Broadview
www.broadview.com.cn

Functional Programming in Scala

Scala 之父 Martin Odersky 作序

Scala函数式编程

[美] Paul Chiusano　Rúnar Bjarnason　著
王宏江　钟伦甫　曹静静　译

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Functional Programming in Scala

Scala函数式编程

[美] Paul Chiusano　Rúnar Bjarnason　著
王宏江　钟伦甫　曹静静　译

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

内容简介

函数式编程越来越多地从学术界走向工业界，很多和人们日常相关的重要系统背后都有函数式编程的身影，并且比例越来越高。在这种大的趋势下，甚至很多指令式编程语言也受其影响加入了对一些函数式特征的支持，比如 Java 8 终于将 lambda 加了进来。

这本书对想要接触函数式编程，或在实际业务中已经使用函数式但想要系统巩固函数式编程知识的程序员来说，是一本非常有价值的书。它以 Scala 为载体，涵盖了函数式的基础和高阶特性。尤其里面的一些高阶特性是在其他书籍中极少介绍到的。因而，非常值得亟待解决高并发问题，或大数据领域从业的开发人员学习。

版权贸易合同登记号 图字：01-2015-3996

图书在版编目（CIP）数据

Scala 函数式编程 /（美）基乌萨诺（Chiusano, P.），（美）比亚尔纳松（Bjarnason, R.）著；王宏江，钟伦甫，曹静静译 . — 北京：电子工业出版社，2016.4

书名原文：Functional Programming in Scala

ISBN 978-7-121-28330-7

Ⅰ.①S… Ⅱ.①基…②比…③王…④钟…⑤曹… Ⅲ.①JAVA 语言 – 程序设计Ⅳ.①TP312

中国版本图书馆 CIP 数据核字 (2016) 第 053663 号

责任编辑：张春雨
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
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：16.75　　字数：402 千字
版　　次：2016 年 4 月第 1 版
印　　次：2016 年 4 月第 1 次印刷
定　　价：69.00 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。

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。

服务热线：（010）88258888。

推荐序 1

我可能是全中国程序员圈子里最不适合给《Scala 函数式编程》写序的人。

三年前我写过《Scala 七大死穴》，算是把 Scala 批判了一番。前几天我则在准备 ThoughtWorks 咨询师大会上的讨论话题《没有函数的函数式编程》，又要杯葛函数式编程的样子。

看起来，我无论对 Scala 还是对函数式编程，都没什么好评嘛。宏江莫不是疯了，居然要我来写序？

等等，事情似乎不是这样。最近几年，ThoughtWorks 的客户在越来越多的项目中采用了 Scala 技术栈，ThoughtWorks 也孵化出了若干基于 Scalaz 的开源项目。我本人也在这些项目中起到了一些作用。

为什么我会做这些“口嫌体正直”的事呢？这得从十年前说起。

我最早是在 C++ 中开始接触到函数式编程的概念。C++ 和 Scala 一样，也是一门多范式语言。C++ 的标准库和 Boost 都提供了许多函数式编程的设施。但是，在我职业生涯初期，给我留下深刻印象的函数式编程库要数 Boost.Egg。

利用 Boost.Egg，你可以写出 my_list|filtered(&is_not_X)|filtered(&is_not_Y) 这样的代码。你会注意到这种用法和 Scala 标准库非常相像，它大致相当于 Scala 的 myList.filter(isNotX).filter(isNotY)，这种 filter 的用法，本书第 5 章中也有讲解。

Boost.Egg 的另一个特点是“非侵入”，比如上例的 filtered 函数，本身并不是 my_list 的成员。相反，我们通过重载 | 运算符给原有的类型添加新功能。这种做法在 Scala 里面相当于隐式转换，本书第 9 章中提供的例子正是利用隐式转换，给字符串添加了中缀操作符。

虽然 Boost.Egg 没能流行起来，但对我个人而言很重要，因为它很大程度塑造了我对代码的品位。

有趣的是，Boost.Egg 的作者 Shunsuke Sogame 近年来的开源项目，都是些 Scala 项目，可能这也是因为 C++ 和 Scala 非常相似的缘故吧。

另一个对我代码品位影响很大的技术是 Lua 中的协程（coroutine）。Lua 的作者 Roberto Ierusalimschy 把协程称为“单趟延续执行流”（One-shot continuation）。有了协程

或者延续执行流，程序员可以手动切换执行流，不再需要编写事件回调函数，而可以编写直接命令式风格代码但不阻塞真正的线程。我的前东家网易在开发游戏时，会大量使用协程来处理业务逻辑，一个游戏程序内同一时刻会运行成千上万个协程。

而在其他不支持协程或者延续执行流的语言中，程序员需要非阻塞或异步编程时，就必须采用层层嵌套回调函数的 CPS（Continuation-Passing Style）风格。这种风格在逻辑复杂时，会陷入“回调地狱”（Callback Hell）的陷阱，使得代码很难读懂，维护起来很困难。

Scala 语言本身并不支持协程或者延续执行流。因此，一般来说，程序员需要非阻塞或异步编程时，就必须使用类似本书第 13 章“外部作用和 I/O”中介绍的技术，注册回调函数或者用 for/yield 语句来执行异步操作。如果流程很复杂的话，即使是 for/yield 语法仍然会陷入回调地狱。

我对 Scala 开源社区的贡献之一是 stateless-future。这个库提供了一些宏，实现了延续执行流，可以自动把命令式风格的代码转换成 CPS 风格。通过这种做法，程序员不再需要手写本书 13.2 节那样的代码了，编写的代码风格更像普通的 Java 或者 PHP 风格，直接像流水账一样描述顺序流程。

后来，我把这种避免回调函数的思路，推广到了其他用途上。比如，我开发了基于 Scala.js 的前端框架 Binding.scala。使用 Binding.scala 的用户，编写普通的 HTML 模板，描述视图中的变量绑定关系，而不需要编写高阶函数就能做出交互复杂的网页。

而我的另一个开源库 Each，则更进一步，支持一切 monad。大多数情况下，使用了 Each 就不需要编写任何高阶函数，我称之为“没有函数的函数式编程”。这意味，本书第 11 章到第 15 章的全部内容，你都可以直接编写类似 Java 的命令式语法，而 Each 则自动帮你生成使用 monad 的代码。

总之，我是 Scala 函数式编程的死对头，我写的 Scala 库，恰恰是为了避免使用本书中谆谆教导的各种高阶函数。如果你是个 Java 程序员，想在最短的时间内用 Scala 开始“搬砖”，那么，从实用角度出发，我建议你合上本书，直接用 Each 即可。因为，虽然 Each 最终会生成 Monad 风格代码，但是，本书中涉及的使用高阶函数的细节，就像汇编语言一样，就算你不知道也照样可以干活。

不过，如果你是个求道者，追求编程艺术的真理，希望刨根到底，理解函数式编程的内在理论和实现机制，那么本书很适合你。

这本书绝不轻易放过每个知识点，全书包含有大量习题，要求你自己实现 Scala 标准库或者 Scalaz 中的既有功能。所以，当你读完本书，做完习题后，虽然你的应用开发能力并不会直接提升，但你会体会到构建函数式语言和框架时的难点和取舍，从而增进你的框架开发和语言设计的能力。

ThoughtWorks Lead Consultant 杨博

参考资料

1. Boost.Egg
2. 关于 Lua 中的协程，参见 A. L. de Moura, N. Rodriguez, and R. Ierusalimschy. Coroutines in Lua. Journal of Universal Computer Science, 10(7):910–925.
3. 关于延续执行体的历史，参见 Reynolds, John C. (1993). “The discoveries of continuations” (PDF). Lisp and Symbolic Computation 6 (3/4): 233–248.
4. 关于 Scala 异步编程的“回调地狱”问题，参见 Business-Friendly Functional Programming – Part 1: Asynchronous Operations

推荐序 2

函数式编程与命令式编程同样源远流长，然而在计算机应用的历史进程中，二者的地位却颇不对等。命令式编程几乎自始至终都是大众的宠儿，函数式编程却长期局限于象牙塔和少数应用领域之内。尽管如此，函数式编程的重要性却从未被忽视，几十年来生机勃勃地发展，静静地等待着逆袭的时刻。事实上，即便是浸淫于命令式编程多年的工程师，也常常会与函数式编程亲密接触而不自知：例如 SQL、C++ 模板元编程，还有 C++ 标准库中的 STL 等，多少都带有一些函数式的色彩。早年，受软硬件水平的限制，函数式语言缺乏高效的编译器和运行时支持，这可能是函数式语言错失流行机会的一大原因。近年来，一方面程序语言理论和实现技术突飞猛进，函数式语言在性能上的劣势越来越不明显；另一方面，随着多核、高并发、分布式场景激增，大众也逐渐开始认识到函数式编程在这些领域得天独厚的优势。然而，流连于主流命令式语言多年积攒下的库、框架、社区等丰富资产，再加上长期的教育惯性与思维惯性，使得人们仍然难以在生产上完全转向函数式语言。

一个新的契机，来自于大数据。社交网络、个人移动设备、物联网等新技术的兴起，使得海量数据处理开始成为家常便饭。人们突然发现，自己在命令式世界的武器库中，竟找不出称手的兵器来攻打大数据这座堡垒。2008 年，Google 发表了跨时代的 MapReduce 论文。尽管学界对 MapReduce 颇有非议[1]，MapReduce 的核心思想仍然旋风般席卷了整个工业界，为大数据技术的发展带来了及时而深远的影响。有趣的是，MapReduce 的核心思想，正是来自于天生擅长高并发和分布式场景的函数式编程。自此以后，各色大数据处理系统层出不穷，而其中的函数式成分也愈加浓重：在用户接口层面，这些系统往往以 DSL 的形式提供一套类 SQL 语义、具函数式特征的申明式查询接口；在实现层面，则仰仗不变性等约束来简化并发和容错。然而，出于种种原因，大部分系统的实现语言仍然以 C++、Java、C# 这些命令式语言为主。可谓操命令式之形而施函数式之实。

自 2009 年起，我先后接触了 Erlang、Scheme、ML 等函数式语言。但出于显而易见的原因，未能有机会将之用于工程实战。2013 年春节前后，我参加了由 Scala 之父 Martin Odersky 在 Coursera 上开设的 Functional Programming Principles in Scala 课程。凑巧的是，就在课程结束后不久，我便得到一个机会加入 Intel 参与有关大数据和 Apache Spark 的工作。函数式语

1 David J. DeWitt and Michael Stonebraker, MapReduce: A major step backwards, https://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.html

言和分布式系统一直是我的两大兴趣点，由 Scala 开发的 Spark 恰恰是二者的一个完美融合。于是，Scala 便成了我的第一门实战函数式语言。近年来 Spark 的火爆，更是对 Scala 和函数式编程的推广起到了推波助澜的作用。

与我所熟悉的其他函数式语言相比，我想 Scala 最大的优点之一就是过渡平滑。立足于 JVM 并将函数式融入如日中天的面向对象，这样的设计带来了两大明显的好处。第一，顺畅地集成了 Java 社区多年积累的财富。第二，Scala 和 C++ 类似，也是一门“广谱”多范式语言；不熟悉函数式编程的 Scala 初学者，随时可以安全地回退，转用熟悉的命令式面向对象范式编程，从而保证交付速度。这个设计的背后，应该与 Martin Odersky “学术工业两手抓、两手都很硬”的风格不无关系。论学术，他师承 Pascal 之父 Niklaus Wirth，在代码分析和程序语言设计方面建树颇丰；论工业应用，他一手打造了 Generic Java 和新一代的 `javac` 编译器。可以说 Martin 既具备了用以高瞻远瞩的理论基础，又十分了解普罗大众的痛点。两相结合，这才造就了 Scala 这样一个平衡于阳春白雪和下里巴人之间的作品。

其实函数式编程本身并没有多难。对于接受过若干年数学训练，却没有任何编程经验的人来说，相较于命令式编程中的破坏性赋值，函数式编程中的不变性、递归等概念反而应该更加亲切。譬如中学做证明题时，我们从不会先令 a = x，隔上几行再令 a = a + 1。真正的困难在于，函数式编程中的一些概念和手法——如用尾递归代替循环——与命令式编程的直觉相冲突。对于那些有着多年命令式语言编程经验，把 Java 用得比母语还溜的工程师们而言，一边要学习新的知识，一边还要克服多年编程训练所造成的思维定势，无异于逆水行舟。而在 Scala 中，你永远有机会在实战中回退至自己的舒适区。实际上，我们完全可以无视 Scala 的函数式特性，仅仅将 Scala 当作语法更加洗练的 Java。因此，对于那些操持主流命令式面向对象语言多年的工程师们而言，Scala 特别适合用作涉猎函数式编程的起步语言。

这本书所讲授的，正是基于 Scala 的函数式编程基础。基于 Scheme、Haskell 等老牌函数式语言的传统教材的问题在于，相关语言的语法和思维方式与读者现有的知识体系迥异，容易造成较为陡峭的入门门槛。此外，由于这些语言本身的实际应用机会不多，初学者也难以在实战中获得宝贵的直觉和经验。而在 Scala 的帮助下，这本书并不要求你抛开现有的思维方式另起炉灶，它所做的更像是为你现有的思维方式添砖加瓦，从而令你如虎添翼。

最后，作为曾经的译者，我深知在国内当前的大环境下技术翻译之不易。每一本优秀技术书籍的中译本背后都是译者数月乃至十数月的艰苦付出。感谢诸位译者为我们带来又一本好书！

Spark committer from Databricks 连城

译序

在近些年新兴起的一些编程语言里，带有函数式特性的语言占据很大的比例，其中Scala的发展无疑是非常突出的。目前来看Scala所取得的成功，主要归功于它抓住了互联网的快速发展、大数据的兴起，以及部分软件企业的产业升级等机遇（尤其在大数据领域，Scala已经成为这个领域的明星语言）；这些领域的发展变化对语言提出了新的要求：提升生产力，满足高并发、高性能及高度抽象——Scala正好都可以满足。Scala内置了很多设计模式（以语法糖的方式），也吸取了很多其他编程语言里已经被验证的优点，比如类型推导、模式匹配等，这极大地简化了开发过程中所要写的代码量（相对于Java），提升了程序员的生产效率。此外，得益于JVM本身的强大威力，加上编译器的一些优化，使它在运行时的效率与Java并没有太大的差距。

作为深受学院派风格影响的语言，Scala本身并不像很多经典的函数式语言那么“纯粹”，而是很大程度地迎合了工业界的需求——保证了与Java极大的兼容。程序员既可以用传统的Java风格（指令式）实现功能，也可以用函数式风格来实现，这为更多程序员提供了相对平滑的过渡方式。

但是即便Scala提供了面向Java程序员的相对平滑的适应方式，依然让很多程序员觉得它有很多高深复杂的地方，即使一个有多年Scala编程经验的程序员也可能对它的很多特性或用法不甚了解。一方面是Scala吸纳百家所长，特性有些多，另一方面是大家对函数式编程了解得还不够。函数式编程的历史很久远，但一直只在业界某些特定领域使用，并未被广泛接受，而最近十余年它才开始逐渐升温。在这之前不管是大学里的普通计算机教育，还是工作实践，多是以C++/Java等指令式编程为主的。大多数程序员并没有对函数式进行系统的了解。函数式编程是一个范围宏大的话题，它像一个参差多态的森林，很多人进入其中也仅是只见树木，不见森林。

五六年前接触Scala的时候，刚开始只是把它当作改良的Java，在一些非核心应用上做了一些简单尝试，并未有特别的感受。直到某天遇到了monad，这个概念一下把我难住了。费了很大的精力才逐渐明白它是怎么回事，之后反思为何这个问题难以理解，很大原因是以往的经验困扰了自己，总是将新概念与原有的经验去匹配，而它事实上并不能与你既有的世界观相匹配。原来还可以用这种思维去编程！于是开始清空以往固有的认识，严肃地去了解函数式和类型系统。在学习类型系统的过程中，发现了RÚNAR的博客，他在博客

上介绍了很多函数式编程的技巧和模式。这些模式不同于我们熟悉的面向对象里常见的设计模式，它多是从类型系统的角度去抽象的，这给我们带来很多新的思考。后来 RÚNAR 把这些内容总结成了这本书，也是机缘巧合，在去年通过老高跟引入这本书中文版权的电子工业出版社取得联系，最终我和同事钟伦甫、曹静静共同完成了对这本书的翻译。

为什么这些大数据的开发者会用这门语言制造了 spark、kafka、algebird/summingbird 等流行产品？并不是说其他语言不可以，而是这些开发者大多都是有学院背景且非常聪明的一群人，他们选择 Scala 除了品味之外，还基于它的抽象能力和它的表达能力。Scala 在这方面比 Java 等语言具备更高的抽象能力，很大一部分原因是它继承自 Haskell 和 ML 里的一些概念。发明 Haskell 的这群数学家为我们提供了另一种抽象问题的方式，这里面有众多的概念都在编程语言领域影响深远。

尽管函数式编程在近十多年用得越来越多，但市面上介绍其高阶特性的书却并不多。这本书在这方面是个重要的补充，它不仅仅面向 Scala 程序员，同样面向用任何编程语言开发的程序员，只要你充满好奇心。

真正参与一本书的翻译之后才发现翻译是要求很高的事，并不只是逻辑上正确与否的问题。这个过程非常感谢高宇翔（程序员老高），以及电子工业出版社的编辑在翻译过程中给出的很多帮助和指正。Scala 领域还有一些英文术语没有形成固定的中文叫法，我们尽量参考了市面上其他几本 Scala 书里的一些叫法。因为水平有限，难免会有错误，恳请读者在发现任何存疑的地方时给予反馈，可以通过邮箱（w.hongjiang@gmail.com）或微博、博客等方式联系我。

王宏江

原推荐序

函数式编程作为书题出现在 Scala 中是个有趣的现象。毕竟，通常 Scala 被称为函数式编程语言，而且在市场上有非常多的 Scala 相关书籍。是不是这些书都缺失了对语言函数式方面内容的描述？为了回答这个问题，我们需要有指导性地进行深挖。什么是函数式编程？对我来说，它是“使用函数编程”的别名，换句话说，是一种聚焦在函数上的编程方式。那么什么是函数？再来探寻更大范围的定义。当一种定义承认函数可能有副作用并返回结果时，纯函数式编程限制函数就像数学里定义的那样：用一种二元关系去映射参数到结果。

Scala 是不纯粹的函数式编程语言，它同时承认非纯粹函数和纯函数，而且没有使用不同的语法或给予不同的类型去区分这两种函数种类。其他函数式语言也有同样的属性。在 Scala 里如果能区分纯函数和非纯函数将是很好的，但我认为我们没有找到轻量级的和无需迟疑的灵活方式来这么做。

可以确信的是，Scala 程序员是被鼓励使用纯函数编程的。副作用也有，比如易变、I/O 或者异常的使用没有被禁止，事实上这些副作用有的时候使用起来十分方便，使用它们的原因有互用性、高效、方便等。但是专家的建议是过度地使用副作用普遍来说不是一种好的方式。然而，因为在 Scala 中非纯函数编程是可能的甚至是方便的，对命令式编程背景的程序员来说，保持他们的风格和不努力采用函数式思维的诱惑就非常大了。事实是，非常有可能将 Scala 编写成没有封号结尾的 Java 程序。

那么要学习 Scala 中的函数式编程，是不是需要先学习纯函数式编程，比如 Haskell？任何关于方法的争论都在本书的出现后被极大地削弱了。

Paul 和 Rúnar 所做的是简单地将 Scala 作为纯函数式编程语言。可变变量、异常、经典的输入输出和所有其他的非纯函数被消除了。假如你想知道在没有这些便捷方式下如何编写有用的代码，你需要阅读此书。从第一个原理扩展到增量的输入输出，本书展示了如何使用纯函数表达每一个概念。而且不仅仅是展示了可能性，也同样引导你去编写优美的代码和深入探索计算的本质。

本书是充满挑战的，不仅仅是因为它需要对细节的注意，同样是对你编程思想的挑战。通过阅读本书和完成推荐的练习，你将更好地认识纯函数式编程是什么，能表达什么，优点是什么。

本书让我特别喜欢的是它的自成体系。它开始于最简单的表达式，然后从细节解释每个抽象，再在其基础上进一步抽象。在某种程度上，本书开发了另一个 Scala“宇宙”，这里可变状态是不存在的，所有函数是纯的。普遍使用的 Scala 库的实现和这有些偏离，通常它们是部分按照命令式实现的，（大多数）外层是函数式接口。Scala 容许在函数式接口中封装可变状态，我认为这是一个优点。但是这种能力通常也被滥用。假如发现自己过多地使用它，那么本书是一种强力的解药。

MARTIN ODERSKY

Scala 的创造者

序言

编写好的软件很难。在各种方法论中纠结多年，我们俩发现并爱上了函数式编程（FP）。尽管它与众不同，但它就是能引领我们编写出一致连贯、灵活组合、美丽优雅的程序。

我们俩都是波士顿地区 Scala 爱好者群（Boston Area Scala Enthusiasts）的成员，这个群会定期在剑桥聚会。起初，群里主要是一些 Java 程序员，他们一直寻求一些更好的东西。后来大部分的人都表示，没有一个好的方法去学习如何用 Scala 进行函数式编程。我们学习的过程几乎都很随意，写一些函数式的代码，向其他 Scala 和 Haskell 程序员请教学习，阅读一些文章、博客和书籍。我们始终觉得应该有比这更简单的学习方法，直到 2010 年 4 月，群的组织者之一 Nermin Šerifović，建议我们写一本关于 Scala 函数式编程的书。本以为基于我们学习的经验，写一本期望中思路清晰的书是一件又快又容易的事情，没想到我们花了 4 年多才完成。要是我们当初学习函数式编程时，有这样一本书该多好啊。

希望这本书能够带给你一种兴奋刺激的感觉，犹如我们第一次遇到函数式编程那样。

致谢

这里我们要感谢很多曾经参与本书编写过程的人。首先是 Nermin Šerifović，我们波士顿 Scala 群的好朋友，没有他，我们不会开始写这本书。

然后是 Capital IQ 这个优秀的团队，感谢你们的支持，感谢你们自告奋勇地帮助测试了本书课程的第一个版本。

特别感谢 Tony Morris，他在本书早期阶段上的工作仍然非常有价值，尤其是在 Scala 函数式编程的实践上做出的贡献。

还有 Martin，感谢他为本书撰写的精彩前言，更感谢他创造了如此强大的编程语言并改变了整个行业。

感谢一直以来社区里那些热衷于函数式编程的 Scala 用户的鼓励。对于本书的审阅者、MEAP 的读者，以及每个为本书提供反馈建议的人，我们同样感谢你们，没有你们就没有这本书今天的样子。

还要尤其感谢开发编辑 Jeff Bleiel、图形编辑 Ben Kovitz、技术校对 Tom Lockney，以及每一个让本书变得更好的 Manning 的工作人员，包括以下开发阶段稿件的审阅者：Ashton Hepburn、Bennett Andrews、Chris Marshall、Chris Nauroth、Cody Koeninger、Dave Cleaver、Douglas Alan、Eric Torreborre、Erich W. Schreiner、Fernando Dobladez、Ionut、G. Stan、Jeton Bacaj、Kai Gellien、Luc Duponcheel、Mark Janssen、Michael Smolyak、Ravindra Jaju、Rintcius Blok、Rod Hilton、Sebastien Nichele、Sukant Hajra、Thad Meyer、Will Hayworth 和 William E. Wheeler。

最后，感谢 Sunita 和 Allison，正是因为你们一直以来的支持，才让这历经数年的冒险路程得以圆满。

关于本书

这不是一本关于 Scala 的书。这是一本介绍函数式编程（FP）的书——一种彻底的原则性很强的编写软件的方法。我们使用 Scala 作为交通工具到达那里，但是你可以在课程中使用任何其他编程语言。当你读完这本书，我们的目标是让你牢固地掌握函数式编程的概念，在写纯函数式程序时感到舒适，并能够为这些主题吸收新的素材，超越本书所覆盖的这些内容。

这本书的结构

这本书由四部分组成。在第一部分，我们讨论究竟什么是函数式编程，并引入一些核心概念。在第一部分的章节里做一个基础的技术总览，例如怎么组织小的函数式程序、定义纯函数式数据结构、错误处理，以及怎么和状态打交道。

在这基础之上，第二部分是一系列的函数式设计教程。我们通过一些实际的函数式库里的例子，揭露设计它们的思维过程。

通过第二部分的库函数的练习，会让你清楚这些库遵循某种模式并有一些冗余。这将突出我们对一种新的更高阶的抽象方式的需要，高阶的抽象能够写出更加泛化的库函数，我们将在第三部分引入这些抽象。这些都是对你的代码推导非常具有威力的工具。一旦你掌握它们，会让你成为一个极其富有生产力的程序员。

第四部分则填补了面向真实世界应用的其余部分的空缺，这些部分包括执行 I/O（例如写数据库、文件或视频显示器）、使用可变状态等，所有这些都是以纯函数式的风格来实现的。

在本书的学习过程中，我们依赖大量的编程练习，仔细、有序的练习能够帮助你消化这些资料。要理解函数式编程，不仅仅是学习理论，你必须立即打开你的文本编辑器写一些代码，你必须将那些你已经学习的理论，转换为工作中的实践。

我们还提供了所有章节的在线笔记。每一章都有一段与之相关的讨论，里面提供了一些进一步阅读的材料链接。这些章节笔记是可以被社区的读者们扩展的，是一个可编辑的 wiki：https://github.com/fpinscala/fpinscala/wiki。

受众人群

本书适合至少有一些编程经验的人参考阅读。在我们心中有一些特定种类的读者——那些对函数式编程好奇的，并且达到中级水平的 Java 或 C 程序员。但我们相信这本书也适合任何语言和任何经验水平的程序员。

在读这本书时你以前的专长经验并不一定比你的动机和好奇心更重要。函数式编程充满乐趣，但它也是一个有挑战性的课题，尤其是对于那些更有经验的程序员来说，因为它的思维方式可能不同于你以往有过的经验。不管你编程多久，你必须准备以一个初学者的姿态来学习。

读这本书并不需要之前有任何 Scala 经验，但我们不会花大量的时间和篇幅讨论 Scala 的语法和语言特性。我们将按照我们的需要，以最小化的方式来介绍 Scala，内容主要涉及的是其他素材。这些对 Scala 的介绍应该足以让你开始练习。如果你对 Scala 语言有进一步的问题，你应该用另一本 Scala 的书作为你的补充阅读（http://scala-lang.org/ documentation/books.html），或者在 Scala 语言文档（http://scala-lang.org/documentation/）里查找具体的问题。

如何阅读这本书

你可以按序阅读这本书，这四个部分的顺序也是有意设计的，你可以按照自己舒适的方式在这些部分之间先休息一下，把你所学的应用到工作中，然后再回来继续阅读下一部分。例如，第四部分的内容，只有你阅读完第一、二、三部分，并且非常熟悉函数式风格的编程之后才有意义。在第三部分之后，休息一下，尝试做一些比章节里更多的函数式程序练习，或许是个好主意。当然，最终都取决于你自己。

这本书里大部分章节都有相似的结构。先引入一些新的思想或技术，用例子来解释，然后通过一些练习来掌握。我们强烈建议你下载练习的源代码，在学习每一章的时候做做练习。练习、提示以及答案都可以在这里获取：https://github.com/fpinscala/fpinscla。我们也鼓励你访问 scala-functional Google 用户组（https://groups.google.com/forum/#!topic/scala-functional/），以及在 irc.freenode.net 上的 #fp-in-scala IRC 频道参与讨论。

练习题会根据难度和重要性来标记。我们会按照自己的理解来标记题目是否很难，或者它对章节材料的理解是否可选。带有困难标记的，我们会尝试给你一些期待的思路——这只是我们所猜测的，或许你会发现一些没有标记困难的问题很难解决，一些标记为困难的问题反而很容易。带有可选标记的练习主要为增进你的见识，在不妨碍你阅读后续的内容时你可以跳过。这些练习用下面的图标表示是否可选：

◈ **练习 1**

在一个被填充的四边形图标后边的练习表示它们是关键的练习。

◇ **练习 2**

一个空心的四边形表示练习是可选的。

这本书会贯穿很多例子，这些例子你要去尝试实践，而非仅仅只是阅读。在你开始之前，应该确保 Scala 解释器能运行。我们鼓励你用自己的方式去试验一下你看到的各种例子。帮你把事情搞明白的一个方法就是对它稍作改造，看看所做的改变对输出结果有什么影响。

有时我们会用 Scala 解释器会话来演示一些代码的运行或求值结果。这些将以解释器的提示符 scala> 开头。在提示符后边紧跟的是要输入或粘贴到解释器的代码，而下一行则是解释器的响应，就像这样：

```
scala> println("Hello, World!")
Hello, World!
```

代码规范和下载

在列表或文本中的所有代码都是*类似这样的等宽字体*，以便于把它们和普通文本区分开。Scala 里的关键字使用*类似这样的等宽粗体*。很多清单里都会包含代码注释，用于突出重要概念。

要下载这本书里的样例代码，以及练习题和章节注释里的代码，请到 https://github.com/fpinscala/fpinscala 或者出版商网站：www.manning.com/FunctionalProgramminginScala。

设定预期

虽然函数式编程对我们编写软件过程中的每个层面都产生了深远的影响，但它仍需要时间。这是一个渐进的过程。在第 1 章先不要指望能被神奇的函数式编程所惊艳到。安排在开头的一些原则都非常微妙，甚至看起来跟常识没什么区别。如果你觉得“这个我在不知道函数式编程的时候已经能够做到了”，那就对了，这正是重点所在。大多数程序员已经在某种程度上做过函数式编程的事情，只是他们不知道而已。大多数人认为最佳实践的很多东西（例如，让函数只具有单一职责，或让数据不可变）都受到函数式编程的启发。我们只是发现在这些最佳实践下面的原则，找出它们的逻辑结论。

在读这本书的时候有很大的可能你将同时要学习 Scala 语法和函数式编程。一开始这些代码可能看起来很陌生，这些技术也显得不太自然，并且练习题也很费脑子。这很正常，不要被吓到。如果你被问题卡住，看一下提示和答案，或把问题抛到 Google 用户组（https://groups.google.com/forum/#!topic/scala-functional/），或 IRC 频道（#fp-in-scala on irc.freenode.net）。

总之，我们希望这本书对你来说是有趣的、有益的体验，函数式编程能让你的工作更轻松、更愉悦，正如它已经带给我们的收益。这本书的目的是当你把里面所有说过的都尝试过，能帮助你在工作时更有效率。它应该让你感到不像是在写一些丑陋、不符合设计原理且难维护的软件，而更像是要创建一个美观实用的东西。