# 内容提要

本书示例代码下载：<https://github.com/oreillymedia/functional_thinking>

本书2015年8月第1版

本书脱离特定的语言特性，关注各种00P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 Fold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。

Clojure是一个在JVM平台运行的动态函数式编程语言，其语法接近于LISP语言，在JVM平台运行的时候，会被编译为JVM的字节码进行运算。

Scala是一门多范式的编程语言，一种类似java的编程语言 [1] ，设计初衷是实现可伸缩的语言 [2] 、并集成面向对象编程和函数式编程的各种特性。

# 为什么

我们用几分钟来想家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

## 范式转变

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向计期科学先驱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。

* 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;

}

}

对于提倡以步进方式处理集合(如例中正则表达式的匹配结果)遍历的语言来说，这是司空见惯的编码风格。

Java 8新增了Stream API和以Lambda块方式实现的高阶函数(后文将会详细介绍)，我们利用这些新的编程手段来改写上面的例子，就得到例1-2.

例1-2词频统计的Java 8实现

private List<String> regexToList(String words, String regex) {

List<String> wordList = new ArrayList<>();

Matcher m = Pattern.compile(regex).matcher(words);

while (m.find())

wordList.add(m.group());

return wordList;

}

public Map< String, Integer > wordFreq(String words) {

Map<String, Integer> wordMap = new TreeMap<>();

regexToList(words, "\\w+").stream()

.map(w -> w.toLowerCase())

.filter(w -> !NON\_WORDS.contains(w))

.forEach(w -> wordMap.put(w, wordMap.getOrDefault(w, 0) + 1));

return wordMap;

}

虽然将命令式风格的例1-1改为对集合进行三次循环遍历(第一遍把所有的单词变成小写，第二遍滤除虚词，第三遍计算词频)也能达成目的，但这种写法的效率会惨不忍睹。例1-1在一个迭代块里完成三项操作，这是牺牲了代码的清晰来换取执行性能。哪怕这种牺牲再稀松平常，我总是不情愿的。

Cljure语言(http://clojure.org/)的发明人Rich Hickey在Strange Loop会议上做过一堂题为“Simple Made Easy”的演讲(<http://www.infoq.com/presentations/Simple-Made-Easy>)，他翻出了一个已经很少用到的老词——“交织”(complect)：穿插缠绕地合为一体，使错综复杂。命令式编程风格常常迫使我们出于性能考虑，把不同的任务交织起来，以便能够用一次循环来完成多个任务。而函数式编程用map()、filter()这些高阶函数把我们解放出来，让我们站在更高的抽象层次上去考虑问题，把问题看得更清楚。后文我们将看到许多函数式思维破解交织现象的例子。

## 1.2跟上语言发展的潮流

如果我们关注各种语言的发展情况就会发现，所有的主流语言都在进行函数式方面的扩充。早走一步的Groovy已经具备了丰富的函数式特性，包括像“记忆”。(memoization，指运行时自动缓存函数返回值的能力)这样的高级特性在内。随着lambda块(也就是高阶函数)被纳入Java 8，Java语言也终于披挂上函数式的武器。Javascript，这种也许算得上使用最为广泛的语言，本身就拥有不少函数式特性。就连最老成持重的C++语言，也在 2011 年版的语言标准里增加了 lambda 块，引人关注的 Boost. Phoenix (http://dwz.cn/phoenix-library)等类库，更是透露出函数式思潮已经对C++语言有了更深入的影响。

不论你用的是Clojure这类新语言，还是日常相伴的老语言，都有可能遇到相关的特性，而只有学会这些新的编程范式，你才能从容地利用它们。我会在第2章讨论如何转变思维，运用这些先进的工具去大展拳脚。

## 1.3把控制权让渡给语言/运行时

在计算机科学短短的发展历史上，有时候会从技术主流分出一些枝杈，有源于实务界的，也有源于学术界的。例如在20世纪90年代个人电脑大发展的时期，第四代编程语言(4GL)也出现了爆发式的流行，涌现了dBASE、Clipper、FoxPro、Paradox等不可胜数的新语言。这些语言的卖点之一是比C、Pascal等第三代语言(3GL)更高层次的抽象。换言之，4GL下的一行命令，3GL可能要用很多行才写得出来，因为4GL自带了更丰富的编程环境。像从磁盘读取流行的数据库格式这样的功能，4GL天生就具备，并不需要使用者特意去实现。

函数式编程也是这样一根横生出来的枝杈，是学术界那些乐于为新思路和新范式寻找表达手段的计算机科学家们的发明，分出来的枝杈偶尔会重新汇入主流，函数式编程当前正好是这种情况。函数式语言不仅在Java虚拟机（JVM）平台上迅速地崭露头角，例如最有代表性的Scala和Clojure语言，.NET平台也不例外，F#已经是堂堂正正的平台一员。那么，为什么所有的平台都在拥抱函数式编程呢?

20世纪80年代早期，我还在上大学的时候，用的编程环境叫作Pecan Pascal。Pecan Pascal的独门绝技是可以在Apple和IBM PC上运行相同的Pascal代码。为了做到这一点，Pecan的工程师祭出了神秘的“字节码”(bytecode)。在编译的时候，开发者写下的Pascal源代码会被编译成这种在“虚拟机”上执行的“字节码”，而“虚拟机”在每一种运行平台上都有专门的原生实现。Pecan Pascal用起来让人痛不欲生、就算最简单的编程习题，编译出来的代码都慢得无法忍受。当时的硬件水平还没有准备好迎接这样的挑战。

Pecan Pascal被淘汰了，但它的架构我们都很熟悉。十年之后Sun发布了采用同样设计的Java，在20世纪90年代中期的硬件环境下勉力取得了成功。Java还带来了其他一些救开发者于水火的特性，自动垃圾收集即是其中之一。我从此再也不想碰那些没有垃圾收集的语言。亲身经历告诉我，最好还是把时间花在更高层次的抽象上，多考虑怎样解决复杂的业务场景，少去费心复杂的底层运作。我为Java纾解了人工管理内存的痛苦而欣喜，同时期冀在别的方面也能找到这样的利器。

人生苦短，远离malloc

随着时间的推移，开发者们越来越多地把乏味单调的任务托付给语言和运行时。对于我日常编写的应用程序类型来说，失去对内存的直接控制没什么可惋惜的，放弃这些反而让我能够专注于更重要的问题。Java接管内存分配减轻了我们的负担，函数式编程语言让我们用高阶抽象从容取代基本的控制结构，也有着同样的意义。

将琐碎的细节交托给运行时，令繁冗的实现化作轻巧，这样的例子本书中比比皆是。

## 1.4 简洁

Working with Legacy Code的作者Michael Feathers用寥寥数语(<https://twitter.com/mfeathers/status/29581296216> )捕捉到了函数式抽象和面向对象抽象的关键区别：

面向对象编程通过封装不确定因素来使代码能被人理解；函数式编程通过尽量减少不确定因素来使代码能被人理解。—— Michael Feathers

请回想一下你熟悉的封装、作用域、可见性等面向对象编程(OOP)构造，这些机制的存在意义，都是为了精细地控制谁能够感知状态和改变状态。而当涉及多线程的时候，对状态的控制就更复杂了。这些机制就属于Michael Feathers所谓的“不确定因素”(moving parts)。大多数函数式语言在这个问题上采取了另一种做法，它们认为，与其建立种种机制来控制可变的状态，不如尽可能消灭可变的状态这个不确定因素。其立论的根据是这样的：假如语言不对外暴露那么多有出错可能的特性，那么开发者就不那么容易犯错。我会提示各种例子来说明函数式编程是怎样消除变量、抽象和其他不确定因素的。

在面向对象的命令式编程语言里面，重用的单元是类和类之间沟通用的消息，这些都可以用类图(class diagram)来表述。这个领域的代表性著作《设计模式：可复用面向对象软件地基础》(Design Patterns:Elements of Reusable Object-Oriented Software, 作者Erich Gamma、Richard Helm、Ralph Johnson、Jonh Vlissides) 就在每一个模式的说明里都附上了至少一幅类图。OOP的世界提倡开发者针对具体问题建立专门的数据结构，相关的专门操作以“方法”的形式附加在数据结构上。函数式编程语言实现重用的思路很不一样。函数式语言提倡在有限的几种关键数据结构(如list、set、map)上运用针对这些数据结构

高度优化过的操作，以此构成基本的运转机构。开发者再根据具体用途，插入自己的数据结构和高阶函数去调整机构的运转方式。

我们来分析下面截取自例1-2的片段

regexToList(words, "\\w+").stream().filter(w -> !NON\_WORDS.contains(w))

这里为了取得列表的一个子集而调用了filter方法，并向filter方法传入已被转换为stream的列表内容，以及定义了筛选条件的高阶函数(即行中裹上了语法糖衣的(w -> !NON\_WORDS.contains(w)))。运转机构高效率地按照指定的条件实行筛选，返回筛选后的列表。

比起一味创建新的类结构体系，把封装的单元降低到函数级别，更有利于达到细粒度的、基础层面的重用。反面例子如Java世界的数十种XML类库，每一种都有自己定义的内部数据结构。相比之下，Clojure就享受到了使用高层次抽象的好处。不久前Clojure库中的map方法经过创造性的重写，获得了自动并行的能力，也就是说，所有Clojure开发者不需要动一行代码，就自动享受到了map操作的性能提升。

函数式程序员喜欢用少数几个核心数据结构，围绕它们去建立一套充分优化的运转机构。面向对象程序员喜欢不断地创建新的数据结构和附属的操作，因为压倒一切的面向对象编程范式就是建立新的类和类间的消息。把所有的数据结构都封装成类，一方面压制了方法层面的重用，另一方面鼓励了大粒度的框架式的重用。函数式编程的程序构造更方便我们在比较细小的层面上重用代码。

# 第2章 转变思维