Web 开发经典丛书

Clojure 高级编程

Jeremy Anderson
Michael Gaare
[美] Justin Holguín 著
Nick Bailey
Timothy Pratley
蒋 楠 译

清华大学出版社

北 京

Jeremy Anderson, Michael Gaare, Justin Holguín, Nick Bailey, Timothy Pratley
Professional Clojure
EISBN: 978-1-119-26727-0

图书在版编目(CIP)数据

Clojure 高级编程/(美)杰里米 • 安德森(Jeremy Anderson) 等著；蒋楠 译. —北京：清华大学出版社，2017
(Web 开发经典丛书)
书名原文：Professional Clojure
ISBN 978-7-302-47111-0

Ⅰ. ①C…　Ⅱ. ①杰… ②蒋…　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 103623 号

责任编辑：王　军　韩宏志
装帧设计：孔祥峰
责任校对：曹　阳
责任印制：王静怡

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

印 装 者：三河市金元印装有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：16.75　　**字　　数**：397 千字
版　　次：2017 年 6 月第 1 版　　**印　　次**：2017 年 6 月第 1 次印刷
印　　数：1~3000
定　　价：49.80 元

产品编号：072433-01

译 者 序

作为一门植根于 JVM 平台的函数式编程语言，Clojure 近年来受到越来越多的关注。无论 C++11 还是 Java 8，都提供针对函数式编程的语法支持。由于不存在可变状态，函数是引用透明的，没有副作用，程序代码易于推理和调试。

无论学习哪种语言，“它能做什么”是不少用户都会问的问题。全球零售业巨头沃尔玛采用 Clojure 开发数据管理系统，自动化运维管理服务提供商 Puppet 采用 Clojure 创建 Trapperkeeper 框架，一站式后端云服务提供商 LeanCloud 采用 Clojure 构建存储 API、推送等核心功能。根据 Clojure 官方网站的统计，目前已有花旗银行、Facebook、LINE、Netfix、Red Hat 等近 300 家企业在自己的系统中采用了 Clojure 或 ClojureScript。此外，在空前重视人工智能的今天，使用 Clojure 进行数据挖掘也是一个热门话题。

任何一门语言都有其拥趸和质疑者，“PHP 是最好的语言”也并非总是笑谈。Clojure 的优点是简洁、专注和实用，但由于 JVM 较慢的启动速度，Clojure 在处理日常事务时或许不那么得心应手。加之 Clojure 和 JVM 平台的绑定是如此之深，要想真正掌握 Clojure，就需要深入理解 Java，这在一定程度上提高了 Clojure 的学习成本。

摆在读者面前的这本书并不厚，却凝结了业内多位专家的宝贵经验。作者尝试以简洁的语言和丰富的示例阐述 Clojure 的独到之处，并培养读者从函数式编程的角度思考问题。不过，本书的目标群体是具有一定经验的开发人员，缺少相关背景的读者可先参阅其他资料，以夯实基础。无论官方文档、论坛博客、主题演讲还是 GitHub，都是很好的学习资源。“一万小时定律”或许并非放之四海而皆准，但唯有不断实践，才能真正掌握一门语言。

非常感谢清华大学出版社的王军老师给予译者翻译本书的机会，他为本书中文版面世所付出的辛勤劳动令译者十分感动。由于译者水平有限，疏漏之处在所难免，恳请读者批评指正，也希望读者提出宝贵意见和建议。

蒋 楠

作 者 简 介

Jeremy Anderson 就职于美国密歇根州的 Code Adept，这是一家提供高品质软件交付的咨询公司，业务涵盖软件开发、敏捷教导与培训服务。Jeremy 是一名 Clojure 爱好者，对多种 Clojure 库的开发都有贡献。Jeremy 对向用户提供编程培训极为热心，并作为志愿者在当地中学协助讲授计算机课程。

Michael Gaare 就职于美国一家提供金融技术服务的初创公司 NextAngles，担任平台技术负责人。从 2012 年起，Michael 就采用 Clojure 开发专业的 Web 服务、数据处理系统与各种库(而非框架)。Michael 爱好参加歌剧演出，大部分闲暇时间都与妻子和两个女儿度过。

Justin Holguín 在美国波特兰的 Puppet Labs 担任软件工程师，负责 Clojure 后端服务的开发。Justin 热爱函数式编程，对高级类型系统、基于属性的测试等能够提高软件稳定性的技术情有独钟。

Nick Bailey 是一名 Clojure 爱好者，也负责 Clojure java.jmx 库的维护。Nick 在总部位于美国加州的 DataStax 担任软件架构师，使用 Clojure 开发用于管理分布式数据库的企业级软件。Nick 从 2010 年起开始接触 Clojure，并由此成为这门语言的拥护者。

Timothy Pratley 从 2008 年起开始使用 Clojure，是这门语言的贡献者和倡导者。Timothy 目前就职于美国旧金山的 Outpace Systems，负责开发基于 Clojure、ClojureScript 和 Clojure Android 的解决方案。Timothy 已有 15 年的专业软件开发经验，接触过许多编程语言、框架和数据库，热爱 Clojure、Datomic 数据库、结对编程(pair programming)[1]，喜欢思考。

[1]一种敏捷软件开发方法，两名程序员在一台计算机上共同工作，一人负责输入代码，另一人负责审查代码。——译者注

技术编辑简介

Justin Smith 是一名活跃于 Clojure 社区的全职 Clojure 开发人员，他将全部精力都投入到 Clojure 开发中。

Zubair Quraishi 来自丹麦，是一名 UX/设计师与营销黑客。在过去 20 年中，Zubair 建立的初创公司已有两家被收购，并投资了超过 30 个初创团队。Zubair 从 2011 年起开始接触 Clojure 和 ClojureScript。Zubair 曾在美国、欧洲、亚洲的多家初创公司和财富 500 强企业工作，他的博客是 www.zubairquraishi.com。

Alex Ott 就职于德国帕德博恩的 Intel Security(其前身为 McAfee)，担任软件架构师，负责信息安全方面的工作。从 2009 年 Clojure 1.0 版发布起，Alex 就使用这门语言开发多种原型、内部服务与开源项目(如 Incanter)。

Doug Knight 具有 18 年的专业编程经验，是 Microsoft 技术方面的专家。2014 年，Doug 加入总部位于美国华盛顿的 LivingSocial，开始接触 Ruby on Rails。从 2015 年起，Doug 开始在工作中使用 Clojure 进行开发。

致　　谢

首先，Jeremy 感谢上帝赐予他能随心所欲做自己喜欢的事情的权利。其次，Jeremy 感谢家人对他的支持和理解，使他可以在晚上和周末将自己关在办公室里疯狂写作。Jeremy 还要感谢 Christina Rudloff 和 Troy Mott 在本书筹备过程中所做的辛勤工作，以及邀请他加入写作团队。正是由于所有作者的共同努力，本书才得以面世。最后，感谢所有技术审稿人在百忙之中抽出时间阅读书稿，并提出宝贵的意见和建议。

Nick 感谢所有为本书面世付出努力的人士：负责组织工作的 Troy 和 Christina、其他撰写和审校的作者以及提供宝贵反馈的技术审稿人。Nick 还要感谢 DataStax，使他有机会撰写这样一部专业的 Clojure 著作。

Michael 感谢 Lara、Charlotte 与 Juliette 对他的关爱、支持和理解，以及 Keith 提供的宝贵帮助。Michael 还要感谢 Christina 和 Troy 的耐心，并给予他撰写本书的机会。此外，感谢 Rich 为写作提供的有趣素材，与他合作实乃人生快事。

Justin 感谢家人对他的引导以及莫大的帮助，鼓励他投身写作和编程之中。Justin 还要感谢他在 Puppet Labs 的诸多好友和同事，他们的鼓舞让 Justin 坚定了学习 Clojure 的信心，通过不断努力，最终成为这门语言的专家。

Timothy 感谢最终合作伙伴 Shin Nee。此外，还要感谢读者对编程发展做出的贡献，以及 Clojure 社区创造了这样一个友善、有益和愉悦的生态系统。最后，Timothy 感谢父母为他的人生提供了诸多机会。

源代码

在读者学习本书中的示例时，可以手工输入所有的代码，也可以使用本书附带的源代码文件。本书使用的所有源代码都可从本书合作站点 http://www.wrox.com/。登录站点 http://www.wrox.com/，使用 Search 工具或使用书名列表就可以找到本书。接着单击本书细目页面上的 Download Code 链接，就可以获得所有的源代码。还可以直接访问 www.wiley.com/go/professionalclojure 或 https://github.com/backstopmedia/clojurebook 下载本书源代码。也可以访问 www.tupwk.com.cn/downpage，输入本书中文书名或中文 ISBN，下载各章的源代码。此外，可扫描本书封底的二维码，直接下载。

注意：

由于许多图书的标题都很类似，所以按 ISBN 搜索是最简单的，本书英文版的 ISBN 是 978-1-119-26727-0。

下载代码后，只需要用自己喜欢的解压缩软件对它进行解压缩即可。另外，也可以进入 http://www.wrox.com/dynamic/books/download.aspx 上的 Wrox 代码下载主页，查看本书和其他 Wrox 图书的所有代码。

目　录

第 1 章

保持初学者的心态

本章内容

- 理解命令式编程和函数式编程的区别
- 学习从函数式编程的角度思考问题
- 介绍 Clojure 对待面向对象编程的独特视角

若保持通透的头脑，就能随时准备好去接受，并对所有的可能性敞开大门。初学者的脑中总是存在无限可能，而专家的脑中却只有零星几个。

——铃木俊隆[1]

对于在过去 30 年中流行的大部分编程语言来说，它们的共通之处超过了相互之间的差异。事实上，一旦掌握了一门语言，学习另一门语言就并非难事。开发人员只需要掌握两门语言在语法上的细微差别，并了解前一门语言所不具备的某些新特性。由于当今不少最流行的语言存在诸多相似之处，一名“掌握多门语言的程序员”并不鲜见。

然而，Clojure 与大部分目前流行的语言完全不同，它从 Lisp 编程语言发展而来。与基于 C 的语言相比，Clojure 的语法和编程风格大相径庭。读者必须舍弃“所有语言都是相通的”这一先入为主的观念，才能更好地学习 Clojure 以及 Lisp 家族的其他语言。

请读者忘记所有先前的编程经验，将 Clojure 当作入门的第一门编程语言。唯有如此，才能领会这门语言的精髓。否则，读者只是学习了若干新语法，写出来的 Clojure 代码风格将偏向 Java/C/Ruby，而背离了 Clojure 原本的设计思想。学习 Clojure/Lisp 甚至将影响读者使用其他语言的方式，特别是在 Java 8 和 Scala 变得越来越流行的今天。

---

[1]铃木俊隆(1905-1971)，将禅宗思想介绍到西方的日本僧人，全球最有影响力的禅宗大师之一。其著作《禅者的初心》(*Zen Mind, Beginner's Mind*)备受苹果公司前 CEO 史蒂夫·乔布斯的推崇。——译者注

1.1 函数式思维

C、C++、C#、Java、Python、Ruby 乃至 Perl，这些语言的语法都极为类似。它们使用相同的构造，强调命令式编程风格。这种风格非常适合在冯·诺依曼体系结构中使用，因为上述语言均以此为基础设计并执行。从 C 语言中可以很明显地看出这一点：用户负责为变量分配和释放内存，并直接在内存中操作指针。其他命令式语言试图隐藏这种复杂性，也取得了不同程度的成功。

在计算机科学中，命令式编程(imperative programming)是一种使用语句来改变程序状态的编程范式。

在过去几十年中，由于类 C 风格的程序设计思想能很好地适应主流硬件结构体系，因此它占据了主导地位。一直以来，程序都能高效执行并利用内存。不过，这种高效是以复杂的语义和语法作为代价的。由于程序过度依赖执行时的内存状态，用户也越来越难以推断执行结果，导致并发处理极为困难且易于出错。近年来，随着内存价格的降低和多核架构的广泛应用，命令式编程的不足已逐渐显露出来。

函数式编程(functional programming)则有所不同，它基于数学概念而非任何给定的计算架构。虽然可将具有 Lisp 血统的 Clojure 称为通用语言，但 Clojure 的确具备大量函数式特性，能很好地支持函数式编程。与命令式语言相比，Clojure 不仅语义更简单，语法也非常简洁。如果没有 Lisp 的背景，阅读和理解 Clojure 代码可能会花些时间。Clojure 极其强调不可变性(immutability)，用户不必手动在内存中管理锁，也不必担心同时读取值的多个线程，这使得并发处理更简单且不易出错。Clojure 不仅具备所有函数式特性，也能很好地胜任面向对象编程任务。在这方面，Clojure 甚至优于 Java。

1.1.1 以值为导向

Clojure 提倡一种新的编程思想，称为“以值为导向的编程”(value-oriented programming)。Clojure 的作者 Rich Hickey 并非首位使用这个短语描述函数式编程的人士，但他在 JAX Conf 2012上做过一次名为“值的价值”(The Value of Values)的主题演讲[2]，详细阐述了以值为导向的编程。

在以值为导向的编程中，我们关注的是值(value)而非可变对象(mutable object)，它是对内存中的位置及其当前状态的一种抽象。函数式编程消除了变动(mutation)，从而避免了不稳定状态。这种思想的强大之处在于，用户不必担心哪段代码在何时访问或修改了数据，这使得在命令式语言中占有重要地位的并发变得微不足道。

采用命令式语言进行编程时，在将(可变)对象传递给方法之前，通常需要保存其副本，以防止使用过程中对象的值发生变化。而函数式语言处理的是不可变的值，因此可以安全地进行传递而不需要保存副本。可以想到，使用 Clojure 将让代码维护的工作量大大减少。

[2]参见 https://www.youtube.com/watch?v=-6BsiVyC1kM

在面向对象编程中，开发人员主要关心如何隐藏信息或限制访问某个对象的数据，这一般通过封装来实现。而 Clojure 处理的是值而非可变对象，因此封装在 Clojure 中变得无关紧要。从语义上讲，数据对用户是透明的，不必再对数据进行严格控制。由于可以采用替代模型简化复杂的函数，这种层次的透明允许开发人员对代码进行分析。典型的例子如下所示，我们定义了一个名为 sum-of-squares 的函数，通过替换它的值将该函数简化。

```
(defn square [a] (* a a))
(defn sum-of-squares [a b] (+ (square a) (square b))

; evaluate the expression (sum-of-squares 4 5)

(sum-of-squares 4 5)
(+ (square 4) (square 5))
(+ (* 4 4) (* 5 5))
(+ 16 25)
41
```

如果使用具有引用透明性(referential transparency)的函数，就能利用 Clojure 提供的记忆化(memorization)机制。这种机制将那些潜在的、需要消耗大量资源的计算任务进行缓存，从而提高程序的执行速度。我们以斐波那契数列为例演示记忆化机制的应用，以下示例来自麻省理工学院的经典教科书《计算机程序的构造和解释》(SICP)。

```
(defn fib [n]
  (cond
    (= n 0) 0
    (= n 1) 1
    :else (+ (fib (- n 1))
             (fib (- n 2)))))
```

观察图 1-1 中树的执行过程，并使用函数求解第 5 个值。可以看到，为计算第 5 个斐波那契数(fib 5)，需要调用 fib 4 和 fib 3。同理，为计算第 4 个斐波那契数(fib 4)，需要调用 fib 3 和 fib 2。这涉及大量数值计算，读者可能已经知道结果(如图 1-1 所示)。

计算 fib 5 时，程序的执行速度很快。但如果计算 fib 42，执行速度将变得非常慢。

```
(time (fib 42))
"Elapsed time: 11184.49583 msecs"
267914296
```

我们利用记忆化机制重写函数，以便显著缩短程序的执行时间。更新后的代码如下：

```
(def memoized-fib
  (memoize (fn [n]
             (cond
               (= n 0) 0
               (= n 1) 1
               :else (+ (fib (- n 1))
                        (fib (- n 2))))))))
```

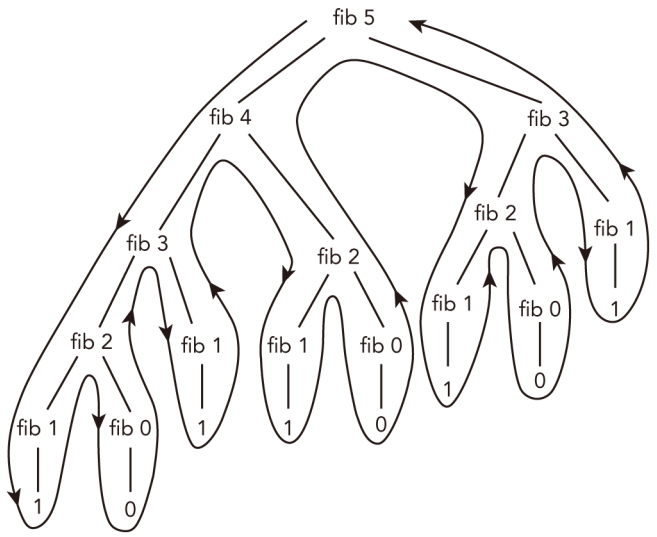

图 1-1

第一次运行重写后的函数时，用户可能感觉不到速度的提升。不过后续调用函数时，程序将在瞬间执行完毕。

```
user> (time (memoized-fib 42))
"Elapsed time: 10586.656667 msecs"
267914296
user> (time (memoized-fib 42))
"Elapsed time: 0.10272 msecs"
267914296
user> (time (memoized-fib 42))
"Elapsed time: 0.066446 msecs"
267914296
```

注意，使用依赖于可变共享状态的函数存在一定风险。但是，由于改进后的函数以值为中心且具有引用透明性，你将能利用 Clojure 提供的某些强大特性。

1.1.2 从递归的角度考虑问题

对于大部分命令式语言来说，递归(recursion)并非一个特别重要的概念，不过函数式语言和递归的关系则十分紧密。学习函数式编程时，应多从递归的角度考虑问题。如果读者对递归的概念不太熟悉，或不习惯从递归的角度考虑问题，可以参考 Daniel P. Friedman 和 Matthias Felleisen 撰写的 *The Little Schemer*[3]一书。该书采用苏格拉底式的教学风格，两位作者通过问答形式向读者介绍如何编写递归函数。

我们来看一个求解阶乘的简单例子。如果采用 Java 编写程序，典型的代码如下所示。首先创建一个用于保存最终结果的局部变量，然后遍历每个数字，直至找到目标数字。接

[3]麻省理工学院出版社(MIT Press)于 1995 年出版。——译者注

下来将最后的结果与 for 循环定义的计数器变量相乘，并保存在局部变量中。

```
public long factorial(int n) {
  long product = 1;
  for ( int i = 1; i <= n; i++ ) {
    product *= i;
  }
  return product;
}
```

由于 Clojure 处理的是值和不可变数据结构，它依赖递归进行循环与迭代。采用 Clojure 计算阶乘时，首先定义一个简单的递归函数 factorial：

```
(defn factorial [n]
    (if (= n 1)
        1
        (* n (factorial (- n 1)))))
```

如下所示，如果阶乘函数的输入为 6，观察程序的执行情况可以看到，JVM 需要在 n 增加时维护栈中的每一次连续操作，直至函数到达某个点，停止递归并返回一个值。如果不小心让函数无限运行下去，最终将导致栈溢出。这种递归通常被称为线性递归(linear recursion)。

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

为解决这个问题，可使用一个名为 recur 的特殊运算符，以尾递归(tail recursion)的方式重写函数。Clojure 文档对 recur 的描述如下：“通过重新绑定并跳到最近的封闭循环或函数帧，实现恒定空间的递归循环”。换句话说，recur 试图模拟一次尾调用优化(tail call optimization)，而 JVM 并不支持该操作。我们采用尾递归的方式重写之前的阶乘函数，得到一个新函数 factorial2：

```
(defn factorial2 [n]
   (loop [count n acc 1]
     (if (zero? count)
         acc
       (recur (dec count) (* acc count)))))
```

在 factorial2 中，可通过 loop 构造定义一个匿名的 lambda 表达式，在一开始就绑定局部变量 count，并将其值传递给阶乘函数 factorial2，从而将累加器的初始值设置为 1。factorial2 的其余部分仅是一个条件语句，用于检查基线条件(base case)并返回累加器的当前值，或使用 recur 完成一次递归调用。观察 factorial2 的尾部可以看到，它不需要运行时跟踪之前的状态，只是采用计算后的值调用 recur。阶乘函数 factorial2 的执行情况如下所示：

```
(factorial2 6)
(loop 6 1)
(loop 5 6)
(loop 4 30)
(loop 3 120)
(loop 2 360)
(loop 1 720)
720
```

对 factorial2 的调用可通过更少的指令完成，且不必像 factorial 那样，每次迭代时都在栈中进行新的调用。

不过，如何处理相互递归(mutual recursion)呢？例如，用户希望定义一个判断数字奇偶性的函数。这可以通过相互定义来实现：对于某个给定的数字，如果自减后为奇数，则该数字被定义为偶数。函数将递归性地调用自身，直至达到 0。如果给定数字被判定为偶数，函数将返回 true；如果被判定为奇数，函数将返回 false。上述两个相互递归函数 my-odd? 和 my-even?的代码如下：

```
(declare my-odd? my-even?)

(defn my-odd? [n]
  (if (= n 0)
    false
    (my-even? (dec n))))

(defn my-even? [n]
  (if (= n 0)
    true
    (my-odd? (dec n))))
```

这个例子与第一个例子存在同样的问题：对于每一次连续的递归调用，都需要在栈中保存状态信息以进行计算，当计算较大的值时可能会导致栈溢出。为解决这个问题，可使用一个称为蹦床(trampoline)的特殊运算符。如下所示，我们对原始代码进行修改，返回一个包含调用递归函数的函数：

```
(declare my-odd? my-even?)

(defn my-odd? [n]
```

```
  (if (= n 0)
    false
    #(my-even? (dec n))))

(defn my-even? [n]
  (if (= n 0)
    true
    *(my-odd? (dec n))))
```

注意第一行的 declare 函数。如下所示，可通过蹦床运算符调用函数：

```
(trampoline my-even? 42)
```

如果调用函数(这里是 my-even?)后返回了另一个函数，trampoline 将继续调用返回的函数，直到返回一个原子值(atomic value)。这使得用户可以对函数进行相互递归调用，而不必担心发生栈溢出。不过这里仍然存在一个问题：如果用户希望使用 my-even?和 my-odd?，就必须了解到需要用 trampoline 调用这两个函数才能防止栈溢出。为解决这个问题，我们再次对函数进行修改：

```
(defn my-even? [n]
  (letfn [(e? [n]
            (if (= n 0)
              true
              #(o? (dec n))))
          (o? [n]
            (if (= n 0)
              false
              #(e? (dec n))))]
    (trampoline e? n)))

(defn my-odd? [n]
  (not (my-even? n)))
```

经过修改，函数对用户有效隐藏了需要使用 trampoline 的细节。

1.1.3　高阶函数

判断一门语言是否属于函数式语言，其中一个条件是能否将函数作为头等对象(first-class object)进行处理。换句话说，函数不仅可以使用值作为参数并返回，也可以作为参数传递给其他函数，还可以被其他函数作为返回值返回。Clojure 提供 map、filter、reduce、remove 和 iterate 等多种常用的高阶函数，用户也可以自定义函数。

我们以 Java 为例进行说明。如果希望将住在某个州的客户过滤出来，需要创建一个保存过滤客户列表的局部变量，手动对客户列表进行迭代，再将客户加入到之前创建的局部变量中。我们不仅需要指定希望过滤的客户，也需要指定对列表进行迭代的方式。Java 代码如下所示：

```
public List<Customer> filterByState(List<Customer> input, String state) {
    List<Customer> filteredCustomers = new ArrayList<>();

    for(Customer customer : input) {
        if (customer.getState().equals(state)) {
            filteredCustomers.put(customer);
        }
    }

    return filteredCustomers;
}
```

接下来，我们采用 Clojure 解决上述问题。从下面的代码可以看到，Clojure 并不关心过滤细节，整个代码更为简洁和声明化。Clojure 的语法看起来或许有些奇怪，但用户只需要采用匿名函数调用过滤函数，告诉程序希望过滤的内容和序列。

```
(def customers [{:state "CA" :name "Todd"}
                {:state "MI" :name "Jeremy"}
                {:state "CA" :name "Lisa"}
                {:state "NC" :name "Rich"}])
(filter #(= "CA" (:state %)) customers)
```

命令模式(Command Pattern)是面向对象编程中一种常见的设计模式，通常在没有头等函数和高阶函数时使用。为实现命令模式，首先需要定义一个接口，后者定义了一种执行命令的方式，它是一种伪函数式对象(pseudo-functional object)。接下来将这个命令对象传递给方式，并在合适的时候调用该对象。这种方案的缺点在于，必须定义多个具体实现以涵盖所有需要执行的功能，或者定义一个用于包装功能的匿名内部类(anonymous inner class)。

```
public void wrapInTransaction(Command c) throws Exception {
   setupDataInfrastructure();
   try {
      c.execute();
      completeTransaction();
   } catch (Exception condition) {
      rollbackTransaction();
      throw condition;
   } finally {
      cleanUp();
   }
}

public void addOrderFrom(final ShoppingCart cart, final String userName,
                         final Order order) throws Exception {
   wrapInTransaction(new Command() {
```

```
        public void execute() {
            add(order, userKeyBasedOn(userName));
            addLineItemsFrom(cart, order.getOrderKey());
        }
    });
}
```

而 Clojure 可以向函数传递任何值。如果仅需要在内部进行声明，可使用匿名的 lambda 表达式。我们采用 Clojure 重写上例：

```
(defn wrapInTransaction [f]
    (do
        (startTransaction)
        (f)
        (completeTransaction)))

(wrapInTransaction #(
    (do
        (add order user)
        (addLineItemsFrom cart orderKey))))
```

换言之，命令式语言处理“如何做”的问题，而函数式语言(如 Clojure)处理“做什么”的问题。Clojure 允许用户在不同层次上对事物进行抽象，这种抽象在大部分命令式语言中都难以实现。

1. 偏函数

在面向对象编程中，可以采用建造者模式(Builder Pattern)逐步创建对象，或采用抽象工厂模式(Abstract Factory Pattern)创建相关类型的对象。而在 Clojure 中，由于抽象的主要方式为函数，也可以通过将某些参数设置为固定值来创建新函数。偏函数(partial function)的作用就在于此。

下面这个简单的例子显示了偏函数 `partial` 的用法。

```
(def add2 (partial + 2))
```

更有实际意义的例子如 `clojure.java.jdbc` 库。下面显示了一种典型模式，它定义了用户数据库的连接属性以及一些简单的查询包装。注意，每次调用 `jdbc/query` 和 `jdbc/insert!`时，`spec` 变量均作为第一个参数。

```
(ns sampledb.data
  (:require [clojure.java.jdbc :as jdbc]))

(def spec {:classname "org.postgresql.Driver"
           :subprotocol "postgresql"
           :subname "//localhost:5432/sampledb"})
```

```
(defn all-users []
  (jdbc/query spec ["select * from login order by username desc"])))

(defn find-user [username]
  (jdbc/query spec ["select * from login where username = ?" username]))

(defn create-user [username password]
  (jdbc/insert! spec :login {:username username :password password :salt
    "some_salt"}))
```

上例中的定义略显重复，它实际只包含三个用于数据库查询的函数。可以想到，一个复杂的应用需要进行多少次类似的定义。为避免重复操作，可使用 partial 定义新函数(第一个参数已和 spec 变量绑定)：

```
(ns sampledb.data
  (:require [clojure.java.jdbc :as jdbc]))

(def spec {:classname "org.postgresql.Driver"
           :subprotocol "postgresql"
           :subname "//localhost:5432/sampledb"})

(def query (partial jdbc/query spec))
(def insert! (partial jdbc/insert! spec))

(defn all-users []
  (query ["select * from login order by username desc"]))

(defn find-user [username]
  (query ["select * from login where username = ?" username]))

(defn create-user [username password]
  (insert! :login {:username username :password password :salt "some_salt"}))
```

partial 也可以用在 map 等高阶函数中。map 接受仅有一个参数的函数来应用于集合中的对象。partial 接受一个函数及其参数(该函数通常有不止一个参数)作为参数，然后创建一个新函数，新函数将这些参数作为自己的一个参数。以乘法函数*为例，如果只有一个参数，则没有意义，但可以通过 partial 指定希望相乘的每一项：

```
(defn apply-sales-tax [items]
  ((map (partial * 1.06) items)))
```

partial 唯一的不足之处在于，参数和值必须按顺序绑定，这意味着参数的顺序很重要。如果希望将最后一个参数与函数绑定，则无法使用 partial。为解决这个问题，可以再定义一个包含原始函数调用或使用 lambda 表达式的函数。

2. 函数组合

Clojure 另一个有用的功能是可将多个函数组合在一起，从而创建一个新函数。需要再次强调，Clojure 是一门基于数学概念的函数式语言。例如，给定两个函数 f 和 g，可以将二者组合，将 f 的输出作为 g 的输入。这与在 Unix 命令行中，通过管道(pipe)和重定向将多个函数进行组合的原理相同。具体来说，如果函数调用类似于 g(f(x))，则在 Clojure 中的表达为((comp g f) x)。

接下来，我们讨论一个更实用的例子：用户希望减小JavaScript代码的尺寸，或在读取JavaScript文件时删除所有换行和多余的空格，以便服务器向浏览器传递信息时能更精简。为此，可将三个常见的Clojure字符串函数str/join、str/trim与str/split-lines组合在一起，如下所示：

```
(defn minify [input]
  (str/join (map str/trim (str/split-lines input))))
```

我们可以使用 comp 函数重写 minify：

```
(def minify (comp str/join (partial map str/trim) str/split-lines))
```

向 comp 进行传递时，注意三个字符串函数的顺序保持原始顺序不变：第三个函数首先传递给 comp，其次是第二个函数，最后是第一个函数。此外，我们略加修改，将 partial 与 map 和 str/trim 进行合并，创建一个可以在集合中使用的函数。这是因为 str/trim 仅能用于单个字符串。

1.1.4　拥抱惰性

和 Haskell 有所不同，Clojure 本身并非一门惰性语言(lazy language)，但它的确支持创建和使用惰性序列(lazy sequence)。事实上，map、filter、reduce 等大部分内置函数在使用过程中都会生成惰性序列，只是用户可能觉察不到而已。例如：

```
user> (def result (map (fn [i] (println ".") (inc i)) '[0 1 2 3]))
#'user/result

user> result
.
.
.
.
(1 2 3 4)
```

当执行第一个表达式时，控制台并没有任何输出。如果这是一个非惰性序列，由于在创建序列的同时就会执行 println 表达式，屏幕上将立即显示输出结果。与之相反，除非要求 Clojure 显示结果符号之后的内容，否则程序不会输出序列中的数据。这是一个非常有用的特性。因为传递给 map 的函数中可能包含需要消耗大量资源的操作，这些操作本身

或许不存在问题，不过一旦开始执行操作，应用的执行速度可能会降低几十甚至几百倍。

惰性序列还可以用在计算无穷量的数字时。例如，对于一个包含全部实数或全部质数的集合，可以在斐波那契数列中进行定义，如下所示：

```
(def fib-seq
    (lazy-cat [1 1] (map + (rest fib-seq) fib-seq)))

(take 10 fib-seq)
-> (1 1 2 3 5 8 13 21 34 55)
```

Clojure 采用惰性序列定义上述序列，并计算序列中的前 10 个数字。如果一门语言不支持这种惰性特性，则无法进行惰性计算。

另一个实用的例子是有限集合中的无限循环。例如，将若干人名分成 4 组，并对每个人名进行标识(为一个集合中的每个值分配一个序号)。大致代码如下：

```
(def names '["Christia" "Arline" "Bethann" "Keva" "Arnold" "Germaine"
             "Tanisha" "Jenny" "Erma" "Magdalen" "Carmelia" "Joana"
             "Violeta" "Gianna" "Shad" "Joe" "Justin" "Donella"
             "Raeann" "Karoline"])

user> (mapv #(vector %1 %2) (cycle '[:first :second :third :fourth]) names)
[[:first "Christia"] [:second "Arline"] [:third "Bethann"] [:fourth "Keva"]
  [:first "Arnold"] [:second "Germaine"] [:third "Tanisha"] [:fourth "Jenny"]
  [:first "Erma"] [:second "Magdalen"] [:third "Carmelia"] [:fourth "Joana"]
  [:first "Violeta"] [:second "Gianna"] [:third "Shad"] [:fourth "Joe"]
  [:first "Justin"] [:second "Donella"] [:third "Raeann"] [:fourth
  "Karoline"]]
```

当需要映射到多个集合时，执行上述函数以遍历所有集合中的所有项。首先遍历所有集合的第一项，然后遍历所有集合的第二项，依此类推，直至遍历所有集合的所有项。因此，如果希望在不了解集合中人名具体数量的情况下将`:first`、`:second`、`:third`、`:fourth`这 4 个值映射给所有人名，就需要在集合中反复执行循环操作。`cycle`与无限惰性集合的用处就在于此。

1.1.5 当变动成为必需时

尽管 Clojure 主要处理对值的操作，但并不意味着可变状态被完全取消。Clojure 只是极大限制了可变状态的使用，转而在特定代码段中使用隔离。Clojure 提供了多种管理可变状态的机制。

1. 原子类型

原子类型(atom)是 Clojure 提供的第一种，也是最简单的一种可变状态管理机制，它使用同步、非协调或独立的方式管理共享状态。因此，如果一次只需要对一个可变状态进行

管理，原子类型是个不错的选择。

到目前为止，我们的注意力都放在值上，不过原子类型的定义和使用方法有所不同。由于原子类型用于管理可变状态，因此它必须作为不可变结构的某种引用。下例定义了一个原子类型：

```
user> (def app-state (atom {}))
#'user/app-state
user> app-state
#atom[{} 0x1f5b7bd9]
```

我们为 app-state 定义了一个包含空映射的原子类型，它有一个保存的引用。观察 repl 的输出可以看到，原子类型将一个内存位置保存在映射中。上面仅对原子类型进行了定义，下面将值关联到映射中：

```
user> (swap! app-state assoc :current-user "Jeremy")
{:current-user "Jeremy"}
user> app-state
#atom[{:current-user "Jeremy"} 0x1f5b7bd9]
user> (swap! app-state assoc :session-id "some-session-id")
{:current-user "Jeremy", :session-id "some-session-id"}
user> app-state
#atom[{:current-user "Jeremy", :session-id "some-session-id"} 0x1f5b7bd9]
```

Clojure 提供了两个用于修改 app-state 的函数 swap!和 reset!，二者能原子化地修改指向引用 app-state 的值。swap!接受一个对保存在引用中的值进行操作的函数，并将该值与函数执行后的返回值进行交换。在上例中，对于某个给定的关键字，我们通过 swap!和 assoc 将一个新值关联到映射中。

reset!函数用于替换 app-state 中作为引用的值，将其设置为一个新值保存在原子类型中，如下所示：

```
user> (reset! app-state {})
{}
user> app-state
#atom[{} 0x1f5b7bd9]
```

可以看到，app-state 再次引用了一个空映射。

介绍完在原子类型中保存共享状态的方法后，读者可能希望了解如何获取这些值。可通过 deref 函数或@读取宏(reader macro)读取保存在原子类型中的值，如下所示：

```
user> (swap! app-state assoc :current-user "Jeremy" :session-id
"some-session-id")
{:current-user "Jeremy", :session-id "some-session-id"}
user> (:current-user @app-state)
"Jeremy"
user> (:session-id @app-state)
```

```
"some-session-id"
user> (:foo @app-state :not-found)
:not-found
```

通过 deref 或@取消对原子类型的引用后，可再次对 app-state 进行操作，此时它又相当于一个映射了。

2. Ref

原子类型用于管理单个值的共享可变状态，但不适合在多个对象之间协调变化。一个经典例子是，将资金从一个账户转到另一个账户时，需要同步两个账户的余额。这种情况下需要使用事务引用(transaction reference)，简称 Ref。Ref 与数据库事务的原理类似，使用一种称为软件事务内存(Software Transactional Memory，STM)的并发模型。事实上，Ref 遵循 ACID 原则[4]的前三个要素：原子性(Atomicity)、一致性(Consistency)与隔离性(Isolation)。由于事务对数据库所做的修改被保存在内存中，因此 Clojure 本身不涉及持久性(Durability)。

下面的例子演示了在处理协调读取数据时，为何无法仅使用原子类型。

```
user> (def savings (atom {:balance 500}))
#'user/savings
user> (def checking (atom {:balance 250}))
#'user/checking
user> (do
       (swap! checking assoc :balance 700)
       (throw (Exception. "Oops..."))
       (swap! savings assoc :balance 50))
Exception Oops...  user/eval9580 (form-init13345619561481318l9.clj:66)
user> (:balance @checking)
700
user> (:balance @savings)
500
```

我们定义了 savings 和 checking 两个原子类型，但在 do 代码段中对二者进行修改时出现抛出异常错误(throw exception)，导致两个账户的余额无法同步。而如果在上例中使用 Ref，其效果如下：

```
user> (def checking (ref {:balance 500}))
#'user/checking
user> (def savings (ref {:balance 250}))
#'user/savings
user> (dosync
       (commute checking assoc :balance 700)
       (throw (Exception. "Oops..."))
```

[4]数据库在更新过程中，为保证事务正确可靠所必须具备的 4 个要素。——译者注

```
        (commute savings assoc :balance 50))
Exception Oops...  user/eval9586/fn--9587
(form-init13345619561481318l9.clj:6)
user> (:balance @checking)
500
user> (:balance @savings)
250
```

可以看到，创建 Ref 并读取值的过程与原子类型类似，不过二者的细微差别在于如何更新保存在 Ref 中的值。我们用 `commute` 代替 `swap!`，所有更新操作均在 `dosync` 代码段的 Ref 中执行。

1.1.6　Nil 双关

经验丰富的 Java 程序员一定不会对棘手的 `NullPointerException`(空指针异常)错误感到陌生，它可能是 Java 开发中最常见的一个问题。几年前，空引用(Null reference)的作者 Tony Hoare 甚至做过一次演讲[5]，专门阐述这个困扰了不少开发人员的错误。略显奇怪的是，除原始值之外的所有值都是对象，但 `null` 不是。不少语言都提供了相应的解决方案，如 Java 采用 `Optional`，Groovy 采用空安全对象导航(null safe object navigation)。甚至在 Objective-C 中，用户可以向 `nil` 发送消息，然后将其忽略即可。

而源于 Lisp 的 Clojure 采用 `nil` 双关(nil punning)的设计思想。与 Java 有所不同，Clojure 中的 `nil` 表示“无应答”。此外，在不同的语境中，`nil` 也具有不同含义。

与众多动态类型语言类似，当用于布尔表达式计算时，`nil` 等效于 `false`。

```
user> (if nil "true" "false")
"false"
```

也可以将 `nil` 视为一个空的 `seq`。如果对 `nil` 调用 `first`，由于第一个元素并不存在，结果将返回 `nil`。如果对 `nil` 调用 `last`，不出意外也将返回 `nil`。但是，这个 `nil` 并非 `seq`，因为对 `nil` 调用 `seq` 时将返回 `false`。

```
user> (first nil)
nil
user> (last nil)
nil
user> (second nil)
nil
user> (seq? nil)
false
```

与 Lisp 家族的大部分语言不同，Clojure 不会将空列表、向量与映射视为 `nil`。

```
user> (if '() "true" "false")
```

[5]参见 https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

```
"true"
user> (if '[] "true" "false")
"true"
user> (if '{} "true" "false")
"true"
```

由于 nil 具备多重含义，用户必须了解它在何时表示 false，在何时表示 nil。例如，在映射中搜索某个值时，某个键的值恰好也可能是 nil。为确定这个值(而非这个键的值)是否存在，需要返回一个默认值。

```
user> (:foo {:foo nil :bar "baz"})
nil
user> (:foo {:foo nil :bar "baz"} :not-found)
nil
user> (:foo {:bar "baz"} :not-found)
:not-found
```

与 Java 不同，nil 在 Clojure 中无处不在，for 函数也会返回 nil。大部分函数都被设计成可以处理传递过来的 nil 值。Eric Normand 在一篇博文中写道，“在 Clojure 代码中，nil 使用不当一般属于类型错误而非 NullPointerException 错误，就像将数字作为函数属于类型错误一样”[6]。

1.1.7 函数式 Web

观察 Web 编程的发展历史是一件有趣的事情。各种范式粉墨登场又销声匿迹，某些范式出类拔萃，另一些则平淡无奇。这种发展仍在进行中。20 世纪 90 年代是动态 Web 发展的早期阶段，那时流行 CGI 等技术，以及 Perl、PHP、ColdFusion 等语言。随着面向对象编程的兴起，CORBA、EJB 等分布式对象技术开始出现，随之而来的是 SOAP 等以对象为中心的 Web 服务技术，以及 ASP.NET、JSF 等面向对象的 Web 编程框架。

近年来，人们越来越青睐更符合 REST 原则和基于微服务的架构。CORBA 已退出历史舞台，甚至 SOAP 在不少人眼中也成了一个令他们生厌的字眼。Web 编程开始进入 HTTP 时代，其无状态本质和以值为中心的特性备受推崇。拜当今多核处理器的流行和并行编程的需求所赐，函数式编程越来越普及。与之类似，Web 编程也面临如何横向而不仅仅是纵向发展的问题。

那么近年来，Web 编程和函数式编程存在哪些共通之处呢(如果有的话)？从本质上说，可以将 REST 端点视为一个函数，其输入是 HTTP 请求，输出是 HTTP 响应。HTTP 协议本身实际上也是无状态的。Cookies 和会话的确如此，但它们只不过是在客户端和服务器端之间通过共享密钥模拟了某种状态而已。REST 端点通常都具备引用透明性，这也是缓存技术相当流行的原因。

[6]参见 http://www.lispcast.com/nil-punning

但是，这并非表示 Web 编程不那么“函数式”。显而易见，如果不修改状态，程序就很难执行下去。较之面向对象编程，这似乎意味着 Web 编程与函数式编程存在更多共通之处。与基于组件的技术相比，组合和重用在 Web 编程中更常见。

1.2　改进面向对象编程

尽管面向对象编程的设计初衷是实现组件的重用，不过在许多方面未能达到这一目标，而函数式编程恰恰实现了这点。其实，早在 Java 出现之前，Lisp 就已应用于面向对象编程。顾名思义，大部分面向对象语言将事物都定义为对象。不过这样一来，对象存在的唯一目的只是作为方式的“陪同”。Steve Yegge 在《名词王国的执行》(*Execution in the Kingdom of Nouns*)[7]一文中巧妙地解释了这个问题。

在 Java 王国，根据国王颁布的敕令，动词(Verb)为名词(Noun)所拥有。但是，动词并非单纯的宠物，它们承担了王国中所有的家务活和体力劳动。动词实际上是王国的奴隶——起码是农奴和契约佣工。Java 王国的居民对这一状况很满意，它们的确很少意识到，情况会有所不同。

虽然动词负责 Java 王国中的所有工作，但没人瞧得起它们，它们甚至无法独自在街上行走。如果动词想去公共场合，身边必须有名词的陪同。

当然需要“陪同”——动词自己没有机会裸奔，它们必须寻找一个“动词陪同者”(VerbEscorter)以推动陪同的实现。那么，“寻找”和“推动”又是什么呢？推动者(Facilitator)和采购者(Procurer)是两个相当重要的名词，它们通过推动(Facilitation)和采购(Procurement)来监护卑微的动词。

国王就此事征询了太阳神的意见，并不时威胁要将所有动词从王国中驱逐出去。如果这成为事实，王国的居民们肯定需要至少一个动词来承担所有家务活。而具有残酷幽默感的国王指出，它将选择最稳妥的一个动词——执行(execute)。

这个名叫“执行”的动词与“run”、“start”、“go”、“justDoIt”、“makeItSo”等它的同义表亲一起，可以完成任何其他动词能完成的工作——只需要将那个动词替换为一个合适的执行者(Executioner)，并调用 execute()即可。等一下？调用 Waiter.execute()。要刷牙？调用 ToothBrusher(myTeeth).go()。倒垃圾？调用 TrashDisposalPlanExecutor.doIt()。没有动词是真正安全的，所有动词都可以被一个名词替换掉。

面向对象编程的核心是通过类来描述事物，类中包含特定对象的名称、方式、功能等属性。接下来，可以通过继承创建更为具体的类。一个典型的例子如图 1-2 所示，这是一个用于描述形状的绘图程序。

[7]参见 http://steve-yegge.blogspot.ca/2006/03/execution-in-kingdom-of-nouns.html

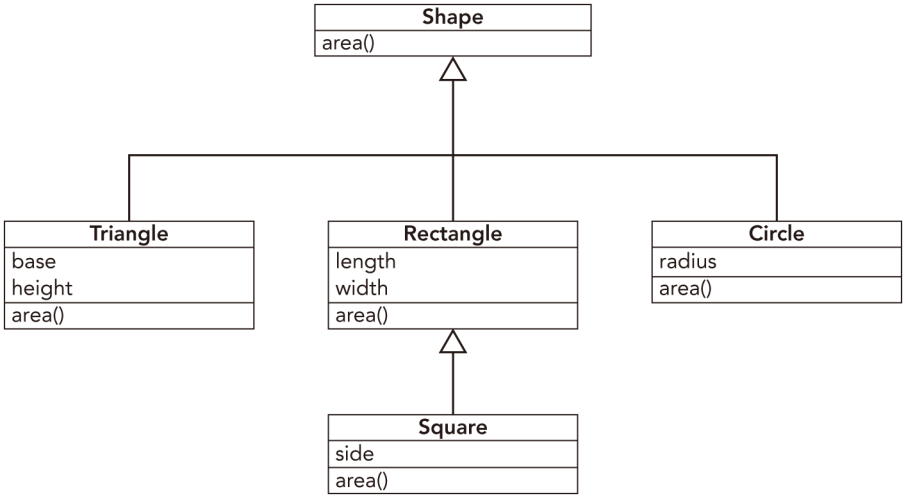

图 1-2

从上图可以看到，我们首先定义了一个名为 Shape 的泛型类(generic class)，然后创建了若干个继承自 Shape 的类，且每个类都定义了自己的 area() 实现。绘图程序向每个形状发送消息询问其面积，然后在运行时根据形状的类型决定调用哪一个实现。这通常被称为多态(polymorphism)。需要注意以下几点：行为属于类本身，并由类本身定义；方式由特定对象进行调用；具体实现由对象的类决定。

1.2.1 利用 defmulti 实现多态调度

与 Lisp 家族的许多语言类似，Clojure 采用一种完全不同的方式处理多态，这种方式称为泛型函数(generic function)。借助于泛型函数，Clojure 能实现不少面向对象语言无法实现的功能。在 Clojure 中，运行时多态(runtime polymorphism)可在类型、值、元数据以及一个或多个参数之间的关系上实现。

我们重写图 1-2 中的绘图程序。首先为 area 定义一个泛型函数，如下所示：

```
(defmulti area (fn [shape & _]
                 shape))
```

这个泛型函数由两部分构成。第一部分是函数名，第二部分是调度函数(dispatch function)，用于确定调用哪个实现。在上例中，首先检查函数的第一个参数的值，然后就可以实现各种面积函数，以求解不同形状的面积。

```
(defmethod area :triangle
  [_ base height]
  (/ (* base height) 2))

(defmethod area :square
  [_ side]
  (* side side))
```

```
(defmethod area :rectangle
  [_ length width]
  (* length width))

(defmethod area :circle
  [_ radius]
  (* radius radius Math/PI))
```

我们为 area 函数定义了 4 个实现。定义时没有使用 defn 关键字，而是以 defmethod 特殊形式，后跟泛型函数名，以及希望匹配的调度函数的值。注意，在每个函数的参数中，可以安全地忽略传递给函数的第一个参数，因为后者仅用于调度。实际运行结果如下：

```
user> (area :square 5)
25
user> (area :triangle 3 4)
6
user> (area :rectangle 4 6)
24
user> (area :circle 5)
78.53981633974483
```

读者可能会问，这比面向对象编程究竟好在哪里呢？我们来看另一个例子。假设用户需要定义一个根据客户地址计算附加费用的函数：当客户住在纽约时，将额外收取 5%的费用；当客户住在加利福尼亚州时，将额外收取 4.5%的费用。一个非常简单的客户发票如下：

```
{:id 42
 :issue-date 2016-01-01
 :due-date 2016-02-01
 :customer {:name "Foo Bar Industries"
            :address "123 Main St"
            :city "New York"
            :state "NY"
            :zipcode "10101"}
 :amount-due 5000}
```

采用 Java 定义这个函数时，大致代码如下：

```
public BigDecimal calculateFinalInvoiceAmount(Invoice invoice) {
    if (invoice.getCustomer().getState().equals("CA")) {
        return invoice.getAmount() * 0.05;
    } else if (invoice.getCustomer.getState().equals("NY")) {
        return invoice.getAmount() * 0.045;
    } else {
        return invoice.getAmount();
```

```
        }
    }
```

如果需要添加另一个州及相应的附加费用，必须修改已有的方式，再增加一个 else if 条件语句。接下来，我们改用 Clojure 定义这个函数，大致代码如下：

```
(defmulti calculate-final-invoice-amount (fn [invoice]
                                           (get-in invoice [:customer :state])))

(defmethod calculate-final-invoice-amount "CA" [invoice]
  (let [amount-due (:amount-due invoice)]
    (+ amount-due (* amount-due 0.05))))

(defmethod calculate-final-invoice-amount "NY" [invoice]
  (let [amount-due (:amount-due invoice)]
    (+ amount-due (* amount-due 0.045))))

(defmethod calculate-final-invoice-amount :default [invoice]
  (:amount-due invoice))
```

今后需要添加其他州及附加费用时，只需要在 Clojure 代码中增加一个额外的 defmethod 即可。

1.2.2 使用 deftype 和 defrecord 定义类型

对于具有面向对象编程经验的开发人员来说，当编写 Clojure 代码时，可能仍然习惯通过自定义类型来描述对象，就像使用面向对象语言进行开发一样。尽管 Clojure 强烈鼓励用户坚持使用内置类型，但自定义数据类型有时更占优势，以使用类型驱动多态(type-driven polymorphism)等特性。在大部分使用面向对象语言编写的程序中，用户定义的类通常分为两种：一种用于实现某种功能，另一种用于描述某种功能。

而在 Clojure 中，deftype 定义实现某种功能的类，defrecord 定义描述某种功能的类。类型和记录的定义与使用极其相似，但二者也存在细微差别(如表 1-1 所示)。

表 1-1 deftype 和 defrecord 的对比

deftype	defrecord
支持可变字段	不支持可变字段
仅提供构造函数	与 PersistentMap 的功能类似，并提供以下默认实现： ● 基于 hashCode 和 equals 的值 ● 元数据支持 ● 关联支持 ● 针对字段的关键字访问方式

(续表)

deftype	defrecord
提供读取器语法以进行对象的实例化(使用完全限定名和参数向量)。将参数向量直接传递给构造函数。例如：#my.type[1 2"a"]	提供另一种读取器语法以进行对象的实例化(使用完全限定名和参数映射)。例如：#my.type{:a "foo" :b "bar"}
提供一种特殊的函数->YourType(YourType 是自定义类型的名称)，用于将其参数传递给自定义类型的构造函数	提供一种特殊的函数 map->YourRecord(YourRecord 是自定义记录的名称)，用于接受映射并构造一条记录

在介绍如何定义与使用 deftype 和 defrecord 之前，我们先来讨论一下 Clojure 中的协议。

1.2.3　协议

经验丰富的Java开发人员通常认为协议与接口非常类似，它们是若干命名的函数及其参数的集合。实际上，Clojure会为所有用户定义的协议生成一个对应的Java接口，后者包含与协议中所定义的函数相对应的方式。

再来看一下计算形状面积的例子。我们首先创建一个名为 Shape 的协议，如下所示：

```
(defprotocol Shape
  (area [this])
  (perimeter [this]))
```

接下来，为 Square 和 Rectangle 创建用于实现 Shape 协议的记录。

```
(defrecord Rectangle [width length]
  Shape
  (area [this] (* (:width this) (:length this)))
  (perimeter [this] (+ (* 2 (:width this)) (* 2 (:length this)))))

(defrecord Square [side]
  Shape
  (area [this] (* (:side this) (:side this)))
  (perimeter [this] (+ (* 4 (:side this)))))
```

然后定义并调用函数，计算正方形和矩形的面积，如下所示：

```
user> (def sq1 (->Square 4))
#'user/sq1
```

```
user> (area sq1)
16
user> (def rect1 (->Rectangle 4 2))
#'user/rect1
user> (area rect1)
8
```

也可以使用 map->Rectangle 和 map->Square 函数创建记录，如下所示：

```
user> (def sq2 (map->Square {:side 3}))
#'user/sq2
user> (def rect2 (map->Rectangle {:width 4 :length 7}))
#'user/rect2
user> (into {} rect2)
{:width 4, :length 7}
user> rect2
#user.Rectangle{:width 4, :length 7}
user> (into {} rect2)
{:width 4, :length 7}
user> (:width rect2)
4
user> (:length rect2)
7
user> (:foo rect2 :not-found)
:not-found
```

观察上一节讨论的记录可以发现，它们基本属于对 PersistentMap 的包装。这意味着用户可以对记录进行操作，如同对映射进行操作一样。用户可将对象 Rectangle 视为一种映射并访问其中的成员，甚至使用 into 构造新的映射。

1.2.4 reify

如果希望在不必自定义类型或记录的情况下实现某个协议，可以使用 Clojure 提供的 reify，如下例所示：

```
(def some-shape
  (reify Shape
    (area [this] "I calculate area")
    (perimeter [this] "I calculate perimeter")))

user> some-shape
#object[user$reify__8615 0x221f1bd "user$reify__8615@221f1bd"]
user> (area some-shape)
"I calculate area"
user> (perimeter some-shape)
"I calculate perimeter"
```

可以将 Clojure 中的 `reify` 视为 Java 中的匿名内部类。实际上，也可通过 `reify` 创建匿名对象以扩展 Java 接口。

1.3　可持久化数据结构

就本质而言，大部分命令式语言所用的数据结构都具有破坏性，因为它们会对当前的值进行修改。由于其他操作可能修改现有对象的值，这种破坏性的数据结构无法确保值不变。

我们以 Java 为例进行说明。如图 1-3 所示，当更新列表 L1 的第二个节点的索引时，索引的值将发生变化，L1 不再是原来的列表。因此，凡是需要调用 L1 进行计算的函数都要相应变化。

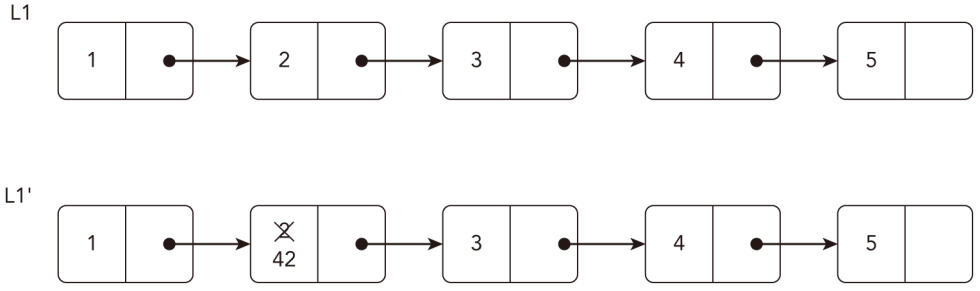

图 1-3

而侧重于操作值的 Clojure 则可以持久性地实现各种数据集合。也就是说，某个集合被修改后，Clojure 将返回一个全新集合，新集合可能与原来的集合共享某些结构。读者或许认为，采用上述方式在结构之间共享元素会产生问题，但由于这些元素本身是不可变的，因此不必担心。

如图 1-4 所示，我们改用可持久化数据结构对列表 L1 进行更新。可以看到，当更新 L1 的第二个节点的索引时，Clojure 将创建一个新的列表 L2，然后将第二个节点之后的所有节点复制过来。换言之，L2 和 L1 共享第二个节点之后的结构。

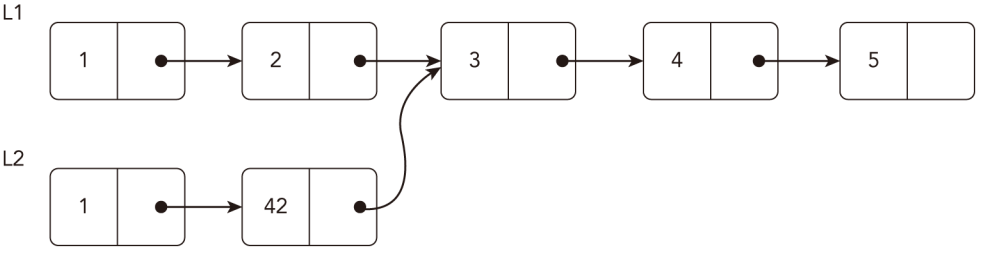

图 1-4

如果将 L2 添加到 L1 的末尾，Clojure 将返回一个新的列表 L3。L3 实际是 L1 和 L2 的结合，其中 L1 的最后一个节点指向 L2 的第一个节点。L1 和 L2 的完整性都得以保证，任何使用这两个列表的函数都能继续使用它们，不必担心列表中的数据发生变化。

我们来看另一个例子。图 1-5 显示了一个简单的二叉搜索树(binary search tree)。当向树 L1 中插入一个值为 6 的新节点时，所有节点的副本都包含到达新节点的路径，只需要让新树 L2 的根节点指向 L1 的右子树即可。

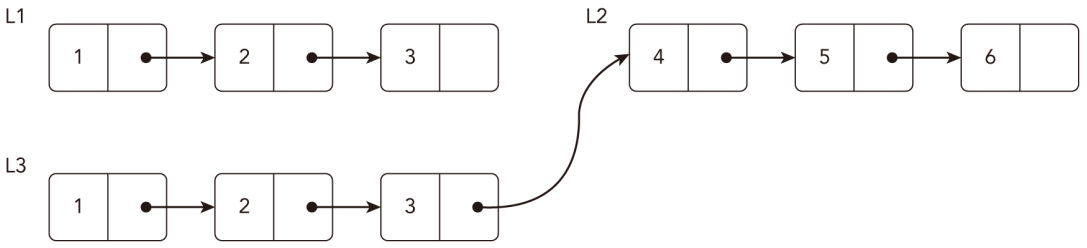

图 1-5

接下来讨论如何在 Clojure 中实现二叉搜索树。首先定义一个名为 INode 的协议，如下所示：

```
(defprotocol INode
  (entry [_])
  (left [_])
  (right [_])
  (contains-value? [_ _])
  (insert-value [_ _]))
```

接下来看几个例子。我们定义一个用于检索左子树和右子树的函数，且任何情况下都能向树中插入新节点。如下所示，定义 INode 后，可以使用 deftype 定义一个新类型，以实现二叉搜索树：

```
(deftype Node [value left-branch right-branch]
  INode
  (entry [_] value)
  (left [_] left-branch)
  (right [_] right-branch)
  (contains-value? [tree v]
    (cond
      (nil? tree) false
      (= v value) true
      (< v value) (contains-value? left-branch v)
      (> v value) (contains-value? right-branch v)))
  (insert-value [tree v]
    (cond
      (nil? tree) (->Node v nil nil)
      (= v value) tree
      (< v value) (->Node value (insert-value left-branch v) right-branch)
      (> v value) (->Node value left-branch (insert-value right-branch v)))))
```

插入一个新节点后，执行结果如下：

```
user> (def root (Node. 7 nil nil))
#'user/root
user> (left root)
nil
user> (right root)
nil
user> (entry root)
7
user> (contains-value? root 7)
true
```

到目前为止，一切顺利。接下来，我们查找树中是否包含值为 5 的节点，执行结果如下：

```
user> (contains-value? root 5)
IllegalArgumentException No implementation of method: :contains-value? of
   protocol:
   #'user/INode found for class: nil  clojure.core/-cache-protocol-fn
  (core_deftype.clj:554)
```

为什么会显示错误呢？观察错误消息可知，`nil` 并没有实现协议，因此不知道如何调用函数 `contains-value?`。可以通过将协议扩展到 `nil` 来解决这个问题，如下所示：

```
(extend-protocol INode
  nil
  (entry [_] nil)
  (left [_] nil)
  (right [_] nil)
  (contains-value? [_ _] false)
  (insert-value [_ value] (Node. value nil nil)))
```

通过对类型 Node 进行重构，可以消除对 `nil` 的冗余检查。

```
(deftype Node [value left-branch right-branch]
  INode
  (entry [_] value)
  (left [_] left-branch)
  (right [_] right-branch)
  (contains-value? [tree v]
    (cond
      (= v value) true
      (< v value) (contains-value? left-branch v)
      (> v value) (contains-value? right-branch v)))
  (insert-value [tree v]
    (cond
```

```
    (= v value) tree
    (< v value) (Node. value (insert-value left-branch v) right-branch)
    (> v value) (Node. value left-branch (insert-value right-branch v)))))
```

解决这个问题后，再次运行程序，结果如下：

```
user> (contains-value? root 5)
false
```

非常好。接下来，我们创建一个拥有更多节点的树。

```
user> (def root (Node. 7 (Node. 5 (Node. 3 nil nil) nil) (Node. 12
 (Node. 9 nil nil) (Node. 17 nil nil))))
#'user/root
```

运行上例中的代码，应创建一个和 L1 具有相同结构的树，如图 1-6 所示。

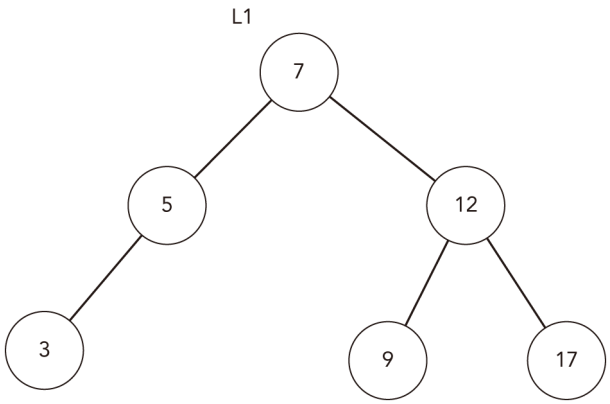

图 1-6

观察程序的运行结果，可以证实这一点。

```
user> (left root)
#object[user.Node 0x5cedcfe8 "user.Node@5cedcfe8"]
user> (entry (left root))
5
user> (entry (left (left root)))
3
user> (entry (right root))
12
user> (entry (right (right root)))
17
```

可以看到，计算根节点的左子树节点，其值为 5；计算节点 5 的左子树节点，其值为 3。接下来，我们从 root 开始，观察左子树和右子树的标识值(identity value)。

```
user> (identity (left root))
#object[user.Node 0x5cedcfe8 "user.Node@5cedcfe8"]
```

```
user> (identity (right root))
#object[user.Node 0x124ee325 "user.Node@124ee325"]
```

读者在运行程序时，得到的结果可能与上例略有区别，但基本类似。接下来，向树中插入一个值为 6 的新节点，它应该位于节点 5 的右侧。插入新节点后，再次观察新树从根节点开始的标识值。

```
user> (def l (insert-value root 6))
#'user/l
user> (identity (left l))
#object[user.Node 0x167286ec "user.Node@167286ec"]
user> (identity (right l))
#object[user.Node 0x124ee325 "user.Node@124ee325"]
```

可以看到，在新树的左子树中插入了一个新节点，但新列表却指向原始树的右子树中同样的实例。插入后的结构如图 1-7 所示。原始列表仍然是完整的，新列表与原始列表共享某些结构。

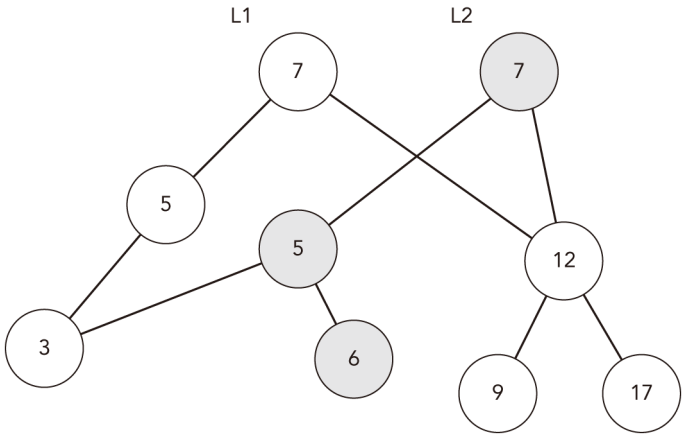

图 1-7

如果读者希望了解有关 Clojure 实现可持久化数据结构的更多信息，可以参考以下两篇优秀文章：Understanding Clojure's PersistentVector implementation[8]和 Understanding Clojure's PersistentHashMap[9]。

1.4　塑造语言

关于对 Lisp 的描述，读者可能听过以下说法：它是一种“可编程的编程语言”，具备

[8]参见 http://blog.higher-order.net/2009/02/01/understanding-clojures-persistentvector-implementation

[9]参见 http://blog.higher-order.net/2009/09/08/understanding-clojures-persistenthashmap-deftwice

同像性(homoiconicity)[10]，以“代码即数据，数据即代码”作为设计思想。那么对于程序员来说，这究竟意味着什么呢？具有 C 语言背景的开发人员可能很熟悉术语“宏”，不过就像本章开头提到的那样，学习 Clojure 时，请忘记所有先前关于宏的知识。Clojure 中的宏是一种功能强大的构造，远超任何命令式语言。有不少书籍专门论述 Clojure 中的宏，如 *Let Over Lambda*、*Mastering Clojure Macros* 等。

在 Clojure 中，用户可以使用宏重写正常的语法规则，通过对语言进行“重塑”，以更好地满足需要。尽管其他语言提供某些可以使用元编程(metaprogramming)[11]的机制，也支持对默认行为进行修改，但它们都难以比肩 Clojure 强大的宏系统。不过，由于宏的功能过于强大，不少语言的框架似乎都在滥用这种能力，从而导致排错困难，或混淆某些特定功能的来源。因此，决定使用宏时必须谨慎。

那么，宏适合在哪些情况下使用呢？一个典型例子是 Compojure 框架使用的路由库，它利用宏创建了一个功能强大、富有表现力的 DSL。Compojure 路由库定义的宏用于构建 Ring 处理函数映射。下例显示了一个名为 `defroutes` 的宏以及其他映射到 HTTP 动词(HTTP verb)的宏：

```
(defroutes app-routes
 (GET "/" [] (index))
 (GET "/books" [] (get-books))
 (GET "/books/:id" [id] (find-book id))
 (POST "/books" [title author] (add-book title author))
```

通过使用宏，Compojure能改变Clojure的语义，从而以一种更容易理解的方式定义URL和处理函数之间的映射。`defroutes`允许用户创建一组命名路由，也就是上例中的`app-routes`。接下来，Ring对列表中的路由逐一进行评估，直至找到一个匹配的路由。路由本身通过一个用于指定HTTP动词的宏来定义，后跟一个可以定义路径变量绑定的路由。接下来列出所有将被绑定的变量，它们可能来自URL参数，也可能来自POST请求中的语句参数。最后，既可以定义实际的内联处理函数(如果不麻烦的话)，也可将路由分配给其他位置定义的某个函数。

Honey SQL库[12]同样能体现宏的强大，并展示如何创建自然流畅的API。Honey SQL允许用户采用Clojure内置的数据结构定义SQL查询，并提供大量能将这些查询转换为`clojure.java.jdbc`的函数和宏。此外，Honey SQL还提供兼容Clojure的参数化SQL，后者可以直接传递给`jdbc/query`和`jdbc/insert`。下例取自Honey SQL文档：

```
(def sqlmap {:select [:a :b :c]
             :from [:foo]
             :where [:= :f.a "baz"]})
```

[10]程序的结构与其句法是相似的，因此可通过阅读程序来推测其内在涵义。一门具备同像性的语言，其源代码与抽象语法树具有相同的结构。Lisp 语言是典型的同像性语言。——译者注

[11]编写在运行时对语言构件进行操作的代码。换言之，所写的代码是用来生成代码的。——译者注

[12]参见 https://github.com/jkk/honeysql。

```
(sql/format sqlmap)

=> ["SELECT a, b, c FROM foo WHERE (f.a = ?)" "baz"]
```

Honey SQL 甚至定义了一个名为 `build` 的辅助函数(helper function)，以帮助用户定义这些映射对象，如下所示：

```
(sql/build :select :*
           :from :foo
           :where [:= :f.a "baz"])

=> {:where [:= :f.a "baz"], :from [:foo], :select [:*]}
```

`build` 函数允许用户保持指定查询的风格不变，但不要求像之前那样使用额外的括号，这使得代码看起来更简洁。

Clojure 还通过 `defprotocol` 支持某些不错的元编程功能。Clojure 中的协议与 Java 接口类似，它定义了一组特定的函数及其签名。协议甚至会生成一个对应的、可在 Java 中使用的接口，后面将对此进行介绍。此外，协议还能扩展现有类型(包括 Java 中的 Final 类)，这意味着用户可以向 Java 中的 `String` 类添加新的方式：

```
(defprotocol Palindrome (is-palindrome? [object]))

(extend-type java.lang.String
  Palindrome
  (is-palindrome? [s]
    (= s (apply str (reverse s)))))

(is-palindrome? "tacocat")

=> true
```

上例定义了一个名为 `Palindrome` 的协议，它仅由一个名为 `is-palindrome?`的函数构成。我们对 `java.lang.String` 类进行扩展，为 Java 内置的 String 类增加功能。接下来，调用值为`"tacocat"`的 `is-palindrome?`来展示结果。

前面曾经提到过，需要谨慎处理对语言及其类型的修改，因为这常导致与滥用元编程同样的问题，且不易判断事物的定义位置。如果仅通过定义普通函数就能解决问题，则应尽量避免对语言和类型进行修改。

为满足需要，Clojure 提供多种非常有效的方法来塑造语言，如何使用这些方法取决于开发人员的偏好。

1.5　小结

本章介绍了 Clojure 与当今大部分主流语言的不同之处。就像本章开头提到的那样，如

果不能保持通透的头脑并从 Clojure 的角度考虑问题，那么写出来的代码只是语法不同而已。虽然掌握 Clojure 并非一朝一夕之功，但只要抱着开放的心态去学习，就可能从根本上改变自己的编程习惯。正如埃里克·雷蒙德(Eric Raymond)[13]所说：“Lisp 是一门值得认真钻研的语言。当你最终掌握它时，将从中获得深刻启发。即便从未真正使用 Lisp 本身编写代码，这种经验也会让你的程序员生涯更加精彩。”

[13]埃里克·雷蒙德(1957-)，美国著名程序员，开源运动的领军人物，其著作《大教堂与集市》(*The Cathedral and the Bazaar*)被奉为开源运动的《圣经》。——译者注