# 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

学习一门新的编程语言一点都不难，你只要知道怎么把熟悉的概念用新的语法表达出来就行了 。比如说你打算学Javascript，那么第一步会去找份资料，看看Javascript是怎么表达if语句的。通常程序员可以通过套用自己已经在别的语言中掌握的知识来学习新的语言。与之相比，学习一种新的范式是困难的——我们必须学会为熟悉的问题找到新的解答方法。

换用Scala、Clojure之类的函数式编程语言并不是写出函数式代码的必要条件，转变我们看待问题的角度才是必不可少的。

## 2.1普通的例子

当垃圾收集成为主流，一下子将若干难以调试的错误类别连根拔起，程序员也因为“**运行时”**接管了复杂且容易出错的内存管理而获得解脱。函数式编程希望在算法编写上给予程序员同样的帮助，一方面程序员得以在更高的抽象层次上工作，另一方面运行时也有了执行复杂优化的自由空间。开发者从中获得的好处体现在更低的复杂性和更高的性能，这点与垃圾收集相同，不过，函数式编程对个人的影响更直接，因为它改变的是你的解答思路。

### 2.1.1命令式解法

命令式编程是按照“程序是一系列改变状态的命令”来建模的一种编程风格。传统的for循环是命令式风格的绝好例子先确立初始状态，然后每次迭代都执行循环体中的一系列命令。

为了形象说明命令式编程与函数式编程的差异，我会从二个普通的问题和它的命令式解法说起。假设我们有一个名字列表，其中一些条目由单个字符构成。现在的任务是，将除去单字符条目之外的列表内容，放在一个逗号分隔的字符串里返回，且每个名字的首字母都要大写。实现这个算法的Java代码见例2-1.

例2-1典型的公司业务处理例子(Java实现)

public class TheCompanyProcess {

public String cleanNames(List<String> listOfNames) {

StringBuilder result = new StringBuilder();

for(int i = 0; i < listOfNames.size(); i++) {

if (listOfNames.get(i).length() > 1) {

result.append(capitalizeString(listOfNames.get(i))).append(",");

}

}

return result.substring(0, result.length() - 1).toString();

}

public String capitalizeString(String s) {

return s.substring(0, 1).toUpperCase() + s.substring(1, s.length());

}

}

命令式编程鼓励程序员将操作安排在循环内部去执行。本例中我做了三件事：一、filter，筛选列表，去除单字符条目；二、transform，变换列表，使名字的首字母变成大写，三、convert，转换列表，得到一个字符串。这三种操作可以说是我们在列表上施展的"三板斧"。在命令式语言里，这三种操作都必须依赖于相同的低层次机制(对列表进行迭代)，而函数式语言为这些操作提供了针对性的辅助手段。

### 2.1.2函数式解法

函数式编程将程序描述为表达式和变换，以数学方程的形式建立模型，并且尽量避免可变的状态。函数式编程语言对问题的归类不同于命令式语言。如上一小节所列的几种操作(filter、transform、convert)，每一种都作为一个逻辑分类由不同的函数所代表，这些函数实现了低层次的变换，但依赖于开发者定义的高阶函数作为参数来调整其低层次运转机构的运作。于是，上一小节的问题可以概念性地表达为例2-2的伪代码。

* 1. 伪代码表示的“公司业务处理过程”

listOfEmps

-> filter(x.length > 1)

-> transfor(x.capitalize)

-> conver(x + “,” + y)

函数式语言可以帮助我们轻松搭建出上面的概念性解答模型，同时又不必操心各种实现细节。

使用Java 8实现，如果担心 某些元素可能 为null，多加一条检查即可，如下：

return names.stream().filter(name -> name != null)

.filter(name -> name.lenth() > 1)

.map(name -> capitalizeFirstLetter(name))

.collect(Collectors.joining(“,”));

Java 运行时会聪明地将null检查和针对长度的筛选合并成一次操作，这 样既不 妨碍我们把意图清楚，又不损失代码的执行效率。

Groovy语言也能实现，不过命名上更接近Ruby等脚本语言，如下：

public static String cleanUpNames(listOfNames) {

listOfNames.findAll { it.length() > 1 }

.collect{ it.capitalize() }

.join ‘,’

}

Groovy规定用it关键字代表唯一参数，无需定义，当然也可以手动定义参数名。Groovy中的collect方法相当于前面的map方法

我们在更高的抽象层次上做事情，运行时才好去优化低层的细节。编写带垃圾收集的工业级虚拟机实在是一项异常复杂的任务，开发者很乐意交出这方面的职责。另一边的JVM工程师则尽力封装起垃圾收集，让它从开发者的日常考虑事项中消失，大大减轻了开发者的负担。

不要再让那些迭代、变化、化约如何进行的低层细节占据你的思维，多想想哪些问题其实可以归结为这几样基本操作的排列组合吧。

## 2.2 案例研究：求完美数

一个数的真约数（即除了自身以外的所有正约数）之和等于自身的数称为完美数。如果和大于自身则是过剩数，如果小于自身则是不足数，例如下面的是完美数：

6 = 1 + 2 + 3 ，6除这3个数都能整除

28 = 1 + 2 + 4 + 7 + 14，28除这5个数都能整除

### 2.2.1 使用命令式求解完全数

例2-10

/\*\* 完美数计算器 \*/

public class PerfectNumberCalculator {

private int number;

private int factorsSum;

public PerfectNumberCalculator(int number) {

this.number = number;

}

public boolean isPerfect() {

return getFactorsSum() == number;

}

public boolean isAbundant() {

return getFactorsSum() > number;

}

public boolean isDeficient() {

return getFactorsSum() < number;

}

/\*\* 计算约数和 \*/

public int getFactorsSum() {

if (factorsSum == 0) {

for (int factor: getFactors()) factorsSum += factor;

}

return factorsSum;

}

public Set<Integer> getFactors() {

Set<Integer> factors = new HashSet<>();

for (int i = 1; i < number; i++) if (isFactor(i)) factors.add(i);

return factors;

}

/\*\* 判断n是否是number的约数，即能否被number整除 \*/

public boolean isFactor(int n) {

return number % n == 0;

}

}

内部状态，存放要计算的目标数字

内部缓存，防止重复进行不必要的运算

PerfectNumberCalculator类维护着两个内部状态。其中number字段作用是为一系列函数省下一个参数。Cache则是通过一个Map结构来缓存每个数字是否是完美数，以在后续针对同一个数字的调用中更快的返回结果（查表速度与计算速度的差别）。内部状态在面向对象编程的世界里是受到推崇的平常做法，因为封装被OOP语言视为一 项优势。状态划分往往为一些工程实践提供了便利，比如单元测试的时候我们很容易注入各种取值。

上面的代码经过了精心 的组织，划分成很多个小方法。这是测试驱动开发的副产物，不过也正好把算法的各个组成部分都表现了出来。其中一些部分会在后续的改造中逐渐被替换成更加函数式的写法。

### 2.2.2 稍微向函数式靠拢的完美数解法

例2-10用它的代码组织形态反映了可测试性的编程目标。假如我们还希望加上一 个“最小化共享状态”的目标，该怎么做呢？这时可以去掉类的成员变量，改为通过参数来传递需要的值，如例2-11：

例2-11 稍微向函数式靠拢的完美数求解实现

/\*\* 完美数计算器 \*/

public class PerfectNumberCalculator {

public boolean isPerfect(int number) {

return getFactorsSum(getFactors(number)) == number;

}

public boolean isAbundant(int number) {

return getFactorsSum(getFactors(number)) > number;

}

public boolean isDeficient(int number) {

return getFactorsSum(getFactors(number)) < number;

}

/\*\* 计算约数和 \*/

public int getFactorsSum(Collection<Integer> factors) {

int factorsSum = 0;

for (int factor: factors) factorsSum += factor;

return factorsSum;

}

public Set<Integer> getFactors(int number) {

Set<Integer> factors = new HashSet<>();

for (int i = 1; i < number; i++) if (isFactor(i, number)) factors.add(i);

return factors;

}

/\*\* 判断n是否是number的约数，即能否被number整除 \*/

public boolean isFactor(int n, int number) {

return number % n == 0;

}

}

众多方法都加上了number参数，因为没有可以存放它的内部状态。

所有方法都带public static 修饰，因为它们都是纯函数，因此可以在完美数之外的领域使用。

例子目前存在重复计算约数和的操作，效率低，因为没有缓存

在例2-11稍微向函数式风格靠拢，所有方法都是自足的、带public和static作用域的纯函数(即没有副作用的函数)，而由于类里面根本不存在任何内部状态，也就没有理由去“隐藏”任何一个方法。实际上，getFactors方法在很多其他应用中都有潜在的用途，比如用来寻找素数。

一般来说，面向对象系统里粒度最小的重用单元是类，开发者往往忘记了重用可以在更小的单元上发生。例如，例2-11的getFactorsSum方法的参数类型没有选择某一种具体的列表类型，而是定为Collection<lnteger>。一个兼容于所有数字集合的接口，在函数级别上发生重用的可能性自然更大一些。

这一版的实现没有为求和结果设计缓存机制。缓存意味着持续存在的状态，可是这一版的实现根本没有可以放置状态的地方。例2-11对比例2-10相同功能的实现，效率上要低些，这是因为失去了存放求和结果的内部状态，只好每次都重新计算。我们将在第4章借助“记忆”机制，在保持无状态的前提下，把缓存找回来。现在暂且和它告别吧。

### 2.2.3完美数求解的Java 8实现

lambda块是最令Java 8面目一新的改进，它其实就是高阶函数。多了这么一个小功能，传统函数式语言里的一些高层次抽象就一下子向Java开发者敞开了大门。

例2-12完美数求解的Java 8实现

public class PerfectNumberCalculator {

private static IntStream getFactors(int number) {

return IntStream.range(1, number).filter(n -> number % n == 0);

}

private static int getFactorsSum(int number) {

return getFactors(number).sum();

}

private static boolean isPerfect(int number) {

return getFactorsSum(number) == number;

}

public boolean isAbundant(int number) {

return getFactorsSum(number) > number;

}

public boolean isDeficient(int number) {

return getFactorsSum(number) < number;

}

}

物理上把“机械能”分成储蓄起来的“势能”和释放出来的“动能”。在版本8以前的Java，以及它所代表的许多语言里，集合的行为可以比作动能：各种操作都立即求得结果，不存在中间状态。函数式语言里的stream则更像势能，它的操作可以引而不发。被stream储蓄起来的有数据来源(例中的数据来源是range方法)，还有我们对数据设置的各种条件，如例中的筛选操作。只有当程序员通过forEach()、sum()终结操作来向stream要求值结果的时候，才触发从势能到动能的转换。在动能开始释放之前，stream可以作为参数传递并后续附加更多的条件，继续积蓄它的势能。这里关于势能的比喻，用函数式编程的说法叫作缓求值(lazy evaluation)，我们将在第4章详细讨论。

旧版本的Java语言也有可能写出与例2-12风格类似的代码，不过需要克服一点困难，需要用一些框架辅助才行。

### 2.2.4完美数的Functional Java实现

现在高阶函数已经是新一代语言的标准配备，不过仍然有众多组织因为技术之外的原因在未来的很多年里都无法摆脱旧版本的Java运行时，开源框架Functional Java 针对以上版本的Java运行时，以尽可能低的侵入性为代价引入了尽量多的函数式编程手法，例如Functional Java可以通过泛型和匿名内部类，在Java 1.5时代的 JDK上模拟出它所缺少的高阶函数特性。例2-13是借助Functional Java的惯用法来实现完美数，它看上去又和前面的例子有所不同。

例2-13 使用Functional Java 框架实现完美数

import fj.F;

import fj.data.List;

import static fj.data.List.range;

public class NumberClassifier {

public List<Integer> getFactors(final int number) {

return range(1, number)

.filter(new F<Integer, Boolean>() {

public Boolean f(final Integer i) {

return number % i == 0;

}

});

}

public int getFactorsSum(List<Integer> factors) {

return factors.foldLeft(fj.function.Integers.add, 0);

}

public boolean isPerfect(int number) {

return getFactorsSum(getFactors(number)) == number;

}

public boolean isAbundant(int number) {

return getFactorsSum(getFactors(number)) > number;

}

public boolean isDeficient(int number) {

return getFactorsSum(getFactors(number)) < number;

}

}

Functional Java在其List类中提供的foldLeft()方法为getFactorsSum提供了很大的便利，“fold left”（即左折叠操作）的含义是：

1. 用一个操作（或者叫运算）将初始值（例中为0）与列表中的第一个元素结合
2. 继续用同样的操作将第1步的运行结果与下一个元素结合
3. 反复进行直接消耗完列表中的元素

Functional Java 提供了运算所需的高阶函数（例中的Integers.add函数），也由它负责施用。当然，真正的高阶函数要到Java 8才出现，Functional Java也无法在旧版本的Java里实现完整的高阶函数功能，只是用匿名内部类来模拟高阶函数的编程风格。

例2-13使用了foldLeft方法，它依次向左方，即向着第一个元素合并列表。对于满足交换律的加法来说，折叠的方向并不影响结果。万一我们要使用某些结果与折叠次序相关的操作，还有foldRight方法可供选择。

高阶函数消除了摩擦。

你可能会认为Functional Java版本（例2-13）与Java 8版本（例2-12）的区别无非是一些语法糖衣（其实不止）。可是语法上的便利也是很重要的方面，毕竟我们想表达 的意思都要由语法 来承载。

我跟Martin Fowler在巴塞罗那的一辆出租车上有过一次记忆深刻的讨论，我们聊的是Smalltalk的衰落和Java的兴盛。Fowler在这两种语言上都有深厚的积累，他说，起初他觉得从Smalltalk到Java的变化只是一些语法上的不便，结果却发现被阻碍的还有原先语言所承载的思维方式。在语法处处掣肘下塑造出来的抽象，很难配合我们思维过程而不产生无谓的摩擦。

不要增加无谓的摩擦。

## 2.3 具有普遍意义的基本构建单元

### 2.3.1 筛选

Groovy的筛选操作（叫作findAll()）

Static def getFactors(number) {

(1..number).findAll {number % it == 0};

}

### 2.3.2 映射

映射(map vt. 映射)

Groovy中没有用map()的名字，而是叫作collect()，如下显示了Groovy版本的查找约数的算法：

static def getFactors(number) {

def factors = (1..round(sqrt(number)).findAll({number % it == 0}) 如28：这里的约数为：1 2 4

(factors + factors.collect {number / it}).unique() 如28：这里的约数为：7 14 28，拿28除以上面的1 2 4得出，所以这里两个约数列表进行了相加

}

最后调用了unique()方法来消除列表中的重复项，确保完全平方数的平方根（如16的平方根4）不会在列表中出现两次。

### 2.3.3 折叠/化约

这种函数的名称最多样，而且在几种流行语言里的实现各有微妙的区别。foldLeft和reduce都是catamorphism这种范畴论的态射概念具体应用到列表操作上面的变体 ，catamorphism是对列表“折叠”(fold)概念的推广。

reduce和fold操作在功能上大致重合，但根据具体的编程语言而有微妙的区别。两者都用一个累积量（accumulator）来“收集”集合元素。Reuduce函数一般在需要为累积量设定一个初始值 的时候使用，而fold起始的时候累积量是空的。

在纯函数式语言里，左折叠foldLeft和右折叠foldRight的 实现并不相同。例如右折叠允许操作无限长度的列表，而左折叠则不允许。

化约（reduce）操作在Groovy版完美数的用法如下：

例2-21 Groovy版的reduce()叫作inject()

static def getFactorsSum(number) {

getFactors(number).inject(0, {i, j -> i + j})

}

Groovy的inject方法与列2-18中出现的reduce函数有着相同的签名；第一个参数都是初始值，第二个参数都是接受两个参数，返回一个值的闭包。例中我们给闭包传的两个参数是 i和j。

fold和reduce常常用在需要从一个集合处理产生另一个大小不同（通常较小但不必然）的集合或单一值的情况。

需要把集合分成一小块一小块来处理的时候，用reduce或fold。

## 2.4 函数的同义异名问题

作为函数式编程语言的共同特征，我们可以在每一种语言里找到同样的几大类基本函数。然而当开发者从一种语言换到另一种的时候往往不太顺利，原因就是熟悉的函数突然换了个不认识的名字。继承函数式传统的语言喜欢按照范式术语来命名基本函数，而出自脚本语言背景的则更喜欢使用描述性的名字(有时候还会起多个名字，实质是指向相同函数的别名)。

### 2.4.1筛选

筛选函数将用户(通常以高阶函数的形式)给定的布尔逻辑作用于集合，返回由原集合中符合条件的元素组成的一个子集。筛选操作与査找(find)函数的关系很密切，査找函数返回的是集合中第一个符合条件的元素。

1. Scala

Scala提供了好几种形式的筛选。最简单的一种是在列表上按传人的条件进行筛选。下面的例子首先创建一个数字列表，然后对列表使用filter函数，并在传给函数的代码块中设置筛选条件为可被3整除的元素

val numbers = List.range(1, 11)

numbers filter (x => x % 3.== 0)

Scala的第二种筛选形式是partition()函数，可以把集合分成两个集合，如下：

Numbers partition(\_ % 3 == 0)，则能被3整除的放一个集合，不能被整除的放另一个集合

filter()函数返回所有匹配元素的集合，而find()只返回第一个匹配项：

numbers find(\_ %3 == 0)

takeWhile()函数从集合头部开始，一直取到第一个不满足断言的元素：

List(1, 2, 3, -4, 5 , 6 , 7 ) takeWhile(\_ > 0) // List(1, 2, 3)

2. Groovy

Groovy一般不被看作一种函数式语言，但它具备很多函数式的范式，只是命名上往往带有脚本语言的色彩。例如按照函数式语言的传统叫作filter的函数在Groovy中对应的是findAll（）.

Groovy也有跟partition（）对应的函数，叫作split()：

(1..10).split {it % 3} // [[1, 2, 4, 5, 7, 8, 10], [3, 6 ,9]]

Groovy的find：(1..10).find {it % 3== 0} // 3，返回的是第一个符合条件的元素

Groovy也有takeWhile()和dropWhile()方法，其语义和Scala差不多：

[1, 2, 3, -4, 5, 6, 7, 8].takeWhile{it > 0} // [1, 2, 3] 从头开始取子集，一直取到元素不符合条件，也就是要扔掉后面的。

def words = [“the”, “quick”, “brown”, “fox”, “jumped”, “over”, “the”, “lazy”, “dog”]

words.dropWhile{it.startsWith(“t”)} // [ “quick”, “brown”, “fox”, “jumped”, “over”, “the”, “lazy”, “dog”] 取子集，从前面开始过滤，如果是t开头就不要（如果前面 有连续多个t开头的，则这多个都不要），直到遇到不是t开头的 ，则后面的全部子集都要，即使后面也有包含有t开头的，也就是要扔掉前面的。

### 2.4.2 映射

map操作在Groovy中对应的是collect()函数，示例：(1..5).collect{it += 1} // [2, 3, 4, 5, 6]

Groovy中类似flatMap()的函数：flatten()，用来消除嵌套结构：[[1,2], [3,4,5], [6]].flatten() // [1, 2, 3, 4, 5, 6]

### 2.4.3 折叠/化约

Groovy提供了两个版本的inject()来完成化约操作，分别对应于Scala众多同类方法中的reduce()和foldLeft()。示例如下：

(1..10).inject{a, b -> a + b} // 55

也可以使用带初始值的版本：

(1..10).inject(0, {a, b -> a + b}) // 55

Groovy的函数式类库远不如Scala和Clojure丰富。这一点并不奇怪，毕竟Groovy的定位是一种多范式语言，并不 特别强调函数式编程能力。

# 第3章 权责让渡

## 3.1 迭代让位于高阶函数

## 3.2 闭包

闭包(closure)是所有函数式语言都具备的一项平常特性，可是相关的论述却常常充斥着晦涩乃至神秘的字眼。所谓闭包，实际上是一种特殊的函数，它在暗地里绑定了函数内部引用的所有变量。换句话说，这种函数(或方法)把它引用的所有东西都放在一个上下文里“包”了起来。

例3-1是一个简单的例子，这段Groovy代码演示了闭包的创建和绑定。

例3-1Groovy语言中闭包绑定的简单示例

class Employee {

def name, salary

}

def paidMore (amount) {

return {Employee e -> e. salary > amount)

}

isHighPaid = paidMore(100000)

例3-1首先定义了一个简单的Employee类，类中带有两个字段。接着定义带有amount参数的paidMore函数，其返回值是一个以Employee实例为参数的代码块，或者叫闭包，类型声明Employee可写可不写，这里写出来顺便起到文档的作用。接下来，我们给代码块传入参数值100 000，并赋予isHighPaid的名称，于是数值100 000就随着这一步赋值操作，永久地和代码块绑定在一起了。以后有员工数据被代入这个代码块求解的时候，它就可以拿绑定的数值作为标准去评判员工的工资高低。

例 3-2 执行闭包

def Smithers = new Employee(name:”Fred”, salary:120000)

def Homer = new Employee(name:”Homer”, salary:80000)

println isHightPaid(Smithers)

println isHightPaid(Homer)

// true, false

例3-2创建了两笔员工数据，然后判断其工资是否达到标准线。闭包在生成的时候，会把引用的变量全部圈到代码块的作用域里，封闭、包围起来(故名闭包)。闭包的每个实例都保有自己的一份变量取值，包括私有变量也是如此。也就是说，我们可以创建paidMore闭包的另一个实例，给它绑定另外的数值(当然实例的名字也要另取)，如例3-3所示。

例3-3绑定另一个闭包

isHigherPaid = paidMore (200000)

println isHigherPaid (Snithers)

println isHigherPaid (Homer)

def Burns = new Employee(name:"Monty", salaryl: 1000000)

println isHigherPaid (Burns)

//false, false, true

闭包经常被函数式语言和框架当作一种异地执行的机制，用来传递待执行的变换代码，如map之类的高阶函数。在缺乏闭包特性的旧版Java平台上，Functional Java利用匿名内部类来模仿“真正的”闭包的某些行为，但语言的先天不足导致这种模仿是不彻底的。闭包的执行机制究竟有什么玄机?

我们用一个例子来说明闭包的特殊之处，请看例3-4.

例3-4闭包的原理(Groovy示例)

def Closure makeCounter() {

def local\_variable = 0;

return { return local\_variable +=1 } //

}

c1 = makeCounter() //

c1() //

c1()

c1()

c2 = makeCounter() //

println “C1 = ${c1()}, c2 = ${c2()}” // output：c1 = 4, c2 = 1

函数的返回值是一个代码块，而不是一个值。

c1现在指向代码块的一个实例。

调用c1将递增其内部变量，如果这个时候输出，其结果会是1.

c2现在指向makeCounter()的一个全新实例，与其他实例没有关联。

每个实例的内部状态都是独立的，各自拥有一份local\_variable

makeCounter函数首先定义一个局部变量，明白无误地命名为local\_variable，接着返回一个使用了该局部变量的代码块。注意makeCounter函数的返回类型是Closure，而不是一个单纯的值。代码块的工作仅仅是递增并返回其局部变量的值。方法中两次明确写出了return关键字，其实这两个地方Groovy都允许省略，不过那样的话，代码看起来就有些晦涩了。

为了演示makeCounter函数的用法，我们给代码块分配了一个变量名c1，然后调用了三次。调用代码块的时候用到了Groovy提供的语法糖衣，也就是在代码块变量名后直接跟一对圆括号的写法(否则应该写成c1.call())。接下来，我们第二次调用了角makeCounter，将返回的又一个代码块实例赋给变量c2。最后我们把c1和c2都调用了一次。从运行的结果可以看出来，两个代码块实例都分别持有自己的一份local-variable变量。“闭包”这个名字来源于它创建封闭上下文的行为。虽然局部变量不是在代码块里面定义的，但只要代码块引用了该变量，两者就被绑定在一起，这种联系在代码块实例的全部生命期内都一直保持着。

从实现的角度来说，代码块实例从它被创建的一刻起，就持有其作用域内一切事物的封闭副本，如例3-4的local-variable。当代码块实例被垃圾收集的时候，它持有的引用也同时被回收。

像例3-4那样创建一个闭包仅仅为了修改自身的内部状态，不是值得提倡的闭包用法，我们这样写只是为了演示闭包绑定的运作原理，更常见的用法是绑定常量或者不可变的值(如例3-1)。

在Java 8以前版本的Java语言，或者任意一种支持函数而不支持闭包的语言里面，我们最多能模拟到例3-5的程度。

例3-5 Java版的makeCounter()

class Counter {

public int varField;

Counter(int var) {

varField = var;

}

public static Counter makeCounter() {

return new Counter(0);

}

public int execute() {

return ++varField;

}

}

Counter类还可以有别的一些写法(比如写成匿名的、泛型的，等等)，但不管怎么做，都避免不了要自己去管理状态。闭包在这里表现出来的函数式思维就是“让运行时去管理状态”，比起自己硬着头皮去处理字段创建、呵护状态(包括经受多线程环境的严酷考验)这些繁琐的事务，还不如交出对状态的控制权，让语言和框架悄悄在背后帮我们管理好。

让语言去管理状态。

闭包还是**推迟执行**原则的绝佳样板。我们把代码绑定到闲包之后，可以推迟到适当的时机再执行闭包。这个特点在很多场合都能发挥作用。例如必要的变量和函数可能并不在定义时的作用域里，要到执行的时候才准备好。那么我们把执行上下文放在闭包里保留起来，就可以等到正确的时机再完成执行。

命令式语言围绕状态来建立编程模型，参数传递是其典型特征。闭包作为一种对行为的建模手段，让我们把代码和上下文同时封装在单一结构，也就是闭包本身里面，像传统数据结构一样可以传递到其他位置，然后在恰当的时间和地点完成执行。

抓住上下文，而非状态。

## 3.3柯里化和函数的部分施用

柯里化(currying)和函数的部分施用(partial application)都是从数学里借用过来的编程语言技法(基于20世纪Haskell Curry等数学家的研究成果)。这两种技法以不同的面目出现在各种类型的语言里，在函数式语言当中尤为普遍。柯里化和部分施用都有能力操纵函数或方法的参数数目，一般是通过向一部分参数代入一个或多个默认值的办法来实现的(这部分参数被称为“固定参数”)。大多数函数式语言都具备柯里化和部分施用这两种特性，但实现上各有各的做法。

### 3.3.1定义与辨析

乍看起来，柯里化和部分施用的使用效果是一样的。两者都可以创建有一部分预设参数值的函数。

柯里化指的是从一个多参数函数变成一连串单参数函数的变换。它描述的是变换的过程不涉及变换之后对函数的调用。调用者可以决定对多少个参数实施变换，余下的部分将衍生为一个参数数目较少的新函数。

部分施用指通过提前代入一部分参数值，使一个多参数函数得以省略部分参数，从而转化为一个参数数目较少的函数。这种技法叫作“部分施用”，顾名思义，就是让函数先作用于其中一些参数，经过部分的求解，结果返回一个由余下参数构成签名的函数。

柯里化和部分施用都是在我们提供部分参数值之后，产出可以凭余下参数实施调用的一个函数。不同的地方在于，函数柯里化的结果是返回链条中的下一个函数，而部分施用是把参数的取值绑定到用户在操作中提供的具体值上，因而产生一个“元数”(参数的数目)较少的函数。用元数大于二的函数来套一下这里的解释，它们之间的区别就会比较清楚了。

举个例子，函数process(x,y,z)完全柯里化之后将变成process(x)(y)(z)的形式，其中process(x)和process(x)(y)都是单参数的函数。如果只对第一个参数柯里化，那么process(x)的返回值将是一个单参数的函数，而这个唯一的参数又接受另一个参数的输入。而部分施用的结果直接是一个减少了元数的函数。如果在process(x,y,z)上部分施用一个参数，那么我们将得到还剩下两个参数的函数process(y,z)。

这两种技法的区分很重要而且很容易被错误地理解，可是使用中它们偏偏又经常得到相同的结果。这里还有更加添乱的事情，Groovy实现了部分施用也实现了柯里化，但是它把两者都叫作柯里化。Scala既有部分施用函数(partially applied function)，又有名称相近的偏函数类PartialFunction，可它们是截然不同的两个概念。

### 3.3.2 Groovy的情况

Groovy通过curry()函数实现柯里化，这个函数来自Closure类。

例3-6 Groovy语言中的柯里化

def product = { x, y-> x \* y}

def quadrate = product. curry(4)

def octate = product. curry (8)

println"4x4: ${quadrate. call(4)}"

prtntln"8×5: ${octate(5)}"

调用curry来固定一个参数，返回结果是一个单参数的函数。

octate函数总是对传入的参数乘以8.

quadrate是一个单参数的函数，可以通过Closure类的call方法来调用它。

Groory提供了一层语法糖衣，可以让调用语句的写法更自然一些。

例3-6首先定义接受两个参数的代码块product。我们利用Groovy内建的curry方法，在product的基础上构造出两个新的代码块，quadrate和octate。Groovy为调用代码块提供了特别的便利，我们既可以显式执行call()方法，也可以使用Groovy在语言层面提供的语法糖衣，也就是在代码块的名称后紧跟一对圆括号，参数则写在括号里(如例中octate(5)的写法)。

curry()虽然叫这个名字，它在背后对代码块所做的事情其实属于函数的部分施用，尽管名不副实，但用它来模拟出柯里化的效果还是可行的，做法是通过连续的部分施用使函数变形为一连串单参数的函数，如例3-7所示。

例3-7 Groovy语言中部分施用与柯里化的对比

def volume = {h, w, l -> h \* w \* l)

def area = volume.curry(1)

def lengthPA = volume.curry(1, 1)

def lengthC = volume.curry(1). curry (1)

println "参数取值为2×3×4的长方体，体积为${volume(2,3,4)}"

println "参数取值为3×4的长方形，面积为${area(3,4)} "。

println "参数取值为6的线段，长度为${lengthPA(6)}"

println "参数取值为6的线段，经柯里化函数求得的长度为${lengthC(6)}"

部分施用。

柯里化。

例3-7中volune代码块的作用是按照公式计算长方体的体积。接着我们固定长方体的第一维(即代表高度的参数h)，令其取值为1，从而构造出第二个代码块area(作用是计算长方形的面积)。如果我们继续以volume为基础构造计算线段长度的代码块，那么无论使用部分施用还是柯里化的技法都能完成任务。lengthPA通过部分施用将前两个参数都固定为1。lengthC连续做了两次柯里化，最后算得与lengthPA相同的结果。两种写法只有微妙的区别，最终的计算结果也完全相同，但如果你在一名函数式程序员面前不加区分地使用这两个名词，他一定会纠正你。很不幸，Groovy把这两个密切相关的概念混为一谈了，函数式编程赋予我们另一套新的构造单元，代替以往命令式语言所使用的机制来完成相同的目标。这些构造单元之间的关系经过了细致的安排。复合(composition)，是函数式语言拼组这些构造单元的一般方式，这方面的详细讨论放在第6章。请看例3-8的Groovy代码。

例3-8 Groovy语言中函数的复合

def composite = {f, g, x ->return f(g(x)) }

def thirtyTwoer = composite.curry(quadrate, octate)

println “composition of curried functions yields ${thirtyTwoer(2)}”

例3-8定义了一个复合的代码块，由两个函数构成，或者更准确地说，是在一个函数的返回值上调用另一个函数。然后我们利用它来构造thirtyTwoer代码块，其中运用了部分施用的手法来组合quadrate和octate两个函数

### 3.3.3 Clojure的情况

### 3.3.4 Scala的情况

### 3.3.5 一般用途

尽管有着微妙定义和繁琐的实现，但柯里化和部分施用都在现实的 编程世界中 拥有一席之地。

#### 1、函数工厂

我们在传统面向对象编程中会用到工厂方法的场合，正适合柯里化(以及部分施用)表现它的才干。我们可以用一个Groovy实现的简单加法函数来说明问题，请看例3-17.

例3-17实现的加法函数和递增函数

def adder = { x,y -> x + y}

def incrementer = adder.curry (1)

println"increment 7: ${ incrementer(7)}"// 8

例中从adder函数派生出了incrementer函数。

#### 2、Template Method 模式

GoF模式集里面有一项Template Method(模板方法)模式。其用意是在固定的算法框架内部安排一些抽象方法，为后续的具体实现保留一部分灵活性。部分施用和柯里化也可以起到相同的作用。部分施用技法注入当前已经确定的行为，留下未确定的参数给具体实现去发挥，其思路与模板方法这种面向对象的设计模式如出一辙。

本书第6章将会用一个例子来说明，若干设计模式(包括模板方法在内)怎样因为部分施用和其他函数式技法而失去了存在意义。

#### 3、隐含参数

当我们需要频繁调用一个函数，而每次的参数值都差不多的时候，可以运用柯里化来设置隐含参数。举个例子，我们在操作持久化框架的时候，每次都要在第一个参数里写上数据源的位置。而经过部分施用以后，我们就不需要反复地写出这个参数值,如例3-18所示。

例3-18运用部分施用技法设置隐含参数值

(defn db-connect [data-source query params]

…)

(def dbc (partial db-connect "db/some-data-source"))

(dbc "select \* from %1" "cust")

例3-18的dbc函数在操作数据的时候不需要再提供数据源，数据源已经自动设置好了，面向对象编程中“封装”概念的本质，也就是魔术般出现在每个函数里的隐含上下文this，我们可以在函数式编程中加以模拟，用柯里化的方式把this传递给所有的函数，让this在使用者的面前隐藏起来。

## 3.4递归

## 3.5 Stream和作业顺序重排

从命令式风格转变为函数式风格还有一个潜在的好处，那就是运行时有能力在涉及效率的问题上替我们做决定。

我们可以把第2章用过的“公司业务过程”例子再拿出来看一遍，其Java 8实现如例3-24所示，其中只做了一点微小的改动。

例3-24公司业务过程的Java 8实现

public String cleanNames(List<String> names) {

if (names == null) return "";

return names

.stream()

.map(e -> capitalize(e))

.filter(n -> n.length() > 1)

.collect(joining(","));

}

眼尖的读者会注意到我在这一版的cleanNames()里面调换了操作的顺序(与第2章的例2-4相比)，map操作被提到了filter的前面。按照命令式的思路，我们本能地就会把筛选操作放在映射操作的前面，这样map需要操作的列表会比较小，可以减少工作量。但是实际上很多函数式语言(包括Java 8乃至Functional Java框架)都提供了Stream抽象。Stream很多方面的行为都与集合相似，但它不像集合那样事先就备妥所有的值，而是需要的时候才让数据从源头“流”向目的地。例3-24的数据源头是names集合，最终目的地(或者叫终结操作)是collect()。处在中间的map和filter都是缓求值(lazy)的操作，它们会被尽量地推迟执行。实际上在下游的终结操作“发出要求”之前，它们都不会产生任何结果。

聪明的运行时会替我们重新安排缓求值操作的执行顺序。例3-24将在运行时的主持下调换其缓求值操作的顺序，让筛选操作先于映射操作执行，以取得最佳的运算效率。在使用Java平台上的各种函数式方案的时候，我们必须保证传给filter等函数的lambda块不存在副作用，否则可能导致无法预料的结果。

允许运行时发挥其优化能力的做法，再次印证了我们关于交出控制权的观点：放弃对繁琐细节的掌控，关注问题域，而非关注问题域的实现。

我们会在第4章继续探讨缓求值的问题，Java 8的stream特性则放在第7章讨论。

通过实验证明，上面说的并不对，运行时并不 会把filter先执行，而是对于每一个元素，先执行一下map再执行一下filter。也就是说每一个元素都会执行到map操作和filter操作。先把一个元素开头大写，然后执行过滤，再把下一个元素开头大写，再执行过滤。。。

# 第4章 用七不用蛮

## 4.1 记忆

“memoization”这个词是英国的人工智能研究者Donald Michie生造出来的，指的是在函数级别上对需要多次使用的值进行缓存的机制。目前来说，函数式编程语言普遍都支持记忆特性，有些是直接内建在语言里，也有一些需要开发者自行实现，但实现起来相对容易。

记忆可以用在这样的场合。假设我们有一个反复调用的函数，需要挖掘它的性能潜力。增加一个内部缓存是很容易想到的方案。每次我们根据一组特定参数求得结果之后，就用参数值做査找用的键，把结果缓存起来。以后当函数又遇到相同参数的时候，就不需要重新计算一遍了，可以直接返回缓存的结果。这种缓存函数计算结果的做法，是计算机科学里一种典型的折衷方案：用更多的内存(我们一般不缺内存)去换取长期来说更高的效率。

只有纯(pure)函数才可以适用缓存技术。纯函数是没有副作用的函数：它不引用其他值可变的类字段，除返回值之外不设置其他的变量，其结果完全由输入参数决定。java.lang.Math类里面的方法都是纯函数的绝好例子。很显然，只有在函数对同样一组参数总是返回相同结果的前提下，我们才可以放心地使用缓存起来的结果。

### 4.1.2 引入“记忆”

## 4.2 缓求值

# 第5章 演化的语言