalance(StandardBankAccount sba) =>
    sba switch
    {
        PremiumBankAccount { Balance: > 10000 and <= 20000 } pba =>
            pba.Balance * (pba.InterestRate + pba.BonusInterestRate),
        MillionairesBankAccount mba =>
            (mba.Balance * mba.InterestRate) +
               (mba.OverflowBalance + mba.InterestRate),
        MonopolyPlayersBankAccount {CurrSquare: not "InJail" } mba =>
            (mba.Balance * mba.InterestRate) + PassingGoBonus,
        MonopolyPlayersBankAccount {Player.FirstName: "Simon" } mba =>
            (mba.Balance * mba.InterestRate) + (mba.PassingGoBonus / 2),
        ClosedBankAccount => 0,
        _ => sba.Balance * sba.InterestRate
    };
```

这个稍显滑稽的例子确保在玩大富翁的时候，所有名叫“Simon”的玩家在经过起点时都拿不到太多钱。唉，可怜的我。

现在，建议思考一下本例所展示的函数。想一想，要是不使用模式匹配表达式，那么需要额外编写多少代码！严格来讲，上述函数仅由一行代码组成，一行……真的很长……的代码，其中包含大量换行符来提高可读性。尽管如此，模式匹配表达式的优势依然是显而易见的。

[35] 不过老实说，现实中的银行绝不可能提供这样的利率。

C# 11.0 中的模式匹配

C# 11.0 引入了一个新的模式匹配特性，虽然它的使用场景可能相对有限，但在合适的时候非常有用。.NET 团队新增了根据可枚举对象的内容进行匹配的能力，并且能将它的元素解构为独立的变量。

假设现在要开发一个简单的文字冒险游戏（还记得 MUD 吗？），这类游戏在我年轻的时候非常流行。玩家通过输入命令来玩游戏，类似于《猴岛》（Monkey Island），但完全是文本形式的。在那个时候，玩家需要更多地运用自己的想象力。

第一个任务是从用户那里接收输入并判断他们想做什么。在英语中，命令几乎总是以相关的动词作为句子的第一个词：例如：GO WEST（向西走），KILL THE GOBLIN（杀死哥布林），EAT THE SUSPICIOUS-LOOKING MUSHROOM（吃下看起很来可疑的蘑菇）。这里相关的动词分别是 GO、KILL 和 EAT。

接下来看看如何运用 C# 11 的模式匹配来实现这一任务，如下所示：

```
var verb = input.Split(" ") switch
{
    ["GO", "TO",.. var rest] => this.actions.GoTo(rest),
    ["GO", .. var rest]      => this.actions.GoTo(rest),
    ["EAT", .. var rest]     => this.actions.Eat(rest),
    ["KILL", .. var rest]    => this.actions.Kill(rest)
};
```

`switch` 表达式中的两个点（`..`）表示“我不关心数组中的其他内容；请忽略它们”。我们把一个变量放在这两个点之后，用以存储数组中不符合特定匹配模式的所有内容。

在本例中，如果输入文本 `GO WEST`，那么会调用 `GoTo()`函数，并向其传递一个单元素数组`["WEST"]`，这是因为 `GO` 符合模式匹配条件。

还有另一种利用 C# 11 特性的巧妙方式。假设需要把人名转换为数据结构，这个数据结构包含三个部分：名字（`FirstName`）、姓氏（`LastName`），以及一个用于存储中间名（`MiddleNames`）的数组（我只有一个中间名，但有些人有好几个）。

```
public class Person
{
    public string FirstName { get; set; }
    public IEnumerable<string> MiddleNames { get; set; }
    public string LastName { get; set; }
}

// 神秘博士的演员 Sylvester McCoy 的出生名
var input = "Percy James Patrick Kent-Smith".Split(" ");

var sylv = new Person
{
    FirstName = input.First(),
    MiddleNames = input is [_, .. var mns, _] ? mns : Enumerable.Empty<string>(),
    LastName = input.Last()
};
```

上例会像下面这样实例化一个 Person 类：

```
FirstName = "Percy",
LastName = "Kent-Smith",
MiddleNames = [ "James", "Patrick" ]
```

虽然这一特性的使用场景不多，但真正用上它的时候，我可能会很高兴。它其实非常强大。

只读结构

本书不会过多地讨论结构[36]，因为已经有很多优秀的书籍深入讨论了 C#的语言特性[37]。从 C#的角度看，结构的优点在于，它们在函数之间传值而非传引用。这意味着传递给函数的是一个副本，原始数据保持不变。传统的面向对象编程会将对象传递给函数并在其内部进行修改，这个位置与实例化该对象的函数相距甚远。而这种做法对于函数式编程来说是不可接受的。在函数式编程中，我们基于类创建一个对象，然后就不再对其进行任何修改了。

结构的概念已经存在了相当长的时间了。虽然结构以传值方式传递，但其属性仍然可以修改，因此本质上并非“不可变”，这种情况一直持续到 C# 7.2。

在 C# 7.2 及之后的版本中，开发者可以为结构定义添加一个 readonly 修饰符，从而在设计阶段就确保结构的所有属性都是只读的。任何尝试为属性添加 setter（赋值方法）的行为都会导致编译错误。

由于所有属性都被设置为只读，所以在 C# 7.2 中，所有属性必须在构造函数中设置，如下所示：

```
public readonly struct Movie
{
    public string Title { get; private set; } // 电影名
    public string Director { get; private set; } // 导演
    public IEnumerable<string> Cast { get; private set; } // 演员名单

    public Movie(string title, string director, IEnumerable<string> cast)
    {
        this.Title = title;
        this.Director = director;
        this.Cast = cast;
    }
}

var bladeRunner = new Movie(
    "Blade Runner", // 银翼杀手
    "Ridley Scott", // 雷德利·斯科特
    new []
    {
        "Harrison Ford", // 哈里森·福特
```

[36] 译注：微软中文文档将 struct 称为“结构”，而不是“结构体”。

[37] 《C# 12.0 本质论》（清华大学出版社 2024 年出版）就是一个不错的起点。

```
        "Sean Young" // 肖恩·杨
    }
);
```

这种方法仍然有些不便，因为每次向结构添加属性时都要更新构造函数，但这总比之前要好。

还有一种情况值得讨论。以下代码在结构中添加了一个 `List`：

```
public readonly struct Movie
{
    public readonly string Title;
    public readonly string Director;
    public readonly IList<string> Cast;

    public Movie(string title, string director, IList<string> cast)
    {
        this.Title = title;
        this.Director = director;
        this.Cast = cast;
    }
}

var bladeRunner = new Movie(
    "Blade Runner",
    "Ridley Scott",
    new []
    {
        "Harrison Ford",
        "Sean Young"
    }
);

// 新增一名演员爱德华·詹姆斯·奥莫斯
bladeRunner.Cast.Add(("Edward James Olmos"));
```

这段代码能顺利编译，应用程序也能顺利运行，但在调用 `Add` 函数时会抛出异常。尽管结构的只读特性得到了强制，但我对潜在的未处理异常感到不太满意。

无论如何，开发者现在可以通过 `readonly` 修饰符来明确意图了，这肯定是一件好事。这个修饰符有助于防止任何不必要的可变性被引入结构中——即使这意味着需要添加额外的错误处理机制。

Init-Only Setter

C# 9.0 引入了一种新的自动属性类型，除了熟悉的 `get` 和 `set` 自动属性，现在还新增了 `init` 自动属性。

如果类的属性设置了 `get` 和 `set`，那么表明该属性可读可写。而如果改为 `get` 和 `init`，那么表明那么它的值只能在其所属对象被实例化时设置，之后便不能再更改了。

因此，只读结构（实际上，类也可以这样）现在可以使用更优雅的语法来进行实例化，然后一直保持只读状态。

```
public readonly struct Movie
{
    public string Title { get; init; }
    public string Director { get; init; }
    public IEnumerable<string> Cast { get; init; }
}

var bladeRunner = new Movie
{
    Title = "Blade Runner",
    Director = "Ridley Scott",
    Cast = new []
    {
        "Harrison Ford",
        "Sean Young"
    }
};
```

这意味着不再需要维护一个复杂的构造函数（它需要为每个属性都提供一个参数，而属性可能有几十个）的同时还要维护属性本身，这消除了一个潜在的繁琐样板代码的来源。不过，修改列表或子对象可能会引发异常的问题仍然存在。

记录类型

C# 9.0 引入了记录（`record`）类型，这是我除了模式匹配以外最喜爱的特性之一。如果还没有用过记录类型的话，请务必体验一下，它非常出色。

乍一看，它们似乎和结构差不多。C# 9.0 的记录类型基于类，因此以传引用的方式传递。

从 C# 10 开始，这一情况发生了变化，记录类型的行为更趋近于结构，这意味着它们可以传值了[38]。与结构不同的是，记录类型[39]没有 `readonly` 修饰符，所以不可变性需要由开发者来实现。下面是一个经过更新的《银翼杀手》代码示例：

```
public record Movie
{
    public string Title { get; init; }
    public string Director { get; init; }
    public IEnumerable<string> Cast { get; init; }
}

var bladeRunner = new Movie
{
```

[38] 译注：注意，C# 9.0 只允许创建记录类，编译器只允许写一个 `record` 关键字，后面不能跟 `struct`。但从 C# 10.0 开始，由于新增了用 `record struct` 创建记录结构的功能，所以允许显式声明 `record class` 来创建记录类。为了清晰起见，建议在创建记录类时总是使用 `record class`，而不要简写为 `record`。

[39] 译注：作者这里说的“记录类型”实际是 `record class`。

```
    Title = "Blade Runner",
    Director = "Ridley Scott",
    Cast = new []
    {
        "Harrison Ford",
        "Sean Young"
    }
};
```

代码看起来并没有太大不同，是吧？然而，在需要创建一个修改版时，记录类型的独特优势就显现出来了。现在，假设需要在这个 C# 10 应用程序中为《银翼杀手》的导演剪辑版创建一条新的电影记录。[40]

这一版的记录除了电影名称（`Title`）不同之外，其他方面都与原始记录相同。为了省去定义数据的步骤，我们准备直接复制原始记录的数据，只改变电影名称。如果使用只读结构，可能需要按照以下方式操作：

```
public readonly struct Movie
{
    public string Title { get; init; }
    public string Director { get; init; }
    public IEnumerable<string> Cast { get; init; }
}

var bladeRunner = new Movie
{
    Title = "Blade Runner",
    Director = "Ridley Scott",
    Cast = new []
    {
        "Harrison Ford",
        "Sean Young"
    }
};

var bladeRunnerDirectors = new Movie
{
    Title = $"{bladeRunner.Title} - The Director's Cut",
    Director = bladeRunner.Director,
    Cast = bladeRunner.Cast
};
```

这段代码遵循了函数式编程范式，看起来还算不错，但它包含大量样板代码，而为了确保数据的不可变性，这些样板代码是不可或缺的。

但是，假如维护的是某个状态对象，而该对象需要根据用户的交互或外部依赖项定期更新，那么这个问题就需要重视了。如果使用只读结构，那么将不得不复制大量属性。

记录类型新增了一个极其有用的关键字：`with`。这个关键字提供了一种快速且便捷的方式，允许创建现有记录的副本，并对其中的属性进行修改。

---

[40] 在我看来，这个版本远胜于公映版。

使用记录类型的导演剪辑版代码如下：

```
public record Movie
{
    public string Title { get; init; }
    public string Director { get; init; }
    public IEnumerable<string> Cast { get; init; }
}

var bladeRunner = new Movie
{
    Title = "Blade Runner",
    Director = "Ridley Scott",
    Cast = new []
    {
        "Harrison Ford",
        "Sean Young"
    }
};

var bladeRunnerDirectorsCut = bladeRunner with
{
    Title = $"{bladeRunner.Title} - The Director's Cut"
};
```

是不是很酷？使用记录类型，可以少写大量样板代码。

我最近用函数式 C#编写了一个文字冒险游戏。游戏中有一个关键的 `GameState` 记录类型，记录了玩家的所有最新进度。我用一个庞大的模式匹配语句分析玩家每一轮的行动，接着用一个简洁的 `with` 语句来返回一个经过修改的副本，从而实现状态的更新。这是编写状态机的一种优雅方式，而且剔除大量冗余的样板代码让代码的意图清晰了许多。

记录类型的另一个优点是可以用非常简洁的方式在一行代码内定义，如下所示：

```
public record Movie(string Title, string Director, IEnumerable<string> Cast);
```

使用这种定义方式创建 `Movie` 实例时，必须使用函数而不是大括号，如下所示：

```
var bladeRunner = new Movie(
    "Blade Runner",
    "Ridley Scott",
    new[]
    {
        "Harrison Ford",
        "Sean Young"
    }
);
```

注意，所有属性必须按顺序提供，除非像下面这样使用构造函数标签：

```
var bladeRunner = new Movie(
    Cast: new[]
    {
        "Harrison Ford",
        "Sean Young"
    },
```

```
        Director: "Ridley Scott",
        Title: "Blade Runner");
```

仍然必须提供所有属性，但可以任意调整顺序（虽然看不出有什么意义）。

使用哪种语法完全取决于个人偏好。在大多数情况下，它们是等效的。

可空引用类型（Nullable Reference Type）

可空引用类型虽然乍一看比较新奇，但和记录类型一样，它已经存在了一段时间了，实际是 C# 8 引入的一个编译器选项。这个选项是通过 CSPROJ 文件设置的，如下所示：

```
<PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <IsPackable>false</IsPackable>
</PropertyGroup>
```

如果更喜欢使用图形用户界面，那么也可以在项目属性的“生成”（Build）区域设置“可为 Null 的类型”。

严格来说，启用可空引用类型并不会改变编译器生成的代码的行为，但它会为集成开发环境和编译器添加一套额外的警告消息，以帮助避免空引用的情况。例如，图 3-1 展示了一个警告，提示 `Movie` 记录类型的属性可能为空。

如果尝试将《银翼杀手》导演剪辑版的标题设为 `null`，那么会触发如图 3-2 所示的另一个警告。

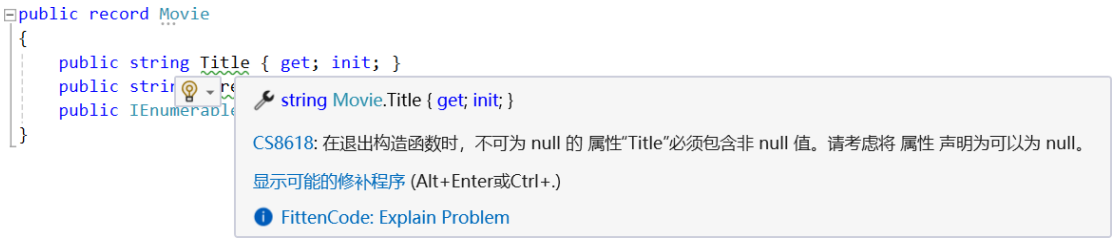


图 3-1 警告记录类型的属性可能为空

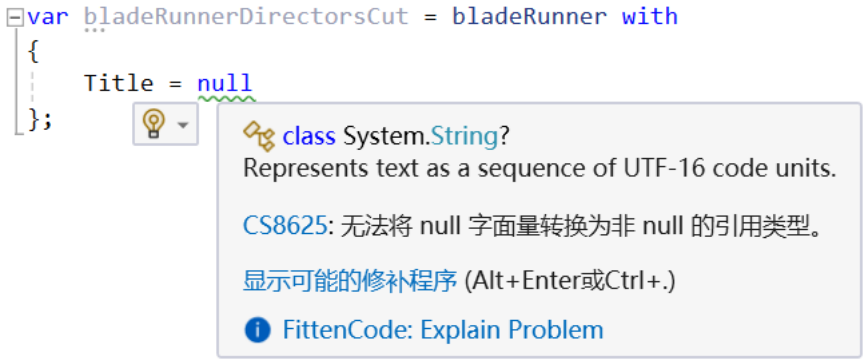


图 3-2 将属性设为 null 时出现的警告

注意，这只是编译器警告，不影响代码的正常执行。它们只是引导我们编写不太可能包含空引用异常的代码，这无疑是一件好事。

无论是否进行函数式编程，避免使用 null 都是一个好的实践。null 被称为“价值十亿美元的错误”。null 的概念由英国计算机科学家托尼·霍尔（Tony Hoare）在 20 世纪 60 年代中期提出。自那时起，null 一直是生产环境中出现 bug 的主要原因之一：将一个对象传递到某处，却意外地发现它是空的，从而导致空引用异常。不需要在这个行业呆太久，就会遇到你的第一个这样的 bug！

在代码库中使用 null 值会带来不必要的复杂性，并引入潜在的错误来源。这就是为什么要重视编译器的警告，并尽可能地避免使用 null。

如果确实有必要将一个值设为 null，那么可以通过在属性的类型声明后添加“?”字符来实现，如下所示：

```
public record Movie
{
    public string? Title { get; init; }
    public string? Director { get; init; }
    public IEnumerable<string>? Cast { get; init; }
}
```

只有在第三方库要求时，我才会向代码库中添加可空属性。但即便在这种情况下，我也绝不允许这个可空属性传递到代码的其余部分。我可能会选择将其隔离在特定区域，只允许负责解析外部数据的代码访问它，然后将其转换为更安全、更可控的结构，再传递到系统的其他地方。

展望未来

写作本书时，C# 11 已经发布，并作为.NET 7 的一部分得到了广泛应用。C# 12 的完整规范也已经公布了。值得一提的是，虽然 C# 12 引入了大量新的通用特性，但没有为函数式编程专门引入新特性。这是多年来的首次。

 C# 12.0 规范：微软官网列出了 C# 12 的所有新特性（https://tinyurl.com/mshp7b3e）。更详细的讲解请参考清华大学出版社 2024 年出版的《C# 12.0 本质论》一书（https://bookzhou.com）。

在 C# 12 中，所有类都支持主构造函数，不再局限于记录类型。这是一个减少代码噪声的好改进，尽管它并不是专门的函数式编程特性。

此外，lambda 表达式现在也可以包含默认值了。这在某些情况下确实使得编写可组合函数变得更简单了，但同样地，这不是专门的函数式编程特性。

尽管 C#在过去一年里没有添加新的功能性特性令人有点失望，但目前已经有许多函数式编程的特性可供我们探索和使用了。此外，微软已经透露，他们将在未来推出一些令人振奋的新东西……

可辨识联合

虽然还不确定 C#是否会引入**可辨识联合**（Discriminated Union），但微软确实在积极研究和开发它。

C# 12.0 规范：如果感兴趣， 可以看看 YouTube 上的“Languages and Runtime Community Standup: Considering Discriminated Unions（语言和运行时社区站会：可辨识联合）”（https://oreil.ly/psirr），微软 C#团队的成员在视频中讨论了他们对于可辨识联合的看法。

这里不会过多地讨论可辨识联合，如果想更全面地了解它的定义和用法，请参阅第 6 章。

目前，在 NuGet 上，我注意到了两个尝试实现这一概念的项目，它们分别是：

- 由哈里·麦金泰尔（Harry McIntyre）开发的 OneOf（https://oreil.ly/bhjGX）
- 由金·胡格纳-奥尔森（Kim Hugener-Olsen）开发的 Sundew.DiscriminatedUnions（https://oreil.ly/Ws3G6）

简而言之，可辨识联合允许定义一个可以同时代表几种不同类型的类型。F#原生支持可辨识联合， 但 C#目前没有提供这样的支持，并且未来也不一定会提供。

截至本书出版，C# 13 正在积极考虑引入可辨识联合，相关的讨论正在 GitHub 上进行（https://oreil.ly/E4_OS），也出了一些相关的提案（https://oreil.ly/bOX3T）。不过，最终结果仍未可知。

活动模式

活动模式（Active Pattern）是 F#的一个特性，我认为它迟早会被引入 C#。它扩展了模式匹配的功能，允许在表达式左侧的“模式”部分执行函数。下面是一个来自 F#的示例：

```
let (|IsDateTime|_|) (input:string) =
    let success, value = DateTime.TryParse input
    if success then Some value else None
```

```
let tryParseDateTime input =
    match input with
    | IsDateTime dt -> Some dt
    | _ -> None
```

如以上代码所示，F#开发者可以为表达式左侧的“模式”部分提供自定义函数。`IsDateTime` 就是一个自定义函数，如代码的第一行所示，它被定义为接收一个字符串参数。如果解析成功，则返回一个值；如果解析失败，则返回一个类似于 `null` 的结果。

模式匹配表达式 `tryParseDateTime()`使用 `IsDateTime` 作为模式。如果 `IsDateTime` 返回了一个值，那么模式匹配表达式中与之相对应的分支就会被选中，然后返回解析出的 `DateTime` 值。

不需要过于关注 F#的语法细节；它不是本书的主题。如果对 F#感兴趣，有很多优秀的书籍和资源可以提供帮助，例如：

- 艾萨克·亚伯拉罕（Isaac Abraham）所著的《F# in Action》或《Get Programming with F#》（Manning 出版社）
- 由伊恩·罗素（Ian Russell）所著的《Essential F#》（LeanPub 出版社，https://oreil.ly/Npcdt）
- 斯科特·瓦拉欣（Scott Wlaschin）创建的“F# for Fun and Profit”网站（https://oreil.ly/NP8XZ）

至于这些 F#特性是否会在 C#的未来版本中出现，目前尚不得而知，但考虑到 C#和 F#共享同一个公共语言运行时（CLR），这些特性的确有移植到 C#中的可能。

小结

本章探讨了自 C# 3.0 和 C# 4.0 开始集成函数式编程以来，C#都引入了哪些函数式特性。我们探讨了这些特性的定义、用法以及它们所带来的好处。

总的来说，函数式编程的这些特性主要分为两大类。

- **模式匹配：**C# 的模式匹配通过一种加强版的 `switch` 语句来实现，它使开发者能编写简洁而强大的逻辑。可以看到，C#的每个版本都在增加更多的模式匹配特性。
- **不可变性：**不可变性是指变量一旦实例化就无法修改。由于需要保持向后兼容性，所以 C#可能永远不会实现彻底的不可变性。不过，C#正在引入一些新特性，比如只读结构和记录类型，使得开发者能在不编写大量样板代码的情况下，实现近似于不可变性的效果。

接下来的各章将深入探讨如何以创新的方式运用 C#现有的特性，进一步丰富你的函数式编程工具箱。

第 4 章 函数式代码：巧干胜过苦干

到目前为止，我所介绍的都是微软 C#团队有意设计的函数式编程特性。这些特性以及相应的示例代码都可以在微软的官方网站上找到。不过，在这一章中，我想采取更创新的方式来使用 C#。

我不知道你怎么样，但我喜欢偷懒——至少，我不喜欢把时间浪费在编写冗长的样板代码上。函数式编程最吸引人的一点就是它那远胜于命令式代码的简洁性。

本章将展示如何进一步发挥函数式编程的潜力，超越 C#的标准功能。此外，还将了解如何在旧版本的 C#中实现一些较新的函数式特性，理想情况下，这能极大地提高日常工作的效率。

本章将探索以下几类函数式编程概念：

- **可枚举对象中的 Func 委托：**虽然 Func 委托似乎用得不多，但它们是 C#中极为强大的特性。本章将展示如何用它们来扩展 C#的功能。具体来说，我们将把 Func 委托添加到可枚举对象中，并通过 LINQ 表达式处理它们。
- **将 Func 委托用作筛选器：**还可以将 Func 委托作为筛选器使用，它们充当了一个中介的角色，帮助筛选出我们真正想要得到的值。采取这种方式，可以编写出简洁而有效的代码。
- **自定义可枚举对象：**本书之前已经讨论过了 IEnumerable 接口的妙用，但你可能不知道的是，还可以对其进行扩展来实现自定义行为。本章将展示具体如何做。

除了上述内容，本章还会探讨其他许多概念。

是时候展现 Func 的魔力了

Func 委托类型实际上是可以作为变量来存储的函数。可以定义它们需要哪些参数，以及应该返回什么类型的值，并像调用普通函数一样调用它们。下面是一个简单的例子：

```
private readonly Func<Person, DateTime, string> SayHello =
    (Person p, DateTime today) => today + " : " + "Hello " + p.Name;
```

尖括号中的最后一个类型是函数的返回类型；在它之前的所有类型都是该函数的参数。在这个例子中，Func 委托接受一个 Person 类型和一个 DateTime 类型作为参数，并返回一个字符串。

从现在起，我们将大量地使用 Func 委托，所以在继续阅读之前，请确保对它们有充分的了解。

可枚举对象中的 Func

尽管我经常看到 Func 委托被用作函数的参数，但许多开发者可能还没有意识到能将这些委托放到可枚举对象中，从而创建出一些有趣的行为。

首先，最直观的用法是将它们放入一个数组中，以便多次对同一数据进行操作：

```
private IEnumerable<Func<Employee, string>> descriptors = new []
{
    x => "First Name = " + x.firstName,
    x => "Last Name = " + x.lastName,
    x => "MiddleNames = string.Join(" ", x.MiddleNames)
};

public string DescribeEmployee(Employee emp) =>
    string.Join(Environment.NewLine, descriptors.Select(x => x(emp)));
```

使用这个技术，我们可以从单一数据源（这里是一个 Employee 对象）生成多条相同类型的记录。本例使用.NET 的内置方法 string.Join 来聚合数据，从而向终端用户展示一个整合后的字符串。

相较于使用一个简单的 StringBuilder，这种方法有几个优势。首先，可以动态构建数组。可以为每个属性及其呈现方式制定多个规则，这些规则可以根据自定义逻辑在一组局部变量中进行选择[41]。

其次，这是一个可枚举对象，采取这种定义方式，我们能够利用可枚举对象的一个特性：惰性求值（参见第 2 章）。可枚举对象不是数组；它们甚至不是数据，而是指向数据提取方式的指针。可枚举对象背后的数据源通常是一个简单的数组，但这并不是绝对的。每次通过 foreach 循环访问可枚举对象的下一个元素时，都需要执行特定的函数。可枚举对象的设计目的是等到最后一刻才转换成实际数据——通常是在 foreach 循环开始迭代时进行。大多数时候，如果可枚举对象的数据源是存储在内存中的数组，惰性求值可能不会带来显著的影响。然而，如果数据源是一个计算成本高昂的函数，或者需要从外部系统检索数据的查询，那么惰性求值就能省去许多不必要的工作。

在可枚举对象的元素被执行枚举的进程使用时，它们会逐一求值。例如，如果使用 LINQ 的 Any 函数来检查可枚举对象中的元素，那么一旦发现第一个符合条件的元素，Any 就会停止枚举，这意味着剩余元素不会被求值。

最后，从代码维护的角度来看，这种技术更易于管理。向最终结果添加信息就像向数组添加新元素一样简单。此外，这种做法对未来的开发人员也起到了一定的约束作用，防止他们在不合适的位置添加太多复杂的逻辑。

超级简单的验证器

现在，来看看一个简单的验证函数的例子。

```
// 判断密码是否有效
public bool IsPasswordValid(string password)
{
    if(password.Length <= 6)
```

[41] 译注：例如，可以添加一个 bool 类型的局部变量 showMiddleNames 来控制是否显示中间名。另外，还可以用一系列局部变量来控制名字的大小写、姓氏的显示顺序、是否添加敬称等等。

```
            return false;

        if(password.Length > 20)
            return false;

        if(!password.Any(x => Char.IsLower(x)))
            return false;

        if(!password.Any(x => Char.IsUpper(x)))
            return false;

        if(!password.Any(x => Char.IsSymbol(x)))
            return false;

        // 密码中不允许包含 Justin Bieber，因为他（贾斯汀·比伯）太有名
        if(password.Contains("Justin", StringComparison.OrdinalIgnoreCase)
           && password.Contains("Bieber", StringComparison.OrdinalIgnoreCase))
            return false;

        return true;
    }
```

对于一套颇为简单的规则来说，上面的代码显得太多了。命令式方法让我们不得不编写大量重复的样板代码。此外，如果要添加一条新规则，那么可能需要通过再新增 4 行代码来实现，而其中真正有意义的只有一行。

如果有一个办法能将这段代码简化成几行就好了。好消息是，确实有这样的办法，如下所示：

```
public bool IsPasswordValid(string password) =>
    new Func<string, bool>[]
    {
        x => x.Length > 6,
        x => x.Length <= 20,
        x => x.Any(y => Char.IsLower(y)),
        x => x.Any(y => Char.IsUpper(y)),
        x => x.Any(y => Char.IsSymbol(y)),
        x => !x.Contains("Justin", StringComparison.OrdinalIgnoreCase)
             && !x.Contains("Bieber", StringComparison.OrdinalIgnoreCase)
    }.All(f => f(password));
```

代码是不是看起来简洁多了？那么，我们都做了什么呢？我们把所有规则都放入了一个 Func 委托数组中（称为**函数数组**），其中每个 Func 委托都接收一个字符串作为输入并返回一个布尔值作为输出——换句话说，每个 Func 都负责检查字符串是否符合某个验证规则。然后，我们调用 LINQ 的.All()扩展方法，将函数数组中的每个 Func 都应用于传入的密码。其中任何一个 Func 返回 false，整个过程就会提前结束，并.All()会返回 false（如前文所述，后续 Func 将不会被访问，惰性求值通过跳过这些元素节省了时间）。只有每个 Func 都返回 true，.All()才会返回 true，表明这是一个有效的密码。

我们相当于以一种更简洁的形式重新实现了第一个代码示例，之前那些样板代码（if 语句和提前返回）现在隐式地存在于结构之中。

这种代码结构的另一个优点就是可维护性。如果需要，甚至可以为它创建一个泛型扩展方法。我经常会这么做，如下所示[42]：

```
public static bool IsValid<T>(this T @this, params Func<T, bool>[] rules) =>
    rules.All(x => x(@this));
```

这进一步减少了密码验证器的代码量，并提供了一个方便的通用结构，可以在别的地方重用。有了扩展方法，就可以直接调用 password.IsValid()，并将所有验证规则作为参数传入，如下所示：

```
public bool IsPasswordValid(string password) =>
    password.IsValid(
        x => x.Length > 6,
        x => x.Length <= 20,
        x => x.Any(y => Char.IsLower(y)),
        x => x.Any(y => Char.IsUpper(y)),
        x => x.Any(y => Char.IsSymbol(y)),
        x => !x.Contains("Justin", StringComparison.OrdinalIgnoreCase)
             && !x.Contains("Bieber", StringComparison.OrdinalIgnoreCase)
    );
```

至此，我希望你已经认识到，编写像第一个代码示例那样冗长而笨重的代码是不可取的。

我认为 IsValid 这种验证方法更易于阅读和维护。但是，如果想要编写与原始代码示例更一致的代码，可以通过使用 Any()而不是 All()来创建一个新的扩展方法，如下所示：

```
// 注意是判断 IsInvalid 而不是 IsValid
public static bool IsInvalid<T>(
    this T @this,
    params Func<T, bool>[] rules) =>
        rules.Any(rule => !rule(@this));
```

这样一来，数组中每个元素的布尔逻辑就可以像原始示例那样被反转：

```
public bool IsPasswordValid(string password) =>
    !password.IsInvalid(
        x => x.Length <= 6,
        x => x.Length > 20,
        x => !x.Any(y => Char.IsLower(y)),
        x => !x.Any(y => Char.IsUpper(y)),
        x => !x.Any(y => Char.IsSymbol(y)),
        x => x.Contains("Justin", StringComparison.OrdinalIgnoreCase)
             && x.Contains("Bieber", StringComparison.OrdinalIgnoreCase)
    );
```

[42] 译注：稍微解释一下这个泛型扩展方法。public static 是定义扩展方法所必须的。this 关键字表示这是一个扩展方法，@this 是被扩展的对象（即要验证的密码）。params 关键字允许传入任意数量的验证规则，每个规则都是一个接受类型 T 的对象并返回布尔值的函数（Func 委托）。rules.All()遍历所有规则。x => x(@this)将当前对象@this 传递给每个规则函数进行验证。如果所有规则函数都返回 true，则.All()返回 true，表示密码有效。

考虑到 `IsValid()`和 `IsInvalid()`这两个函数各有用途，我们希望在代码库中同时保留它们。为此，可以在一个函数中引用另一个函数来减少一些编码量和未来的维护工作，如下所示：

```
public static bool IsValid<T>(this T @this, params Func<T,bool>[] rules) =>
    rules.All(x => x(@this));

public static bool IsInvalid<T>(this T @this, params Func<T,bool>[] rules) =>
    !@this.IsValid(rules);
```

明智地使用这股力量，年轻的函数式编程学徒！[43]

C#旧版本中的模式匹配

模式匹配是近年来 C#最为出色的特性之一，与记录类型并列，但只有最新的.NET 版本对它提供了支持。（若想了解关于 C# 7 及以上版本对模式匹配提供的原生支持，请参见第 3 章。）

那么，是否有办法在不升级 C#版本的情况下使用模式匹配呢？答案是肯定的。虽然这种方法不如 C# 8 的原生语法那样优雅，但有总比没有好。

在下面的例子中，我们将根据英国所得税法的简化版本来计算个人所得税。需要注意的是，这些规则远比现实中的税收体系要简单得多。但是，我不想让大家迷失在复杂的税收规则中。

将应用以下规则：

- 若年收入不超过 12 570 英镑，则不收税。
- 若年收入在 12 571 至 50 270 英镑之间，则需缴纳 20%的税款。
- 若年收入在 50 271 至 150 000 英镑之间，则需缴纳 40%的税款。
- 若年收入超过 150 000 英镑，则需缴纳 45%的税款。

如果采用传统编程方式（非函数式编程）来实现这个逻辑，代码可能是这样的：

```
decimal ApplyTax(decimal income)
{
    if (income <= 12570)
        return income;
    else if (income <= 50270)
        return income * 0.8M;
    else if (income <= 150000)
        return income * 0.6M;
    else
        return income * 0.55M;
}
```

在 C# 8 及以后版本中，可以利用 `switch` 表达式将上述逻辑压缩成短短几行。如果使用的是 C# 7（.NET Framework 4.7）或更高版本，那么可以用以下代码来模拟模式匹配：

[43] 译注：作者改写了《星球大战》中的一句名言。

```
var inputValue = 25000M;
var updatedValue = inputValue.Match(
    (x => x <= 12570, x => x),
    (x => x <= 50270, x => x * 0.8M),
    (x => x <= 150000, x => x * 0.6M)
).DefaultMatch(x => x * 0.55M);
```

上述代码传入一系列元组（稍后作为元组数组来实现），每个元组都包含两个 lambda 表达式。第一个表达式判断输入是否与当前模式匹配；如果匹配，第二个表达式就会转换它的值。如果输入不与任何模式匹配，最后会应用默认模式。

这段代码比原始代码简洁了很多，但它包含了所有相同的功能。元组左侧的匹配模式很简单，但根据需要，它也可以是很复杂的表达式，甚至可以调用包含详细匹配标准的完整函数。

那么，具体要如何实现 `Match()`呢？下面是一个提供了大部分所需功能的简化版本：

```
// 专门创建一个扩展方法类来包含扩展方法
public static class ExtensionMethods
{
    public static TOutput Match<TInput, TOutput>(
        this TInput @this,
        params (Func<TInput, bool> IsMatch,
        Func<TInput, TOutput> Transform)[] matches)
    {
        var match = matches.FirstOrDefault(x => x.IsMatch(@this));
        var returnValue = match?.Transform(@this) ?? default;
        return returnValue;
    }
}
```

利用 LINQ 的 `FirstOrDefault()`方法，我们首先遍历左侧的函数，以找到一个返回 true（即满足特定条件）的函数，然后调用右侧的转换函数 `Func` 来获取修改后的值。

这种方法虽然有效，但如果所有模式都不匹配，那么可能出问题。最可能出现的是空引用异常。

为了避免这种情况，需要确保提供一个默认匹配（相当于传统 `if-else` 逻辑中的 `else` 语句，或者 `switch` 表达式使用的弃元模式匹配“`_`”）。我们的解决方案是让 `Match` 函数返回一个占位符对象，该对象要么容纳由 `Match` 表达式转换的来的值，要么执行 `Default` 模式的 lambda 表达式。

改进后的版本如下所示：

```
public static MatchValueOrDefault<TInput, TOutput> Match<TInput, TOutput>(
    this TInput @this,
    params (Func<TInput, bool>,
    Func<TInput, TOutput>)[] predicates)
{
    var match = predicates.FirstOrDefault(x => x.Item1(@this));
    var returnValue = match?.Item2(@this);
    return new MatchValueOrDefault<TInput, TOutput>(returnValue, @this);
}
```

```
public class MatchValueOrDefault<TInput, TOutput>
{
    private readonly TOutput value;
    private readonly TInput originalValue;

    public MatchValueOrDefault(TOutput value, TInput originalValue)
    {
        this.value = value;
        this.originalValue = originalValue;
    }

    public TOutput DefaultMatch(Func<TInput, TOutput> defaultMatch)
    {
        if (EqualityComparer<TOutput>.Default.Equals(default, this.value))
        {
            return defaultMatch(this.originalValue);
        }
        else
        {
            return this.value;
        }
    }
}
```

在这个改进的实现中：

- Match 方法：该方法仍然接收一个元组数组，每个元组包含一个条件和一个转换函数。方法尝试找到第一个匹配的元组，并尝试执行转换函数。无论结果如何（即使为 null），都将返回一个 MatchValueOrDefault 对象。
- MatchValueOrDefault 类：这个类封装了转换结果和原始输入值。它提供了一种机制，通过 DefaultMatch 方法可以在没有找到有效匹配时执行默认的转换。
- DefaultMatch 方法：这个方法检查封装的值是否等于 TOutput 类型的默认值（即 default(TOutput)），如果是，则调用提供的默认转换函数 defaultMatch。这相当于 switch 表达式的“_”模式匹配或者 if-else 语句的最后一个 else 块。

这种设计的好处在于，它增加了对未匹配结果的处理能力，避免了简单实现中可能出现的 null 返回值问题，并且提供了一个明确的方式来定义默认行为。

通过这样的实现，不仅避免了空引用异常的风险，还增加了代码的健壮性和易于维护性。这个例子演示了在设计使用模式匹配或类似功能的 API 时，考虑到所有可能的情况和边界条件的重要性。

不过，与最新版本的 C#相比，这种方法在功能上有很大的局限性。它不支持对象类型匹配，语法也没那么优雅，但它仍然能够有效地减少样板代码的数量并促进代码的规范性。

对于不支持元组的更旧的 C#版本，可以考虑使用 KeyValuePair<T,T>来实现类似的功能，但它的语法就没有那么讨喜了。什么，你不信我说的话？好吧，那就试试吧。别怪我没提醒你……

扩展方法本身的改动不大，只需稍作修改，用 KeyValuePair 代替原来的元组：

```
public static MatchValueOrDefault<TInput, TOutput> Match<TInput, TOutput>(
    this TInput @this,
    params KeyValuePair<Func<TInput, bool>, Func<TInput, TOutput>>[] predicates)
{
    var match = predicates.FirstOrDefault(x => x.Key(@this));
    var returnValue = match.Value(@this);
    return new MatchValueOrDefault<TInput, TOutput>(returnValue, @this);
}
```

接下来的代码就比较难看了。创建 KeyValuePair 对象所用的语法非常糟糕。

```
var inputValue = 25000M;
var updatedValue = inputValue.Match(
    new KeyValuePair<Func<decimal, bool>, Func<decimal, decimal>>(
        x => x <= 12570, x => x),
    new KeyValuePair<Func<decimal, bool>, Func<decimal, decimal>>(
        x => x <= 50270, x => x * 0.8M),
    new KeyValuePair<Func<decimal, bool>, Func<decimal, decimal>>(
        x => x <= 150000, x => x * 0.6M)
).DefaultMatch(x => x * 0.55M);
```

虽然可以通过这种方式在 C# 4 中实现某种形式的模式匹配，但我不确定这么做能有多少实际的好处。总之，我展示了实现它的方法，具体要不要用就由你自己来判断了。

让字典更有用

函数不仅仅可以用于转换数据的形式，还可以用作筛选器，在开发者获取原始数据或调用原始功能之前进行一些筛选。本节将探讨如何运用函数式筛选来拓展字典的用途。

字典是我最喜欢的 C#特性之一。若使用得当，它们可以用几个简单而优雅的数组式查询来取代大量冗长的样板代码。而且一旦创建，字典能非常高效地查找数据。

但字典存在一个问题，这个问题经常导致我们不得不添加大量样板代码，这违背了我们的初衷，如下例所示[44]：

```
// 神秘博士的演员
var doctorLookup = new []
{
    ( 1, "William Hartnell" ),
    ( 2, "Patrick Troughton" ),
    ( 3, "Jon Pertwee" ),
    ( 4, "Tom Baker" )
}.ToDictionary(x => x.Item1, x => x.Item2);

var fifthDoctorInfo = $"The 5th Doctor was played by {doctorLookup[5]}";
```

这段代码出了什么问题？它涉及字典在 C#中一个令人费解的特性：如果尝试查找一个不存在的条目，程序会抛出一个必须处理的异常。

[44] 译注：事实上，截止 2024 年 5 月，在所有剧集、复活剧、电影中，神秘博士的演员人数已经达到 15 名。英国演员舒提·盖特瓦是第 15 任“神秘博士”。

为了安全地应对这种情况，可以使用 C#提供的几种技术之一，在将键值转换成字符串之前先检查键是否存在于字典中。

```
var doctorLookup = new []
{
    ( 1, "William Hartnell" ),
    ( 2, "Patrick Troughton" ),
    ( 3, "Jon Pertwee" ),
    ( 4, "Tom Baker" )
}.ToDictionary(x => x.Item1, x => x.Item2);

var fifthDoctorActor = doctorLookup.ContainsKey(5)
    ? doctorLookup[5]
    : "An Unknown Actor"; // 未知演员

var fifthDoctorInfo = $"The 5th Doctor was played by {fifthDoctorActor}";
```

或者，在较新的 C#版本中，可以使用 `TryGetValue()`函数来使代码更加简洁。

```
var fifthDoctorActor = doctorLookup.TryGetValue(5, out string value)
    ? value
    : "An Unknown Actor"; // 未知演员
```

那么，是否能使用函数式编程技术来减少样板代码，在不用担心潜在问题的前提下享受字典的各种便利特性呢？当然可以！

首先需要创建一些简单的扩展方法。

```
public static class ExtensionMethods
{
    // 接收一个字典并返回一个函数。返回的函数接收一个键，并尝试在字典中查找这个键
    public static Func<TKey, TValue> ToLookup<TKey, TValue>(
        this IDictionary<TKey, TValue> @this)
    {
        return x => @this.TryGetValue(x, out TValue? value) ? value : default;
    }

    // 在字典中找不到键时，将返回默认值
    public static Func<TKey, TValue> ToLookup<TKey, TValue>(
        this IDictionary<TKey, TValue> @this, TValue defaultVal)
    {
        return x => @this.ContainsKey(x) ? @this[x] : defaultVal;
    }
}
```

稍后会详细解释上述代码，但在此之前，先来看看如何使用扩展方法。

```
var doctorLookup = new []
{
    ( 1, "William Hartnell" ),
    ( 2, "Patrick Troughton" ),
    ( 3, "Jon Pertwee" ),
    ( 4, "Tom Baker" )
}.ToDictionary(x => x.Item1, x => x.Item2)
.ToLookup("An Unknown Actor");

var fifthDoctorInfo = $"The 5th Doctor was played by {doctorLookup(5)}";
// 输出 = "The 5th Doctor was played by An Unknown Actor"
```

注意到区别了吗？仔细观察，会发现现在是用圆括号来获取字典中的值，而不是方括号。这是因为它本质上已经不再是字典了，而是变成了一个函数。

这些扩展方法返回的其实是函数，而这些函数将原始的 `Dictionary` 对象保存在其作用域内。本质上，它们充当了 `Dictionary` 和代码库其他部分之间的一个筛选层，负责判断 `Dictionary` 的使用是否安全。

如此一来，就可以放心地使用 `Dictionary`，而不必担心因为找不到键而引发异常了。我们可以返回该类型的默认值（通常是 `null`），或是自定义一个默认值。这很简单。

这个技术唯一的缺点是，`Dictionary` 实际上已经不再是“字典”了。我们不能对它做进一步的修改或者执行任何 LINQ 操作。但是，如果确定不需要进行这些操作，那么这个技术还是很实用的。

对值进行解析

将字符串解析为其他形式的值是导致代码变得冗长且充满样板代码的国一个常见原因。在.NET 环境下，由于没有 appsettings.json 或 `IOption<T>`等，所以我们可能会用以下方式解析一个设置对象：

```
public Settings GetSettings()
{
    var settings = new Settings();

    var retriesString = ConfigurationManager.AppSettings["NumberOfRetries"];
    var retriesHasValue = int.TryParse(retriesString, out var retriesInt);
    if(retriesHasValue)
        settings.NumberOfRetries = retriesInt;
    else
        settings.NumberOfRetries = 5;

    var pollingHrStr = ConfigurationManager.AppSettings["HourToStartPollingAt"];
    var pollingHourHasValue = int.TryParse(pollingHrStr, out var pollingHourInt);
    if(pollingHourHasValue)
        settings.HourToStartPollingAt = pollingHourInt;
    else
        settings.HourToStartPollingAt = 0;

    var alertEmailStr = ConfigurationManager.AppSettings["AlertEmailAddress"];
    if(string.IsNullOrWhiteSpace(alertEmailStr))
        settings.AlertEmailAddress = "test@thecompany.net";
    else
        settings.AlertEmailAddress = aea.ToString();

    var serverNameString = ConfigurationManager.AppSettings["ServerName"];
    if(string.IsNullOrWhiteSpace(serverNameString))
        settings.ServerName = "TestServer";
    else
        settings.ServerName = sn.ToString();

    return settings;
}
```

对于一个简单的任务来说，这段代码是不是太长了？大量样板代码模糊了代码的意图，只有那些特别熟悉这类操作的人才能看懂。而且，每次添加新设置时，都需要写5~6行新代码，这显然是一种浪费。

不过，可以采取更加函数式的方法来简化这个过程，将复杂的结构隐藏起来，只展示清晰易懂的代码意图。

和之前一样，这可以通过一个扩展方法来实现，如下所示：

```
public static class ExtensionMethods
{
    // 尝试将一个对象转换为整数
    public static int ToIntOrDefault(this object @this, int defaultVal = 0) =>
        int.TryParse(@this?.ToString() ?? string.Empty, out var parsedValue)
        ? parsedValue
        : defaultVal;

    // 尝试将一个对象转换为字符串
    public static string ToStringOrDefault(
        this object @this,
        string defaultVal = "") =>
        string.IsNullOrWhiteSpace(@this?.ToString() ?? string.Empty)
        ? defaultVal
        : @this.ToString();
}
```

这个方法消除了第一个示例中的重复代码，使我们能够转向可读性更高、以结果为导向的编码风格，如下所示：

```
public Settings GetSettings() =>
    new Settings
    {
        NumberOfRetries = ConfigurationManager.AppSettings["NumberOfRetries"]
            .ToIntOrDefault(5),
        HourToStartPollingAt =
            ConfigurationManager.AppSettings["HourToStartPollingAt"]
            .ToIntOrDefault(0),
        AlertEmailAddress = ConfigurationManager.AppSettings["AlertEmailAddress"]
            .ToStringOrDefault("test@thecompany.net"),
        ServerName = ConfigurationManager.AppSettings["ServerName"]
            .ToStringOrDefault("TestServer"),
    };
```

现在，一眼就能看出代码做了什么、默认值是什么，而且可以通过单行代码添加更多设置项。如需处理除 `int` 和 `string` 以外的其他设置值类型，那么只需再创建一个扩展方法，这不是什么大问题。

自定义枚举

许多人都在编程中使用过可枚举对象，但鲜为人知的是，它们背后实际上有一个强大的引擎，我们可以利用这个引擎来实现各种有趣的自定义行为。有了自定义迭代器，就可以用更少的代码在数据遍历过程中实现复杂的操作。

不过，首先需要理解可枚举对象在底层是如何运作的。可枚举对象的底层存在一个“枚举器类”（Enumerator Class），它是驱动枚举过程的引擎，使我们能使用 `foreach` 循环来遍历值。

枚举器有两个关键成员：

- `Current`：这个属性获取可枚举对象中的当前元素。只要不移动到下一项，就可以多次获取这个属性的值。但如果在首次调用 `MoveNext()`之前就尝试获取 `Current` 值，将会引发异常。
- `MoveNext()`：这个方法从当前元素移动到下一个元素，并尝试判断是否还有其他元素可以选择。如果找到下一个元素，则返回 `true`；如果已经到达可枚举对象的末尾，或者一开始就没有元素，则返回 `false`。在首次调用 `MoveNext()`时，它会将枚举器定位到可枚举对象中的第一个元素。

查询相邻元素

让我们从一个相对简单的例子开始。假设需要遍历一个由整数组成的可枚举对象，检查它是否包含任何连续的数字。命令式解决方案可能是这样的：

```
// 生成由随机数构成的一个列表
public IEnumerable<int> GenerateRandomNumbers()
{
    var rnd = new Random();
    var returnValue = new List<int>();
    for (var i = 0; i < 100; i++)
    {
        returnValue.Add(rnd.Next(1, 100));
    }
    return returnValue;
}

// 判断是否包含连续的数字
public bool ContainsConsecutiveNumbers(IEnumerable<int> data)
{
    // 好吧，被你发现了，OrderBy 严格来说不属于命令式编程，
    // 但我不想花大功夫编写排序算法！
    var sortedData = data.OrderBy(x => x).ToArray();

    for (var i = 0; i < sortedData.Length - 1; i++)
    {
        if ((sortedData[i] + 1) == sortedData[i + 1])
            return true;
    }

    return false;
}

var result = ContainsConsecutiveNumbers(GenerateRandomNumbers());
Console.WriteLine(result);
```

和之前一样，为了将上述代码转换为函数式风格，需要创建一个扩展方法。这个方法将接受可枚举对象，提取它的枚举器，并通过枚举器来控制自定义行为。

为了避免使用命令式风格的循环，我们将使用递归。递归（详见第 1 章和第 2 章）是一种通过让函数反复调用自身来实现不定循环的方法[45]。

第 9 章将进一步讨论递归的概念。现在，让我们先使用简单形式的递归。

```
public static bool Any<T>(this IEnumerable<T> @this, Func<T, T, bool> evaluator)
{
    using var enumerator = @this.GetEnumerator();
    var hasElements = enumerator.MoveNext();
    return hasElements && Any(enumerator, evaluator, enumerator.Current);
}

private static bool Any<T>(IEnumerator<T> enumerator,
        Func<T, T, bool> evaluator,
        T previousElement)
{
    var moreItems = enumerator.MoveNext();
    return moreItems && (evaluator(previousElement, enumerator.Current)
       ? true
       : Any(enumerator, evaluator, enumerator.Current));
}
```

那么，这段代码具体是如何工作的呢？从某种意义上来讲，这种方法有点像杂耍。首先，我们提取出枚举器，并将其定位到序列的第一个元素。

私有函数接受三个参数：枚举器（目前已指向第一个元素）、判断是否已经完成的 `evaluator` 函数以及第一个元素的副本。

然后，我们立即移动到下一个元素并执行 `evaluator` 函数，传入第一个元素和新的 `Current` 以进行比较。

这个阶段有两种可能的情况：一种是发现序列中的元素已经全部遍历完毕，另一种是评估函数返回了 `true`，因此当前迭代就可以结束了。如果 `MoveNext()`返回 `true`，我们就会检查 `previousValue` 和 `Current` 是否符合条件（由 `evaluator` 定义）。

如果这两个元素符合条件，就结束并返回 `true`；如果不符合，就进行递归调用，继续检查序列中的剩余元素。

以下是查找连续数字的更新版代码：

```
public IEnumerable<int> GenerateRandomNumbers()
{
    var rnd = new Random();
    var returnValue = Enumerable.Repeat(0, 100)
        .Select(x => rnd.Next(1, 100));
    return returnValue;
}

public bool ContainsConsecutiveNumbers(IEnumerable<int> data)
{
    var sortedData = data.OrderBy(x => x).ToArray();
```

[45] 虽然是“不定”循环，但最好不要真的“无限”循环下去！（译注：之前说过，indefinite 和 infinite 是两个意思。）

```
    var result = sortedData.Any((prev, curr) => curr == prev + 1);
    return result;
}
```

根据这个逻辑创建一个 `All()`方法也很简单，如下所示：

```
public static bool All<T>(
    this IEnumerator<T> enumerator,
    Func<T,T,bool> evaluator,
    T previousElement)
{
    var moreItems = enumerator.MoveNext();
    return moreItems
        ? evaluator(previousElement, enumerator.Current)
            ? All(enumerator, evaluator, enumerator.Current)
            : false
        : true;
}

public static bool All<T>(this IEnumerable<T> @this, Func<T,T,bool> evaluator)
{
    using var enumerator = @this.GetEnumerator();
    var hasElements = enumerator.MoveNext();
    return hasElements
        ? All(enumerator, evaluator, enumerator.Current)
        : true;
}
```

`All()`和 `Any()`唯一的区别在于，它们根据不同的条件来决定是继续迭代还是提前返回结果。`All()`会检查每一对值，只有在发现某对值不满足给定条件时才会提前退出循环。

在满足条件前持续迭代

本节所介绍的技巧本质上是 `while` 循环的替代方案，在掌握这种技巧后，就可以让代码符合函数式编程的风格，再也不使用 `while` 语句了。。

举个例子，让我们想象一下文字冒险游戏的回合制系统可能会是什么样子。这对年轻的读者来说可能有些陌生，但在图形界面出现之前，这类游戏非常普遍。在这种游戏中，你需要输入想执行的动作，然后游戏就会回应你的指令并显示结果——这有点像是一本小说，只不过故事情节的发展由玩家来决定。

 想要亲身体验这种游戏风格的话，不妨尝试一下史诗级冒险游戏《魔域》（Zork）。但要小心，别被格鲁吃了！在线游玩：https://tinyurl.com/nhcfwuzn。

这类游戏的基本结构是这样的：

1. 描述玩家当前所在的位置。
2. 接收玩家的输入。
3. 根据玩家的输入执行相应的命令。

实现这种结构的命令式代码如下所示：

```
var gameState = new State
{
    IsAlive = true,  // 是否存活
    HitPoints = 100  // 生命值
};

while(gameState.IsAlive)
{
    var message = this.ComposeMessageToUser(gameState);
    var userInput = this.InteractWithUser(message);
    this.UpdateState(gameState, userInput);
    if(gameState.HitPoints <= 0)
        gameState.IsAlive = false;
}
```

本质上，我们想要的是一个 LINQ 风格的 `Aggregate()`函数，但它不会在遍历数组中所有元素后就结束。我们希望这个函数能持续运行，直到满足结束条件（玩家死亡）。这里我做了一些简化（显然，在真正的游戏中，还有一个结束条件是玩家取胜）。但这个小游戏就像是人生，而人生是不公平的！

在实现这个扩展方法时，可以利用尾递归优化来提升效率，我将在第 9 章中进一步讨论。现在，为了避免过早引入太多复杂的概念，下面只使用简单的递归（如果游戏有很多回合，这可能会成为问题）：

```
public static class ExtensionMethods
{
    public static T AggregateUntil<T>(
        this T @this,
        Func<T,bool> endCondition,
        Func<T,T> update) =>
         endCondition(@this)
        ? @this
        : AggregateUntil(update(@this), endCondition, update);
}
```

通过这种方式，可以完全不使用 `while` 循环，而是将整个回合过程转化为单一函数，如下所示：

```
var gameState = new State
{
    IsAlive = true,
    HitPoints = 100
};

var endState = gameState.AggregateUntil(
    x => x.HitPoints <= 0,
    x => {
```

```
            var message = this.ComposeMessageToUser(x);
            var userInput = this.InteractWithUser(message);
            return this.UpdateState(x, userInput);
        });
```

虽然这种方式未臻完美，但它符合函数式编程规范。第 13 章将讨论如何以更好的方法处理游戏状态更新的多个步骤，以及如何在函数式编程范式下处理用户交互的问题。

小结

本章探讨了如何使用 Func 委托、可枚举对象和扩展方法来扩展 C#的功能，使得编写符合函数式编程风格代码变得更加简单，并绕开 C#中的一些限制。我相信它们只是冰山一角，还有更多有待挖掘和运用的方法。

下一章将深入讨论高阶函数以及一些能利用这些函数创造更多实用功能性的结构。

第 II 部分 深入函数式编程的腹地

欢迎，勇敢的冒险者！

你已经安全地度过了第一段旅程，现在，你已经从一名函数式学徒晋升为熟练工了[46]。接下来要踏上的是一段更长、更曲折、充满未知的旅途。

但是，不要害怕，只要保持开放的心态，就会发现前方充满了无限的可能性。在这一部分，我们将暂时放下传统的 C#编程思维，深入探讨函数式理论。

这里不会有正式的定义或关于列表理论的详细讨论，因为本书始终着眼于对日常编码有实际帮助的内容。我将引导你一步步走上这个平缓的学习曲线。

启程吧，马儿们已经迫不及待了。是时候翻身上马，继续我们的探险了！

[46] “熟练工”（Journeyman）一词已经有些年代了，但我觉得它非常适合用在这里。你可以根据自己的需求，随意将“man”替换为合适的称谓。

第 14 章 总结

各位看官，书写完了！庆功宴时间到！来来来，大家都动起来！舞女站这边，舞男站那边，然后……

等等，我刚收到奥莱利[110]先生的短信。咱们没钱搞百老汇式的豪华庆功宴了。

那咱们有什么？好吧，就剩我的竖笛了。人人都爱竖笛，是吧？来，一、二……

哎呀，不好意思各位。我刚想起来，上次我吹竖笛，家里的牛奶全都不见了。所以，恐怕只能听我随便讲几句，然后大家就各回各家吧。生活就是这样啊……

读到这里的感受如何？

玩笑到此为止，我真心希望你享受阅读这本书的过程。它耗费了我不少时间，如你所见，我在这本书中倾注了很多个人情感。

本书采用了一种轻松诙谐的文风，这是出于几个原因。首先，市面上已经有许多枯燥（但仍然很优秀）的计算机书籍了，我相信市场可以接纳一本与众不同的书。[111]其次，在函数式编程领域，有许多网站、文章和书籍都专注于讲述正式定义，连我读起来都觉得很艰深晦涩，因此我想展示另一种介绍函数式编程的方式。

在撰写本书的过程中，我努力从初学者的角度出发，逐步深入到一些 F#开发者可能感到熟悉的概念。希望你在这个过程没有遇到太多挫折，并感受到了一些乐趣！

C#是一种混合式编程语言，从前如此，今后亦然。它不太可能支持纯函数式代码的编写。这不是抱怨，这只是一个事实。

如果你对函数式编程的探索之旅意犹未尽，想要在本书的基础上继续前行，那么有几条不错的学习路径可以选择。现在，就让我为你一一介绍这些路径。

接下来应该去向何方？

可以选择多条路径来继续你的旅程。我会大致按照你对这些内容的熟悉程度来为它们排序，从你可能已经有所了解的部分开始，到需要更深入学习的主题。

---

[110] 译注：O'Reilly，即本书英文版的出版社。

[111] 值得一提的是，还有一本名为《Mr. Bunny's Big Cup o' Java》的书，作者是卡尔顿 • 埃格蒙特三世（Carlton Egremont III），由艾迪生韦斯利出版社出版。据可靠消息称，这可能是有史以来最幽默的计算机书籍。我有时候真希望自己是一名 Java 开发者，那样我就能尽情享受阅读这本书的乐趣了！

更多的函数式 C#

请容我再次推荐恩里科·布奥南诺（Enrico Buonanno）的著作《C#函数式编程：编写更优质的 C#代码》（Functional Programming in C#）（Manning 出版社）。这是我读过的最好的编程书之一。如果你跟随本书的脚步踏入了函数式编程的大门，那么布奥南诺的这本书将是进一步学习的绝佳选择。

布奥南诺比我更深入地探讨了函数式理论，还涵盖了一些我没有介绍的内容。与我的书相比，这本书就像是一瓶上好的葡萄酒——值得慢慢品鉴，细细回味。

然而，要想在函数式编程上更进一步，最有价值的做法是不断地实践。学习的本质在于反复练习，直到能自如地运用它。成为一个更优秀的函数式程序员的最佳途径是减少对理论的关注，更多地把精力放在写代码上。

正如本书所展示的那样，不需要完全采用函数式编程范式。按照自己的喜好，从小处着手，逐步深入。偶尔看看一些优秀的书籍，以寻找尝试新技术的灵感。

不过，我建议你与团队讨论一下函数式编程的话题。本书前几章的内容不太可能引起争议，但如果你突然开始在代码中使用单子，那么可能会让团队成员感到措手不及。当然，这取决于你的团队的接受程度。事先进行沟通总是好的，这至少可以避免在代码库中全面采用函数式编程风格后，才有团队成员提出反对的尴尬局面，从而省去了漫长、成本高昂且枯燥的代码重写过程。

深入探索了函数式 C#，并通过大量实践成为了该领域的专家之后，就可以考虑学习一种新的编程语言了。

学习 F#

对于函数式 C#开发者来说，学习 F#是一个自然而然的选择。它是一种.NET 语言，并且可以轻松地与 C#互操作，这意味着可以根据项目需求，在解决方案中灵活地混合使用这两种语言的代码。

有许多优质的学习资源可以帮助你学习 F#。我推荐从斯科特·瓦拉欣（Scott Wlaschin）的杰出教学网站“F# For Fun and Profit”（https://oreil.ly/r8uvQ）开始你的学习之旅。

由伊恩·罗素（Ian Russell）撰写的电子书《Essential F#》（https://oreil.ly/hxY9w）是免费的，你可以根据个人意愿决定是否赞赏作者。伊恩在校对和验证本书内容的准确性方面做出了巨大贡献，我对此深表感谢。如果你有机会遇见他，请代我向他问好。

我也很喜欢萨克·亚伯拉罕（Isaac Abraham）的《F# in Action》（Manning 出版社），这本书读起来非常轻松愉快。

纯函数式编程语言

如果已经掌握了 F#并且想要进一步探索，可以考虑学习一些更纯粹的函数式编程语言。需要注意的是，这些语言与.NET 框架之间无法实现互操作，因此它们在日常的.NET 工作中的实用性可能有限。

如果选择学习像 Elm 或 Haskell 这样的语言，你可能会追求以下目标：

- 了解更纯粹的函数式编程范式，以便带着全新的编程思维回归到.NET 开发中。
- 离开.NET 领域，也可以考虑开启一条新的职业道路。这很可能意味着你 加入新的组织，除非你当前的组织非常开明。
- 纯粹是出于对学习的热爱和对新知识的好奇心。学习本身就是一种乐趣。这也是为什么我在从大学毕业二十年后，仍然留在这个行业。

我将把决定权留给你，但就个人而言，我对学习新的语言并没有太大兴趣，也无法为你推荐特定的语言。

如果你正在寻找学习资源，米兰·利波瓦察（Miran Lipovača）的个人网站和她撰写的《HASKELL 趣学指南》（Learn You a Haskell for Great Good!）（No Starch 出版社)（https://oreil.ly/8V8CS）可能是开启你的新旅程的最佳起点。多年以来，有许多人都向我推荐过这本书。

那你呢？

我？嗯，我这个夜幕下的表演者也该继续我的旅程了——带着我的木偶和戏法，一路向前。如果你想继续保持联系，欢迎访问我的个人网站（http://www.thecodepainter.co.uk），或者在各种软件开发大会和聚会中找到我（虽然我主要在欧洲活动，但有时也会去其他地方）。如果你碰巧遇到我，请不要默默走开。尽管过来打个招呼，我可能还会请你喝上一两轮啤酒呢！

看了看时间，我们应该还有时间做最后一个戏法。

注意，我这只袖子里空无一物。是真的，不信自己看。等等，你凑得也太近了！真是调皮。另一只袖子里也没有东西。但是，我即将在你的眼前凭空消失。

请看仔细了，我要开始了。

现在你能看到我，但……[112]

[112] 译注：“Now you see me（现在你能看到我）”是魔术表演中常见的台词，后面往往跟着一句“Now You Don't（现在你看不到我了）”。值得一提的是，电影《惊天魔盗团》的英文名就是“Now you see me”。

关于作者

西蒙·J·佩恩特（Simon J. Painter）自 2005 年以来一直深耕与软件开发领域，他使用过.NET 的每一个版本（甚至包括 Compact Framework——还记得它吗？）并在许多不同的行业工作过。在日常工作之余，他热衷于参加用户小组和会议，并经常发表有关函数式编程和.NET 的演讲。自从读懂了他父亲的 Sinclair ZX Spectrum BASIC 手册开始，西蒙就成为了编程爱好者。除了编程以外，他还喜欢演奏音乐、玩填字游戏、玩《战斗幻想》（Fighting Fantasy）游戏书以及喝大量的咖啡，尽管这可能对健康不太好。他目前和妻女住在英国的一个小镇上。

封面说明

《C#函数式编程》封面上的动物是东部郊狼（Canis latrans var.），也被称为“coywolf”。

东部郊狼是居住在美洲的 19 种郊狼亚种之一，实际上是东部狼（C. lycaon）、郊狼（Canis latrans）和家犬的杂交种，因此它的体型比西部郊狼更大，平均体重在 45~55 磅之间。东部郊狼的领地范围更为广阔，遍布美国东部和加拿大的大部分地区，从东海岸的纽芬兰和拉布拉多地区一直延伸到南部的乔治亚州。

作为一种机会主义的杂食性动物，东部郊狼以可获得的任何食物为食，范围从蚱蜢到驼鹿不等。它们通常以小家庭（由一对成年郊狼和幼崽组成）的形式生活和狩猎，不过如果有幸在夜间听到它们嚎叫，你很可能会以为它们是群体狩猎者，就像它们的狼亲戚那样：在必要时，东部郊狼能够制造出相当喧闹的声音（https://oreil.ly/IGHUv）！

尽管全球人口在不断增长，但郊狼目前并不属于濒危物种，至少从生态保护的角度来看是这样的。不过，出现在 O'Reilly 书籍封面上的许多动物都是濒危物种；每一个物种对这个世界的生态平衡都极为重要。

本书的封面图片由凯伦·蒙哥马利（ Karen Montgomery ）绘制，灵感来源于理查德·莱德克（Richard Lydekker）的《皇家自然史》（The Royal Natural History）中的一幅经典线雕画。

现在你看不到我了。

你怎么还在这里？这本书已经结束了。你的家人可能正在想念你呢。花些时间陪陪他们吧！

快去吧，别磨蹭了！