O'REILLY®

TURING 图灵程序设计丛书

函数式编程思维

Functional Thinking

[美] Neal Ford 著
郭晓刚 译

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

函数式编程思维

Functional Thinking

[美] Neal Ford 著
郭晓刚 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　京

图书在版编目（CIP）数据

函数式编程思维 /（美）福特（Ford, N.）著；郭晓刚 译. -- 北京：人民邮电出版社，2015.8
（图灵程序设计丛书）
ISBN 978-7-115-40041-3

Ⅰ. ①函… Ⅱ. ①福… ②郭… Ⅲ. ①函数－程序设计 Ⅳ. ①TP311.1

中国版本图书馆CIP数据核字(2015)第176648号

内 容 提 要

本书脱离特定的语言特性，关注各种 OOP 语言的共同实践做法，展示如何通过函数式编程解决问题。知名软件架构师 Neal Ford 展示了不同的编程范式，帮助我们完成从 Java 命令式编程人员，到使用 Java、Clojure、Scala 的函数式编程人员的完美转变，建立对函数式语言的语法和语义的良好理解。

本书适合 Java、Clojure、Scala 及其他想要提高工作效率、关注函数式编程的程序员阅读。

◆ 著 [美] Neal Ford
译 郭晓刚
责任编辑 岳新欣
执行编辑 冯雪松
责任印制 杨林杰

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京圣夫亚美印刷有限公司印刷

◆ 开本：800×1000 1/16
印张：10.25
字数：242千字 2015年8月第1版
印数：1－3 500册 2015年8月北京第1次印刷
著作权合同登记号 图字：01-2015-2577号

定价：49.00元

读者服务热线：(010)51095186转600 印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第 0021 号

版权声明

O'Reilly Media, Inc.介绍

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

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

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

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

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

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

——*Linux Journal*

译者序

函数式编程不是屠龙技。过去在一般开发者的认识里，函数式编程是一种仅仅存在于某些偏门语言里的学究气的概念。然而我们观察当今的主流语言，会发现函数式编程已经成为了标配，唯其存在形式发生了变化，从固执于“纯”函数式语言，转变为让一些关键的函数式特征或深或浅地融入到各式语言中去。

函数式编程的普及趋势，我以为主要应该归因于纯函数、一等函数、高阶函数等特征迎合了人们提高语法表现力和解决大规模并发问题的需要。函数式编程进入主流语言，意味着我们实际上已经在不同程度地使用着函数式编程。比如，你不一定用 F#，但 LINQ 实在是太方便了；你可能觉得 Clojure 太怪异，但 map、filter、reduce 任何时候都是必备的利器。

不同语言的函数式能力可以有很大的差别。那么在一些只能迂回模拟个别函数式特征的语言里面，去谈论函数式编程是否有意义？我对同行提到这本书用 Java 8 来解说函数式编程的时候，立即被编出了“只有这样才能写一本书”的笑话。笑点显然是因为用 Haskell、Lisp 来解说的话，写一章就够了。作者 Neal Ford 大概有不一样的看法，因为他故意用了 Scala、Clojure、Groovy、Java 8 这些函数式程度各异的语言，乃至在 Java 5 的极端环境下的 Functional Java 框架来证明，即使只是函数式编程的一个很小的子集，已经能够满足很大一部分需要，发挥很大的作用。毕竟，不管语法和实现上如何笨拙，函数式编程为我们开启的是另一个广阔的思考维度。不负责任地说，就算只学到了 map、filter、reduce 三板斧，你花在这本书上的时间都是值得的。

那么，要不要来学一学函数式编程呢？我想，开发者总不能比 Java 进步得还慢吧。

我把这本书翻译完了，而且，我敢保证，书里面没有一句话是你看不懂需要去翻原文的。把一本书从头到尾好好地译完，这件事情就算做过再多次，仍然值得我大大地夸一下自己，特别是我同时还要照顾两岁的郭寄傲小朋友。我的孩子要尝试 10 次、20 次才肯接受一种新的食物。我们接受一种新的范式，大概不会比这个简单。

绕了一个大圈子，我其实想说：靡不有初，鲜克有终。请不要只是买了这本书，而是真的学会函数式思维吧！

郭晓刚

2015 年 7 月

前言

我第一次认真研究函数式编程是在 2004 年。当时我受到 .NET 平台上一些非常规语言的吸引，开始摆弄 Haskell 和若干早于 F# 的 ML 家族语言。到了 2005 年，我开始在一些会议上做关于“.NET 和函数式语言”的演讲，不过那时候的语言多半还只是概念性的，即使说成是“玩具”也不为过。但不管怎么说，能够试探在一种新的编程思维范式下推演铺陈的可能性，已然令我乐在其中，而且这段经历改变了我在常规语言里对一些问题的处理方法。

2010 年我再次涉足这个研究领域，是因为目睹当时崛起的一批语言，例如 Java 生态圈里的 Clojure 和 Scala，一下子让我重温了五年前亲历的那些函数式世界的精妙所在。于是我在一个午后打开维基百科，顺着链接一页一页地翻阅着，半天时间下来，我已经完全沉迷其中。就这样，我一头钻入函数式编程的世界，开始了走遍各种思维分枝的探索历程。作为研究的成果，我于 2011 年在波兰举办的“33rd Degree Conference”大会（http://33degree.org/）上第一次做了题为“函数式编程思维”的演讲，又在 IBM developerWorks 网站上开设了同名的系列文章（http://dwz.cn/dev-works-ft-series）。在接下来的两年时间里，我按照每个月写一篇文章的进度，制订对函数式编程的研究和探索计划，并且坚持了下来。至今，我的函数式编程思维的演讲仍在继续，并且根据反馈不断完善。

这本书是对“函数式编程思维”演讲和系列文章中所有观点的总结。我发觉磨砺素材最好的办法是将之反复地呈现给观众，因为我每次做演讲或者写文章都会学到一些新东西。有些关联或者共性只有深入研究和被迫思考（截稿时间特别能让人集中精神！）之后，才会发现。

我在上一本书 *Presentation Patterns*（http://presentationpatterns.com/）中说过视觉象征对于会议演讲的重要性。因此我在做“函数式编程思维”演讲的时候，特意用了黑板和粉笔的形象（来引申出与函数式编程概念的数学联系）。到演讲结束的时候，我会呈现一张半截粉笔摆在黑板下方的图片，暗示观众自己拿起这半截粉笔，继续探索演讲中提到的观点。

我做的演讲，写的系列文章以及这本书，目的都是想针对那些在命令式的、面向对象的语言中浸淫已久的开发者，用一种他们能够理解的方式来介绍函数式编程的核心观点。希望我提炼的这些观点能引发你的兴致，并且拿起粉笔来继续你自己的探索。

——Neal Ford，2014 年 6 月于亚特兰大

本书结构

本书每一章都会演示函数式思维的例子。第 1 章“为什么”提供了概述和若干贯穿全书的思维转换的例子。第 2 章“转变思维”为程序员描绘了一个渐进的转变过程，让你从面向对象、命令式的观察角度过渡到函数式的观察角度。为了形象地展示这种思维转变，我分别用命令式风格和函数式风格来解决同一个常见问题以作对比。然后又通过一个详尽的案例分析来说明函数式的观察角度（以及若干辅助语法）如何帮助你向函数式的思维方式转变。

第 3 章“权责让渡”列举了一些可以放心托付给语言或运行时去处理的日常杂务。状态是 Michael Feathers 所谓的“不确定因素”之一，通常在非函数式的语言里需要直接明确地进行管理。闭包（closure）允许我们将一部分状态管理工作交托给运行时；我举了一些例子来说明这个状态处理机制背后的工作原理。这一章还会展示如何按照函数式思维在一些细节方面放权，例如把集合操作交给递归。这些思路将对代码重用的粒度产生影响。

第 4 章“用巧不用蛮”着重讨论两个延续“消灭不确定因素”精神的例子，它们利用运行时来缓存函数的结果，从而获得缓求值（laziness）的特性。很多函数式语言都包含“记忆”（memoization）特性（可能直接支持，也可能通过库，或者用一点小技巧就能实现），可以作为一种常用的性能优化手段。我在第 2 章“完全数分类器”例子的基础上比较了几种不同层次的优化手段，有手工进行的，也有利用语言提供的记忆特性来完成的。如果你想提前知道比赛的结果，记忆特性是最后的赢家。缓求值（lazy）的数据结构把运算推迟到最后时刻才去执行，这个特点让我们有机会换一个角度来看待各种数据结构。我演示了如何实现缓求值数据结构（甚至可以用非函数式语言来实现），以及如何利用语言已经具备的缓求值特性。

第 5 章“演化的语言”反映各种语言是怎样朝着加强函数式特征的方向演变的。本章还会讨论若干革命性的语言发展趋势，如操作符重载和方法调用之外的新的分派（dispatch）方式，讨论让语言去迎合问题（而不是反过来）的观点，以及 Option 等常见的函数式数据结构。

第 6 章“模式与重用”通过一些例子来展示解决问题的一般思路。我分析了传统的设计模式在函数式编程的世界里是怎样蜕变（或者消失）的。我还详细对比了通过继承和通过组合这两种代码重用方式，并从耦合的角度对它们进行了由表及里的分析。

第 7 章“现实应用”详细展示了 Java 开发工具包（JDK）新增的几项人们期待已久的函数式特性。从分析中可以看到，Java 8 也像别的语言一样接纳了函数式思维，它的高阶函数（即 lambda 块）用法就是表现之一。我还讨论了 Java 8 在保持向后兼容上使用的一些巧妙而优雅的手法。Stream API 是特别提到的一个发扬了函数式思维的亮点，它能够以描述性的语言简洁明了地表达工作流。最后我介绍了 Java 8 新增的 Option 结构，它解决了 null 返回值含义模糊的潜在问题。我还用了一些篇幅来讨论函数式架构和数据库的主题，分析函数式的视角怎样改变了它们的设计。

第 8 章“多语言与多范式”叙述了函数式编程对于当前这个多语言世界的影响。我们一直在各种项目中遭遇和容纳越来越多的语言。很多新的语言都是多范式（polyparadigm）的，同时支持若干种不同的编程模型。例如 Scala 支持面向对象编程和函数式编程。作为最后一章，我们探讨了活在一个有更多范式可以选择的世界里有什么好处和坏处。

排版约定

本书使用了下列排版约定。

- 楷体
  表示新术语或突出强调的内容。
- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。
- 加粗等宽字体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。

该图标表示一般注记。

使用代码示例

补充材料（示例代码、练习等）可以从 https://github.com/oreillymedia/functional_thinking 下载。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无需联系我们获得许可。比如，用本书的几个代码片段写一个程序就无需获得许可，销售或分发 O’Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无需获得许可，将书中大量的代码放到你的产

品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和ISBN。比如："*Functional Thinking* by Neal Ford (O'Reilly). Copyright 2014 Neal Ford, 978-1-449-36551-6."

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过permissions@oreilly.com与我们联系。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将Safari Books Online视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解Safari Books Online的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

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

中国：

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

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

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com。

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

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

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

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

致谢

感谢 ThoughtWorks 大家庭，这是我能找到的最好的工作环境。感谢和我一起参加各种会议的讲师们，尤其是“No Fluff, Just Stuff”会议的讲师们，给了我许多思想的碰撞。感谢这些年来出席“函数式编程思维”演讲的观众，你们的反馈帮我磨砺了本书的素材。特别感谢本书的技术审阅人，他们给出了中肯的实质性建议。尤其感谢那些花时间提交勘误的早期读者，你们为我揭示了许多视而不见的晦涩之处。感谢数不清的朋友和家人充当了我坚实的后盾，特别感谢 John Drescher 在我们离家的时候帮忙照顾猫咪们。当然，还要感谢一直忍耐包容我的夫人 Candy，她早就不指望我能放下对编程的迷恋了。

目录

第1章

为什么

我们用几分钟来想象一下自己是一名伐木工人，手里有林场里最好的斧子，因此你是工作效率最高的。突然有一天场里来了个推销的，他把一种新的砍树工具——链锯——给夸到了天上去。这人很有说服力，所以你也买了一把，不过你不懂得怎么用。你估摸着按照自己原来擅长的砍树方法，把链锯大力地挥向树干——不知道要先发动它。“链锯不过是时髦的样子货罢了”，没砍几下你就得出了这样的结论，于是把它丢到一边重新捡起用惯了的斧子。就在这个时候，有人在你面前把链锯给发动了……

学习一种全新的编程范式，困难并不在于掌握新的语言。毕竟能拿起这本书的读者，学过的编程语言少说也有一箩筐——语法不过是些小细节罢了。真正考验人的，是怎么学会用另一种方式去思考。

本书探讨函数式编程的话题，但重点并不放在函数式编程语言上。请别误会，我并不打算空谈理论，书里会有用很多种语言写成的大量代码，实际上整本书都是围绕着代码来展开的。用“函数式”的方式编写代码牵涉到诸多方面，我会用具体的例子来解说各方面的要旨，包括设计上的种种取舍、不同重用单元的作用等。比起语法，我更看重思路，因此解说会从 Java 语言入手，毕竟这是最大的开发者群体的最基本的共同语言，而且会掺杂 Java 8 和旧版 Java 的例子。我会尽可能地用 Java 语言（或其近亲）来解释函数式编程概念，仅仅用其他语言来演示一些独有的特性。

也许你对 Scala 和 Clojure 一点都不感兴趣，下半辈子能有现在用着的语言就心满意足了，可是你的语言并不会停下来，反而时刻都在变得更加函数式，也径直带着你一起。所以说，现在快来学学函数式编程范式吧，这样，当有一天（不是假如）函数式降临你日常使

用的语言的时候，你才知道如何驾驭它。我们不妨先了解一下，为什么所有的语言都日渐向函数式靠拢。

1.1 范式转变

计算机科学的进步经常是间歇式的，好思路有时搁置数十年后才突然间变成主流。举个例子，第一种面向对象的语言 Simula 67 是 1967 年发明的，可是直到 1983 年诞生的 C++ 终于流行起来以后，面向对象才真正成为主流。很多时候，再优秀的想法也得等待技术基础慢慢成熟。早年 Java 总被认为太慢，内存耗费太高，不适合高性能的应用，如今硬件市场的变迁把它变成了极具吸引力的选择。

函数式编程的发展轨迹与面向对象编程十分相似，它也是诞生在学院里，然后用几十年的时间悄悄浸染了所有的现代编程语言。不过，仅仅在语言里加入一些新语法，并不足以让开发者完全发挥出这种新思维的全部力量。

我们的讨论可以从两种风格的对比开始，尝试分别用传统编程风格（命令式的循环）和函数式特征更明显的方式来解决同一道题目。这道题目出自计算机科学史上的著名事件，是当年 *Communications of the ACM* 杂志“Programming Pearls”专栏的作者 Jon Bentley 向计算机科学先驱 Donald Knuth 提出的挑战。涉猎过文本操作的开发者会很熟悉这道题目：*读入一个文本文件，确定所有单词的使用频率并从高到低排序，打印出所有单词及其频率的排序列表*。对于问题中的词频统计部分，我给出了一个“传统”Java 的解答，见例 1-1。

例 1-1 词频统计的 Java 实现

```
public class Words {
    private Set<String> NON_WORDS = new HashSet<String>() {{
        add("the"); add("and"); add("of"); add("to"); add("a");
        add("i"); add("it"); add("in"); add("or"); add("is");
        add("d"); add("s"); add("as"); add("so"); add("but");
        add("be"); }};

    public Map wordFreq(String words) {
        TreeMap<String, Integer> wordMap = new TreeMap<String, Integer>();
        Matcher m = Pattern.compile("\\w+").matcher(words);
        while (m.find()) {
            String word = m.group().toLowerCase();
            if (! NON_WORDS.contains(word)) {
                if (wordMap.get(word) == null) {
                    wordMap.put(word, 1);
                }
                else {
                    wordMap.put(word, wordMap.get(word) + 1);
                }
            }
        }
        return wordMap;
```