Broadview
www.broadview.com.cn

MANNING

如何用函数式编程技术改进Java程序

Java函数式编程

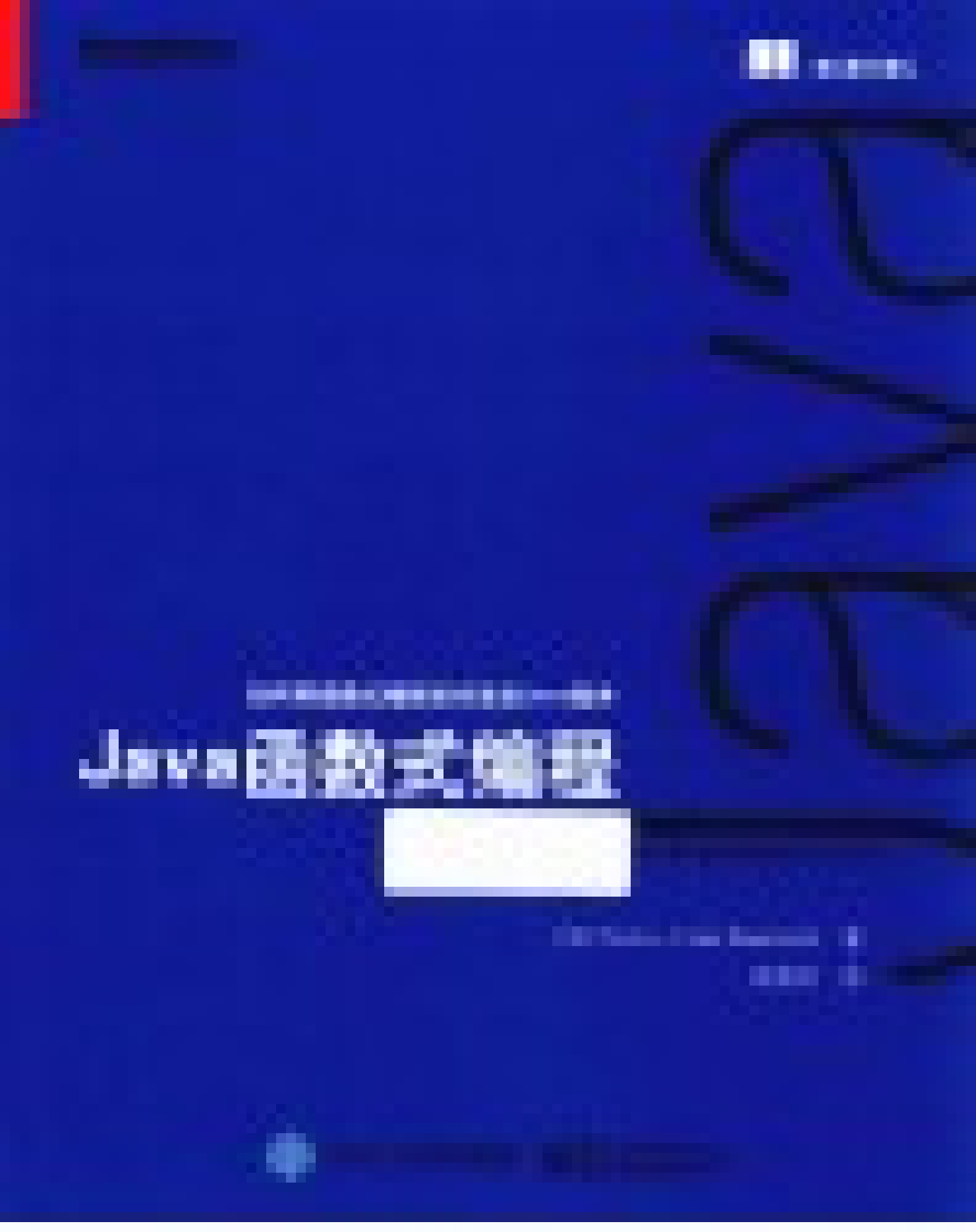
al Programming in Java

[法] Pierre-Yves Saumont 著

高清华 译

中国工信出版集团

電子工業出版社



Java函数式编程

onal Programming in Java

[法] Pierre-Yves Saumont 著

高清华 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

《Java函数式编程》并不是一本关于Java的书，而是一本关于函数式编程的书。作者由浅入深地介绍了函数式编程的思维方式，并引导读者通过易于掌握的例子、练习和图表来学习和巩固函数式编程的基本原则和最佳实践。读者甚至可以在阅读的同时编写出自己的函数式类库！

本书非常适合对Java有所了解的程序员，无须任何基础的数学理论或是函数式编程经验即可快速上手！

版权贸易合同登记号　图字：01-2017-2241

图书在版编目（CIP）数据

Java函数式编程/（法）皮埃尔-伊夫斯·索蒙特（Pierre-Yves Saumont）著；高清华译. —北京：电子工业出版社，2018.1
书名原文：Functional Programming in Java
ISBN 978-7-121-33021-6

Ⅰ. ①J… Ⅱ. ①皮… ②高… Ⅲ. ①JAVA语言—程序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字（2017）第277090号

策划编辑：张春雨
责任编辑：刘　舫
印　　刷：北京京科印刷有限公司
装　　订：北京京科印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　　邮编：100036
开　　本：787×980　1/16　　印张：32.25　　字数：578千字
版　　次：2018年1月第1版
印　　次：2018年1月第1次印刷
定　　价：119.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819　faq@phei.com.cn。

译者序

有幸受邀翻译本书。初见书名，心中不免有几分疑虑，难道又是一本教授怎么使用 Java 8 lambda 来进行函数式编程的书吗？翻了几页，方觉自己大误。本书其实意在如何从零开始，逐步理清函数式编程的思维方式并编写基础类库，不仅授之以鱼，而且授之以渔。只不过由于 Java 的受众实在太广，所以才使用这门语言罢了。

函数式编程有一个至关重要的前提，那就是函数的输出只能取决于函数的参数（我们会在书中看到生成随机数的例子）。初看上去似乎与 Java 这门面向对象的语言不搭，但语言只是工具而已，正如你也可以在 Haskell 中编写命令式风格的代码一样。在一个不太复杂甚至非并发的常规 Java 系统中，由于程序内部状态的改变，多次调用同一个方法的返回值很可能是不一样的，更不用说所带来的副作用了。在函数式编程中，确定的输入决定了确定的输出，这意味着只要参数对了，结果一定在预期中。也就是说，函数式编程没有无法重现的 bug。在这样的前提下，单元测试相对容易实现，而且能极大地增强你的信心。（想想你对目前所在项目的单元测试有多大的信心？）许多个这样的函数复合起来，在不改变信心的同时能够提供更多、更强大的功能，进而带来更大的收益，如无状态的线程安全、必要时才计算的惰性求值、加快多次执行速度的记忆化等。

传统的命令式编程是计算机硬件的抽象，源自图灵机，其实就是外部输入、内部状态、对外部的输出以及对内部状态的改变。函数式编程源自 λ 演算，即将变

量和函数替换为值或表达式并根据运算符进行计算。函数式编程相比命令式编程，代码更简洁、可读性更强，这是因为它的思维方式更倾向于描述什么，而不是怎么做。所以学习过程反而更加自然，并且不需要多么高深的数学基础。可是我们也知道，软件开发没有银弹。新的方法论也会带来新的问题，需要运用新知识来解决。幸运的是，新知识的坑已经有人帮你踩过了，高阶函数、偏应用函数、复合函数、柯里化、闭包……软件开发从来不缺术语。幸好它们并非高不可攀，作者在第 2 章中帮你扫清疑虑，并在后续章节中挑战惰性求值、记忆化、状态处理、应用作用还有 actor 等更高级的技术。你说 Monad？作者才不告诉你它究竟是什么，但是看完本书你自然就领悟了。

函数式编程不是万能药。它有自己擅长的领域，也有自己的弱项。函数式编程是级别更高的抽象。高级别抽象带来的收益就是易读、好写，可是有些低级别的事情（如果你真的需要的话）可能就不容易完成。函数式编程没有副作用，导致无法完成输入 / 输出操作。尽管如此，你也会在本书中看到一些解决办法。函数式编程没有变量，因此无法改变循环的终止条件，故而没有循环，严重依赖于用递归来抽象循环。在某些情况下可能会影响性能，所以你还会在本书中看到一些性能与情怀之间的权衡。绝大部分的编程最佳实践都是针对某个特定场景而言的，因此脱离业务场景直接讨论技术并不可取。拥有函数式编程的思维，你就拥有了解决问题的另一种选择，但是条条大路通罗马，千万别钻牛角尖。程序是对现实世界的建模，“不要让世界适应你的模型，要让你的模型适应世界。”

高清华

译者简介

高清华

亚马逊软件研发工程师。工作十多年来，在简洁代码、自动化测试、持续交付、DevOps 等方面都有着丰富的经验。他是《DevOps 实践》译者之一。其技术博客的网址是 http://qinghua.github.io/，希望能以通俗易懂的语言普及 IT 技术。

为什么要函数式编程

函数式编程中没有赋值语句，因此变量一旦有了值，就不会再改变了。更通俗地说，函数式编程完全没有副作用。除了计算结果，调用函数没有其他作用。这样便消除了 bug 的一个主要来源，也使得执行顺序变得无关紧要——因为没有能够改变表达式值的副作用，可以在任何时候对它求值。这样便把程序员从处理控制流程的负担中解脱出来。由于能够在任何时候对表达式求值，所以可以用变量的值来自由替换表达式，反之亦然——程序是“引用透明”的。这样的自由度让函数式的程序比它们的一般对手在数学上更易驾驭。

—John Hughes
Why Functional Programming Matters

我称之为十亿美元的错误……我当时的目标是确保所有引用的使用都应该绝对安全，由编译器自动执行检查。但是我无法抵制引入一个空引用的诱惑，仅仅是因为它很容易实现。这导致了无数的错误、漏洞和系统崩溃，它很可能在过去的四十年中造成了十亿美元的痛苦和损害。

—Tony Hoare

测试程序可以是一个证明 bug 存在的有效方式，但是对于证明 bug 不存在还是无可奈何。

—Edsger W. Dijkstra

测试本身并不能提高软件质量。测试结果是质量的指标，但它们并不能提高质量。尝试通过增加测试来提高软件质量就像是尝试通过经常称自己的体重来减肥一样。

—Steve McConnell

注释的合理运用是为了补偿我们无法成功地在代码中表达自己。

—Robert C. Martin

编程的难点并不是解决问题，而是决定要解决什么问题。

—Paul Graham

面向对象编程通过封装变化的部分使代码容易理解。函数式编程通过最小化变化的部分使代码容易理解。

—Michael Feathers

我总是发现计划没有用，但是制订计划不可或缺。

—Dwight D. Eisenhower

设计软件有两种方法：一种是简单化，使其明显没有缺陷；另一种是复杂化，使其没有明显缺陷。前者要难得多。

—Tony Hoare

如果我们询问客户他们想要什么，他们只会说："更快的马。"

—Henry Ford

尽管一些声明式程序员对等式推理（equational reasoning）只会耍嘴皮子，但函数式语言的用户每次运行编译器时都会用到它们，无论他们是否注意到。

—Philip Wadler
How to declare an imperative

我们并不试图争取 Lisp 程序员；我们在追赶 C++ 程序员。我们设法在他们到达 Lisp 的半路上拽走他们。

—Guy Steele

人们"得到"类型，一直用着它们。告诉某人他不能用香蕉来敲钉子并不令其感到惊讶。

—Unknown

TDD 代替了类型检查器……正如烈酒代替了悲伤。

—byorgey

首先，调试的难度是编写代码的两倍。因此，即使你把代码写得精彩绝伦，根据定义，你还不足以聪明地调试代码。

—Brian W. Kernighan 和 P. J. Plauger

一旦开始编程，令我们感到惊讶的是，很难让程序按预想的方案正确地工作。不得不靠调试来弄明白。我还记得当意识到我此生的大部分时间都会用于寻找自己程序中的错误时的那一刻。

—Maurice Wilkes (1949)

序言

编程序既有趣又多金。许多人为了乐趣而编程，还能赚钱。从这种意义上说，程序员有点像演员、音乐家或职业足球运动员。似乎是一个梦想，直到作为程序员的你开始负起真正的责任。从这个角度上说，编写游戏或办公应用程序都没有什么大不了的。如果应用程序有一个 bug，你只修复它并发布一个新版本即可。但如果你编写的程序被人们依赖，并且还不能简单地发布一个新版本后让用户们自行安装，那就是另一种情况了。当然，Java 并非用于编写监控核电站或控制飞机的应用程序，或者是一个简单的 bug 就可能会置人类生命于风险中的系统。但如果你的程序用于管理互联网骨干，你一定不会愿意在奥运会开幕前一天发现一个讨厌的 bug，导致整个国家的电视传输失败。对于这样的程序，你希望确保它可以被证明是正确的。

大多数命令式程序无法被证明是正确的。测试只允许我们在测试失败时证明程序不正确。成功的测试说明不了什么问题。你无法证明发布的程序是不正确的。就单线程程序而言，大量的测试也许能够说明你的代码大部分是正确的。但是对于多线程应用程序，条件组合的数量使测试成为不可能。显然，我们需要另一种不同的方式来编写程序。在理想情况下，这种方法将允许我们证明程序是正确的。因为这一般不是完全可能的，所以一个很好的折中是明确分离程序中可以被证明为正确的部分和不能证明为正确的部分。这就是函数式编程技术所能提供的。

函数式编程的定义与函数式程序员一样多。有人说函数式编程是用函数编程。

这是对的，但它无助于你了解这种编程范式的优势。更重要的是，函数式编程需要将抽象推至极致的理念。它允许明确分离程序中可以被证明为正确的部分与输出取决于外部条件的其他部分。通过这种方式，函数式程序是不太容易产生 bug 的程序，它的 bug 只会驻留在特定的受限区域。

可以采用许多种技术来实现这一目标。使用不可变数据虽然不仅限于函数式编程，但却是这样一种技术。如果数据不能更改，你将不会有任何（不良的）意外，数据不会过期或损坏，没有竞争条件，无须锁住并发访问，也不会有死锁的风险。可以毫无风险地共享不可变数据。你不需要生成防御性副本，那就没有了忘记这样做的风险。另一种技术是抽象控制结构，因此你不必冒着混淆循环索引和退出条件的风险一次又一次地编写相同的结构。完全删除使用 null 引用（无论是隐式还是显式）将把你从臭名昭著的 NPE（空指针异常，NullPointerException）中解放出来。通过所有这些技术（还有更多），你可以确信：如果程序通过编译，那它就是正确的（也就是说，它的实现没有 bug）。虽然这样做并不能消除所有可能的 bug，但它更加安全。

计算机从一开始就基于寄存器中的可变值使用了命令式范式。正如其他许多被称为“命令式语言”的编程语言一样，Java 似乎在很大程度上依赖于这种范式，但根本没有必要。如果你是有经验的 Java 程序员，你可能会惊讶地发现无须更改变量的值就可以编写有用的程序。这并非函数式编程的必要条件，但是函数式程序员几乎总是自由自在地用着不可变数据。你可能也难以相信，可以在不使用 `if…else` 结构、`while` 还有 `for` 循环的情况下编写程序。再说一次，避免这样的结构并不是使用函数式范式的一个条件，但是如果你愿意，就可以避免这种结构，从而得到更安全的程序。所以即使 Java 通常被视为“命令式语言”，但它其实不是。没有命令式语言，也没有函数式语言。相信它们的存在就像是认为英语更适用于商业文档，而意大利语更适用于歌剧，法语更适用于情诗，德语更适用于哲学（或你能想象到的任意组合）。差异可能存在，但它们大多是文化方面的，编程语言也是如此。Java 是一门命令式语言，这是因为大多数 Java 程序员是命令式程序员，而 Java 的文化也是命令式的。与此相反，Haskell 程序通常以函数式风格编写，因为程序员们都想要选择这门语言来进行函数式编程。但是可以在 Haskell 中编写命令式程序，也可以用 Java 编写函数式程序。不同之处在于，Haskell 比 Java 更加“对函数式友好”。

所以问题是：“你应该用 Java 来进行函数式编程吗？”令人惊讶的是（在给定本书主题的情况下），答案为否。要是可以自由地选择任何语言，我会建议你不应该为

这个目的而选择 Java。但是你通常没有这样的自由。在撰写关于使用 Java 进行函数式编程的文章时，我收到的大多数负面评论都是“你不应该为此使用 Java。这样用不是 Java 的本意。”或者“为什么要用 Java，用 Haskell、Scala 或其他什么不是更好？”

在现实中，你通常无法选择语言。如果你在一个公司工作，可能必须使用公司的语言，或者至少是你所在的团队为你正在工作的项目选择的语言。从这个角度上看，Haskell 从来都不是一个可选项。通常，除了 Java，你别无选择。如果你处于选择语言的位置，除了使用自己熟悉的语言或是使用允许重用一些旧代码或适合环境或是其他一些条件的语言之外，可能也别无选择。本书面向的正是你，除了使用 Java 之外别无选择的 Java 程序员，尽管你希望能受益于函数式编程的安全性。

在 Java 中使用函数式编程技术往往会导致你反对所谓的“最佳实践”。实际上，这些实践中有许多都没有用，有些还确实是非常糟糕的做法。从不捕获错误就是其中之一。作为 Java 程序员，你可能已经了解到不应该捕获 OOME（内存不足的错误，Out Of Memory Error）或是你无法处理的其他类型错误。也许你甚至知道不应该捕获 NPE（空指针异常，NullPointerExceptions），因为它们表示有 bug，你应该让应用程序崩溃并修复它。不幸的是，OOME 和 NPE 都不会使应用程序崩溃。它们只会使所在的线程崩溃，并留下处于某种不确定状态的应用程序。即使它们发生在主线程中，如果某些非后台线程正在运行，它们也可能无法使应用程序崩溃。当所有的应用程序都是单线程时，这种“最佳实践”是正确的。而现在它是一个非常糟糕的实践。你应该捕获所有的异常，尽管可能不在一个 `try...catch` 块中。函数式编程的真言是“始终捕获，绝不抛出”。

在我们的函数式编程之旅中还将挑战其他许多最佳实践。其中一个是：“不要重复发明轮子”，尽管它与 Java 或命令式编程没有直接关系。想想看。某次，有人发明了轮子。那时候，大概是用刚性材料制造并随轴转动的圆。从那时起，轮子已经被重复发明许多次了。如果没有，那就不会有汽车，不会有火车，几乎没有什么东西会使用轮子。所以你应该继续努力一次次地重复发明轮子。不仅在将来会给我们带来更好的轮子，而且它具有挑战性，有回报，还有乐趣。（如果你认为当今的汽车轮子是圆形的，那么最好再思考一下，没有车能够开在圆形的轮子上！）[1]

1 车轮与地面接触的部分是平的。作者以此为例，意在表达我们认为理所当然的事情并不一定是正确的。——译者注

致谢

我要感谢许多参与制作本书的人。

首先，非常感谢我的策划编辑 Marina Michaels。非常荣幸与你一起工作，此外，还有你在手稿上的出色工作。

也非常感谢我的技术编辑 Mark Elston 和我的技术审校 Alessandro Campeis，你们帮我把这本书制作得比我自己弄要好得多。

还要谢谢所有的评论者、MEAP 的读者以及其他所有提供反馈和评论的人！如果没有你们的帮助，这本书不会是今天这个样子。特别的，我要感谢所有花时间审校和评论这本书的人：Aditya Kumar、Al Krinker、Andy Kirsch、Andy Knight、Anthony Moralez、Arun Allamsetty、Barry Kern、Boris Vasile、Bruce Hernandez、Charles Feduke、Chris Kirk、David Drummond、Davide Fiorentino lo Regio、Erwin van Eijk、Gualtiero Testa、Ivan Milosavljević、Jan Vorwerk、Jérôme Baton、Joshua McAdams、Julian Templeman、Maria Gemini、Norbert Kuchenmeister、Philippe Charrière、Piotr Bzdyl、Rambabu Posa、Sebastian Hähnel、Sebastian Metzger、Simeon Leyzerzon、Tarin Gamberini、Ursin Stauss、William Wheeler、Zach Schwartz 和 Zorodzayi Mukuya。

关于本书

这不是一本关于 Java 的书。这本书是关于函数式编程的，它是讲解编写软件程序的不同方法的。“不同”意味着它与称为“命令式范式”的编写软件的“传统”方式不同。本书涉及应用函数式范式进行 Java 编程。

没有像“函数式语言”这样的东西，只有或多或少对函数式友好的语言。虽然我在本书中使用 Java，但你可以将我所教的所有原则应用于任何其他语言。只是实现这些原则的方式有所不同。你可以用任何语言编写函数式程序，甚至是那些据说根本不是函数式的语言；同样，你也可以使用对函数式最友好的语言来编写命令式程序。

随着 Java 8 的发布，一些函数式特性已被添加到了 Java 语言中。但是正如本书不是关于 Java 的，它也不是关于这些具体的 Java 8 特性的。在这本书中，我大量地使用了某些特性，并几乎忽略了其他特性。如果你的目标是学习如何使用 Java 8 的函数式特性，本书并不合适。Urma、Fusco 和 Mycroft 编写的 *Java 8 in Action*（Manning，2014）会是一个更好的选择。

另一方面，如果你想了解什么是函数式编程，如何构建函数式数据结构，以及函数式编程范式如何帮助你编写更好的程序（有时使用 Java 8 的特性，有时又避免使用它们），那么这就是为你量身定做的书。

本书面向的读者

本书面向具有一些 Java 编程经验的读者，需要对 Java 泛型有较好的理解。如果你发现自己不了解 Java 的构造（例如实现为方法的泛型常量或者是参数化方法调用），请不必担心：我将解释它们的含义以及为什么需要它们。

你不必具备函数式编程的经验，或是了解基础的数学理论。第 2 章将会让你回忆起什么是一个函数。不会再用其他的数学知识了。

我介绍的所有函数式技术都与它们相应的命令式相关，所以我希望你具有在 Java 中使用命令式编程的经验。

如何使用本书

希望读者能顺序阅读本书，因为每一章都建立在前文概念的基础之上。第 14 章和第 15 章是仅有的例外，不会使用你在第 12 章和第 13 章中学习的内容。也就是说，如果你愿意，可以跳过第 12 章和第 13 章；它们提供了更先进的技术，理解了会很有用，但是你可能不会在自己的程序中使用。

我用了“阅读”这个词，但这本书并不只是为了阅读而生，它只有极少部分的理论。为了充分利用这本书，请在电脑前阅读，随时动手做练习。每一章都包含了一些练习，并提供了必要的说明和提示，以帮助你找到答案。所有的代码都可以从 GitHub（http://github.com/fpinjava/fpinjava）和出版商的网站 https://www.manning.com/books/functional-programming-in-java 上免费下载。每个练习都附带一个推荐答案和 JUnit 测试，你可以用它来验证你的答案是否正确。

为了导入 IntelliJ（推荐）、NetBeans 或 Eclipse 中，代码附带了所有必需的要素，虽然在撰写本文时，Eclipse（Mars 4.5.1）尚未完全兼容 Java 8。可以通过源代码或者使用 Gradle 导入项目。Gradle 的任何版本都可以，因为它能够自动下载正确的版本。

请注意，我并不希望你仅靠阅读文章就能够理解本书中出现的大部分概念。做练习可能是学习过程中最重要的部分，所以我鼓励你不要跳过任何练习。有些练习可能做起来很困难，而你可能会忍不住想看看推荐答案。这样做完全没问题，但是你应该在那之后回到练习中，不看答案再做一次。如果只是翻答案，你可能会在之后尝试解决更高级的练习时再次遇到问题。

这种方式不需要烦琐地打字，因为几乎没有什么要复制的。大部分练习由编写

方法的实现组成，我为此提供了环境和方法签名。没有多于十几行代码的练习；大多数都是四五行。

一旦你完成了练习（即你的实现通过编译），只需运行相应的测试来验证它是否正确即可。

值得注意的是，每一章中的各个练习都是独立的，所以在同一章中，上一个练习创建的代码会复制到下一个练习中。这种做法的必要性在于，因为每个练习通常会建立在前面的练习基础之上，因此虽然可以使用相同的类，但是实现不同。所以，不要在完成前面的练习之前先看后面的练习，因为你会看到还未做的练习的答案。

你可以下载代码的存档文件，也可以用 Git clone。我强烈建议用 clone，因为代码可能会更改，通过简单的 pull 命令来更新代码比重新下载完整的存档文件效率更高。

练习的代码都被分到了各个模块中，它们的名字与章节标题而不是章节号大体上相对应。因此，IDE 将按字母顺序排列它们，而不会按照它们在书中出现的顺序排序。为了帮助你了解每个章节对应的模块，我还在随书附带代码（http://github.com/fpinjava/fpinjava）的 README 文件中提供了一个包含相应模块名称的章节列表。

设定期望

函数式编程并不会比命令式编程更难，只是不同而已。你可以用这两个范式来解决相同的问题，但是从一个转换到另一个时可能效率较低。学习函数式编程就像是学习一门外语。正如你想着一门语言并翻译到另一门语言不可能效率较高那样，你也不能想着命令式并把代码翻译为函数式。并且正如你需要学会用新的语言来思考那样，你也需要学会函数式地思考。仅靠阅读不足以学习函数式地思考，编写代码必不可少。所以你必须多加练习。

这就是为什么我不希望你只是通过阅读来了解这本书，为什么我要提供这么多练习？你需要做练习，以完全掌握函数式编程的概念。并不是因为这个主题很复杂，不可能通过阅读来理解它，而是因为如果你只通过阅读（不做练习）就能理解它，那你很可能不需要这本书。

出于所有的这些原因，练习是充分利用本书的关键。我鼓励你在继续阅读之前尝试解决每个练习。如果你找不到答案，请再试一试，而不是直接翻阅我提供的答案。

如果你理解一些东西很困难，请在论坛上提问（见下一小节）。在论坛上提出问题和获得答案不仅可以帮助你，还可以帮助回答问题的人（以及其他有同样问题的人）。我们都通过回答问题（顺便说一下，主要是我们自己的问题）来学习，而较少通过问问题的方式。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误**：您对书中内容的修改意见可在*提交勘误*处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **与我们交流**：在页面下方*读者评论*处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/33021