

Clojure编程

Clojure Programming

Chas Emerick, Brian Carper
& Christophe Grand 著

徐明明 杨寿勋 译

電子工業出版社

Publishing House of Electronics Industry

北京 · BEIJING

内 容 简 介

Clojure 是一种实用的通用语言，它是传奇语言 Lisp 的方言，可与 Ruby、Python 等动态语言相媲美，更以无缝 Java 库、服务，以及拥有 JVM 系统得天独厚的资源优势而胜出。本书既可以用来熟悉 Clojure 基础知识与常见例子，也可了解其相关的实践领域与话题，更可以看到这一 JVM 平台上的 Lisp 如何帮助消除不必要的复杂性，为我们在编程实践中解决最具挑战性的问题开辟新的选择——更具灵活性、更适合于 Web 编程和操作数据库，可以应付更为苛刻的应用程序安全要求，更有效的并发性和并行处理、数据分析能力，以及在未来云环境下的更大的发展潜力。

©2012 by O'Reilly Media, Inc. Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Publishing House of Electronics Industry, 2013. Authorized translation of the English edition, 2012 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

本书简体中文版专有版权由 O'Reilly Media, Inc. 授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号 图字：01-2013-1664

图书在版编目（CIP）数据

Clojure 编程 / (美) 埃默里克 (Emerick,C.), (美) 布卡珀 (Carper,B.), (美) 格兰德 (Grand,C.) 著；徐明，杨寿勋译。—北京：电子工业出版社，2013.4

书名原文：Clojure programming

ISBN 978-7-121-19718-5

I. ①C… II. ①埃… ②布… ③格… ④徐… ⑤杨… III. ①程序语言—语言设计 IV. ①TP312

中国版本图书馆 CIP 数据核字（2013）第 040832 号

策划编辑：张春雨

责任编辑：刘 航

封面设计：Karen Montgomery 张 健

印 刷：北京中新伟业印刷有限公司

装 订：三河市皇庄路通装订厂

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：39.25 字数：911 千字

印 次：2013 年 4 月第 1 次印刷

印 数：3000 册 定价：99.00 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。

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

还记得第一次接触 Clojure 是在 JavaEye 论坛上，那是庄晓丹同学的一篇名为《几行代码解决淘宝面试题之 Clojure 版》^{注1} 的帖子，给我留下了很深的印象。淘宝那道面试题目是：

“有一个很大的整数链表，需要求这个链表中所有整数的和，写一个可以充分利用多核 CPU 的代码来计算结果。”

用 Java 来解答这道题需要七八十行代码^{注2}；程序员要自己手动把这个大链表切割成一个个子链表，自己手动创建多个线程去分别计算切割出来的子链表，然后再等待所有线程做完之后对结果进行归约以计算出最后的结果。确实是一道很难的面试题，涉及的知识点包括线程、线程池、线程同步等，一般人还真不一定能把它写出来。而在 Clojure 看来完全不能作为面试题，因为利用 Clojure 来解决太简单了，稍微学过一点 Clojure，掌握了 map、reduce、pmap 这几个基本函数的同学都能搞定。这个对比是很震撼的，Clojure 解决起问题来是那么简洁、优雅、自然，这算是我对 Clojure 的初体验吧。

在 Clojure 之前我也尝试过几种函数式语言，比如“最纯的函数式语言” Haskell、“天生擅长高并发的” Erlang，这些都是很棒的语言，但是学过一段时间之后就忘了，因为工作、生活中实在是用不到。而 Clojure 对我则有独特的吸引力，首先，因为它是 Lisp——一门富有传奇色彩的语言，一直希望有机会可以学习一门 Lisp 的方言；其次，Clojure 是一门接地气的语言，它运行在 JVM 这个最成功、应用最广泛的平台之上，能够跟 Java 代码无缝互操作，JVM 上所有的资源都可以为 Clojure 所用。

Clojure 是这样的有潜力、接地气，那么如果你要选择一门新语言来玩玩，不选它选什么？

对 Clojure 有了初次接触之后就有了深入学习的想法，在网络上到处寻找资料，找到的资料几乎都是英文的，中文资料少之又少。最终发现一篇比较简单而又相对完整的教程^{注3}，当时就想如果把这个教程翻译成中文既可以加深自己对于 Clojure 概念的理解，也可以丰富 Clojure 的中文资料，便于以后对这门语言感兴趣的同學查阅，于是边学边把这篇教程翻译了一下。^{注4} 目前用 Google 搜索 Clojure 中文资料，这篇文章还是排在第一位的，深感欣慰。

注 1：参见 <http://www.iteye.com/topic/713259>。

注 2：参见 <http://www.iteye.com/topic/711162>。

注 3：参见 <http://java.ociweb.com/mark/clojure/article.html>。

注 4：参见 <http://xumingming.sinaapp.com/302/clojure-functional-programming-for-the-jvm-clojure-tutorial/>。

“空有一身的 Clojure 本事，没地方施展啊！”学了一段时间 Clojure 之后有了这样的感觉，相信这也是很多利用业余时间学习新语言的同学的困惑。这时恰逢 Twitter 开源了他们的实时计算框架 Storm，我对实时计算很感兴趣而它又是用 Clojure 开发的。太好了，正愁没有项目练手呢，于是去读它的源代码。看到真正在生产环境中使用的 Clojure 代码才发现自己还有很多东西是不会的，对于 Clojure 只是学会了形，没有学到神。比如，Clojure 对于函数作为头等公民的强调、Clojure 对于值语义的注重，这些对于从 Java 世界过来的程序员来说，不是那么容易转换思维的（本书有专门一节强调要转换思维，讲解如何转换思维，读了之后获益良多，建议大家重点看一下）。而通过学习 Storm 源码，学到了很多这方面活生生的例子，对自己的水平有很大的提高，这里也建议每个程序员都应该参与到开源项目中去，会收益良多的。

2012 年 5 月，张春雨同学在 Clojure 中文邮件列表中发帖，要给本书找译者，因为自己之前也看过一些，而且也算有一定的 Clojure 编码经验、有一定的翻译经验，于是就把这个活应了下来。这本书大体可以分为两部分，前半部分重点讲解 Clojure 的基本概念、原理，后半部分则介绍用 Clojure 来解决实际编程中的各种问题的实例。你既可以从最基本原理开始，“自底向上”地去读；也可以从实例入手，去看看怎么用 Clojure 去解决你感兴趣的问题，遇到不懂的概念再去前半部分找对应的章节深入阅读。

翻译这本书的过程也是一个学习的过程，之前零散地学了很多 Clojure 知识，但都不是很系统。《Clojure 编程》这本书给我最大的帮助是帮我把这些零碎的知识点串了起来，形成了一个系统。它从最基本的原理讲起，在讲解各个知识点的同时由浅入深地把背后的原理一点点娓娓道来。翻译的过程中不时地发出这样的感叹：“原来那么实现是因为这个原因啊”，每天都会发现自己又多懂了那么一点点，这算是翻译过程中最大的乐趣吧，相信大家去读这本书的时候也会有类似的体会。

翻译的过程中得到了很多人的帮助，首先感谢我的老爸老妈，是你们把我带到了这个世界，才有机会去探索这神奇的计算机世界；感谢我自己，这几个月你格外辛苦，没想到你的名字也会出现在一本书的封面上，干得不错；感谢小废同学，如果没有你的捣乱，这本书可能去年就已经面市了；感谢我的合译者杨寿勋同学，没有你的通力合作，这本书不会这么快上市；感谢我们的审稿同学 haungz、孙宁、庄晓丹，是你们使得这本书的质量提高了一个档次，帮我挑出了那么多技术上的、语法上的错误；感谢策划编辑张春雨同学，没有你，这本书的译者可能就是别人了；感谢责任编辑刘舫，虽然从来没正式地打过交道，但是，从一次次反馈中感受到了你编辑工作的严谨；感谢所有帮助过我的人。

```
{:author      "徐明"
 :date: "2013 年 3 月 6 日"
 :city:      "杭州"}
```

目录

前言	xvii
第 1 章 进入 Clojure 仙境	1
为什么要选择 Clojure?	1
获取 Clojure	3
Clojure REPL	3
不！括号真的不会让你瞎了眼	6
表达式、操作符、语法以及优先级	7
同像性	10
Clojure Reader	12
标量字面量	13
注释	18
空格和逗号	19
集合字面量	20
reader 的一些其他语法糖	20
命名空间	21
符号求值	23
特殊形式	24
阻止求值 : quote	25
代码块 : do	26
定义 Var : def	27

本地绑定 : let	27
解构 (let, 第 2 部分)	29
定义函数 : fn	36
条件判断 : if	42
循环 : loop 和 recur	43
引用 var : var	44
和 Java 的互操作 : . 和 new	45
异常处理 : try 和 throw	45
状态修改 : set!	46
锁的原语 : monitor-enter 和 monitor-exit	46
小结	46
eval	47
这只是开始	48

第 1 部分 函数式编程以及并发

第 2 章 函数式编程	51
所谓函数式编程, 到底意味着什么?	52
谈谈值的重要性	52
关于值	53
值与可变对象的比较	54
一个关键性的选择	58
作为头等公民的函数以及高阶函数	59
Apply, Partial	64
函数 (功能) 的组合	68
编写高阶函数	71
利用可组合的高阶函数构建一个日志系统	72
纯函数	76
为什么纯函数很有意思?	77
现实生活中的函数式编程	79
第 3 章 集合类与数据结构	81
抽象优于实现	82

集合 : collection	85
序列 : Sequence	87
Associative	97
索引集合 : Indexed	101
栈 : stack	103
set	103
有序集合 : sorted	104
访问集合元素的简洁方式	109
习惯用法	111
集合、key 以及高阶函数	111
数据结构的类型	113
列表 : List	113
vector	114
set	115
map	116
不可变性和持久性	121
持久性与结构共享	122
易变集合	128
元数据	133
用 Clojure 的集合来小试牛刀	135
标识符和循环引用	135
思维方式的改变 : 从命令式到函数式	137
遍历、更新以及 Zipper	151
总结	157
第 4 章 多线程和并发	159
计算在时间和空间内的转换	160
delay	160
future	162
promise	163
简单的并行化	166
状态和标识	168
Clojure 的引用类型	170

并发操作的分类.....	172
原子类型 (Atom).....	174
通知和约束	176
观察器	177
校验器	179
ref.....	180
软件事务内存.....	180
对 ref 进行修改的机制	181
STM 的一些缺点.....	192
var.....	198
定义 var.....	199
动态作用域	202
var 不是变量	207
前置声明	208
agent.....	209
处理 agent action 中的错误	212
I/O、事务以及嵌套的 Send.....	215
使用 Java 的并发原语	225
Locking	226
总结	226

第 2 部分 构建抽象

第 5 章 宏.....	229
宏到底是什么?	229
宏不是什么	231
有什么是宏能做而函数不能做的.....	232
宏 vs.Ruby 的 eval.....	234
编写你的第一个宏	235
调试宏.....	237
宏扩展	237
语法	240
引述和语法引述	240

反引述与编接反引述.....	241
什么时候使用宏.....	243
宏卫生.....	245
Gensym 来拯救	246
让宏的用户来选择名字.....	249
重复求值	249
宏的常见用法和模式	251
隐藏参数 : &env 和 &form.....	252
&env.....	253
&form.....	254
测试上下文相关的宏.....	259
深入 -> 和 ->>	261
总结	264
第 6 章 数据类型和协议	265
协议	266
扩展已有的类型	268
定义你自己的类型	272
记录.....	274
类型.....	280
实现协议	282
内联实现	284
重用实现	287
协议自省	291
协议函数分派的边界场景	292
自己实现一个 set	294
总结	302
第 7 章 多重方法	303
多重方法基础	303
通往层级之路	306
层级	308
独立层级	310

真正实现多重！	313
还有几件事	315
多重继承	315
内省多重方法	316
type 或 class；或者说，映射的报复	316
转发函数的范围是无限的	318
最后的思考	319

第 3 部分 工具、平台以及项目

第 8 章 Clojure 项目的组织与构建	323
项目布局	323
定义与使用命名空间	323
位置、位置、位置	334
Clojure 代码库的功能组织	335
构建	337
提前编译	338
依赖管理	340
Maven 的依赖管理模型	340
构建工具与配置模式	345
最后的思考	353
第 9 章 Java 及 JVM 互操作	355
JVM 是 Clojure 的基础	356
Java 类、方法和字段的使用	356
便利的互操作工具	359
异常与错误处理	361
摆脱检查型异常	363
with-open：finally 的挽歌	364
为了效率进行类型提示	365
数组	369
定义类、实现接口	370

匿名类的实例 : proxy	370
定义具名的类	373
注解	380
在 Java 里使用 Clojure	384
使用 deftype 和 defrecord 类	386
实现协议接口	389
乐于合作的伙伴	391
第 10 章 面向 REPL 的编程	393
交互式开发	393
持续、演化的环境	397
工具集	398
最简的 REPL	399
Eclipse	402
Emacs	405
在 REPL 里调试、监测和打补丁	410
“已部署” REPL 的特别考虑	412
重定义结构的限制	413
小结	415

第 4 部分 实战

第 11 章 数字与数学	419
Clojure 中的数字	419
Clojure 首选 64 比特（或更大）的表示	420
Clojure 的混合数字模型	420
有理数	422
数字传播规则	423
Clojure 中的数学	425
有限与任意精度	425
无检查的操作	428
任意精度的小数操作的刻度和取整模式	429

相等与等值	430
对象相同 (<code>identical?</code>)	431
引用相等 (=)	431
数字等值 (==)	432
优化数字效率	434
声明函数接受和返回原始类型	435
合理使用原始类型的数组	439
用 Clojure 可视化芒德布罗集	447
第 12 章 设计模式	455
依赖注入	457
策略模式	460
责任链	462
面向方面的编程	464
最后的思考	468
第 13 章 测试	469
不可变值与纯函数	469
模拟	470
<code>clojure.test</code>	471
定义测试	472
测试“套件”	475
Fixtures	477
HTML DSL 的成长	480
依赖断言	484
前条件和后条件	486
第 14 章 使用关系数据库	489
<code>clojure.java.jdbc</code>	489
<code>with-query-results</code> 解析	492
事务	494
连接池	494

Korma	496
前奏	496
查询	497
为什么不嫌麻烦地用 DSL ?	498
Hibernate	501
设置	501
持续数据	504
运行查询	505
去掉样板	506
最后的思考	507
第 15 章 使用非关系型数据库	509
安装 CouchDB 和 Clutch	510
基本的 CRUD 操作	510
视图	512
一个简单的 (JavaScript) 视图	512
用 Clojure 写的视图	514
_changes : 把 CouchDB 滥用做消息队列	517
可随意点选的消息队列	519
最后的思考	522
第 16 章 Clojure 与 Web	523
Clojure 栈	523
基石 : Ring	524
请求与应答	525
适配函数	527
处理函数	528
中间件	529
用 Compojure 路由请求	531
使用模板	541
Enlive : 基于选择器的 HTML 转换	542
最后的思考	550

第 17 章 部署 Clojure Web 应用程序	551
Java 与 Clojure Web 架构.....	551
Web 应用程序打包	554
在本地运行 Web 应用	559
Web 应用程序部署	560
在 Amazon 的 Elastic Beanstalk 上部署 Clojure 应用	560
超越简单 Web 应用程序部署	563
 第 5 部分 杂项	
第 18 章 明智地选择 Clojure 类型定义形式	567
第 19 章 在工作场所引进 Clojure.....	571
事实是.....	571
强调生产效率	573
强调社区	574
审慎	575
第 20 章 下一步?	577
(dissoc Clojure 'JVM)	577
ClojureCLR	577
ClojureScript	578
4Clojure	578
Overtone	578
core.logic	579
Pallet	580
Avout	580
Heroku 上的 Clojure	581
索引	583

前言

Clojure 是一种动态的、强类型的、寄居在 Java 虚拟机（JVM）上的语言，如今已经是它的第 5 个年头了。现今它已经被各种背景的程序员热情地应用到几乎所有的领域。Clojure 提供了一些很引人注目的特性来帮助我们解决现代编程所面临的各种挑战：

- 函数式编程的基础，包括一套性能可以和典型的可变数据结构媲美的持久性数据结构。
- 由 JVM 所提供的成熟、高效的运行时环境。
- 跟 JVM/Java 的互操作能力使得很多架构、运维方面的需求可以得到满足。
- 一套提供并发、并行语义的机制。
- 是一种 Lisp 方言，因此提供了非常灵活、强大的元编程能力。

Clojure 为那些受编程语言和环境限制而痛苦的程序员提供了一个实际可行的新选择。我们将会通过展示 Clojure 与大家每天都会使用的现有技术、类库或者服务的无缝交互来证明这一点。我们会逐步介绍 Clojure 的基础知识，从大家最熟悉的一些方面开始而不是从一些枯燥的原理开始。

这本书的目标读者是谁？

我们在写这本书的时候，脑子里面是有一些目标读者的，希望你是其中之一。

Clojure 跟你现在最喜欢的语言在表达力、简洁性和灵活性方面可以匹敌——通常还有所超越，同时还使你可以不费任何力气就能获得由 JVM 所提供的性能、类库、社区以及运维稳定性。这也使得 Clojure 很自然地成为 Java 程序员（以及使用解释型、不是那么快的非 Java 语言的 JVM 程序员）的“下一代”语言，因为它们不想因为脱离 JVM 而导致写出来的程序性能下降，或者不想因此而放弃之前在 JVM 平台上的投资。而对于那

些不想放弃语言强大表达力，但是又希望获得一个更可靠、更高效的平台以及更多可以选择的高质量类库的 Ruby、Python 程序员来说，Clojure 也是很自然的选择。

职业 Java 程序员

这个世界上有成千上万的 Java 程序员，但是有一小部分 Java 程序员工作在极限条件下解决十分重大的问题，通常是一些领域相关的问题。如果你就是这样的一个 Java 程序员，那么你可能一直在寻找着更好的工具、技术或者实践方法来提高工作效率以贡献更多的价值给你的团队、公司或者社区。而且你可能对于 Java 跟其他语言相比所表现出来的种种局限而苦恼，但是你舍不得离开 JVM 这个生态系统：它的流程成熟度、大量的第三方库、软件供应商的支持服务以及大量熟悉 Java 的程序员资源，即使其他语言的特性是那么诱人。

你会发现 Clojure 是一个很不错的选择。它以优异的性能运行在 JVM 之上，能够跟现在你已经有的所有的类库、工具以及应用进行交互，而且它比 Java 简单很多，同时又更富表达力、更简洁。

Ruby、Python 以及其他程序员

从任何一个方面来讲，Ruby 和 Python 都已经不算什么新语言了，但是最近这些年获得了极大的关注（应该可以算是“主流语言”了吧）。不难看出其中的缘由：它们都是极富表达力的动态语言、有很活跃的社区、在很多领域都鼓励最大化程序员的效率。

Clojure 将是你理想的“下一代语言”。作为一个 Ruby 或者 Python 程序员，你肯定不愿意放弃这些语言所提供的优秀特性，但是又想让你的程序在一个更强劲的平台上运行，可以有更好的运行时性能，更多的类库可供选择。高效寄居于 JVM 之上的 Clojure 完全满足这些条件，通常在程序简洁性以及工作效率方面更会超出你的期望。



在书中我们会经常把 Clojure 跟 Java、Ruby 以及 Python 进行比较来把一些概念“翻译”成你的专业领域中对应的概念。在这些比较中，我们比较的总是这些语言的权威实现：

- Ruby MRI（也称为 CRuby）
- CPython
- Java 6/7

如何阅读这本书

在规划整本书结构的时候，我们准备给大家介绍两方面的内容：一是有关 Clojure 语言的细节；另外就是一些实际的、大家都会碰到的实际问题的例子。但是我们努力避免那些不是很成功的规划办法。具体一点来说就是，我们避免以一个例子贯穿整本书，这种

方式的结果是，很多时候例子跟要叙述的概念往往不是那么相关，而且这一个例子可能对于读者来说不是那么熟悉。

有了这个概念之后，我们把这本书分成了两部分，从最基本的概念开始介绍 Clojure 的语言细节，这部分大概占全书的三分之二；从第 4 部分开始介绍一些具体的、实际生活中的例子。这种清晰的分隔符合了“*duplex book*”的要求（这个概念最早是 Martin Fowler 在 <http://martinfowler.com/bliki/DuplexBook.html> 里面提到的）。因此，你可以以两种方式来阅读这本书。

从 Clojure 的实际应用例子开始

学习一门语言最好的办法通常就是看怎么用它来解决实际的问题。如果你喜欢这种方式，那么希望在本书第 2 部分举的例子有跟你日常做的事情相关的例子，这样你可以自己做个比较：用你自己熟悉的语言怎么做这件事情，再看看书里介绍的用 Clojure 如何解决这个问题。在这个过程中你会遇到很多未知的概念以及语言结构，每当遇到这些问题的时候，你可以以当前的上下文作为线索去本书的第 1 部分查找对应的章节阅读。

自底向上——从 Clojure 最基本的概念开始

有时候弄懂一个东西的唯一办法是从基本的概念开始学习，如果你习惯这种方式的话，那么从第 1 章的第 1 页开始读是最好的办法了。我们尝试以渐进的方式阐述 Clojure 所有基本的原则以及结构，这样做的结果是你很少需要先阅读书的后面章节内容才能理解前面章节的内容。而当你准备自己动手实践的时候，你可以到后面的实践部分挑一个你最感兴趣、跟你要做的事情最相关的例子进行阅读。

我们是谁？

我们是通过不同途径认识并且喜欢上 Clojure 的三个程序员。在写这本书的时候，我们试图把我们对于为什么使用以及如何使用 Clojure 的经验告诉大家以让你也可以很好地使用它。

Chas Emerick

Chas 从 2008 年的早期开始就一直活跃于 Clojure 社区。他给 Clojure 语言核心贡献过代码，参与过十多个 Clojure 开源项目，并且经常会撰写或者演讲一些有关 Clojure 或者软件开发方面的东西。

Chas 维护了 Clojure Atlas (<http://clojureatlas.com>)，这是一个以交互式的、可视化的方法帮助大家学习 Clojure 及其标准库的一个网站。

Chas 创建了 Snowtide 公司 (<http://snowtide.com>)，它坐落于马萨诸塞州西部，Chas 的主要领域在于非结构化数据的提取，并且特别擅长与 PDF 相关的数据提取。他经常在他的博客 (<http://cemerick.com>) 上发表一些有关 Clojure、软件开发、企业管理等方面的文章。

Brian Carper

Brian 是一个从 Ruby 转过来的 Clojure 语言爱好者。他从 2008 年开始就使用 Clojure 了，不管是自己业余的小项目还是工作中的项目，涉及的领域从 Web 开发到数据分析再到桌面应用编程等。

Brian 是 Gaka (<https://github.com/briancarper/gaka>) 的作者，这是一个把 Clojure 代码转换成 CSS 的编译器；他还是 Oyako (<https://github.com/briancarper/oyako>) 的作者，这是一个对象关系映射的类库。他在他的博客 <http://briancarper.net> 上发表了 Clojure 以及其他主题的一些文章。

Christophe Grand

Christophe 是一个迷失在 Java 世界的函数式编程爱好者，直到 2008 年遇到 Clojure，一见钟情！他是好几个软件的作者：Enlive (<http://github.com/cgrand/enlive>) 是一个 HTML/XML 转换、提取以及模板库；Parsley (<http://github.com/cgrand/parsley>) 是一个递增式的 parser 生成器；Moustache (<http://github.com/cgrand/moustache>) 是一个为 Ring 设计的路由及中间件应用的 DSL。

作为一个独立咨询者，他开发并且提供 Clojure 培训。他在他的博客 <http://clj-me.cgrand.net> 上撰写了有关 Clojure 的文章。

致谢

本书跟其他任何图书一样，如果没有那么多人不知疲倦的帮助的话是不会有这本书的。

首先要感谢 Clojure 语言的作者 Rich Hickey。在短短的几年里，他设计、实现并且把 Clojure 引入了这个世界。对于很多人来说这不只是另外一个工具而已，它重新激起了我们对编程的热爱。除此之外，他个人也教给了我们很多东西——当然是有关编程的，不过同时也学到了他的耐心、品格和眼光。多谢！Rich。

Dave Fayram 和 Mike Loukides 对于本书雏形的形成起了至关重要的作用。当然，如果没有我们的编辑 Julie Steele 以及所有 O'Reilly 那些好人，大家现在也看不到这本书，他们帮我们搞定了出版过程中的一切事宜。

如果没有审校人员的把关，这本书不会有这么高的质量。审校人员有：Sam Aaron、Antoni Batchelli、Tom Faulhaber、Chris Granger、Anthony Grimes、Phil Hagelberg、

Tom Hicks、Alex Miller、William Morgan、Laurent Petit 以及 Dean Wampler。同时我们也想感谢那些通过 O'Reilly 论坛、E-mail、Twitter 等途径给本书早期预览版提出反馈和建议的读者们。

Michael Fogus 和 Chris Houser 在很多方面给了我们灵感。其中之一就是他们在 *The Joy of Clojure* 一书中对于跟 REPL 交互内容的显示格式，我们直接照搬过来了。

如果这里少提了任何人，那么请接受我们默默的感谢以及诚挚的歉意，最后我们很高兴能把本书带给大家！

最后，当然感谢得还远远不够

Clojure 社区已经作为我的另外一个家好多年了。这个社区里面的成员都很热情，给我以帮助，是我的榜样。特别是经常在 #clojure 频道里出现的人们，不但跟他们成了好朋友，也从他们那里学到了从别的地方学不到的东西。

感谢我的合作者，Christophe 和 Brian：跟你们一起撰写本书对我来说是莫大的荣耀。没有你们的话，我是绝对完不成这本书的。

感谢我的父母，Charley 和 Darleen：我对于事物运作原理无法抑制的好奇心、我对于语言以及修辞的热爱、我对于商业的兴趣跟你们早年对我的影响是分不开的。没有你们影响的话，我是不会走上现在的路的：创建自己的软件公司以及撰写这本书。

最后，感谢我的妻子 Krissy：为了让我能追求自己的理想你做出了那么多牺牲。此时说什么都显得那么苍白，我只说一句：我爱你。

—Chas Emerick, 2012 年 2 月

感谢所有帮助创建 Clojure 这门语言的社区成员：感谢你们不知疲倦的辛苦工作，你们的工作使得我的职业以及个人编程生活变得那么有趣，使得我的眼界更加开阔。

感谢我的合作者 Christophe 和 Chas：我从来没有跟比你们聪明的人工作过。跟你们一起工作我很荣幸。

感谢我的妻子 Nicole：我每天晚上打字使得你不能入眠。

—Brian Carper, 2012 年 2 月

感谢 Rich Hickey，你创建了 Clojure 并且培育了这么一个友好的社区。

感谢 Clojure 社区，你们帮助我提高了我的水平。

感谢我的合作者，Brian 和 Chas：跟你们合作我很荣幸。

A mon professeur Daniel Goffinet, et à ses exercices improbables, qui a radicalement changé mon approche de la programmation et de l'informatique—sur ces sujets je lui suis plus redévable qu'à nul autre.

(感谢 Daniel Goffinet 教授，他从根本上改变了我对于编程以及计算的看法——在这些主题上我最感谢他。)

A mes parents pour votre amour bien sûr mais aussi pour tout le temps à s'inquiéter que je passais trop de temps sur l'Amstrad.

(感谢我的父母：感谢你们对我的爱以及帮我买的那个 8-bit 的计算机，你们还担心我在上面花太多时间呢。)

A ma compagne Emilie, et mon fils Gaël, merci d'être là et de m'avoir supporté pendant l'écriture de ce livre.

(感谢我的妻子 Emilie 和我的孩子 Gaël：感谢你们在我写作过程中对我的支持。)

—Christophe Grand, 2012 年 2 月

本书使用的一些格式约定

我们在本书中使用如下一些约定：

斜体 (*Italic*)

表示一个新词组、URL、邮件地址、文件名或者文件扩展名。

等宽字体 (**Constant width**)

用来表示程序代码，也会用在一个普通段落中用来引用变量或者函数名、数据库、数据类型、环境变量、表达式以及关键字。

以分号 (;) 开头的行

表示这行文字是由在 REPL 中运行的代码打印出来的（也就是说，打印到标准输出或者标准错误）。

以分号 + 等号 (;=) 开头的行

用来表示这是 REPL 中运行的代码的返回值。

加粗的等宽字体 (**Constant width bold**)

表示命令或者其他一些用户应该原封不动地输入的文本。

斜体的等宽字体 (*Constant width italic*)

表示这个文本应该被替换成用户提供的值或者它的值是由上下文确定的。



这个图标表示提示、建议或者一般的标注。



这个图标表示警告或提示。

中文版书中切口以“**□**”表示原书页码，便于读者与原英文版图书对照阅读，本书的索引中所列的页码为原英文版页码。

示例使用

本书的宗旨就是帮助你完成工作。一般而言，你可以在自己的程序和文档中随意使用书中的代码。除非你原样引用大量代码。否则无须联系我们获得授权。例如，在编写程序时引用了本书若干代码片段是无须授权的，然而销售或分发 O'Reilly 图书示例光盘则是需要授权许可的。通过引用本书内容以及代码的方式来答疑解难是不必有授权的。而将书中的代码大量加入到你的产品以及文档中，则需要授权。

如果你在引用书中内容时注明出处，我们将不胜感激，但是这不是必需的。引用声明通常是包含了标题、作者、出版商和 ISBN。例如：“*Clojure Programming* by Chas Emerick, Brian Carper, and Christophe Grand (O'Reilly). Copyright 2012 Chas Emerick, Brian Carper, and Christophe Grand, 978-1-449-39470-7.”

如果你发现自己对书中代码的使用有失公允，或是违反了前述条款，敬请通过 permissions@oreilly.com 与我们联系。

Safari® Books Online

Safari Books Online 是一个按需索取的电子图书馆，让你能够轻松地在超过 7500 本技术及创新参考书和视频中进行检索，快速找到你需要的答案。

订阅之后，你可以在我们的在线图书馆中阅读图书并观看视频；在你的手机或移动设备上读书；在印刷前看到新标题，独家阅读撰写中的原稿，向作者进行反馈；复制粘贴代码示例，管理你的收藏，下载章节内容，在关键部分加上书签，添加笔记，打印页面，并从大量其他省时的特性中获益。

O'Reilly Media 已将本书英文版上传至 Safari Books Online。要想访问本书或其他来自 O'Reilly 和其他出版商的类似主题，可以在 <http://my.safaribooksonline.com> 进行免费注册。

如何联系我们

请将对本书的评价和存在的问题通过如下地址告知出版者：

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站，你可以在那找到关于本书的相关信息，包括勘误列表、示例代码以及其他的信息。本书的网站地址是：

<http://shop.oreilly.com/product/0636920013754.do>

对于本书的评论和技术性的问题，请发送电子邮件到：

bookquestions@oreilly.com

关于本书的更多信息、会议、资料中心和 O'Reilly 网络，请访问以下网站：

<http://www.oreilly.com/>
<http://www.oreilly.com.cn/>

进入Clojure仙境

如果你正在读这本书，那么说明你已经准备学习一门新的语言了。同时你也应该意识到，这会花费你一些时间和精力在上面，作为回报，它会使你工作更高效，使你的团队工作更高效，使你的公司更有生命力、竞争力。

我们相信你在学习、使用 Clojure 的过程中会有一种互相反馈的感觉：Clojure 要求你提高自己的水平，而在你学习 Clojure 的过程中，它也会让你的水平越来越高。

作为软件开发人员，我们会有各自钟爱的一些工具和语言。到底使用哪个工具和语言来做个项目很多时候是由工具或者语言的实用程度以及遗留系统决定的。但是有一些东西是共通的：我们都希望使用那些能够最大化生产力，使得我们可以编写出有价值的、优雅的软件系统的工具。总结一句话：希望我们的工具可以让容易的事情保持容易，让困难的事情变得可能。

为什么要选择 Clojure？

Clojure 就是一门符合上面那个标准的语言！它融合了各种优秀语言的最好的特性——包括各种 Lisp 的实现，Ruby、Python、Haskell，等等。Clojure 提供了一些工具来解决程序员需要面对的最头痛的一些问题。同时 Clojure 不会让你做太大的改变——你无须学习新的架构或平台，Clojure 是架构在 JVM 平台上的，而 JVM 平台是当前用得最多、最广泛的平台。

为了让大家先尝点甜头，这里先给大家列举一些 Clojure 的主要特性。

Clojure 是运行在 JVM 上的一种语言

2

Clojure 代码可以使用任何 Java 类库，反之 Clojure 库也可以被任何的 Java 代码使用，

而且 Clojure 代码可以像 Java 代码一样被打包，然后被部署到任何 Java 应用可以部署的地方：Web 应用服务器、桌面应用 Swing、SWT 以及命令行等。这同时也表明，Clojure 所使用的运行时就是 Java 的运行时——世界上最高效、运转最稳定的平台之一。

Clojure 是 Lisp

跟 Java、Python、Ruby、C++ 以及其他编程语言不一样，Clojure 是 Lisp 方言之一。但是，要学习 Clojure 的话，你要把你所知道的有关 Lisp 的一切都忘掉：Clojure 保留了 Lisp 最好的那些特性，同时也去掉了其他 Lisp 方言的那些缺点。同时，作为 Lisp 的方言，Clojure 里面有宏，这是一种进行元编程以及语法扩展的强大工具，几十年来一直是衡量其他类似系统的基准。

Clojure 是一种函数式编程语言

Clojure 鼓励使用作为“头等公民”的高阶函数，并且提供了一些高效的、不可变的数据结构。选择设计成一门函数式语言使我们可以避免由于对状态的不加控制的修改而导致的一些常见 bug，并且也使 Clojure 成为一种进行并行、并发编程的完美语言。

Clojure 提供了进行并行、并发编程的创新式解决方案

现在多核、多 CPU 以及分布式计算已经逐步成为趋势，使得我们迫切需要一种在设计的时候就把并发编程的思想融入进去的语言和库。Clojure 的引用类型强制我们把对象的状态和对象的标识区分开。Clojure 对于多线程编程的支持使得我们不用手动加锁、解锁也能编写多线程的代码，就如同 Java 的自动垃圾回收功能使我们不用显式地请求、释放内存一样。

Clojure 是一种动态编程语言

Clojure 是动态的，同时也是强类型的（和 Python、Ruby 类似），Clojure 里面的函数调用会被编译（编译速度很快）成 Java 的方法调用。Clojure 另一种意义上的动态是指它支持在运行时更新现有代码或者加载新的代码，加载的代码可以在本地，也可以在远端。这对于交互式的开发以及调试，甚至在不停机的情况下给远端应用打补丁都很有用。

当然，我们并不指望你可以一下子明白上面说的这些优点，但是希望这些优点对你有些吸引力。如果是这样的话，继续读吧。读完这一章，你就可以写出简单的 Clojure 代码了，并且应该可以理解并且运用 Clojure 去挖掘你的潜能！

获取 Clojure

3

要跟我们一起练习本章中提到的代码的话，你需要两样东西：

1. Java 运行时 (JRE)。你可以在 Oracle 的官方网站上免费下载 JRE 的 Windows 以及 Linux 版本 (<http://java.com/en/download/>)；而对于 Mac OS X 来说，JRE 是系统自带的。Clojure 需要 JRE 1.5 或以上版本，而使用最新发布的 1.6 或 1.7 版本则更好。
2. Clojure 本身，可以在 [clojure.org 上面下载](http://clojure.org/downloads) (<http://clojure.org/downloads>)。这本书里面的代码需要 Clojure 1.3 以上版本才能运行，并且在 Clojure 1.4 上测试通过。^{注1} 在你下载的 zip 文件中会找到一个名为 *closure-1.4.0.jar* 的文件，这就是你所需要的全部文件。



对于各种常用的编程工具 IDE，比如 Eclipse 和 Emacs，都有很多 Clojure 插件；
398 页^{注2}“工具集”一节有关于 Clojure 各种工具的介绍。虽然 Clojure 自带的命令行 REPL 对于简单地尝试 Clojure 已经足够了，但是我们鼓励你使用你最喜欢的编辑器——如果你喜欢的 IDE 有很好的 Clojure 插件支持，否则选一个有很好插件支持的 IDE。

如果你不想使用某个特定的 IDE 的话，你至少应该使用 Leiningen，它是最受欢迎的 Clojure 项目管理工具。它会帮你自动下载 Clojure，而且它所带的 REPL 比 Clojure 自带的 REPL 要好很多，而且你将来也会每天都使用 Leiningen 来管理你的项目。347 页的“Leiningen”一节有关于它的详细介绍。

如果你现在还不想下载任何东西的话，可以在线运行本书里面的很多例子，<http://tryclj.com> 上有一个浏览器内的 Clojure 实现。

Clojure REPL

很多语言都有 REPL，它通常也被称为解释器：Ruby 有 irb，Python 有它的命令行解释器，Groovy 有它的控制台，就连 Java 也有 BeanShell。REPL 其实是几个单词的缩写：

1. 读入 (Read)：代码被作为字符串从输入源读入（通常是标准输入，但是如果你是从 IDE 或者其他非控制台环境里面运行 REPL 的话，那就不一定了）。
2. 求值 (Eval)：代码被求值，产生一个结果。
3. 打印 (Print)：求值的结果被打印到某个输出设备（通常是标准输出，但是如果你是从 IDE 或者其他非控制台环境运行 REPL 的话，那就不一定了）。

4

注1：因为 Clojure 的发展一直保持了很好的向后兼容性，所以本书中所讲述的概念以及提到的代码示例也适用于未来新版本的 Clojure。

注2：本书正文中标注的参见页码及索引中标注的页码均为英文原书页码，英文原书页码在正文切口处以“□”形式标注。

4. 循环 (Loop)：控制重新跳回到读入 (read) 阶段。

Clojure 语言也有 REPL，但是和其他语言是很不一样的。Clojure 的 REPL 不是单纯的解释器或者是一个受限制的、轻量级的 Clojure 的子集：你在 Clojure REPL 中输入的 Clojure 代码都会被编译成 JVM 的字节码——跟直接执行一个 Clojure 源代码文件产生的字节码是一样的。在这两种情况下，“编译”都是在运行时进行的——不需要一个单独的“编译”步骤^{注3}来触发。实际上，Clojure 从来没有被解释执行过。这也就意味着下面两件事情：

1. 在 REPL 中执行的代码是在“全速”运行。也就是说，在 REPL 中运行代码与把代码写在 Clojure 源码文件中再运行相比没有任何额外运行时开销。
2. 一旦你理解了 Clojure REPL 的工作原理（特别是它的读入和求值阶段），就从根本上理解了 Clojure 本身是如何工作的。

知道了上面说的第二点，让我们深入 Clojure REPL 去看看能不能一窥它的原理。



Clojure 最有效的编程方式中大量使用了 REPL，跟其他语言的典型开发方式不一样的是：它使我们的开发过程有尽可能多的交互性。很强的交互性可以大大提高工作效率，并且非常有意思——都是 Clojure 带来的哦。我们会在第 10 章详细介绍这一点。

示例 1-1：从命令行启动 Clojure REPL

```
% java -cp clojure-1.4.0.jar clojure.main  
Clojure 1.4.0  
user=>
```

上面的这个命令启动了一个新的 JVM 进程，而启动的 classpath 包含了当前目录中的 *clojure-1.4.0.jar* 文件，并且以类 *clojure.main* 作为它的主入口^{注4}。如果你还不知道什么是 classpath 的话，那么我们在 331 页的“classpath 入门”一节有详细的介绍。就现在来说，你可以把 classpath 当成 Python 里面的 **PYTHONPATH**，Ruby 里面的 **\$:**，或者 shell 里面的 **PATH**，其实就是一些指定的目录或者文件，而 JVM 会从这些地方加载类文件和资源文件。

当你看到命令提示符 *user=>* 的时候，REPL 就已经加载完成了，你可以开始输入 Clojure 代码了。Clojure REPL 的命令行提示符中 => 之前的部分 (*user*) 是当前命名空间 (namespace) 的名称。可以把命名空间理解成模块 (module) 或者包 (package)，我们会在 20 页的“命名空间”一节详细介绍。Clojure 的 REPL 会话始终是从默认的 *user*

注 3：如果有需要的话，你也可以进行“预先编译”，把 Clojure 代码编译成 Java class 文件。337 页的“提前编译”一节会有详细介绍。

注 4：或者你可以使用 *java -jar clojure.jar*，但是 -cp 标志以及程序入口 *clojure.main* 都是非常重要的，大家要知道。我们在第 8 章会详细介绍。

命名空间开始的。

下面让我们来看一些真实的代码吧，一个计算平均数的函数，分别用 Java、Ruby 和 Python 来实现。

示例1-2：分别用Java、Ruby和Python来求平均数

```
public static double average (double[] numbers) {  
    double sum = 0;  
    for (int i = 0; i < numbers.length; i++) {  
        sum += numbers[i];  
    }  
    return sum / numbers.length;  
}  
  
def average (numbers)  
    numbers.inject(:+) / numbers.length  
end  
  
def average (numbers):  
    return sum(numbers) / len(numbers)
```

下面是 Clojure 的实现：

```
(defn average  
  [numbers]  
  (/ (apply + numbers) (count numbers)))
```

① defn 在当前的命名空间里定义了一个名叫 average 的函数。

② 这个 average 函数接受一个名为 numbers 的参数。要注意的是，在这里我们没有给参数指定类型，所以可以传给这个函数任何类型的集合或者数组，并且集合或者数组中的数字可以是任何数字类型 (long、double、float 等)，这个函数都可以正常工作。

③ 这个 average 函数的函数体通过 (apply + numbers) 来对它们求和，^{注5} 然后把这个和与数字的个数相除，数字的个数是通过 (count numbers) 来获取的。相除的结果即为我们要求的平均数。

我们把这个函数的定义代码输入到 REPL 里面，然后调用这个函数，传给它一个包含数字的 vector，下面是结果：

```
user=> (defn average  
          [numbers]  
          (/ (apply + numbers) (count numbers)))
```

注5： 注意，这里的 + 并不像其他语言中作为一个特殊的语言操作符存在。它只是一个普通的函数，跟我们自己定义的函数没有任何区别。apply 同样也是一个函数，它以提供给它的集合参数（这里是 numbers）来调用传给它的函数；因此这里的 (apply + [a b c]) 调用跟 (+ a b c) 是等价的。

```
#'user/average
user=> (average [60 80 100 400])
160
```

6

有关 REPL 交互方式的一些约定

从这里开始，为了简化我们的表达，后面对于一段代码的输出结果会用 ;= 前缀来表示，比如：

```
(average [60 80 100 400])
;= 160
```

而代码输出到 stdout 的内容我们则用 ; 前缀来表示，比如：

```
(println (average [60 80 100 400]))
; 160
;= nil
```

上面的 160 是代码打印的内容，所以用 ; 前缀来表示。而 nil 是 println 函数的返回值，所以用 ;= 前缀来表示。

在 Clojure 中，以 ; 开头的语句其实是注释。所以你可以把本书中提供的这些示例代码直接输入到 REPL，我们在后面举例子的时候都不会包含命令提示符 user=>，这也使得大家在复制粘贴代码的时候更加方便。

不！括号真的不会让你瞎了眼

许多没有用过 Lisp 或者上一次用 Lisp 还是在大学的程序员对于 Lisp 的语法很不习惯。典型的原因包括：

1. 使用小括号 (...) 来定义代码范围，而不是更常见的大括号 {...} 或者 do ... end。
2. 使用前缀表示法：(+ 1 2)，而不是更常见的中缀表示法：1 + 2 来表示一个操作。

这些反对的声音通常只是简单地因为他们对 Lisp 不熟悉。Java（以及 C、C++、C#、PHP 等）用来划定作用域的花括号看起来不是很好吗？为什么要用一个看起来很奇怪的小括号来划定作用域呢？类似的，我们从小开始都是使用中缀表示法来做数学计算，中缀表示法是多么自然啊！我们为什么要用前缀表示法呢？我们都是习惯的生物，除非是

为了了解事物之间的差异才去理解新的事物、新的方式，我们通常总是喜欢我们所熟悉的事物。

对于上面两个质疑，我们的回答都是 Clojure 不是简单地从其他 Lisp 实现中把语法搬过来，我们所搬过来的语言特性都非常棒，是值得我们对习惯做细微改变的。从以下两个方面来看：

- 统一使用前缀表示法可以极大地简化 Clojure 的语法，并且消除了复杂表达式产生二义性的可能。
- 而对于小括号的使用（是 Lisp 里面列表的表示方法）是 Clojure 作为一个同像性语言的自然结果。关于什么是同像性我们会在第 9 页的“同像性”一节详细介绍，它的好处是很多的。同像性使得在其他语言中不可能做的事情，如开发和进行元编程以及领域特定语言（DSL），在 Clojure 中变成可能。

只要对 Clojure 稍加熟悉之后，你就会渐渐喜欢上 Clojure 的语法，因为它会大大降低你读 / 写代码时所需要思考的时间。比如，在 Java 里面，`<<`（位左移）跟 `&`（位与）到底谁先执行，谁后执行？每当程序员读到这种代码的时候，都要停下来想一想，有时还要查一查手册。每当程序员需要停下来，在这些代码的两边加上括号“以防万一”的时候，他的大脑中其实就产生了一个“Page Fault”。并且，每当程序员忘记考虑这个优先级的事情的话，一个潜在的 bug 就被引入他的代码了。想象一下，如果你有一门语言完全不需要考虑优先级会有多爽吧！Clojure 就是这样的一门语言！

你可能会说：但是 Clojure 中的小括号也太多了吧？

其实在任何合适的地方，Clojure 都从其他语言中借用了许多语法——比如 Ruby，Clojure 从它里面借用了数据字面量（Data Literals）。而其他的 Lisp 方言则是到处使用小括号，Clojure 提供了数据和集合（vector、map、列表、set）的丰富的字面量，还包括记录类型（可以认为是 Clojure 中的 struct）。

如果你数数同等规模的 Clojure、Java、Ruby 以及 Python 代码中这些划定作用域的字符个数的话（`()`、`[]`、`{}` 以及 Ruby 的 `||` 和 `end` 等），你会发现 Clojure 不会比其他语言多，而由于 Clojure 语言很简洁，通常会更少。

表达式、操作符、语法以及优先级

所有的 Clojure 代码都是由表达式组成的，每个表达式会求值产生一个值。这跟其他很多语言依赖于大量无值控制语句不一样，比如跟 `if`、`for` 以及 `continue` 来命令式地控制

程序流程不一样，Clojure 中的这些控制性的语句都是有值的表达式，跟其他普通表达式没有本质区别。

我们前面看过 Clojure 里面几个表达式的例子了：

- 60
- [60 80 100 400]
- (average [60 80 100 400])
- (+ 1 2)

这些表达式都会求值产生一个值。而如何对这些表达式进行求值的规则跟其他语言相比要简单很多：

- 8
1. 列表（用小括号表示）表示函数调用，列表中的第一个值表示操作符，剩下的值表示参数。我们通常称列表中的第一个元素是函数位置（因为这个位置是我们指定函数名，或者命名被调用函数的符号的位置），整个调用表达式被求值成这个调用的返回值。
 2. 符号（比如前面看到的 `average` 和 `+`）会被求值成它们在当前作用域中的值，这个值可能是一个函数，一个本地绑定，比如 `average` 例子里面的 `numbers`，一个 Java 类，一个宏，一个特殊形式（special form）。我们会在后面介绍特殊形式，现在你只要把它当成函数就好了。
 3. 所有其他表达式求值成它们的字面量：是什么就是什么，比如 1 就是数字 1，“james”就表示一个字符串。



Lisp 里面的列表通常被称为 S 表达式（s-expression）或者 sexprs——符号表达式（symbolic expressions）的简称。正确的并且能够被成功求值的 S 表达式我们称为形式（form）；比如 `(if condition then else)` 是一个 if 形式，`[60 80 100 400]` 是一个 vector 形式等。并不是所有 S 表达式都是形式，比如 `(1 2 3)` 是一个表示有三个数字的正确的 S 表达式，但是它没办法被成功地求值。因为这个列表里的第一个元素是一个数字，不能被调用，因此不是一个形式。

上面的第 2 点和第 3 点在其他语言中基本是等价的（虽然我们稍后会看到 Clojure 的字面量的表达能力更强），但是我们只要稍微看一下在其他语言比如 Java、Ruby 中方法调用的原理，就可以看出它们语法的复杂性。

表1-1: Clojure、Java、Python和Ruby中的函数调用语法比较

Clojure 表达式	对应的 Java 语法	对应的 Python 语法	对应的 Ruby 语法
(not k)	!k	not k	not k 或者 !k
(inc a)	a++, ++a, a += 1, a + 1 ^a	a += 1, a + 1	a += 1
(/ (+ x y) 2)	(x + y) / 2	(x + y) / 2	(x + y) / 2
(instance? java.util.List al)	al instanceof java.util.List	isinstance(al, list)	al.is_a? Array
(if (not a) (inc b) !a ? b + 1 : b - 1 (dec b)) ^b	b + 1 if not a else !a ? b + 1 : b - 1	b - 1	
(Math/pow 2 10) ^c	Math.pow(2, 10)	pow(2, 10)	2 ** 10
(.someMethod someObj "foo") ^d	someObj.someMethod("foo")	someObj.someMethod("foo", ("foo", otherObj))	someObj.someMethod
(.otherMethod otherObj otherObj 0))	otherObj.otherMethod(0))	otherObj.otherMethod(0))	otherMethod(0))

^a 递增和递减操作在 Clojure 中没有直接对应的函数，因为 Clojure 不支持这种对于值的随意修改，在第 2 章，特别是第 52 页“值的重要性”一节我们会详细解释这么设计的好处。

^b 记住，即使是那些影响控制流程的形式在 Clojure 中也会跟其他表达式一样求值成一个值的，包括 if 和 when。这里 if 表达式的值要么是 (inc b)，要么是 (dec b)，具体取决于 (not a) 的值。

^c 这里可能是第一次给大家展示如何从 Clojure 中调用 Java 代码。我们会在第 9 章详细介绍。

我们可以看到表格中充斥着各种语法（下面以 Java 为例，Python 和 Ruby 也是一样的）：

- 我们使用了一些中缀表达式（比如 a + 1, al instanceof List），但是任何稍微复杂一点的代码都需要使用大量的小括号来覆盖默认的优先级规则，使得代码执行优先级更加明显。
- 对于一元表达式的使用很随意，你可以使用前缀方式（比如 !k 和 ++a），也可以使用后缀表达式（比如 a++）。
- 对于静态方法的调用，我们把要调用的静态方法所在的类放在整个表达式的最前面，比如 Math.pow(2, 10)，但是……
- 对于实例方法的调用使用的则是一种在 Java 中不常见的中缀表达式，把该方法所作用的对象放在了“中间”——就是那个 this，它被放在方法名称之后，所有正式参数之前。^{注6}

形成鲜明对比的是，Clojure 的调用表达式遵循一个极其简单的规则：列表里的第一个值是操作符，其余的都是这个操作符的参数。从来不使用什么中缀表达式或者后缀表达

注 6： Python 也使用类似的中缀表达式来表示实例方法调用，但是跟 Java 不一样的是，它要求用户手动传递这个参数，这个参数通常名为 self。

式，也没有什么很复杂的操作符优先级需要背，这种简化使得 Clojure 的语法非常容易学，非常容易记，同时也使得 Clojure 的代码非常容易阅读。

同像性

Clojure 的代码是由 Clojure 自身的数据结构：原子值（字符串、数字等）和集合的字面量来表示的。这种特征的学名叫做“同像性”，我们一般称为“代码即数据”。^{注7} 这相对于其他语言是极大的简化，同时也使得 Clojure 语言很适合用来进行元编程——而其他语言则不行。要想理解这里面的道理的话，我们要从一般意义上来说说语言和语言的内部表示之间的关系。

还记得 REPL 的第一个阶段是读入你输入的代码吗？每种语言都提供了一种机制去把字符形式的代码转换成一种可以被编译、求值的内部表示。大多数语言是把表示代码的文本转换成抽象语法树（Abstract Syntax Tree, AST）。这个听起来好像很复杂的东西其实很简单：抽象语法树是一种用来表示文本形式的代码所表达含义的正式模型的数据结构。比如图 1-1 所示的是一些文本代码和其所对应的抽象语法树。^{注8}

10 从文本表示到 AST 的转换是一门语言设计的核心，语言的表达力如何，语言与它的目标问题领域有多契合，大多数领域相关语言的优点都在这里，如果你设计的一门语言是为了解决特定领域的问题，那么这个领域的专家使用起你的语言将会很简单，并且比使用那些通用语言（Java、C++ 之类的）更容易表达。

11 这种方式的缺点是大多数语言没有提供用来控制其抽象语法树的办法。语言的文本表示与它们抽象语法树之间的对应关系完全由语言的实现者定义。这迫使一些聪明的程序员使用一些看上去很“聪明”的办法来提高他所写代码的表达力和他使用该语言所拥有的工具的集合：

- 代码生成。
- 文本宏和预处理器（C 和 C++ 程序员用了几十年了）。
- 编译器插件（比如 Scala, Java 里面的 Lombok, Groovy 的 AST 转换以及 Haskell 的模板）。

注 7：Clojure 并不是唯一一个同像性语言，同像性本身也不是什么新概念。其他的同像性语言包括所有 Lisp 方言、各种机器语言（包括汇编语言在内）、Postscript、XSLT、XQuery、Prolog、R、Factor、Io 等。

注 8：关于自然语言解析树可以看这里：http://en.wikipedia.org/wiki/Parse_tree。

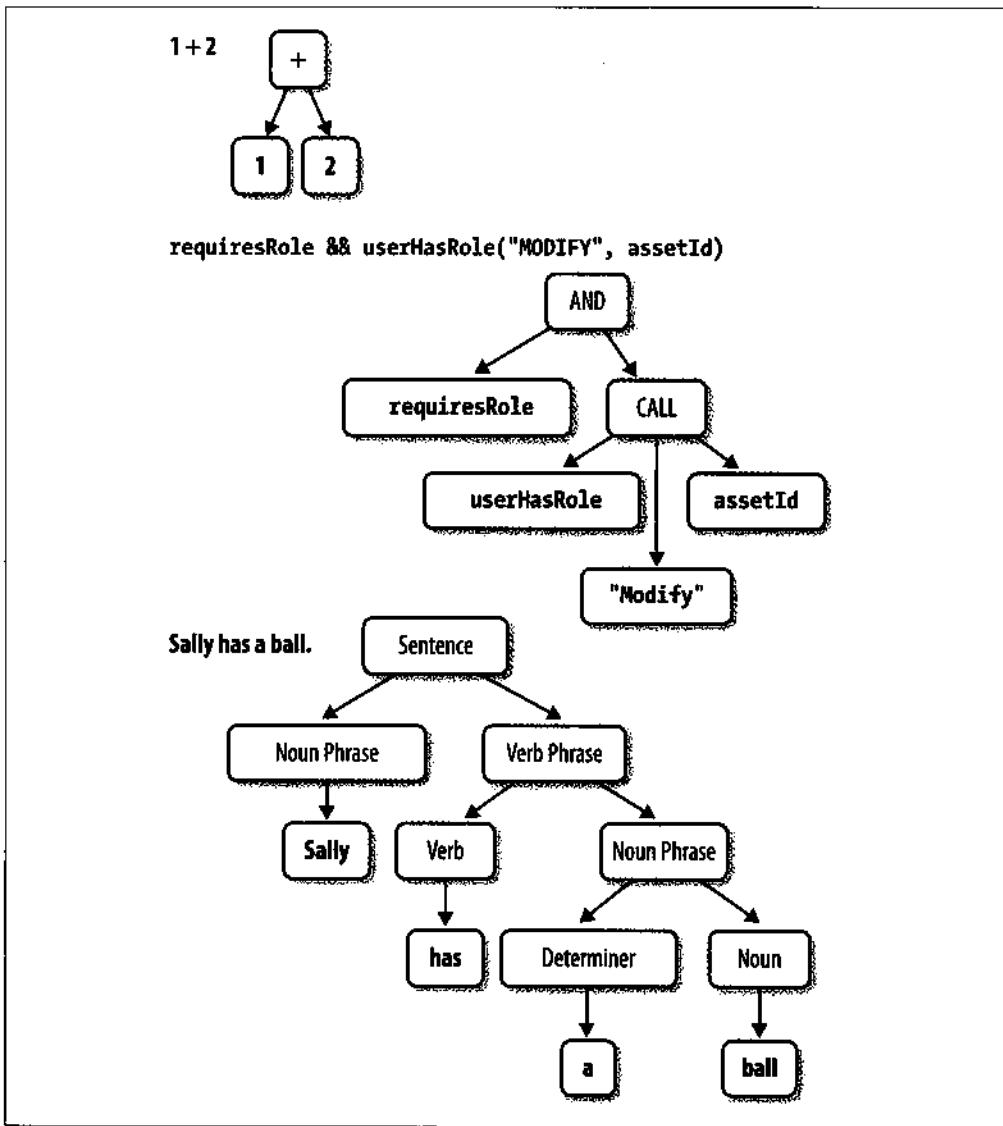


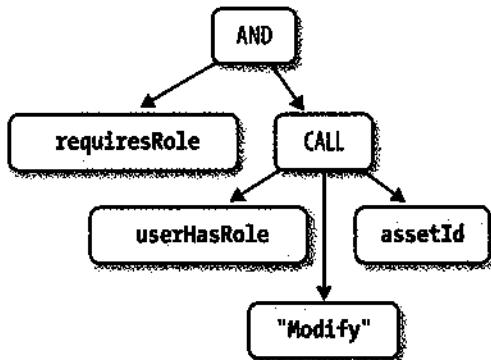
图1-1：文本代码到语法树的转换示例

这些通常都是非常复杂的——之所以复杂是因为语言的设计者把语言的文本语法当做语言的唯一表示方式，使得语言的内在表示方式跟作者的实现方式强烈相关（虽然这些内在表达方式已经暴露了）。

Clojure 和所有的 Lisp 一样使用了一种不一样的方式：没有定义一种将会被转换成 AST 的语法，Clojure 代码是直接用表示抽象语法树的 Clojure 数据结构来写的——也就是说，你写下的 Clojure 代码相当于其他语言里面的抽象语法树。回忆一下图 1-1 里面提到

的 `requiresRole`, 再来看看它对应的 AST 是什么样子的（回忆一下我们前面说的，在 Clojure 中，列表的第一个元素是函数位置）。

```
(and requiresRole (userHasRole "MODIFY" assetId))
```



Clojure 使用数据来表示语言代码的特性使得 Clojure 代码可以很容易地用来编写和转换其他 Clojure 代码。这是宏（Macro）的基础，Clojure 中的元程序编程工具要比 C 语言中提供的那种宏以及其他文本预处理器要强劲很多。也是当我们对于语言表达能力以及 DSL 有强烈需求的时候最终拯救我们的一大利器。我们将在第 5 章详细学习 Clojure 中的宏（Macro）。

从实用的角度来看，代码和数据之间这种直接的对应关系也意味着你在 REPL 里面写的代码根本就不是简单的文本代码：你在使用 Clojure 的数据结构在编程。回想一下示例 1-2 里提到的 `average` 函数：

```
(defn average
  [numbers]
  (/ (apply + numbers) (count numbers)))
```

这不是一堆简单的文本，然后通过某个神奇的黑盒子最后转换成了一个函数定义。这整个是一个列表数据结构，这个列表包含了 4 个元素：一个符号 `defn`、一个符号 `average`，一个 vector 表示函数的参数，最后一个元素也是一个列表，这个列表就是函数的函数体，而对这个列表进行求值其实就是对函数的定义。

Clojure Reader

虽然 Clojure 的编译和求值全部是对 Clojure 的数据结构进行的，但是对于普通 Clojure 程序员来说代码还是写成文本格式的。那么我们就需要一种机制来把文本形式的代码处理成 Clojure 的那些数据结构。这也就是我们这一节要谈的 Clojure reader 的职责。

reader 的所有操作是由一个叫做 `read` 的函数定义的，这个函数从一个字符流⁹里读入代码的文本形式的，产生这个文本形式所对应的数据结构。Clojure 的 REPL 就是使用 `reader` 来读入文本代码的；`read` 函数读出的每个完整数据结构都会传给 Clojure 的运行时来求值。

一个对于我们尝试 Clojure reader 原理更方便的函数是 `read-string`，这个函数与 `read` 函数做的是同样的事情，但是接受一个字符串作为参数：

```
(read-string "42")
;= 42
(read-string "(+ 1 2)")
;= (+ 1 2)
```

`reader` 的作用其实可以看做是一种反序列化机制。Clojure 的数据结构和其他字面量有特定的文本表示，而 `reader` 则把这些特定的文本表示反序列化成对应的值和数据结构。

你可能已经意识到了，Clojure REPL 打印出来的 Clojure 的值跟你输入 REPL 的形式是一样的：数字和其他原子值还是你输入的那样，列表以小括号分隔，vector 以中括号分隔。这是因为与 `read` 和 `read-string` 对应的两个函数：`pr` 和 `pr-str`，它们把 Clojure 里面的值打印到 `*out*`¹⁰，并且把这个值返回。所以 Clojure 的数据结构和值序列化之前都是既是对人可读，也对机器可读的：

```
(pr-str [1 2 3])
;= "[1 2 3]"
(read-string "[1 2 3]")
;= [1 2 3]
```



Clojure 应用不像 Java 应用那样使用 XML 或者 `java.io.Serializable` 来做序列化，它通常用 Clojure 本身的 `reader` 来做序列化，特别是当序列化的格式要让人可读的时候。

<13

标量字面量

标量字面量是 Clojure 中非集合类型值的语法形式。很多字面量你在其他语言比如 Java、Ruby、Python 中都已经接触过了，而另外一些则是 Clojure 所独有的并且有它独特的语义。

注 9：从技术上来说，`read` 需要一个类型为 `java.io.PushbackReader` 的参数，不过这是一个实现细节。

注 10：`*out*` 默认指向标准输出 `stdout`，但是我们可以很简单地把它重定向到别的地方。72 页的“利用可组合的高阶函数来构建一个日志系统”一节提供了一个例子。

字符串

Clojure 中的字符串就是 Java 的字符串（也就是说，是 `java.lang.String` 的实例），而且表示形式也是一模一样的，左右两边以双引号括起来：

```
"hello there"  
:= "hello there"
```

Clojure 的字符串天然支持多行，你不用使用任何特殊的语法形式（跟 Python 类似），比如：

```
"multiline strings  
are very handy"  
:= "multiline strings\nare very handy"
```

布尔值

和 Java、Ruby、Python 一样，Clojure 中用 `true` 和 `false` 来表示布尔值。

nil

Clojure 中的 `nil` 跟 Java 中的 `null`，Ruby 里面的 `nil`，Python 里面的 `None` 是类似的。在 Clojure 的条件判断中，与 Ruby 和 Python 类似的是，`nil` 是逻辑 `false`。

字符

Clojure 中的字符字面量是通过反斜杠加字符来表示的：

```
(class \c)  
:= java.lang.Character
```

而且对于 Unicode 编码和 octal 编码，我们可以使用对应的前缀：

```
\u00ff  
:= \ÿ  
\o41  
:= \!
```

同时对于一些特殊字符，也有对应的常量：

- `\space`
- `\newline`
- `\formfeed`
- `\return`

- \backspace
- \tab

关键字

关键字求值成它们自身，经常被当做访问器来获取它们对应的值，比如下面的 map 和 record：

```
(def person {:name "Sandra Cruz"
            :city "Portland, ME"})
:= #'user/person
(:city person)
:= "Portland, ME"
```

这里我们创建了一个包含两对值的 hashmap，两个键分别是 :name 和 :city，然后以 :city 作为键去查它所对应的值。之所以可以这么做是因为关键字本身是函数，它的作用就是查找它所对应的值。

从语法上来说，关键字始终以冒号开头，它可以包含任意的非空字符。如果关键字里面包含 /，表示这个关键字是命名空间限定的，而如果一个关键字以两个冒号 (::) 开头的话，那么表示是当前命名空间的关键字，如果一个关键字以两个冒号开头，同时又包含了 / 的话，比如 ::alias/kw，那么表示是某个特定命名空间里面的关键字。这个设计与 XML 里面的命名空间实体的用法和动机是一样的。也就是为了让同一个名字在不同的命名空间里面可以有不同的值和语义。^{注11}

```
(def pizza {:name "Ramunto's"
            :location "Claremont, NH"
            ::location "43.3734,-72.3365"})
:= #' user/pizza
pizza
:= {:name "Ramunto's", :location "Claremont, NH", :user/location
"43.3734,-72.3365"}
(:user/location pizza)
:= "43.3734,-72.3365"
```

15

这使得同一个应用的不同模块、同一个组织的不同部门可以安全地、自由地使用关键字，而不必为了避免名字冲突而设计复杂的领域模型或者一些复杂的命名规范。

关键字是一种命名类型，之所以这么说是因为它们有自己内的名字（可以通过 name 函数访问），以及一个可选的命名空间（通过 namespace 函数访问），比如：

^{注11}： 命名空间限定的关键字在多重方法以及 isa? 体系里面也有很多使用，我们将在第 7 章详细讨论。

```
(name :user/location)
:= "location"
(namespace :user/location)
:= "user"
(namespace :location)
:= nil
```

另一种命名类型的值是我们下一节要介绍的符号（Symbol）。

符号

跟关键字一样，符号也是一种标识符，但不同的是，符号的值是它所代表的 Clojure 运行时里面的那个值。这个值可以是 var 所持有的值（持有的可以是函数以及其他值）、Java 类、本地引用等。我们再回过头来看看示例 1-2：

```
(average [60 80 100 400])
:= 160
```

这里的 average 就是一个符号，代表是一个名叫 average 的 var 所指向的函数。

符号不能以数字开头，但是跟 Java 以及其他语言中不大一样的是，符号的名字中不但可以包括数字、字符，还可以有 *、+、!、-、_ 以及 ? 等特殊字符。如果一个符号里面包含有斜杠 (/)，那么表示这个符号是命名空间限定的符号，而它所代表的值也会求值成这个符号在那个命名空间里面的值。符号到底求值成什么值是由它所在的上下文和命名空间决定的。我们会在 20 页的“命名空间”一节详细介绍命名空间以及符号的求值。

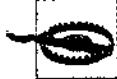
数字

Clojure 提供了很多表示数字的方法（详见表 1-2）。有一些是你在其他编程语言里已经见过的，而另一些则是 Clojure 所特有的，这些特有的数字表示法会简化一些特定算法的实现，特别是当这些算法要用到数字的某种特殊表示法（八进制、二进制、有理数以及科学计数法等）时。

16 表1-2：Clojure中的数字字面量

字面量语法	数字类型
42、0xff、2r111、040	long(64位带符号整数)
3.14、6.0221415e23	double(IEEE 标准的 64 位浮点数)
42N	clojure.lang.BigInt(任意精度的整数 ³)
0.01M	java.math.BigDecimal(任意精度的浮点数)
22/7	clojure.lang.Ratio

³ 当需要的时候，clojure.lang.BigInt 会被自动转换成 java.math.BigInteger。详细细节还是参见第 11 章有关 Clojure 对数字处理方法的深入讨论吧。



虽然 Java 运行时中支持很多种范围不同的数字类型（比如 32 位的 int, 64 位的 long），Clojure 也支持和这些不同长度的数字类型进行互操作，但是 Clojure 中的数字都使用比较长的 long 和 double，代价就是 int、byte、short、int、float 这些短数字占了不必要的位数。这意味着当要和 Java 程序进行互操作（比如调用一个 Java 方法）的时候，Clojure 会从大数 (long) 转成对应的需要的小数 (int、byte 等)，但是纯 Clojure 操作都默认使用大数来表示。

对于大多数的编程领域来说，你不需要关心这个。如果数字精度对于你所写的代码很重要，你可以去第 11 章看看，我们在第 11 章详细介绍了 Clojure 对于各种数字类型的处理以及相关的一些其他主题。

在任何一个数字前面加上一个横线 (-) 就变成了相对应的负数。

下面我们来看一些有趣的数字表示法。

十六进制

跟大多数语言一样，Clojure 支持十六进制表示法：0xff 表示 255，0xd055 表示 53333 等。

八进制

以 0 开头是八进制数字的表示法。比如，040 表示十进制中的 32。

“任意”进制

你可以以下面这种形式给数字指定任意进制：B r N，其中 N 是你要表示的数字，而 B 则表示进制。比如可以使用 2r 表示二进制的数字（2r111 表示 7），16r 表示十六进制（16rff 表示 255）等，最高支持 36 进制。^{注 12}

任意精度的数字

Clojure 支持任意精度的数字，关于这方面的更详细的信息请查看 428 页的“有限与任意精度”一节，我们会在那里解释为什么这么设计，以及这种设计在什么时候特别有用。

有理数

Clojure 直接支持有理数，有理数又称为比例数，Clojure 中的有理数表示方法用的就是我们学的那个数学表示法：分子、分母中间有一个斜杠 (/)。

想了解更多有关 Clojure 的有理数，以及 Clojure 中有理数如何跟 Clojure 中其他数字类型交互的话，可以看看 422 页的“有理数”一节。

注 12：这个限制是 java.math.BigInteger 类造成的。注意，虽然我们在求值的时候是用 BigInteger 这个类求值的，但是 reader 求值出来的数字的具体类型还是跟其他 Clojure 的数字字面量是统一的：要么是一个 long 或者是一个支持任意精度的大整型。

正则表达式

Clojure 里面把以 # 开头的字符串当做正则表达式：

```
(class #"(p|h)ail")
:= java.util.regex.Pattern
```

这跟 Ruby 里面的 /.../ 语法类似，只是模式分隔符有一点小区别。实际上，Ruby 和 Clojure 在处理正则表达式方面是非常相似的：

```
# Ruby
>> "foo bar".match(/(...) (...)/).to_a
["foo bar", "foo", "bar"]

;; Clojure
(re-seq #"(...)" "...") "foo bar")
:= ([["foo bar" "foo" "bar"]])
```

Clojure 中的正则表达式表示方法使我们不需要像在 Java 中那样对反斜杠进行转义：

```
(re-seq #"\d+-\d+" "1-3")      ;; 在 Java 里面需要写成 "\\\d+-\\\\d+"
:= ([["1-3" "1" "3"]])
```

Clojure 中正则表达式语法所创建出来的 `java.util.regex.Pattern` 跟在 Java 中创建出来的没有什么区别，因此可以在 Clojure 代码中使用 `java.util.regex` 中提供的任何方法。^{注13} 这样你就可以直接调用 `Pattern` 类上的任何方法，但是你也会发现，Clojure 中定义的正则表达式的相关函数更简单易用（比如 `re-seq`、`re-find`、`re-matches` 以及 `clojure.string` 命名空间中定义的其他方法）。

18

注释

Clojure 支持两种注释：

- 以分号 (;) 开头的单行注释，在这一行中，分号后面的所有内容都会被忽略。这个和 Java、JavaScript 中的 // 以及 Ruby、Python 中的 # 作用是一样的。
- 形式级别的注释 `#_` 宏。这个宏告诉 reader 忽略下一个 Clojure 形式。

```
(read-string "(+ 1 2 #_(* 2 2) 8)")
:= (+ 1 2 8)
```

为什么一个有 4 个数字的列表——(+ 1 2 4 8) 被求值成只有三个数字呢？因为第三个形式使用了 `#_` 宏，因此被忽略了。

由于 Clojure 的代码是使用 Clojure 自身的数据结构定义的，所以这种形式级别的注释要

注 13： 查看 `java.util.regex.Pattern` 美的文档以了解 Java 正则表达式所支持的各种正则语法：<http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html>。

比 Java 以及其他语言里面的多行注释 /* */ 要有用。比如看下面的例子，为了调试，我们使用 `println` 打印了一些信息到标准输出 `stdout`：

```
(defn some-function
  [...arguments...]
  ...code...
  (if ...debug-conditional...
    (println ...debug-info...)
    (println ...more-debug-info...))
  ...code...)
```

当调试好了怎么把这些 `println` 去掉呢？只要在 `if` 调用的前面加上 `#_`，然后重新加载一下这个函数定义就可以了，`#_` 的好处在于，它注释的边界是一个形式，而不管你的形式是几行还是几百行。



还有另外一种注释代码的方法是使用 `comment` 宏。比如：

```
(when true
  (comment (println "hello")))
:= nil
```

`comment` 里面可以包含任意数量的代码，但是跟 `#_` 不同的是，`comment` 里面的代码并没有被 `reader` 彻底忽略，`comment` 形式的返回值始终是 `nil`，大多数时候这个是无关紧要的。但是有些时候也有些不方便，比如把前面 `#_` 的例子用 `comment` 宏改写一下，它就要抛出 `NullPointerException` 了，因为 `nil` 对于 `+` 函数来说不是一个合法参数。

```
(+ 1 2 (comment (* 2 2)) 8)
;= #<NullPointerException java.lang.NullPointerException>
```

空格和逗号

19

你可能意识到了，在形式之间、函数的参数之间、数据结构的元素之间都没有使用逗号分隔的：

```
(defn silly-adder
  [x y]
  (+ x y))
```

这是因为对于分隔参数、分隔形式的目的来说，空格就已经够了。而且在 `reader` 的眼里，逗号就是空格。下面的这段代码跟上面的代码是一样的：

```
(defn silly-adder
  [x, y]
  (+, x, y))
```

我们来证明一下：

```
(= [1 2 3] [1, 2, 3])
:= true
```

到底要不要写逗号完全是个人的喜好。一般来说，逗号一般会在希望提高代码可读性的时候派上用场，比如当一个 map 里面有多对值，而多对值写在同一行的时候：^{注14}

```
(create-user {:name new-username, :email email})
```

集合字面量

reader 给 Clojure 中最常用的一些数据结构提供了对应的语法：

```
'(a b :name 12.5)      ;; list
['a 'b :name 12.5]     ;; vector
{:name "Chas" :age 31}  ;; map
#{1 2 3}                ;; set
```

由于 Clojure 中使用列表来表示函数调用，所以当要表示数据结构的时候要在列表的前面加一个单引号，以防止列表被求值成一个函数调用。

我们会在第 3 章详细讨论 Clojure 的集合类型。

20 reader 的一些其他语法糖

reader 还提供了一些其他的语法糖，这些语法糖要么是用来使语法更简洁，要么为了使特定代码与 Clojure 的整体风格保持一致。

- 通过在一个形式前面加上单引号 (') 可以使这个形式不被求值。详情请查看 24 页的“阻止求值：quote”一节。
- 你可以使用 #() 语法来定义一个匿名函数。详情请查看 40 页的“函数字面量”一节。
- 虽然 var 本身会被求值成 var 所代表的值，我们也可以通过在 var 的名字前面加上 #' 来得到 var 的本身；44 页的“引用 var：var”一节会详细讨论。
- 你可以在 Clojure 的引用类型前面加上一个 @ 符号来得到这个引用所指向的值。我们会在 170 页的“Clojure 的引用类型”一节详细讨论。
- reader 给宏定义提供了三个特殊的语法：`、~ 以及 `@。我们会在第 5 章详细讨论宏。
- 虽然从严格的技术意义上来说，Clojure 只提供了两种跟 Java 互操作的形式，但是

注 14：有关风格的问题很难给出一个确定的答案，但是我们很少看到在一行里面写一个 map 的多个键值对，还有一些，比如 let，我们总是每行一组绑定，以保持代码清晰。

Clojure 还提供了一些包装这两种形式的语法糖来方便我们编写互操作的代码。44 页的“和 Java 的互操作：和 new”一节会详细讨论。

- Clojure 的所有数据结构和引用类型都支持元数据——有关数据的一些数据。同样的 Clojure 值如果只是元数据不同，那么它们语义上还是相同的。我们可以用元数据来做很多事情，比如 Clojure 语言本身就使用元数据——比如表明一个函数是不是私有的、表明一个函数的类型或一个函数的返回类型——这些东西在其他语言中是通过关键字来做的。Clojure 允许我们使用 ` 语法来给一个值添加元数据，详情请查看 135 页“元数据”一节。

命名空间

到这里为止，我们了解了 Clojure REPL 最核心部分（从而也就理解了 Clojure 本身）的工作原理。

- 读入：Clojure reader 读入代码的文本形式，生成对应的数据结构（比如列表、vector 等）和原子值（比如符号、数字、字符串等）。
- 求值：reader 所生成的很多值求值成它们本身（包括大多数的数据结构和标量值，比如字符串和关键字）。我们已经在前面第 7 页的“表达式、操作符、语法以及优先级”一节学习过列表会被求值成对于函数位置上的操作符的函数调用。

现在只剩下符号的求值过程，我们就彻底搞清楚 Clojure REPL 的工作原理了。到目前为止，我们使用这些符号来命名、引用函数、本地绑定等。除了用来引用本地绑定，符号的求值跟所在的命名空间是密切关联的，◀ 21 命名空间是 Clojure 最基本的代码模块组件。

所有的 Clojure 代码都是在一个命名空间中被定义和求值的。命名空间可以看成 Ruby 和 Python 中的 module，Java 中的 package。^{注 15} 从本质上来说，命名空间是符号跟 vars 或者引入的 Java 类的一个映射关系。

Clojure 中的一种引用类型^{注 16}——var，是一种可修改的内存地址，从而可以保存任何值。在 var 被定义的命名空间里，var 和一个符号相关联，然后我们就可以通过这个符号来使用这个 var，从而得到这个 var 的值。

在 Clojure 中，var 是用特殊形式 def 来定义的，def 只作用于当前命名空间。^{注 17} 下面让我们在当前命名空间 user 中定义一个名叫 x 的 var，var 的名字是我们在当前命名空间

^{注 15} 其实，当在 Clojure 的命名空间中定义类型的时候，Clojure 的命名空间精确地对应到 Java 的包。比如如果在一个名为 app.entities 的命名空间中定义一个名为 Person 的类型，那么它会产生一个全限定名为 app.entities.Person 的类。我们会在第 6 章详细介绍定义类型和记录类型。

^{注 16} 我们会在 170 页的“Clojure 的引用类型”一节详细讨论 Clojure 为并发编程提供的各种引用类型。

^{注 17} 记住，Clojure REPL 启动时候默认的命名空间始终是 user。

找到它的钥匙：

```
(def x 1)
:= #'user/x
```

可以通过这个符号来访问到这个 var 的值：

```
x
:= 1
```

这里在引用符号 x 的时候没有指定命名空间，所以它会被求值成当前命名空间中的符号。我们也可以对这个符号进行重定义，这对于支持 REPL 中的交互式开发是非常重要的：

```
(def x "hello")
:= #'user/x
x
:= "hello"
```



var 不是变量

我们应该只在交互式的语境下定义 var——比如 REPL，或者是在一个 Clojure 源代码文件中用来定义函数，或者其他常量等。特别要提醒的是，全局的 var（在所在的命名空间中都可以访问，由 def 以及它的变种定义）只应该被全局的表达式来定义，决不要在某个 Clojure 程序里的正常分支的函数里定义一个全局的 var，这会导致意想不到的事情发生。

我们会在 206 页的“var 不是变量”一节进一步讨论。

我们在引用符号的时候也可以指定符号所在的命名空间，这样得到的值将是它在指定的命名空间求值得到的值，而不是当前命名空间：

```
*ns*
:= #<Namespace user>
(ns foo)
:= nil
*ns*
:= #<Namespace foo>
user/x
:= "hello"
x
:= #<CompilerException java.lang.RuntimeException:
:= Unable to resolve symbol: x in this context,compiling:(NO_SOURCE_PATH:0)>
```

① 当前的命名空间始终会绑定到 *ns*。

这里我们用 ns 宏创建了一个新的命名空间（同时这个宏也会把当前的命名空间切换到新

的命名空间), 然后我们用 user/x 去引用 user 这个命名空间里面的符号 x 的值。因为我们刚刚创建了 foo 这个新的命名空间, 所以在这个命名空间里面的符号 x 是没有绑定值的, 所以报了一个不能解析 x 的异常。



要想高效地使用 Clojure, 必须要知道如何创建、定义、组织、操作命名空间。有一整套的函数专门做这个事情; 我们会在 322 页的“定义与使用命名空间”一节详细介绍。

前面我们还提到 Clojure 里面的命名空间还维护符号和被引入 (import) 的 Java 类之间的映射。因为 `java.lang` 这个包里面的类默认被引入到每个 Clojure 命名空间中, 所以可以不加包名直接访问这些 Java 类; 而对于那些没有引入的类, 则必须指定包名才能访问它们。任何指向 Java 类的符号都会被求值成那个 Java 类:

```
String
:= java.lang.String
Integer
:= java.lang.Integer
java.util.List
:= java.util.List
java.net.Socket
:= java.net.Socket
```

同时, 每个命名空间都会默认引入 Clojure 的核心库 `clojure.core` 中的所有 var, 比如 `clojure.core` 命名空间中定义了一个名为 `filter` 的函数, 可以不指定命名空间直接访问它:

```
filter
:= #<core$filter clojure.core$filter@7444f787>
```

这里介绍的是 Clojure 命名空间的最基本的一些概念, 我们会在 322 页的“[定义与使用命名空间](#)”一节进行更加详细的介绍。 ◀ 23

符号求值

在对命名空间有了一个最基本的理解之后, 我们再回过头来看看示例 1-2 里面的 `average` 函数, 现在应该对这个函数的求值过程有了更深的理解了:

```
(defn average
  [numbers]
  (/ (apply + numbers) (count numbers)))
```

我们在第 9 页的“同像性”一节介绍过，这些代码其实就是 Clojure 的数据结构。在这个函数体中有好几个符号，它们要么是指向当前命名空间的一个 var，要么是一个所在作用域的本地绑定：

- /、apply、+ 和 count 都被求值成 clojure.core 命名空间中的函数。
- numbers 这个符号稍微复杂一点，当它在参数数组中的时候 ([numbers])，它定义这个函数的唯一参数。^{注 18} 当在这个函数体中引用它的时候 ((apply + numbers) 和 (count numbers))，它求值成这个符号的值。

在知道了这个之后，并且回忆一下在列表里面函数位置的符号代表的是函数调用的操作符，我们对这个函数如何求值就有了全面的认识：

```
(average [60 80 100 400])
;= 160
```

这里符号 average 求值成它的值 #'average，也就是当前命名空间中定义的那个函数。我们传给这个函数一个包含数字的 vector——在 average 函数内部，我们通过 numbers 这个名字来引用这个 vector，调用这个函数的返回值是 160——会被返回给调用者，在这里调用者是 REPL，REPL 会把这个返回值打印到标准输出 stdout。

特殊形式

让我们先把跟 Java 的互操作性放在一边，Clojure 里面的函数位置的符号只可能求值成两个结果：

- 24 > 1. 一个 var 或者本地绑定的值，这个我们前面已经见过了。
2. Clojure 里面的特殊形式。^{注 19}

特殊形式是 Clojure 里面组成计算的基本构建单元，Clojure 里面的其余部分都是基于这些特殊形式构建起来的。这跟早期的其他 Lisp 方言是一样的：先定义一些最基本的操作原语，然后语言的其他部分都是由这些原语搭建而成的。^{注 20} 而且特殊形式有它们自己的语法（比如，它们本身不接受参数）以及求值语义。

^{注 18}： 我们会在 36 页“定义函数：fn”一节深入讨论如何定义函数，以及有关函数参数的所有细节。

^{注 19}： 在求值到函数位置的时候，Clojure 总是首先检查它是不是特殊形式。比如你可以定义一个名字叫做 def 的 var，但是如果把它用在函数位置上的話，它将始终被求值成特殊形式，而不是你定义的 var，当然，如果不是用在函数位置上还是可以正常求值的。

^{注 20}： Paul Graham 的 *The Roots of Lisp* (<http://www.paulgraham.com/rootsoflisp.html>) 是一个简洁但是又足够长的有关计算的基本操作的一本著作，最开始是 John McCarthy 提出来的。虽然这个有关计算的分类是在 50 年前提出的，但是现在你还可以在 Clojure 语言中看到它们的影子。

就像你已经看到的那样，在其他语言中经常被描述成基本原语的操作或者表达式，比如加、减，在 Clojure 中不是基本原语。在 Clojure 中除了那些特殊形式，其他都是利用特殊形式来实现出来的。^{注 21} 这带来的好处就是，Clojure 本身不限定语言本身的“语法”，如果你喜欢的话，可以设计你自己喜欢的语法。^{注 22}

因为整个 Clojure 是构建在这些特殊形式上面的，所以你需要理解这些特殊形式每个是什么的，很多特殊形式你都会经常使用。下面我们会逐一介绍。

阻止求值：quote

quote 阻止 Clojure 表达式的求值。跟这个关系最密切的是指向 var 的符号——如果不使用 quote 的话，符号求值的结果就是对应 var 的值。而使用了 quote 之后，Clojure reader 不再求值这个符号所指向的 var，而是求值成符号本身（就跟字符串、数字等一样）：

```
(quote x)
;= x
(symbol? (quote x))
;= true
```

quote 有对应的 reader 语法：单引号 (')，reader 在求值到单引号的时候会把它解析成一个 quote：

```
'x
;= x
```

25

任何 Clojure 形式都可以被 quote，包括数据结构。如果使用数据结构，那么返回的是数据结构本身（数据结构内的表达式将不作求值操作）：

```
'(+ x x)
;= (+ x x)
(list? '(+ x x))
;= true
```

比如列表在 Clojure 里面一般是求值成函数调用的，而用了 quote 之后，则求值成列表本身——一个真正数据结构意义上的列表：它包含三个元素 '+'、'x 和 'x。这跟我们利用 list 函数手动组装一个列表的效果是一样的：

```
(list '+ 'x 'x)
;= (+ x x)
```

注 21：如果你打开 Clojure 的核心库 core.clj 文件的话，会看到 when、or、defn 以及 = 都是用 Clojure 语言实现的。确实，如果你想的话，你可以利用特殊形式从头实现你自己的 Clojure（或者另外一种语言）。

注 22：这种语法扩展一般需要借助宏的帮助，我们会在第 5 章详细介绍。



quote 的一个有趣的应用就是探查 reader 对于任意一个形式的求值结果，我们首先来看看 quote 本身：

```
'`x  
:= (quote x)
```

再来看看 @ 的求值结果：

```
'@x  
:= (clojure.core/deref x)  
'#(+ % %)  
:= (fn* [p1_3162792#] (+ p1_3162792# p1_3162792#))  
'` (a b `c)  
:= (seq (concat (list (quote user/a))  
:= (list (quote user/b))  
:= (list c)))
```

❶

- ❶ 虽然 deref 是 clojure.core 命名空间里面的，但是这里为了易读性还是把命名空间前缀加上了。

代码块：do

do 会依次求值你传进来的所有表达式，并且把最后一个表达式的结果作为返回值。比如：

```
(do  
  (println "hi")  
  (apply * [4 5 6]))  
; hi  
;= 120
```

这些表达式的值除了最后一个都被丢弃了，但是它们的副作用还是发生了（比如我们这里打印字符串到标准输出，或者操作当前作用域里面的有状态变量）。

- 26 要提醒的一点是，很多其他形式，包括 fn、let、loop、try、defn 以及这些形式的变种都隐式地使用了 do 形式，所以你可以在这些形式中使用多个表达式。比如 let 表达式，在我们这个例子中，定义了两个本地绑定，并且 let 体中有多个表达式，它们都隐式地被包在一个 do 里面：

```
(let [a (inc (rand-int 6))  
     b (inc (rand-int 6))]  
  (println (format "You rolled a %s and a %s" a b))  
  (+ a b))
```

这使得我们可以在 `let` 体中使用任意多个表达式，而最后一个表达式的值决定 `let` 的结果。而如果 `let` 实现的时候没有用 `do` 的话，则需要自己加这个 `do`：^{注23}

```
(let [a (inc (rand-int 6))
      b (inc (rand-int 6))]
  (do
    (println (format "You rolled a %s and a %s" a b))
    (+ a b)))
```

定义 Var : def

我们在前面已经使用过 `def` 了，^{注24} 它的作用是在当前命名空间里定义（或重定义）一个 `var`（你可以同时给它赋一个值，或者也可以不赋值）：

```
(def p "foo")
:= #'user/p
p
:= "foo"
```

Clojure 中很多其他形式都间接地创建或者重定义，而内部都是使用 `def` 来实现的。这些形式一般都以 `def` 开头，比如 `defn`、`defn-`、`defprotocol`、`defonce`、`defmacro` 等。



虽然我们说创建或者重定义 `var` 的函数的名字都以 `def` 开头，但是并不是所有以 `def` 开头的函数都创建或者重定义 `var`，比如 `deftype`、`defrecord` 和 `defmethod`。

◀ 27

本地绑定 : let

`let` 让我们可以在 `let` 形式内部定义本地绑定。换句话说，`let` 定义本地绑定。比如看看下面这个简单的 Java 方法：

```
public static double hypot (double x, double y) {
    final double x2 = x * x;
    final double y2 = y * y;
    return Math.sqrt(x2 + y2);
}
```

跟下面的 Clojure 函数是等价的：

```
(defn hypot
  [x y]
  (let [x2 (* x x)
```

注 23： 另外一个办法是让 `let`（以及所有隐式使用了 `do` 的函数、宏）自己实现（或者重新实现）“做几件事并且把最后一个表达式作为返回值”的语义；但是这样做是完全没有必要的。

注 24： 20 页“命名空间”一节介绍了有关 `var` 的典型用法：作为在命名空间内对于值的恒定引用；198 页的“`var`”一节有关于 `var` 的更详细的介绍，包括动态作用域以及线程本地引用。

```
y2 (* y y)]  
(Math/sqrt (+ x2 y2))))
```

上面的 `x2` 和 `y2` 在 Java 方法和 Clojure 函数里面的作用是一样的：给一个中间变量 / 值命名。



有很多词汇可以用来描述由 `let` 创建的命名引用：

- 本地值
- 本地绑定
- 或者 `let` 绑定

注意，我们这里提到的绑定跟 `binding` 宏是完全不一样的，后者是用来控制动态作用域的线程本地值的，我们在 201 页的“动态作用域”一节会详细介绍。

在任何需要使用本地绑定的地方都间接地使用了 `let`。举个例子，`fn`（以及所有基于它进行函数定义的形式，比如 `defn`）使用 `let` 来绑定函数参数作为函数体内的本地绑定。比如上面例子里面的 `hypot` 函数，`x` 和 `y` 就是 `defn` 利用 `let` 绑定的。因此不管这个定义绑定 `vector` 被用来定义函数参数，还是被用来定义本地绑定，它们都遵循同样的语义。

28



有时候，我们会在 `let` 的绑定数组里面对一个表达式求值，但是我们对于表达式的值并不关心，也不会去使用这个值。在这种情况下，通常用下划线（`_`）来指定这个名字，这样 `reader` 就知道对于这样的表达式，我们是不关心它的返回值的。

使用这种技巧的大多数情况是，这些表达式有副作用，我们所关心的是它的副作用，而不是返回值，一个很典型的例子就是在 `let` 形式中调用 `println` 来打印一些信息。比如：

```
(let [location (get-lat-long)  
      _ (println "Current location:" location)  
      location (find-city-name location)]  
  ...display city name for current location in UI...)
```

这里先调用 API 去获取当前的经度和纬度，而在把它转化成我们肉眼能看懂的具体地理地址之前，我们希望把这个经度和纬度打印出来。我们经常会在 `let` 的绑定数组里面把同一个名字绑定好几次以完成对它的计算，而我们如果想把这些中间值打印出来的话，把这些打印语句绑定到下划线 `_`，表示只想打印，而不关心打印语句的值。

`let` 定义的绑定跟其他语言中的本地变量有两个显著区别。

- 所有本地绑定都是不可变的。你可以在一个嵌套的 let 形式里面把一个本地绑定绑定到另外一个值，或者可以在一个绑定数组的后面把它绑定到别的值，但是没有办法改变一个 let 绑定的值。这使得我们可以避免大量的编程错误，同时也不会太过限制语言的可用性：
 - loop 和 recur 这两个特殊形式满足了在每个循环都对本地值进行改变的需求，我们在 43 页的“循环：loop 和 recur”一节进行了详细介绍。
 - 如果你真的需要“可修改”的本地绑定，Clojure 提供的引用类型具有这种可变语义，170 页的“Clojure 的引用类型”一节有详细介绍。
- let 的绑定数组在编译期可以对通用集合类型进行解构，利用“解构”，可以大大简化从绑定数组中抽取想要的数据的操作。

解构 (let, 第 2 部分)

使用 Clojure 编程很多时候需要跟各种类型的数据结构实现打交道，而顺序性数据结构和 map 是其中最关键、最常用的两种。很多 Clojure 函数都接受这两种类型作为参数或者返回它们作为返回值，而且接受或者返回的是抽象类型，而不返回这些类型的某个具体的实现——绝大多数 Clojure 应用都是依赖这些抽象类型而不是具体的数据结构实现、Java 实现等。这使得我们的函数在调用 Clojure 类库的时候不需要额外的代码去对接具体的数据结构实现，也就不需要一些黏胶代码（glue code）来做类型转换之类的事情，可保持代码简单。

使用抽象集合的一个挑战是如何简洁地去访问一个集合中的多个数值，比如下面是一个 Clojure 的 vector：

```
(def v [42 "foo" 99.2 [5 12]])
;= #'user/v
```

下面是访问这个 vector 中元素的几种方法：

```
(first v)          ❶
;= 42
(second v)
;= "foo"
(last v)
;= [5 12]
(nth v 2)         ❷
;= 99.2
(v 2)             ❸
;= 99.2
(.get v 2)         ❹
;= 99.2
```

- ① Clojure 提供了方便的函数来访问顺序集合的第一个、第二个以及最后一个值。
- ② 同时还提供了一个 `nth` 函数来返回顺序集合的指定下标的元素。
- ③ Clojure 里面的 `vector` 本身也是函数，它接受数组下标作为参数，返回该下标中保存的元素。
- ④ 所有的 Clojure 的顺序集合都实现了 `java.util.List` 这个接口，所以你还可以使用 `List` 接口的 `.get` 方法来返回顺序集合中的元素。

所有的这些函数使得对于一维集合里面元素的访问都很方便，那如果集合里面的元素本身是一个集合，要访问这个内嵌集合里面的元素，这时候就不是很方便了：

```
(+ (first v) (v 2))  
;= 141.2  
  
(+ (first v) (first (last v)))  
;= 47
```

Clojure 的解构特性提供了一种简洁的语法来声明式地从一个集合里面选取某些元素，并且把这些元素绑定到一个本地 `let` 绑定上去。并且因为解构这个特性是由 `let` 提供的，它可以在任何间接使用了 `let` 的地方使用，比如 `fn`、`defn`、`loop`。

`let` 支持两种类型的解构：对于顺序集合的解构以及对于 `map` 的解构。

顺序解构

顺序解构可以对任何顺序集合进行解构，包括：

- Clojure 原生的 `list`、`vector` 以及 `seq`。
- 任何实现了 `java.util.List` 接口的集合（比如 `ArrayList` 和 `LinkedList`）。
- Java 数组。
- 字符串，对它解构的结果是一个个字符。

下面是一个最基本的例子，解构的是我们上面讨论的那个 `v`：

示例1-3：最基本的顺序解构

```
(def v [42 "foo" 99.2 [5 12]])  
;= #'user/v  
(let [[x y z] v]  
  (+ x z))  
;= 141.2
```

在 `let` 最简单的用法中，`let` 的绑定数组中包含的是一个个键值对，但是这里我们指定了三组名字：`[x y z]`——而不是一个名字。这么写的意思是让它对这个顺序集合 `v` 进行

解构，而第一个元素绑定到 x 这个名字，第二个元素绑定到 y，第三个元素绑定到 z。然后我们就可以像使用其他本地绑定一样使用它们。上面的代码跟下面这些是等价的：

```
(let [x (nth v 0)
      y (nth v 1)
      z (nth v 2)]
  (+ x z))
;= 141.2
```



Python 中有和 Clojure 的顺序解构类似的 *unpacking* 机制。比如上面的例子用 Python 写是这样的：

```
>>> v = [42, "foo", 99.2, [5, 12]]
>>> x, y, z, a = v
>>> x + z
141.19999999999999
```

Ruby 中也差不多：

```
>> x, y, z, a = [42, "foo", 99.2, [5, 12]]
[42, "foo", 99.2, [5, 12]]
>> x + z
141.2
```

31

Clojure、Python 和 Ruby 的解构从表面上看起来是差不多的，但是深入看一下的话就会发现，Ruby 和 Python 所提供的解构跟 Clojure 的解构完全不在一个层次上面。

Clojure 中的解构被设计成能够清晰反映被解构的集合的结构。^{注25} 所以如果把我们的解构形式和被解构的集合排在一起的话，你可以很清楚地看出来每个要解构的符号所对应的值：^{注26}

```
[x y z]
[42 "foo" 99.2 [5 12]]
```

解构的形式还支持嵌套解构形式，所以我们可以很轻松地解构嵌套集合：^{注27}

```
(let [[x _ _ [y z]] v]
  (+ x y z))
;= 59
```

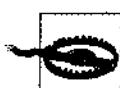
如果再把我们的解构形式和要被解构的集合摆在一起，你会看到，各个符号的值依然很简单就能看出来：

注 25： 因此有了这个名字：解构是创建的反过程。

注 26： 源集合中没有对应绑定名字的就不会在 let 中被使用到；你不需要指定所有的名字以和源集合的结构一一对应，但是顺序解构的话确实需要我们从集合的第一个元素开始解构。

注 27： 这里再注意一下下划线（_）的使用，用它来表示我们忽略某个绑定，跟这章前面讨论过的用法一样。

```
[x _           [y z ]]  
[42 "foo" 99.2 [5 12]]
```



如果内嵌的 vector 中还有一个内嵌的 vector，仍然可以用同样的方式继续内嵌解构形式。解构机制对于内嵌的层数是没有限制的，但是为了保持代码的可读性，最好不要内嵌太多层次。如果内嵌 4 层以上，那么别人来看你的代码可能根本看不懂，你写过之后可能自己都看不懂。

顺序解构还有另外两个特性，大家应该有所了解：

保持“剩下的”元素

可以使用 & 符号来保持解构剩下的那些元素，这个跟 Java 里面不限制参数个数的方法的机制很类似，同时也是 Clojure 函数里面的剩余参数的基础：

```
32 > (let [[x & rest] v]  
       rest)  
     := ("foo" 99.2 [5 12])
```

这对于操作一个序列来说太方便了，不管你是做递归调用，或者是用 loop 形式来做循环。注意，这里的 rest 是一个序列，而不是一个 vector，虽然被解构的参数 v 是一个 vector。

保持被解构的值

你可以在解构形式中指定 :as 选项来把被解构的原始集合绑定到一个本地绑定：

```
(let [[x _ z :as original-vector] v]  
  (conj original-vector (+ x z)))  
 := [42 "foo" 99.2 [5 12] 141.2]
```

这里，original-vector 被绑定到未做修改的集合 v。如果要解构的集合是一个函数调用的返回值，你想对这个返回值集合进行解构，同时又想保持对于这个集合的绑定，那么这个特性就很方便了。如果没有这个特性的话，你可能就要写类似下面的代码：

```
(let [some-collection (some-function ...)  
      [x y z [a b]] some-collection]  
  ...do something with some-collection and its values...)
```

map 解构

map 解构跟顺序解构在概念上是一样的：通过解构形式从 map 中抽出一些元素。map 解构对于下面几种数据结构有效。

- Clojure 原生的 hash-map、array-map，以及记录类型。^{注 28}

注 28：272 页的“记录”一节会详细介绍。

- 任何实现了 `java.util.Map` 的对象
- `get` 方法所支持的任何对象。比如：
 - Clojure 原生 vector
 - 字符串
 - 数组

先来看一个最简单的 map 解构：

```
(def m {:a 5 :b 6
        :c [7 8 9]
        :d {:e 10 :f 11}
        "foo" 88
        42 false})
;= #'user/m
(let [{a :a b :b} m]
  (+ a b))
;= 11
```

33

这里把 map `m` 里面的 `:a` 所对应的值绑定到了 `a`, `:b` 所对应的值绑定到了 `b`。跟前面在介绍顺序解构所做的一样，同样把解构形式和 map 摆在一起，我们可以很清楚地看出解构形式和要解构的集合之间的对应关系：

```
{a :a b :b}
{:a 5 :b 6}
```

要注意的是，可以在 map 解构中用做 key 的不止是关键字，可以是任何类型的值，比如字符串：

```
(let [{f "foo"} m]
  (+ f 12))
;= 100
(let [{v 42} m]
  (if v 1 0))
;= 0
```

而如果要进行 map 解构的是 vector、字符串或者数组的话，那么解构 key 则是数字类型的数组下标。^{注29} 这个在你操作一个用 vector 表示的矩阵时会很方便，因为你可能只需要取矩阵中间的某个值，这时候如果用顺序解构会很麻烦：

```
(let [{x 3 y 8} [12 0 0 -18 44 6 0 0 1]]
  (+ x y))
;= -17
```

^{注29}：这是由 `get` 的多态行为导致的，它利用指定的 key 到指定集合中去查找对应的值；对于这种有索引的顺序集合，`get` 以索引为 key 去集合中查找数据。我们会在 99 页的“Associative”一节详细介绍 `get`。

跟顺序解构一样，map解构也可以处理内嵌map：

```
(let [{e :e} :d] m]
  (* 2 e))
:= 20
```

这个外部map解构——`{e :e} :d`——直接作用在要解构的集合m的顶级元素上，获取:d所对应的元素：`{:e 10 :f 11}`。而内部解构——`{e :e}`，则是作用在`{:e 10 :f 11}`上的，所以我们得到的值是10。

还可以把顺序解构和map解构结合起来：

```
(let {[x _ y] :c} m)
  (+ x y))
:= 16
(def map-in-vector ["James" {:birthday (java.util.Date. 73 1 6)}])
:= #'user/map-in-vector
(let [[name {bd :birthday}] map-in-vector]
  (str name " was born on " bd))
:= "James was born on Thu Feb 06 00:00:00 EST 1973"
```

34

map解构也有一些额外的特性。

保持被解构的集合。跟顺序解构一样，可以在解构形式中使用`:as`来获得对于被解构的集合的引用，然后你可以像使用任何其他`let`绑定一样使用这个被解构的集合：

```
(let [{r1 :x r2 :y :as randoms}
      (zipmap [:x :y :z] (repeatedly (partial rand-int 10)))]
  (assoc randoms :sum (+ r1 r2)))
:= {:sum 17, :z 3, :y 8, :x 9}
```

默认值。你可以使用`:or`来提供一个默认的map，如果要解构的key在集合中没有的话，那么默认map中的值会作为默认值绑定到我们的解构符号上去：

```
(let [{k :unknown x :a
       :or {k 50}} m]
  (+ k x))
:= 55
```

这使得我们不必在进行map解构之前把源map和默认map合并，或者手动给没有解构到的符号设置默认值，这对于追求简洁的Clojure程序员来说是不能容忍的：

```
(let [{k :unknown x :a} m
      k (or k 50)]
  (+ k x))
:= 55
```

更进一步说，`:or` 能区分到底是没有赋值，还是赋给的值就是逻辑 `false` (`nil` 或者 `false`)，看下面的例子：

```
(let [{opt1 :option} {:option false}]
  opt1 (or opt1 true)
  {opt2 :option :or {opt2 true}} {:option false}]
  {:opt1 opt1 :opt2 opt2})
:= {:opt1 true, :opt2 false}
```

绑定符号到 `map` 中同名关键字所对应的元素。通常，对于一个特定意义的值会给它一个固定的名字，而通常在 `let` 中绑定的时候也会倾向于用 `map` 中与 `key` 的名字一样的符号来绑定 `key` 所对应的值，但是这样写的话，代码会变得很冗长：

```
(def chas {:name "Chas" :age 31 :location "Massachusetts"})
:= #'user/chas
(let [{:name name :age age :location location} chas]
  (format "%s is %s years old and lives in %s." name age location))
:= "Chas is 31 years old and lives in Massachusetts."
```

要把每个 `key` 的名字写两遍也跟我们追求简洁的精神相悖。在这种情况下，Clojure 提供了 `:keys`、`:strs` 和 `:syms` 来指定 `map` 中 `key` 的类型：`:keys` 表示 `key` 的类型是关键字，`:strs` 表示 `key` 的类型是字符串；`:syms` 表示 `key` 的类型是符号。35 它们的用法如下：

```
(let [{:keys [name age location]} chas]
  (format "%s is %s years old and lives in %s." name age location))
:= "Chas is 31 years old and lives in Massachusetts."
```

这样对于 `key` 的名字就只需要写一遍了，再看一个 `:strs` 和 `:syms` 的例子：

```
(def brian {"name" "Brian" "age" 31 "location" "British Columbia"})
:= #'user/brian
(let [{:strs [name age location]} brian]
  (format "%s is %s years old and lives in %s." name age location))
:= "Brian is 31 years old and lives in British Columbia."

(def christophe {"name" "Christophe" 'age 33 'location
"Rhône-Alpes"})
:= #'user/christophe
(let [{:syms [name age location]} christophe]
  (format "%s is %s years old and lives in %s." name age location))
:= "Christophe is 31 years old and lives in Rhône-Alpes."
```

将来你会发现你使用 `:keys` 会远远多于 `:strs` 和 `:syms`，因为在 Clojure 中我们习惯使用关键字来作为 `map` 中的 `key`。

对顺序集合的“剩余”部分使用 map 解构。前面介绍过可以用 & 来把顺序集合的剩余部分收集进一个集合，并且组合使用顺序解构和 map 解构来深入解构任何数据结构。下面例子里面的集合，前几个是普通元素，后面则是关键字 / 值对：

```
(def user-info ["robert8990" 2011 :name "Bob" :city "Boston"])
:= #'user/info
```

这种数据其实是很常见的，但是处理这种数据的办法却从来都不优雅。在 Clojure 中，如果我们“手动”解构这个数据，大概会是这样的：

```
(let [[username account-year & extra-info] user-info          ①
      {:keys [name city]} (apply hash-map extra-info)]           ②
  (format "%s is in %s" name city))
:= "Bob is in Boston"
```

① 我们会先用顺序解构解构前面几个普通数据，并且把数组的剩余部分收集起来。

② 然后把剩下的部分生成一个 map，最后用 map 解构来解构这个 map。

但是，我们 Clojure 程序员追求简洁、优雅的精神让我们还是不能容忍这样的代码。

36 Clojure 提供的更好的办法是直接用 map 解构来解构集合的剩余部分——如果剩余部分的元素个数是偶数的话，顺序解构会把剩余部分当做一个 map 来处理，神奇吧？

```
(let [[username account-year & {:keys [name city]}] user-info]
  (format "%s is in %s" name city))
:= "Bob is in Boston"
```

这跟前面手动方式的结果是一样的，但是简洁多了，这个也是后面第 39 页要介绍的 Clojure 的“关键字参数”的基础。

定义函数：fn

函数在 Clojure 中是“头等公民”。我们用特殊形式——fn 来创建函数，fn 本身也有与 let 和 do 类似的语义。

下面是一个把传入参数加 10 再返回的一个简单函数：

```
(fn [x]           ①
  (+ 10 x))       ②
```

① fn 接受 let 样式的那种绑定数组，绑定数组定义函数的参数的名字以及参数的个数；我们在第 28 页的“解构（let，第 2 部分）”讨论的解构在这里也同样可以适用。

② 绑定数组后面的所有的形式组成了函数体。整个函数被隐式地包括在一个 do 形式中，所以函数体中可以包含任意多个形式；同样因为是隐式地使用了 do，所以函数体中的

最后一个形式的值就是整个函数的返回值。

函数定义时的参数与调用函数时实际传递的参数之间的对应是通过参数位置来完成的：

```
((fn [x] (+ 10 x)) 8)
;= 18
```

8是这个函数的唯一参数，并且绑定到函数参数x，所以这个函数调用跟下面的let形式的效果是一样的：

```
(let [x 8]
  (+ 10 x))
```

你也可以定义接受多个参数的函数：

```
((fn [x y z] (+ x y z))
  3 4 12)
;= 19
```

在这种情况下，函数调用跟下面的let形式是等价的：

```
(let [x 3
      y 4
      z 12]
  (+ x y z))
```

同时函数还可以有多个参数列表。这里我们用一个var来保持函数的引用，使得我们可以通过var来多次调用这个函数：

```
(def strange-adder (fn adder-self-reference
                        ([x] (adder-self-reference x 1))
                        ([x y] (+ x y))))
;= #'user/strange-adder
(strange-adder 10)
;= 11
(strange-adder 10 50)
;= 60
```

在定义多参数列表的函数时，每套参数 - 函数体都要放在一个单独的括号内。函数调用在决定到底执行哪个函数体的时候是根据传入的参数个数来决定的。

在上面的例子中，你可能已经注意到了，fn的第一个参数是一个可选的函数名字adder-self-reference。这个可选的函数名字使我们可以在函数体中引用函数自己，这样一个参数的实现就可以调用两个参数的实现来实现自己的逻辑。



用 letfn 来解决函数定义互相引用的问题

具名函数（比如上面的 `adder-self-reference`）使我们可以很简单地创建自递归的函数。一个更变态的情况是定义两个相互引用的函数。

对于这种比较极端的例子，Clojure 提供了特殊形式：`letfn` 来解决，它允许同时定义多个具名函数，而且这些函数可以互相引用。看看下面重新实现的 `odd?` 和 `even?`：

```
(letfn [(odd? [n]
             (even? (dec n)))
         (even? [n]
             (or (zero? n)
                 (odd? (dec n))))]
        (odd? 11))
      := true)
```

① 可以看出这个 vector 由多个普通的 fn 函数体组成，只是 fn 本身被省略了。

`defn` 是基于 `fn` 的。前面已经见过使用 `defn` 的例子了，而且刚才的那个例子大家看着应该很眼熟吧？`defn` 是一个封装了 `def` 和 `fn` 功能的宏，它使我们可以很简洁地定义一个具名函数，并且把这个函数注册到当前的命名空间里去，比如下面的两个定义是等价的：

```
38 > (def strange-adder (fn strange-adder
           ([x] (strange-adder x 1))
           ([x y] (+ x y))))  
  
(defn strange-adder
  ([x] (strange-adder x 1))
  ([x y] (+ x y))))
```

也可以定义只有一个参数列表的函数，因为只有一个参数列表，那么多余的括号就可以省掉了。下面两个定义是等价的：

```
(def redundant-adder (fn redundant-adder
           [x y z]
           (+ x y z))  
  
(defn redundant-adder
  [x y z]
  (+ x y z)))
```

我们在后面会大量使用 `defn` 而不是 `fn` 来演示函数的功能，因为用 `defn` 来定义具名函数比 `fn` 的可读性更好一些。

解构函数参数

得益于 `defn` 使用 `let` 来做函数参数的绑定，对于函数参数，Clojure 也支持解构，你可以再温习一下前面讨论的解构的所有特性，下面我们会讨论一些在解构函数参数时非常常用的用法。

可变参函数。函数可以把调用它传入的多余的参数收集到一个列表里去，这里使用的机制跟顺序解构使用的机制是一样的。这样的函数叫做可变参函数，被收集起来的参数通常称为剩余参数或者不定参数。下面的这个例子中，函数接受一个固定位置参数，剩下的所有参数都作为“剩余参数”：

```
(defn concat-rest
  [x & rest]
  (apply str (butlast rest)))
 ;;= #'user(concat-rest
 (concat-rest 0 1 2 3 4)
 ;;= "123"
```

而“剩余参数”列表可以像其他序列一样进行解构，在下面的例子中，我们对函数的参数进行解构，使得它使用起来像是一个没有定义参数的函数。

```
(defn make-user
  [& [user-id]]
  {:user-id (or user-id
    (str (java.util.UUID/randomUUID))))}
 ;;= #'user(make-user
 (make-user)
 ;;= {:user-id "ef165515-6d6f-49d6-bd32-25eeb024d0b4"}
 (make-user "Bobby")
 ;;= {:user-id "Bobby"}
```

39

关键字参数。定义一个接受很多参数的函数时通常有一些参数不是必选的，有一些参数可能有默认值；而且有时候我们希望函数的使用者不必按照某个特定的顺序来传参。^{注30}

`fn`（以及 `defn`）对于这种情况通过关键字参数来支持，这个是构建在 `let` 对于剩余参数的 `map` 解构的基础上的。关键字参数是跟在固定位置参数后面的一个个参数名和参数默认值的对子。这些关键字参数会被收集到一个 `map` 中，然后函数通过 `map` 解构来获取具体某个参数的值：

```
(defn make-user
  [username & {:keys [email join-date]
    :or [{join-date (java.util.Date.)}}]}
  {:username username} ●
  ●
```

注 30：Python 是一种普遍支持这种用法的语言，你可以给每个参数命名，然后在调用函数的时候以任意顺序传入，而且参数的默认值可以在定义函数的时候设置。

```

:join-date join-date
:email email
;; 2.592e9 -> one month in ms
:exp-date (java.util.Date. (long (+ 2.592e9 (.getTime join-date)))))

;= #'user/make-user
(make-user "Bobby")
;= {:username "Bobby", :join-date #<Date Mon Jan 09 16:56:16 EST 2012>,
;= :email nil, :exp-date #<Date Wed Feb 08 16:56:16 EST 2012>}
(make-user "Bobby")
;join-date (java.util.Date. 111 0 1)
:email "bobby@example.com"
;= {:username "Bobby", :join-date #<Date Sun Jan 01 00:00:00 EST 2011>,
;= :email "bobby@example.com", :exp-date #<Date Tue Jan 31 00:00:00 EST 2011>}

```

- ❶ 上面定义的 `make-user` 函数接受一个固定位置的参数 `username`，而剩下的其他参数则被当成一个个参数名 - 参数值的对子被收集到一个 `map` 中，然后在 `&` 符号后面对这个 `map` 进行解构。
- ❷ 在这个 `map` 解构中，我们给 `join-date` 定义了一个默认值——当前时间。
- ❸ 当我们传一个参数去调用 `make-user` 的时候，返回的是一个 `user map`，`user map` 中的内容是用户名，默认的是 `join-date`、`expiration-date`，以及值为 `nil` 的 `email`——因为我们没有提供 `email`。
- ❹ 给 `make-user` 提供的多余的参数会被当做 `map` 来解构，而且不考虑这些参数的顺序。

40



因为关键字参数是利用 `let` 的 `map` 解构的特性来实现的，所以关键字的参数名字理论上可以用任何类型的值（比如字符串、数字甚至集合），比如：

```

(defn foo
  [& {k ["m" 9]}]
  (inc k))
;= #'user/foo
(foo ["m" 9] 19)
;= 20

```

`["m" 9]` 在这里被当做一个关键字参数的名字。

虽然说是这么说，但是从没有人这么干过。在 Clojure 中，大家都习惯用关键字来作为 `map` 的 `key` 的名字，参数的名字，这也是我们把这种参数称为关键字参数的原因。

前置条件和后置条件。`fn` 提供对函数参数和函数返回值进行检查的前置和后置条件。这个特性在单元测试以及确保参数正确性方面非常有用，我们会在 487 页的“前条件和后条件”一节详细介绍。

函数字面量

我们在 20 页“reader 的一些其他语法糖”一节简要介绍过函数字面量。函数字面量类似 Ruby 中的 blocks 以及 Python 中的 lambda 表达式：当你要定义一个匿名函数，特别是非常简单的函数的时候，函数字面量提供了非常简洁的语法来做这个事情。

比如，下面这些匿名函数表达式是一样的：

```
(fn [x y] (Math/pow x y))
```

```
#(Math/pow %1 %2)
```

后一个形式其实是前一个形式的 reader 语法糖，它在运行时会被扩展成前一种形式；我们可以简单证实一下：^{注 31}

```
(read-string "#(Math/pow %1 %2)")  
:= (fn* [p1_285# p2_286#] (Math/pow p1_285# p2_286#))
```

fn 跟函数字面量的区别在于：

函数字面量没有隐式地使用 do。普通的 fn（以及由它引申出来的所有变种）把它的函数体放在一个隐式的 do 里面，我们在 36 页的“定义函数：fn”一节已经介绍过了。这使得你可以定义下面这样的函数：

```
(fn [x y]  
  (println (str x \^ y))  
  (Math/pow x y))
```

41

而如果使用函数字面量的话，那么你要显式地使用一个 do 形式：

```
#(do (println (str %1 \^ %2))  
      (Math/pow %1 %2))
```

使用非命名的占位符号来指定函数的参数个数。上面 fn 的例子中，我们使用 x 和 y 来指定函数接受参数的个数，同时也利用它们来指定参数的名字。而对于函数字面量，我们则使用非命名的占位符 % 来指定函数参数个数以及引用具体参数，%1 表示第一个参数，%2 表示第二个参数等。而且最大的占位符指定函数的参数个数，因此如果我们要定义一个接受 4 个参数的函数，那么我们要在函数体内引用到 %4 才行。

对于函数字面量的参数，还有两个小技巧告诉大家：

1. 因为很多函数字面量都只接受一个参数，所以可以简单地使用 % 来引用它的第一个参数，所以 #(Math/pow % %2) 和 (Math/pow %1 %2) 是等价的，而 Clojure 中则鼓励

注 31：因为在里参数的名字到底叫什么没什么关系，函数字面量会为它们生成唯一的符号来引用它们，在这个例子里面就是 p1_285# 和 p2_286#。

使用比较短的表达方式。

2. 你可以定义个不定参数的函数^{注32}，并且通过 %& 来引用那些剩余参数。因此下面这些函数是等价的：

```
(fn [x & rest]
  (- x (apply + rest)))

#(.- % (apply + %&))
```

函数字面量不能嵌套使用。虽然下面这个是完全没有问题的：

```
(fn [x]
  (fn [y]
    (+ x y)))
```

但是这个就不行了：

```
#(#(+ % %))
:= #<IllegalStateException java.lang.IllegalStateException:
:= Nested #()'s are not allowed>
```

暂且不说我们设计函数字面量的本意是用来表达那些简短、简单的表达式，把函数字面量嵌套将会使得函数字面量很难读懂；同时也没有一个简短的办法使我们能区分出来 % 参数到底是指向内层的字面量还是外层的字面量的参数。

42 条件判断：if

if 是 Clojure 唯一的基本条件判断函数。语法非常简单，如果 if 的第一个表达式的值是逻辑 true 的话，那么整个 if 的值就是第二个表达式的值，否则 if 形式的值则是第三个表达式的值。第二个表达式和第三个表达式是按需求值的。

Clojure 的条件判断把任何非 nil 或非 false 的值都判断为 true，比如：

```
(if "hi" \t)
:= \t
(if 42 \t)
:= \t
(if nil "unevaluated" \f)
:= \f
(if false "unevaluated" \f)
:= \f
(if (not true) \t)
:= nil
```

需要注意的是，如果一个条件表达式的值是 false，但是 if 的 else 表达式没有提供的话，

注 32： 细节请看 38 页的“可变参函数”一节。

那么整个 if 的值就是 nil。^{注33}

很多其他的一些函数、宏是构建在 if 基础之上的，包括：

- 如果想在条件判断为 false 的时候返回 nil（或者什么动作都不做），那么最好使用 when。
- cond——跟 Java 和 Ruby 里面的 else if，以及 Python 里面的 elif 类似，它使得你可以简洁地指定多个条件判断语句，如果满足则分别返回各自对应的表达式的值。
- if-let 和 when-let，它们分别是 let 跟 if 和 when 的组合，如果判断条件是 true 的话，那么 if 形式的值则绑定到这个本地绑定。

 Closure 还提供了两个谓词 true? 和 false?，但是它们跟 if 条件判断是没有关系的。比如：

```
(true? "string")
:= false
(if "string" \t \f)
:= \t
```

true? 和 false? 检查所给的参数是不是布尔值 true 和 false，而不是逻辑意义上的 true 和 false，逻辑意义上的 true 其实和下面的表达式是等价的：对于任何 x 值的 (or (not (nil? x)) (true? x))。

循环：loop 和 recur

43

Clojure 提供了好几个有用的循环结构，包括 doseq 和 dotimes，它们都是构建在 recur 的基础之上的。recur 能够在不消耗堆栈空间的情况下把程序执行转到离本地上下文最近的 loop 头那里去，这个 loop 头可以是 loop 定义的，也可以是一个函数定义的。我们来看一个简单的循环例子：

```
(loop [x 5]
  (if (neg? x)
    x
    (recur (dec x))))
```

:= -1

- loop 通过一个隐式的 Let 来定义绑定，所以它接受一个绑定数组，跟 let 类似。
- 如果 loop 形式的最后一个表达式产生一个值，那么这个值将作为 loop 形式的值。在这个例子里面，当 x 的值为负数时，loop 形式会返回 x 的值。
- recur 形式会把程序执行的控制权转到上下文里面最近的 loop 头去，在这里就是我们

注 33：对于这样的场景 when 要更适合。

的 `loop` 形式，同时把 `x` 绑定的值变成 `(dec x)`。

函数也可以建立 `loop` 头，如果是函数建立 `loop` 头的话，那么 `recur` 所带的值则会绑定到函数的参数上面去：

```
(defn countdown
  [x]
  (if (zero? x)
    :blastoff!
    (do (println x)
      (recur (dec x))))))
:= #'user/countdown
(countdown 5)
; 5
; 4
; 3
; 2
; 1
:= :blastoff!
```

适当地使用 `recur`。`recur` 是一个非常底层的循环和递归控制操作，通常我们并不需要使用这么底层的工具：

- 比如很多时候我们可以直接使用 Clojure 核心库中提供的函数，比如 `doseq` 和 `dotimes` 来完成我们的循环。
- 如果要遍历一个集合或者列表，`map`、`reduce`、`for` 这些函数比 `recur` 更方便。

因为 `recur` 并不消耗堆栈空间（从而也就避免了堆栈异常错误），`recur` 对于实现某些递归的算法是非常关键的。并且由于它不对数字自动进行装箱操作，所以操作数字的时候性能很好，对于实现一些跟数学和数据相关的操作非常有用。我们会在 449 页的“用 Clojure 可视化芒德布罗集”一节介绍一些这样使用的例子。

最后，也有些情况下我们要累计或者消费一个集合或一堆集合，这个时候如果用纯函数的 `map`、`reduce` 等会非常难或者效率不高。这种情况下，可以用 `recur`（可能还要用 `loop` 来设置 `loop` 头）来解决难题。

引用 var : var

命名一个 `var` 的符号求值成 `var` 所对应的值：

```
(def x 5)
:= #'user/x
x
:= 5
```

但是，有时候你想获得指向 var 本身的引用，而不是 var 的值，var 这个特殊形式就是用来做这个的：

```
(var x)
:= #'user/x
```

我们已经看过很多次一个 var 在 REPL 上面是怎么显示的了：#’后面跟一个符号。Clojure 中也有一个 reader 语法求值成 var 这个特殊形式：

```
#'x
:= #'user/x
```

我们会在 198 页的“var”一节更详细地进行介绍。

和 Java 的互操作：. 和 new

所有跟 Java 的互操作——从初始化、静态和实例方法调用到字段访问都是通过 new 和 . 这两个特殊形式来实现的。Clojure reader 在这些互操作的形式之上还提供了一些语法糖使得跟 Java 的互操作更简洁，并且跟 Clojure 的整体风格保持一致——列表的第一个元素为函数位置。所以，你在 Clojure 代码中将很少看到直接使用 . 和 new，但是随着你看的 Clojure 代码越来越多，你总会碰到它们的。

表1-3: Clojure提供的Java互操作语法糖与原生语法对照表

45

操作	Java 代码	Clojure 语法糖	特殊形式的写法
对象初始化	new java.util.ArrayList(100)	(java.util.ArrayList 100)	(new java.util.ArrayList 100)
调用静态方法	Math.pow(2, 10)	(Math/pow 2 10)	(. Math pow 2 10)
访问静态成员变量	"hello".substring(1, 3)	(. "hello" substring 1 3)	(. "hello" 1 3)
访问实例成员变量	Integer.MAX_VALUE	Integer/MAX_VALUE	(. Integer MAX_VALUE)
成员变量	someObject.someField	(.someField some-object)	(. some-object some-field)

在进行互操作时应该优先使用表 1-3 提供的这些语法糖，而不要直接使用 New 和 .。我们会在第 9 章详细介绍 Java 互操作。

异常处理：try 和 throw

这两个特殊形式使我们可以从 Clojure 代码中使用 Java 的异常处理和抛出机制，我们会在 362 页的“异常与错误处理”一节详细介绍。

状态修改 : set!

虽然 Clojure 强调使用不可变的数据结构和值，但是还是有一些场景中我们需要对一个状态进行改变，最常见的情况是调用一个 Java 对象的 `set` 方法或者任意一个改变对象状态的方法。而对于剩下的那些场景，Clojure 提供了函数：`:set!`，它可以：

- 设置那些没有根绑定 (root binding) 的线程本地值。我们会在 201 页的“动态作用域”一节详细讨论。
- 设置一个 Java 的字段，我们会在 359 页的“访问对象属性”一节详细介绍。
- 设置由 `deftype` 定义的对象的可修改字段，我们会在 277 页的“类型”一节介绍它的用法。

锁的原语 : monitor-enter 和 monitor-exit

这两个是 Clojure 中提供的锁原语，用来同步每个 Java 对象上都有的 `monitor`。你不应该直接使用这两个原语，因为 Clojure 提供了一个宏：`:locking`，它能够保证进行更合适的锁获取和锁释放。我们将在 225 页的“锁”一节中详细讨论。

46

小结

我们在这一章的学习过程中，不停地回过头来复习示例 1-2 这个例子：

```
(defn average
  [numbers]
  (/ (apply + numbers) (count numbers)))
```

我们在“同像性”（参见第 9 页）一节知道了这个函数其实就是一个包含了一些数据的 Clojure 列表。而在“表达式、操作符、语法以及优先级”（参见第 7 页）一节，我们知道在 Clojure 里面列表被求值成方法调用，而函数位置的值则是操作符。在学习了命名空间之后，我们在“符号求值”一节（参见 23 页）知道了符号在 Clojure reader 的求值过程是怎样的。而现在学习了特殊形式，特别是 `def` 和 `fn`，我们理解了对于这个函数求值的全过程（不管它是定义在 REPL 里，还是作为 Clojure 源文件的一部分）。

`defn` 只是它们的一个快捷方式：

```
(def average (fn average
  [numbers]
  (/ (apply + numbers) (count numbers))))
```

在这里，`fn` 创建了 `average` 函数（回忆一下“定义函数：`fn`”一节中（参见第 36 页）提到的 `average` 是一个自引用的值，所以如果有需要的话，在 `average` 内部，它可以递归地调用自己），而 `def` 把这个函数的值绑定到当前命名空间的 `average` 这个符号。

eval

我们在这里讨论的所有求值语义都被包装在 eval 这个函数中，这个函数接受一个形式作为参数。通过调用 eval，我们可以清楚地看到标量以及其他字面量都被求值成什么样的值：

```
(eval :foo)  
:= :foo  
(eval [1 2 3])  
:= [1 2 3]  
(eval "text")  
:= "text"
```

而一个列表则会被 eval 求值成它所表示的函数调用的返回值：

```
(eval '(average [60 80 100 400]))  
:= 160
```



虽然 eval 的语义是整个 Clojure 的基础，它本身在 Clojure 的应用里面是很少被直接使用的。它提供了终极的灵活性——使得你可以求值任何一个正确的 Clojure 表达式，但是通常是不需要的。通常来说，如果在代码里面使用 eval 了，那么你可能是在用牛刀来杀鸡。

◀ 47

大多数使用 eval 的地方，一般都可以用宏来解决，我们会在第 5 章详细讨论。

学到这里，我们其实已经可以自己重新实现 Clojure REPL 了。首先我们知道 read（或者 read-string）可以用来把 Clojure 数据结构的文本表示转换成数据结构：

```
(eval (read-string "(average [60 80 100 400])))  
:= 160
```

然后我们用 recur 来在一个函数里面建立一个循环，再加一个 I/O 相关的函数把表达式求值的结果打印到 REPL 的提示符后面，这样就有了我们自己的 REPL：

示例1-4：Clojure的REPL的简单实现

```
(defn embedded-repl  
  "A naive Clojure REPL implementation. Enter ':quit'  
   to exit."  
  []  
  (print (str (ns-name *ns*) ">>> "))  
  (flush)  
  (let [expr (read)  
        value (eval expr)]  
    (when (not= :quit value)  
      (println value)  
      (recur))))
```

```
(embedded-repl)
; user>>> (defn average2
;           [numbers]
;           (/ (apply + numbers) (count numbers)))
; #'user/average2
; user>>> (average2 [3 7 5])
; 5
; user>>> :quit
;= nil
```

这个简单实现的 REPL 还有很多问题，比如代码抛出的任何错误都会被抛出 `embedded-repl` 定义的循环之外，导致整个 REPL 不再可用，但是这至少是个开始。^{注 34}

48 | 这只是开始

我们已经浏览了 Clojure 的最基本的部分：进行计算的基本操作（特殊形式），代码与数据的可交换性，以及有关交互式开发的冰山一角。在这个基础之上，介绍了利用 JVM 提供的一些特性，Clojure 提供的不可变的数据结构；被明确定义的、易于使用的并发原语、宏等。

我们会帮你在本书的其余部分中理解所有的这些，希望能把 Clojure 带进你每天的生活，并且希望你在读完本书第 5 部分之后能够成为一个 Clojure 程序员。

下面是一些你会经常用到的有用资源：

- Clojure 的核心 API 文档：<http://clojure.github.com/clojure>。
- Clojure 的邮件列表：<http://groups.google.com/group/clojure>，以及 Freenode 上的 IRC 频道 #clojure，^{注 35} 不管你的 Clojure 水平如何，你都能在这些地方得到帮助。
- 我们还为本书建立了一个网站 <http://closurebook.com>，你可以在这个网站上找到一些额外的资源来帮助你更有效率地学习、使用 Clojure。

你准备好学习下一章了吗？

^{注 34} Clojure 自带的 REPL 也是用 Clojure 实现的，命名空间是 `clojure.main`，如果感兴趣的话，你可以去看看以后你每天都要使用的 REPL 是怎么实现的。

^{注 35} 如果你不是经常使用 IRC 客户端的话，可以使用这个在线的版本：<http://webchat.freenode.net/?channels=#clojure>。

函数式编程以及并发

函数式编程

函数式编程在软件开发领域中是一个见仁见智的概念。虽然在该领域中有很多不同的见解，但是 Clojure 无疑是一种函数式语言，而这也是 Clojure 本身许多优点和特性的源泉。

在这一章中，我们会：

1. 介绍到底什么是函数式编程。
2. 解释为什么你应该关注函数式编程。
3. 讨论 Clojure 的具体实现——使得 Clojure 成为一门优秀语言的原因。

在这个学习的过程中，我们希望使得函数式编程——特别是 Clojure 风格的函数式编程不是那种学术研究风格的函数式编程，可以像这些年的过程式编程、面向对象式编程一样改进你的软件设计和开发。

如果你已经对函数式编程非常熟悉（不管是通过 Ruby、JavaScript 或者更纯正的函数式编程如 Scala、F# 或者 Haskell 等），那么接下来介绍的这些对你来说应该都是一些很熟悉的概念，但是这些概念值得再过一遍以便更深入地理解 Clojure 风格的函数式编程。

如果你对函数式编程完全是陌生的，或者一开始你对函数式编程是持质疑态度的，那么我们强烈建议你读一读这一章，你不会后悔的！^{注1} 再回忆一下第 1 章提到的：Clojure 要求你提高你的水平，同时作为让你这样做的回报，它也会使你的水平越来越高；就像如果要学习面向对象编程、Java 或者 Ruby，你要提高自己的水平；学习函数式编程、Clojure，你也需要提高你自己的水平。但是作为回报，你所得到的不只是一个“新的思

注 1： 在你知道了 Clojure 语言会给你提供什么特性之后，你可以看看 Wikipedia 上的一些更深入的介绍：
http://en.wikipedia.org/wiki/Functional_programming。

52 维方式”，而且还会得到一些实用的工具和理论来解决我们日常生活中每天都要遇到的编程挑战。^{注2}

所谓函数式编程，到底意味着什么？

函数式编程是一个比较大的概念，在不同的语言里面有不同的解释。在 Clojure 中，函数式编程意味着：

- 我们希望操作不可变的值，这包括：
 - 我们使用简单抽象的不可变数据结构，而不是有可变状态的数据结构。
 - 把函数本身当做值的一种，从而使得我们可以使用高阶函数。
- 我们更喜欢对数据进行声明式的处理，而不是命令式的控制、遍历。
- 我们喜欢通过对函数进行递增式的组合，使用高阶函数以及不可变数据结构，在更高的抽象级别（或者说正确的级别）来解决复杂的问题。

这些都是你可能听说过的 Clojure 中其他更高级特性的基础，比如 Clojure 对于多线程编程，或者更通用的说法是对于对象标识和对象状态所提供的明确定义的语义，我们会在第 4 章中分别介绍。

谈谈值的重要性

程序状态是一个历史悠久的、很宽泛的概念，但是通常来说，它指的是你的应用中用来表示实体的所有标量变量以及聚合类型的数据结构，再加上你的应用与外界的所有联系（比如打开的文件、socket 等）的总和。一种编程语言的特点往往是由这个语言处理状态的方式决定的：它提供了什么？它鼓励怎么做？它禁止怎么做？

不管是不是通过对象的形式表达状态，大多数语言要么通过一些术语，要么通过一些明显的设计鼓励使用可变的状态。而函数式编程则均鼓励使用不可变的对象——我们称之为值——来表示程序的状态。Clojure 在这一点上面没有什么区别。

53 “不过，等等，”你可能会这么说，“把可变状态完全去除没有意义，我的程序始终是需要对这个世界做点什么，所以状态改变是不可避免的。”在这一点上你说得没错：任何有用的程序都需要跟外部世界进行交互，从外部世界获取输入并且输出点东西给外部世界……

注 2：值得注意的是，这是可能的：在 Java 这种从来不鼓励（有时候甚至积极反对）使用函数式编程的语言里面也可以应用函数式编程的一些原则。如果你已经有一些高质量的持久性数据结构和函数式编程的关键概念的实现，比如 Google Guava (<https://code.google.com/p/guava-libraries/>) 或者 Functional Java (<http://functionaljava.org>) 这些类库，你会发现要应用这些函数式编程的原则很简单。

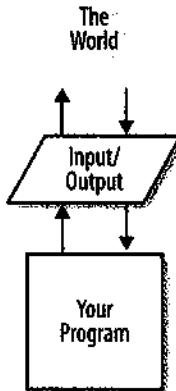


图2-1：所有程序的原理^{注3}

但是，这并没有阻止你使用不可变的值。正相反的是：程序里面值使用得越多，就越容易来推断你的程序的行为——跟使用可变状态相比。我们会在这一章中持续讨论这个问题。

从使用可变状态改为使用不可变值，很多人是不习惯的，但是你可能没有意识到：你在每天的编程过程中已经大量使用过不可变的值了，它们也是你的程序中最可靠、最容易读懂的部分。

关于值

到底什么是值？它们的不可变性和可变的对象比起来到底有什么区别？下面是值的一些例子：

```
true false 5 14.2 \T "hello" nil
```

这些都是 JVM 的标准的布尔值、数字、字符以及字符串，Clojure 方便地重用了它们，它们都是不可变的、都是值，你可以每天使用并且依赖它们。

值的一个关键特性是：值不随时间的改变而改变，比如，如果两个值相等，那么始终相等；两个值不等，那么始终不等。下面的这些关系始终是对的：

```
(= 5 5)
(= 5 (+ 2 3))
(= "boot" (str "bo" "ot"))
(= nil nil)
```

^{注3}：专利未决！

```
(let [a 5]
  (do-something-with-a-number a)
  (= a 5))
```

而这些表达式在 Java、Python 以及 Ruby 中对应的表达式也都始终是对的。^{注4} 而这个也可以帮助我们分析那些跟值有关部分代码的行为。要理解为什么，你可以想象如果这些始终不变的值变得可变了，你的代码行为会发生怎样的变化。

值与可变对象的比较

在应用里面到底使用可变的对象还是不可变的值是一个很重要的决定，这个决定会产生很重大的后果，即使对于最简单的项目也是如此。但是这个决定通常却只是基于习惯、熟悉程度而做出的，而根本没有考虑每种选择的后果。由于由可变对象所保持的状态可能会变，而且变的时候你可能根本就不知道，所以如果我们可以使用不可变的值，却使用可变的对象的话，那么只能说你的选择很危险。

这听起来可能觉得很夸张，特别是如果你现在每天都在使用可变对象，而且好像可变对象用得还挺好的。但是让我们稍微花点时间来考虑一个小例子：我们每个程序员都认为是不变的值，integer 如果变得可变了会怎么样：

示例2-1：用Java实现的可变integer

```
public class StatefulInteger extends Number {
    private int state;

    public StatefulInteger (int initialState) {
        this.state = initialState;
    }

    public void setInt (int newState) {
        this.state = newState;
    }

    public int intValue () {
        return state;
    }

    public int hashCode () {
        return state;
    }

    public boolean equals (Object obj) {
        return obj instanceof StatefulInteger &&
```

注4：除了这个 (= nil nil) 例子；许多语言对于 nil 或者 null 都只提供了有限的支持，而当你把 nil 或者 null 传递给对象的 equals 的时候，一些糟糕的事情如 NullPointerException 就会发生。而 nil 在 Clojure 中也只是一个普通的值而已。

```
    state == ((StatefulInteger)obj).state;  
}  
  
// java.lang.Number 类里面其他 xxxValue() 方法……  
}
```

这个类从根本上来说和 `java.lang.Integer` 是一样的，除了这个类里面提供了一些静态方法。一个重大的不同是：它唯一的字段是可变的（没有被定义成 `final`），并且还给这个字段提供了 `set` 方法：`setInt(int)`。我们来看看怎么来使用一个可变的数字：^{注5}

```
(def five (StatefulInteger. 5))          ①  
:= #'user/five  
(def six (StatefulInteger. 6))  
:= #'user/six  
(.intValue five)                         ②  
:= 5  
(= five six)                            ③  
:= false  
(.setInt five 6)                         ④  
:= nil  
(= five six)                            ⑤  
:= true
```

- ① 我们创建了 `StatefulInteger` 类型的两个对象，一个值是 5，另一个是 6。
- ② 我们可以验证 `five` 包含的值确实是“5”……
- ③ 并且 5 跟 6 不相等。
- ④ 我们把 `five` 的状态改成 6，然后……
- ⑤ `five` 现在跟 `six` 相等了。

你会发现这是非常麻烦的。我们通常的设想是——数字是绝对不可变的值，这是远远超越技术层面的，应该可以认为是一个普遍存在的真理。5 应该始终等于 5，而不应该受制于某个有着虐待狂倾向的程序员的奇怪想法。

最后一个例子，展示一下把可变对象与方法调用一起使用的后果：

```
(defn print-number                      ①  
  [n]  
  (println (.intValue n))  
  (.setInt n 42))  
:= #'user/print-number  
(print-number six)                      ②
```

56

注 5：我们在 Clojure 代码中会经常使用 Java 类，Clojure 中提供了大量与 Java 和 JVM 进行互操作的特性。我们会在第 9 章详细介绍。

```
; 6  
:= nil  
(= five six)  
:= false  
(= five (StatefulInteger. 42))  
:= true
```

❶一个简单的 print-number 函数在字面来看，它应该只是把传给它的数字打印到 stdout。出乎我们意料的是，它还额外地更改了 StatefulInteger 参数的值。^{注6}

❷我们用 StatefulInteger 示例中的 six 来调用 print-number 函数。在这个函数返回的时候，这个对象的值也被改变了。

❸到这里，虽然前面我们的代码展示了 six 跟 five 是相等的，但是现在已经不等了。

从一般意义上来说，把可变参数传给函数来调用，不一定要有什么可疑的代码，也不一定就是某个水平很差的程序员写出来的。对传递给函数的参数进行修改太普遍了，而且你肯定碰到过（甚至自己也这么写过）因为函数对传入参数修改而产生的 bug。对于一些全局对象的修改也是一样的道理。而对于这些情况，通过写文档来保证它们不被修改是不靠谱的，因为文档往往写了之后就没人看了，而且这些参数可修改的陷阱确实也是对象深度复制以及复制构造函数存在的理由。

如果你使用的程序语言中数字是这么工作的话，那么不要干了，辞职做木匠去吧。不幸的是，几乎所有其他的对象都是这么工作的。

在 Ruby 中，其他语言中不可变的字符串也是可变的。这是所有问题的根源：

```
>> s = "hello"  
=> "hello"  
>> s << "*"  
=> "hello*"  
>> s == "hello"  
=> false
```

57 Ruby 的集合类似如 hash 和 set 通过把字符串“冻结”来绕过这个问题。但是，如果要想避免这类潜在的严重 bug，你需要对你写的任何跟字符串打交道的类都使用这种类似的技术：

```
>> h = {[1, 2] => 3}          ❶  
=> {[1, 2]=>3}  
>> h[[1,2]]                  ❷  
=> 3  
>> h.keys[0] << 3          ❸
```

注 6： 我们也可以把它实现成一个 Java 方法；这里实现成 Clojure 的函数只是为了方便。

```
=> [1, 2, 3]
>> h[[1,2]]
=> nil
```

④

- ❶ 我们创建了一个 hash(map)，并且把一个数组映射到了一个数字……
- ❷ 我们期望的是，当我们用这个数组到 map 中查询的时候，返回给我们这个数字。
- ❸ 但是如果这个 map 中的任何 key 被改掉了的话……
- ❹ 那么之前能够成功的查询操作，都会失败。^{注7}

这种问题在任何提供了可变对象的语言中都存在，但是如果在这种语言中不可变的值很少被使用的话，那么这种可变对象将是有害无益的：许多程序员的思路是尽量去避免对可变对象的更改操作，所以对于一个简单的问题，比如把集合作为 map 的 key，在这类语言中要正确地实现都是非常困难的。创建一个新的类来包含这个 map，并且只提供有限的几个对 map 进行操作的方法？使用一些简单实现的不可变数据结构，但是在需要对数据结构进行改变的时候，需要对整个数据结构进行全拷贝？哦，真是痛苦！

其实你不用这么痛苦，在 Clojure 中可以安全地使用集合作为 map 的 key（同样道理，set、vector 和记录类型也可以），而你不需要担心这些作为 key 的集合会被改变，或者在多线程情况下的安全问题，因为 Clojure 提供的数据结构都是不可变的并且是高效的：

```
(def h {[1 2] 3})          ❶
:= #'user/h
(h [1 2])
:= 3
(conj (first (keys h)) 3)    ❷
:= [1 2 3]
(h [1 2])                  ❸
:= 3
h
:= {[1 2] 3}                ❹
```

- ❶ 我们创建了一个 Clojure 的 map，并且把一个 vector 作为 key 映射到一个数字。
- ❷ 我们可以向这个作为 key 的 vector 里面“添加”一个数字，这个添加操作会返回一个“更新的”vector，但是……
- ❸ 但是，这个更新的 vector 对于原来 map 中的那个作为 key 的 vector 完全没有影响，我们还是可以使用原先的 vector 来查找到对应的 value，这是因为……

◀ 58

注7： 是的，我们可以使用 hash 的 rehash 方法，这样确实可以“解决”这个问题。如果你不在意在任何查询之前都对 map 进行 rehash，并且如果这个 hash 同时被多个线程访问的话，还需要在 rehash 的过程中以及对 map 进行更新的过程中维持一个锁。

- ❶ 这是因为这个 `vector` 并没有被改变，而且也不可能被改变；当我们往里面添加一个新的数字的时候，它返回的是一个全新的 `vector` 引用。

这里介绍的都只是冰山的一角。理解并且运用 Clojure 的数据抽象和数据结构的具体实现是灵活运用这门语言的基础。我们在这一章的例子中会不时地用到 Clojure 中的一些数据结构，并会在下一章中详细地介绍它们。

一个关键性的选择

总结来说，使用不受限制的对象状态意味着：^{注 8}

- 可变对象不能被安全地传给方法调用。
- 可变对象不能安全地作为 `map` 的 `key`、`sets` 的元素等。因为它们的相等语义以及查询语义会随着时间的改变而改变。
- 可变对象不能被安全地缓存。
- 可变对象不能在多线程情况下被安全地使用，因为这需要小心地在不同线程之间进行同步。

如果你使用不可变的值的话，那么这些由于可变对象所引起的一系列问题都将不复存在。对象的可修改性所带来的问题其实大家是有思考的，^{注 9} 这些后果对于那些除了可变对象没有别的选择的语言都是存在的。这些问题如此的普遍，在面向对象的世界里，我们开发了一堆拷贝机制来解决这些由于对对象不加控制的修改而引发的问题：

- 为了确保对对象状态的可靠访问，开发了拷贝构造器以及深度拷贝方法。^{注 10}
- 一堆有关跟踪以及管理对象状态的设计模式：包括“观察者”模式、“反映者”模式等。^{注 11}
- 一大堆用于在可变数据结构中树立一层脆弱的“不可变视图”的工具方法，比如 `java.util.Collections` 中所提供的那些。因为这些工具方法只是简单地把所有的方法代理给下层的可变数据解构；所以如果下层的数据结构发生了改变，那么“不可变”的视图也就变了。

注 8： Brian Goetz 在 <http://www.ibm.com/developerworks/java/library/j-jtp02183.html> 这篇文章中详细讨论了这个陷阱。

注 9： 比如 Joshua Bloch——Java 标准库的核心架构师之一，推荐说你应该“最小化可修改性”（<http://www.artima.com/intv/bloch11.html>），并且在他 2001 年写的 *Effective Java* 一书中写道：“类应该保持不可变，除非你有很好的理由让它可变。”

注 10： 当对对象模型没有提供可靠的拷贝构造函数或者复制方法的时候，我们还要使用序列化技术来达到深度拷贝的目的。

注 11： 在第 12 章中，我们会介绍一些 Clojure 工具，这些工具会使大家所熟悉的面向对象领域的那些设计模式变得无足轻重。

- 这些年来，大量的文档以及糟糕建议^{注12}都告诉我们如何对共享的可变状态的并发访问进行手动锁管理，结果就是，死锁和竞争条件变成了业界软件质量问题的最大根源。

最后，软件编程领域最大的挑战其实是如何辨别出那些不变的元素，你可以在你的算法、你的应用以及你的业务规则里面找到那些不变的元素。你找到的不变元素越多，你就可以把越的注意力集中在某段代码的本地改变上面，也就可以把更多的精力集中在你要做的系统上面。不可变的值建立了一块全新的不可变量的领域；使用它们的话，你可以确切地知道你传给一个函数的集合不会在这个函数内部发生改变；多个线程可以同时去访问一个值而不用担心这个值的一致性被破坏，同时也不用因为要使用复杂的锁机制而使得代码变得复杂而低效；而且同样的操作不会因为时间的不同而产生不同的结果。^{注13}这些东西对于不变量如数字来说都是理所当然的，我们没有理由不要求我们的其他数据结构也有这样的特性，比如 Clojure 的不可变集合以及记录类型。

作为头等公民的函数以及高阶函数

不同语言对于“函数式编程”的解读可能不同，但是有一点是一致的：函数它们本身就是值，这样它们就可以像其他的数据一样，作为参数传给其他函数，同时也可以把函数作为返回值。

函数作为数据使得函数式语言中有着其他非函数式语言中所没有的进行抽象的方法。一个简单的例子，如果我们要实现一个函数，它的作用是调用其他函数两次，要求这个 `call_twice` 要足够通用，它可以调用任何函数，传入任何参数。

这个需求对于函数作为头等公民的语言来说是很简单的，比如 Ruby^{注14} 和 Python：

< 60

```
# Ruby
def call_twice(x, &f)
  f.call(x)
  f.call(x)
end

call_twice(123) {|x| puts x}

# Python
def call_twice(f, x):
    f(x)
```

注 12：也许你会回忆起有关双重校验锁的混乱以及不确定，几年前终于解决了，但是在新的 JVM 内存模型的帮助下，并且解决方法非常复杂。具体可参见：<http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html>。

注 13：A.k.a. Heisenbugs，<https://en.wikipedia.org/wiki/Heisenbug>。

注 14：Ruby 中的 `blocks`、通过 `lambda` 所创建的对象、`Proc.new` 的返回值，以及由 `SomeClass.method(:foo)` 所返回的方法都是头等公民函数的变种。而 Clojure 中的函数则实现了它们所有的功能。

```
f(x)  
call_twice(print, 123)
```

Clojure 的实现也是同样简单：

```
(defn call-twice [f x]  
  (f x)  
  (f x))  
  
(call-twice println 123)  
; 123  
; 123
```

相比较而言，要在 Java 中实现这个简单的函数就非常困难了。比较讽刺的是，JVM 上面的大多数语言都不把函数作为头等公民对待。在 Java 中，代码只能存在于方法里面，而方法必须存在于一个类里面，而且由于 Java 的反射机制，方法本身不能被当做一个变量或者值来直接引用。

而为了弥补没有头等公民函数的缺陷，Java 里面会定义那种只包含静态方法的工具类，如：`java.lang.Math`。其他有用的方法，比如字符串操作方法，被定义在 `java.lang.String` 里面，这些函数不是静态方法，所以只能对一个字符串的实例调用这些函数。

```
Math.max(a, b);  
  
someString.toLowerCase();
```

这些听起来好像挺有道理的：把相关的一些工具方法放在一个专门的类中；把有关这个类的操作全都定义在这个类的实例上面；有什么办法比这些方法更简单呢？

其实，有很多办法可以把这些事情做得更简单而且更强大。

比如，在 Java 中你怎么计算一个数组中最大的那个数字？或者把一个列表中的字符串全部都转成小写的？

61> 示例2-2：用Java写的一些静态工具方法

```
public static int maxOf (int[] numbers) {  
    int max = Integer.MIN_VALUE;  
    for (int i : numbers) {  
        max = Math.max(i, max);  
    }  
    return max;  
}  
  
public static void toLowerCase (List<String> strings) {  
    for (ListIterator<String> iter = strings.listIterator(); iter.hasNext(); ) {  
        iter.set(iter.next().toLowerCase());      ●
```

```
}
```

❶ 小贴士：这里我们直接改变 List 里面的字符串，但愿没有人保持着这个列表的一个引用却不知道我们在更改这个列表。

相比较而言，Clojure 中的所有函数都是头等公民。它们独立存在，不依附于任何对象或者命名空间，可以作为参数传给其他函数，也可以被其他函数作为返回值返回。它们是数据，就如同数据结构、数字以及字符串一样。

Clojure 的核心库中定义了 `max` 函数和 `lower-case` 函数（`lower-case` 是定义在 `closure.string` 命名空间中的），它们分别和上面的 `Math.max` 和 `String.toLowerCase` 相对应。^{注 15} 当然，它们可以被直接使用：

```
(max 5 6)
;= 6
(require 'closure.string)
;= nil
(closure.string/lower-case "Clojure")
;= "closure"
```

但是我们想说的不是它们可以被直接使用。更大的区别是：Clojure 函数本身是值，它们可以被高阶函数（HOF）使用。所谓高阶函数，是指接受函数作为参数，或者把函数作为返回值的函数。

Clojure 本身提供了很多高阶函数，在这里不能一一介绍，我们这里会讨论一些典型的高阶函数：`map`、`reduce`、`partial`、`comp`、`complement`、`repeatedly`。首先来看看 `map`，它可能是 Clojure 中使用最多的高阶函数了。^{注 16}

`map`。`map` 接受一个函数、一个或者多个集合作为参数，返回一个序列（sequence）作为结果，这个返回的序列是把这个函数应用到所有集合对应元素所得结果的一个序列。任何的 `map` 调用：`(map f [a b c])` 和 `[(f a) (f b) (f c)]` 是等价的，而 `(map f [a b c] [x y z])` 和 `[(f a x) (f b y) (f c z)]` 是等价的，等等。

62

看一些实际的例子更有助于我们对 `map` 语义的理解：

```
(map closure.string/lower-case ["Java" "Imperative" "Weeping"          ❶
                                  "Clojure" "Learning" "Peace"])
;= ("java" "imperative" "weeping" "clojure" "learning" "peace")
(map * [1 2 3 4] [5 6 7 8])                      ❷
;= (5 12 21 32)
```

注 15：这些 Clojure 函数其实只是简单地代理给对应的 Java 方法。

注 16：这要归功于“序列”这个抽象的、高度可用的工具，我们会在 89 页的“序列”一节详细介绍。

❶ 这可能是对于 map 最简单的应用了：传一个 lower-case 函数以及一个集合的字符串，map 会返回一个集合的小写字符串。

❷ 把 * 和 n 个集合的数字传给 map，map 返回这些集合中对应的数字相乘的结果。

上面的第一个例子是 Java 静态工具方法：toLowerCase 在示例 2-2 中的 Clojure 实现。即使对于这种简单的例子，Java 和 Clojure 的实现区别也是很大的：

- toLowerCase 静态方法会更改参数的状态，而 map 跟其他正确实现的 Clojure 函数一样，返回一个不可变的值。
- 如果想让 toLowerCase 返回一个新的转换后的字符串集合，那么需要显式地声明这个返回的集合，要考虑它的集合类型；而 map 始终返回序列。^{注 17}
- 在 Java 中，我们总是要不断地去考虑命令控制流，从手动遍历输入集合到调用哪个方法来转换集合元素，再到如何对这些转换后的元素进行排序。Python 的列表推导（list comprehension）以及 Ruby 的 each 在这方面有所改善。而 Clojure 则走得更远，Clojure 鼓励把应用中的共性的东西从具体的应用逻辑中剥离出来。比如在 map 的 mapping 过程中，map 不考虑要 map 的到底是一个序列还是一堆序列，map 也不要求数列必须要实现某个函数，map 甚至不要求这个 mapping 的过程是在一个线程内完成的。^{注 18}

map 是 Clojure 中把一个顺序集合转换成另一个顺序集合的最基本的高阶函数，我们有时候也需要把一个集合合并成一个值——而这个单个值可能不是顺序的。这个就是 reduce 的特长了。

63 ➤ reduce。我们把向一个集合应用一个函数而产生单个值的过程叫做：归约。Clojure 中通过高阶函数 reduce 来实现这个概念。^{注 19} 使用 reduce 的最简单的例子可能就是利用 Clojure 来实现示例 2-2 中提到的 maxOf 静态方法：

```
(reduce max [0 -3 10 48])  
;= 48
```

给定一个函数，以及一个要遍历的集合，reduce 对于集合中的每一个元素都调用一下这个函数，并且收集函数的返回值，在对最后一个元素调用完这个函数之后返回这个返回值作为 reduce 函数的返回值。理解 reduce 的关键是理解它是怎么对集合中的每一个元素调用函数的。如果让我们来手动重复 reduce 的调用过程的话，大概是这样的：

注 17：跟 Clojure 中大多数返回序列的函数一样，由 map 所返回的序列是惰性的。我们会在 93 页的“惰性序列”一节详细介绍；这里你就把它当成普通序列理解就行了。

注 18：这个非常有用，它使得实现 pmap 这样的函数成为可能，你可以用这样的函数来简单地并行化一个纯函数对于多个集合的操作。76 页的“纯函数”一节会详细介绍纯函数，而 166 页的“简单的并行化”一节会介绍 pmap 及其他相关的一些并行化组件。

注 19：Ruby 里面对应函数的名字比较容易混淆，叫做 inject。

```
(max 0 -3)
:= 0
(max 0 10)
:= 10
(max 10 48)
:= 48
```

合并成单个表达式来表示：^{注20}

```
(max (max (max 0 -3) 10) 48)
:= 48
```

在 `reduce` 的第一趟遍历过程中，它对集合的头两个元素调用函数，得到一个结果。在那之后，`reduce` 对前一次调用的结果（比如上面的 0），以及集合里面的下一个元素（比如上面的 10，因为我们前面只用了 0 和 -3）调用函数来获取下一个结果，等等。同时你还可以给 `reduce` 提供一个初始值：

```
(reduce + 50 [1 2 3 4])
:= 60
```

这里，`reduce` 并没有做什么不同的事情，只是第一次调用函数的时候，它的参数是我们提供的“初始值”以及集合的第一个元素，后面的过程就没什么区别了。能够提供一个“初始值”给 `reduce` 函数是一个很关键的特性，因为这使得我们可以把一个集合的元素转换成任意类型的值。比如我们可以把一个集合的数字转换成以数字为 `key`，以数字的平方为 `value` 的 `map`：

```
(reduce
  (fn [m v]
    (assoc m v (* v v)))      ❶
  {}
  [1 2 3 4])                  ❷
:= {4 16, 3 9, 2 4, 1 1}
```

64

- ❶ 跟其他例子一样，我们提供一个函数给 `reduce`，但是在这个例子中，我们提供的函数是内联的匿名函数，它接受两个参数，一个 `map`（始终是前一次调用函数的返回值，或者我们提供的初始值），以及在遍历的集合中的下一个元素。我们这里提供的函数简单地把集合中的元素以及元素的平方 `assoc`^{注21} 进这个 `map`；`assoc` 函数返回一个包含了我们元素的新的 `map`，而这个 `map` 会作为下一次调用函数时候的第一个值。

注 20：Clojure 通过很多手段保证 Clojure 的代码中不会出现这样的代码。比如 `max` 本身就已经是一个不定参函数，因此 `(max 0 -3 10 48)` 明显比这个嵌套的 `max` 调用优美、易懂多了。这里这么写只是为了说明问题而已。

注 21：它是 `associate` 的简写，相当于 Java 中的 `java.util.Map.put(Object, Object)` 或者 Python、Ruby 里面的 `hash_map[key] = value`。我们会在第 3 章介绍 `map`、`assoc` 以及 Clojure 所提供的其他丰富的集合。

- ❷ 这里我们提供给 `reduce` 函数的初始值是一个空的 map。{} 是 Clojure 中空 map 的字面量。

什么时候该用匿名函数，什么时候该用函数字面量

前面的例子是用 `fn` 形式来创建的匿名函数。以这种形式创建匿名函数，然后再把这个匿名函数跟高阶函数如 `map`、`reduce` 一起使用在 Clojure 中非常普遍。但是同时你也应该熟悉 Clojure 的函数字面量，就像我们在前面 40 页“函数字面量”一节所学到的那样，它把 `fn` 符号以及那个显式的参数数组都去掉了，因此它比由 `fn` 创建的匿名函数更加简洁，对于定义那些非常简单的函数来说特别合适。

比如下面是我们用函数字面量对前面的 `reduce` 例子进行重写的结果：

```
(reduce
  #(assoc % %2 (* %2 %2))
  {}
  [1 2 3 4])
:= {4 16, 3 9, 2 4, 1 1}
```

到底使用比较长的匿名函数还是使用比较简洁的函数字面量完全是个人的喜好问题。函数字面量要简洁很多，对于那些短小的函数特别合适；而匿名函数允许我们给函数的参数起有意义的名字从而可读性会稍微好一些。不管怎么样，实际生活当中两种形式都有使用，所有我们要对两种定义函数的方式都熟悉以便方便地阅读 Clojure 代码。

就像我们前面对 `map`、`lower-case` 以及 Java 静态方法 `toLowerCase` 所做过的比较，同样可以对 `reduce`、`max` 以及定义在示例 2-2 中的 Java 静态方法 `maxOf` 来做同样的比较：操作和具体应用的隔离，避免定义具体的控制流程等。但是相比较而言最重要的一点是：没有人会在 Clojure 中定义类似 `maxOf` 和 `toLowerCase` 的静态方法。定义一个核心的操作如：`max` 和 `lower-case`，然后在你需要应用它们的地方应用它们，不管你要使用的高阶函数是什么。

Apply, Partial

函数应用不同于普通的函数调用，你给定一个函数，以及要传给这个函数的参数序列，然后来调用这个函数的过程称为函数应用。比如，在 Ruby 和 Python 中（见示例 2-3 和示例 2-4），可以通过在函数名字前面加上一个星号来应用这个函数到一个数组或者列表。

示例2-3: Ruby里面的函数（方法）应用

```
>> interval = [-10, 10]
=> [-10, 10]
>> Range.new(*interval)
=> -10..10
```

```
>> h = {}
=> {}
>> pair = ['a', 5]
=> ["a", 5]
>> h.store(*pair)
=> 5
>> h
=> {"a"=>5}
```

这个能力对于要支持各种函数式编程概念的语言非常重要——特别是支持调用那种要调用的函数是在运行时才确定（比如在某个上下文中作为参数传给你），并且传给这个函数的参数的个数也是不一定的，所以不可能（或者很烦琐，并且很容易出错）简单地像调用普通函数那样把函数的参数传给函数然后直接调用。

Clojure 中通过 apply 来支持函数应用：

```
(apply hash-map [:a 5 :b 6])
;= {:a 5, :b 6}
```

比较方便的是，apply 在传给 apply 一个列表的不定参数之前还可以先传给 apply 几个确定参数。因为在很多情况下，你要应用的函数的前几个参数是确定的、知道的，这时候就不用再创建一个包含确定参数和不定参数的新列表来调用 apply：

```
(def args [2 -2 10])
;= #'user/args
(apply * 0.5 3 args)
;= -60.0
```

函数应用的时候，你是把函数应用到一个序列集合的参数——必须把函数需要的所有参数都提供给 apply，而偏函数应用则是你把函数的一部分参数传给一个函数，这样创建一个新的函数，这个函数需要的参数就是你没有传给那个函数的那些剩余参数。

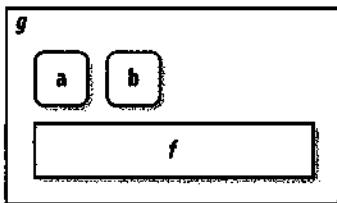
◀ 66

示例2-4: Python里面的偏函数应用

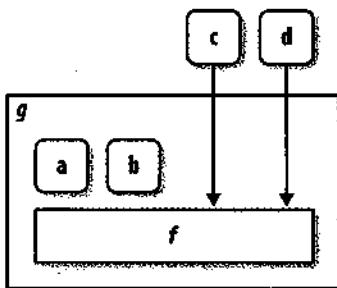
```
>> from functools import partial
>> only_strings = partial(filter, lambda x: isinstance(x, basestring))
>> only_strings(['a', 5, 'b', 6])
['a', 'b']
```

这里发生了什么？partial 接受一个函数（比如这里的 filter，但是可以是任何函数 f ），

以及这个函数的一个或者多个参数（比如这里的字符串类型的谓词，但是可以是任何 a, b, …）作为参数，并且产生了一个新的函数 g，这个函数保持了这些参数以及函数 f 的引用：



当我们调用从 `partial` 返回的这个新函数 g，函数返回的结果就是我们调用函数 f 的结果，而调用的参数则是传给 `partial` 的参数 (a, b, ...) 以及调用 g 的时候传入的参数：



Clojure 中通过 `partial` 来提供这种偏函数应用的机制是：

```
(def only-strings (partial filter string?))
;= #'user/only-strings
(only-strings ["a" 5 "b" 6])
;= ("a" "b")
```

- 67> 偏函数应用在很多情况下都很有用。比如，有一些函数需要一些配置才能进行工作（可能是要连接的数据库的地址，或者要写的文件的路径），这些配置信息通常作为这个函数的头一个或者是头几个参数。这样我们就可以通过 `partial` 来创建一个偏函数把这个配置信息包进去，这样再去写通过配置信息获取一些数据的代码的时候就不用关心这些配置信息到底是怎么来的了，比如：

```
(def database-lookup (partial get-data "jdbc:mysql://..."))
```

`partial` 的这个特点使我们可以创建一些指定我们关心的那些参数的偏函数，而不去理会我们不关心的那些参数，这使得把 `partial` 和 `comp` 放在一起使用很有用。^{注22}

注 22： 我们很快会在 68 页的“函数（功能）的组合”一节详细介绍 `comp`。



有一点要注意的是，跟普通的明确指定函数参数的调用相比，高阶函数 `apply` 和 `partial` 确实会有一些性能损耗，损耗很小，而且只有在参数个数较多的情况下有损耗：对于 `comp` 来说是 3 个以上的参数，对于 `partial` 来说是 4 个以上的参数。在参数较少的情况下，这些函数使用了一些特殊的实现：把这些参数包在一个闭包里面，而不用显式地去对参数列表进行装包、解包。

这些高阶函数之所以有性能损耗是因为当传很多参数给 `apply` 和 `partial` 的时候，这些函数需要把这些传过来的参数列表进行解包，以确定我们要调用的函数的参数个数，以选择具体的函数体（一个函数根据参数的不同可能有多个函数体）。这个当然不能跟调用一个普通的 Clojure 函数的速度相比——它是直接利用 JVM 的（非常快）的方法调用机制。而好的一面是，正是因为 Clojure 使用的底层方法调用机制非常高效，通过 `apply` 来调用函数，以及调用由 `partial` 返回的函数还是比 Python 或者 Ruby 中显式调用方法要快。

偏函数 vs. 函数字面量。你可能已经意识到了，从技术层面来讲，函数字面量提供了 `partial` 所提供的功能的超集：可以通过函数字面量来实现偏函数类似的功能。

```
(#(filter string? %) ["a" 5 "b" 6])
;= ("a" "b")
```

但是，函数字面量并不限制你去指定函数的开头几个参数：

```
(#(filter % ["a" 5 "b" 6]) string?)
;= ("a" "b")
(#(filter % ["a" 5 "b" 6]) number?)
;= (5 6)
```

这里的权衡是函数字面量强制指定要调用的函数的所有参数，而 `partial` 则不强制这一点：◀ 68

```
(#(map *) [1 2 3] [4 5 6] [7 8 9])          ❶
;= #<ArityException clojure.lang.ArityException:
;= Wrong number of args (3) passed to: user$eval812$fn>
(#(map * % %2 %3) [1 2 3] [4 5 6] [7 8 9])    ❷
;= (28 80 162)
(#(map * % %2 %3) [1 2 3] [4 5 6])            ❸
;= #<ArityException clojure.lang.ArityException:
;= Wrong number of args (2) passed to: user$eval843$fn>
(#(apply map * %&) [1 2 3] [4 5 6] [7 8 9])    ❹
;= (28 80 162)
(#(apply map * %&) [1 2 3])
;= (1 2 3)

((partial map *) [1 2 3] [4 5 6] [7 8 9])        ❺
;= (28 80 162)
```

- ❶ 我们必须把所有的参数都提供给 `map` 函数；这里的这个函数字面量其实只接受 0 个参数，因为通过字面量定义的函数最后到底接受多少参数是固定的——由它所引用的最大的占位参数确定。
- ❷ 我们可以通过显式地把所有的参数都传给 `map` 来解决这个问题。
- ❸ 但是，如果我们定义的函数字面量接受的参数的个数跟实际传入的参数的个数不一致的话，这个办法就不行了。
- ❹ 一种可行的办法是使用 `apply`，并且通过 `%&` 来说明这个函数字面量接受的剩余参数。
- ❺ 这里其实只是自己实现了一下 `partial` 的功能，而直接用 `partial` 则更好，不用自己去“实现”这个语法。

就像你看到的一样，有时候使用 `partial` 写出来的代码更可读，而有时候用 `partial` 则是为了给一个不定参数的函数指定几个参数的值。

函数（功能）的组合

可组合性是一个被过度使用的词汇，在我们这里指的是多个小程序组合成一个有用的大组件的能力。

不同的编程模型提供了把小程序组装成大组件的不同的支持。命令式的、过程式的代码通常根本就不能组合；在一个方法中调用其他方法真的不能算。面向对象编程提供了对于组合的基本支持，最典型的就是对象的组合：在一个对象中把另一个类的对象作为字段。代理模式、策略模式则是另一些面向对象领域进行“组合”的方法。

- 69 函数式编程对于可组合性有着更好的支持，使得你可以非常容易地从编写小程序开始，最后组合成一个大组件。我们会在书的第 2 部分深入探索如何在 Clojure 中进行组合、封装，不过进行组合、封装的最简单的就是函数。因为函数通常跟数据完全隔离——而且理想情况下，它们应该是可以处理任何类型的、语义相同的数据——函数是进行组合的最简单、最有力的武器。

函数组合在函数式编程领域有着非常不同的意义：^{注23} 对于给定的任意个数的函数，它们会被组成一个新的函数，所有传给这个新函数的参数会用来调用每个函数，每个函数的返回值会作为参数来调用下一个函数，这些函数调用的顺序则通常与传入的顺序相反。

我们举个简单的例子实现这个功能：给定一个列表的数字，返回这些数字的总和的负数的字符串形式。用 Clojure 来实现的话，非常简单：

注 23： 函数组合还有一些特殊的表示法。比如数学上的函数 f 和 g 的组合表示成 $f \circ g$ ，而 Haskell 里面则表示成 $f . g$ 。

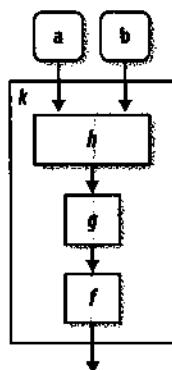
```
(defn negated-sum-str
  [& numbers]
  (str (- (apply + numbers))))
:= #'user/negated-sum-str
(negated-sum-str 10 12 3.4)
:= "-25.4"
```

在 Clojure 中，通过 `comp` 来实现函数组合，用 `comp` 来重写上面代码的话会更加简洁，并且通常也更清楚：

```
(def negated-sum-str (comp str - +))
:= #'user/negated-sum-str
(negated-sum-str 10 12 3.4)
:= "-25.4"
```

这里的两个 `negated-sum-str`，在实现功能上是一样的。不一样的只是函数的组合方法。

`comp` 到底是怎么执行的？你可以把 `comp` 当做一个管道（pipe）：它接受任意数量的函数——比如上面的 `str`、`-` 和 `+`（可以是任意函数 `f`、`g` 和 `h`）——并且生成一个新的函数（图中用 `k` 来表示）。新生成的函数 `k` 接受的参数的个数跟 `comp` 接受的最后一个函数接受的参数个数相等（图中的 `h`，因为生成的 `k` 函数第一个调用的就是管道里的 `h`），然后调用把 `h` 的返回值作为参数调用 `g`，然后再以 `g` 的返回值作为参数来调用函数 `f`，依次类推。



`comp` 生成的函数的返回值是 `comp` 的第一个参数（函数）的返回值。

70

唯一的局限是提供给 `comp` 的函数的返回值必须能够作为下一个函数的参数。所以，如果我们把这些函数顺序倒过来，它就会抛出错误：

```
((comp + - str) 5 10)
:= #<ClassCastException java.lang.ClassCastException:
:=     java.lang.String cannot be cast to java.lang.Number>
```

因为函数调用：`(str 5 10)` 的结果是一个字符串，不能把字符串作为参数来调用 `-`。

可别以为 `comp` 只能用来写写这种“hello world”的小玩意，它可以用来组合很强大的功能。因为 `comp` 返回的还是函数，还是可以进一步被作为参数传递给 `comp` 以进一步组合。比如在 Java、XML 以及其他很多语言中标识符通常都是 CamelCase 式的，我们有时候可能想把这样的标识符放进 Clojure 的 map 中，但是同时也想使这些标识符使用 Clojure 中通用的以横线分隔的小写单词的样式。

要实现这个转换，`comp` 会是一个不错的选择：

```
(require '[clojure.string :as str])  
  
(def camel->keyword (comp keyword  
                           str/join  
                           (partial interpose \-)  
                           (partial map str/lower-case)  
                           #(str/split % #"(?<=[a-z])(?=[A-Z])")))  
  
;; #'user/camel->keyword  
(camel->keyword "CamelCase")  
;; :camel-case  
(camel->keyword "lowerCamelCase")  
;; :lower-camel-case
```

- 71 ◀ ① 我们这里使用 `require` 来加载 `clojure.string` 命名空间，并且给它一个别名：`str`。
② 传给 `comp` 的函数不一定是事先定义好的函数。比如我们这里使用了一个函数字面量：它用一个正则表达式^{注24} 来把提供的 CamelCase 的标识符分隔成一个个单词。



你可以以其他方式来实现 `comp` 的功能：使用 `->` 和 `->>` 宏。^{注25} 由于是宏，它们操作的不是函数，它们把传给它们的代码做了重新调整：使得你传给这些宏的第一个参数作为后面所有函数的第一个参数 (`->`) 或者最后一个参数 (`->>`)。比如下面的实现跟我们上面用 `comp` 实现的功能是一样的：

```
(defn camel->keyword  
  [s]  
  (->> (str/split s #"(?<=[a-z])(?=[A-Z])")  
        (map str/lower-case)  
        (interpose \-)  
        str/join  
        keyword))
```

要实现一个具体的功能到底是使用 `comp` 还是这些串行宏其实只是一个人喜好的问题。^{注26}

注 24：Clojure reader 支持很方便的正则表达式字面量，详见 17 页的“正则表达式”一节。

注 25：我们会在 259 页的“深入 `->` 和 `->>`”一节详细介绍 `->` 和 `->>`。

注 26：`comp` 和 `partial` 使得我们可以进行无参风格（又称为默契编程）编程，它最大的特点就是定义函数的时候不用显式地定义它的参数。

我们可以把 `camel->keyword` 进行进一步组合：定义一个函数，这个函数把传入的 CamelCase 的 key 转化成 Clojure 命名风格的 key：

```
(def camel-pairs->map (comp (partial apply hash-map)
                               (partial map-indexed (fn [i x]
                                                     (if (odd? i)
                                                         x
                                                         (camel->keyword x))))))
 ;;= #'user/camel-pairs->map
 (camel-pairs->map ["CamelCase" 5 "lowerCamelCase" 3])
 ;;= {:camel-case 5, :lower-camel-case 3}
```

编写高阶函数

`comp` 所提供的函数组合方式只是 Clojure 提供的众多函数组合方式中的一个——当然，是非常有用的一个，它之所以应用得这么广泛，是因为它是进行函数组合比较底层的一个工具。虽然 Clojure 提供了一些通用作用的高阶函数，但是我们自己也可以编写自定义的高阶函数。

72

如果想很好地利用高阶函数的话，有时需要自己设计函数和高阶函数的交互，来达到更复杂更有效的功能。一旦你习惯把一个函数当做一个普通的值，那么你会发现编写一个把函数作为返回值或者接受函数作为参数的高阶函数是很自然的事情。

让我们先看一些简单的例子。首先，编写一个高阶函数，这个函数返回某个给定数字与它的参数的和：

```
(defn adder
  [n]
  (fn [x] (+ n x)))
 ;;= #'user/adder
 ((adder 5) 18)
 ;;= 23
```

稍微复杂一点的例子：编写一个高阶函数，它接受一个函数作为参数 (`f`)，同时返回一个函数 (`r`)，`r` 函数的作用是返回 `f` 函数的返回值的两倍：

```
(defn doubler
  [f]
  (fn [& args]
    (* 2 (apply f args))))
 ;;= #'user/doubler
 (def double-+ (doubler +))
 ;;= #'user/double-+
 (double-+ 1 2 3)
 ;;= 12
```

让我们再看一些更有趣的例子，在这个过程中我们也会体会到函数式编程不只对于处理不可变数据以及算法很有用，对于解决跟状态、IO 有关的问题也非常拿手。

利用可组合的高阶函数构建一个日志系统

日志对于任意大小的系统都是一个必备的组件，而对日志系统进行配置通常也因为种种原因很烦人并且复杂。我们这里用高阶函数来实现一个日志系统。^{注27} 过程中我们会用到一些还没介绍过的 Clojure 的概念、细节，但是应该很容易理解。

73 我们都为使用 `System.out.println`、`puts` 或者 `print` 来打印日志而感到羞愧——它们虽然有时候很方便，但是太原始了。为了改进这一点，我们会从最简单的实现开始，实现一个高阶函数，这个高阶函数的返回值是一个函数，它可以把信息打印到任何作为参数传给高阶函数的 `writer` (`java.io.Writer` 的实例)：

```
(defn print-logger
  [writer]
  #(binding [*out* writer]
    (println %)))
```

- ❶ 我们的高阶函数接受一个参数，这个参数可以是任何实现了 `java.io.Writer` 接口的类的一个实例，这个接口是写数据到某个输出设备的最基本的接口。
- ❷ `print-logger` 返回一个函数，这个函数绑定 `*out*`^{注 28} 到我们传入的 `writer`。
- ❸ 返回的函数的函数体把它唯一的参数（要打印的消息）用 `println` 写入到 `*out*`（而 `*out*` 的实际地址已经被我们换成了 `writer`）。

让我们看看它到底是如何工作的：

```
(def *out*-logger (print-logger *out*))          ❶
 ;;= #'user/*out*-logger
 (*out*-logger "hello")                         ❷
 ; hello
 ;;= nil
```

- ❶ 我们把 `*out*` 传给了 `print-logger`，所以所有的消息会被打印到 `*out*`——或者说打印到在定义 `*out*-logger` 时 `*out*` 的值。
- ❷ `print-logger` 总是返回一个接受单个参数的函数，这里我们传一个字符串来调用它。

好吧，我承认，到现在为止我们所实现的只是一个复杂的 `println` 功能，它的作用就是

注 27： 我们这里写的日志函数只是一个小例子，真的要在生产环境中使用的话，还是推荐你使用成熟的 `tools.logging` 库，具体请参见 <https://github.com/clojure/tools.logging>。

注 28： `*out*` 默认就是绑定到标准输出上去的。通过对它进行重新绑定，可以把要打印到标准输出的内容转到我们指定的地方。绑定 (`rebind`) 在 201 页的“动态作用域”一节有详细介绍。

把输入的消息打印到标准输出 (stdout)。一个有趣的实现是把这些消息打印到一个内存 buffer。这里我们使用 `java.io.StringWriter`, 它是 `java.io.Writer` 的一个实现, 但是它不是输出到一个设备, 而是直接保存到内存中的一块区域:

```
(def writer (java.io.StringWriter.))          ❶
:= #'user/writer
(def retained-logger (print-logger writer))
:= #'user/retained-logger
(retained-logger "hello")                     ❷
:= nil
(str writer)                                ❸
:= "hello\n"
```

- ❶ 我们创建并且保持 `StringWriter` 类的一个引用, 创建引用是为了在后面调用 `print-logger` 之后来检查消息是不是真的打印到它里面了。◀ 74
- ❷ 这里调用日志函数, 它不会打印任何内容到标准输出……
- ❸ 因为信息被 `println` 到了我们提供的 `StringWriter` 对象。

这个例子已经比前面的例子有趣了, 但是任何日志框架都应该有能力把消息打印到一个文件。上面定义的高阶函数 `print-logger` 允许我们很容易做到这一点: 对于任意给定的 `Writer`, 它都会返回一个函数, 而这个返回的函数就会把消息打印到我们提供的 `Writer`。所以只要能够得到一个输出设备是我们指定文件的 `Writer` 的引用, 那就搞定了:

```
(require 'clojure.java.io)

(defn file-logger
  [file]
  #(with-open [f (clojure.java.io/writer file :append true)]      ❶
    ((print-logger f) %)))                                         ❷
                                                               ❸
```

- ❶ 高阶函数 `file-logger` 接受单个参数 `file`, 我们的消息会被打印到这个文件里面去。由于 `clojure.java.io/writer` 的语义, 这个 `file` 参数可以是文件的具体路径, 也可以是 `java.io.File`、`java.net.URL` 或者 `java.net.URI` 类的一个对象, 效果是一样的。
- ❷ 由 `file-logger` 所创建的函数字面量返回一个指向 `file` 的 `writer`, 这个 `file` 以追加模式打开 (以使得我们在写入新消息的时候不会把以前写的老消息覆盖掉), 并且把这个 `Writer` 在局部作用域命名为 `f`。^{注 29}
- ❸ 这里没有重复实现把 `*out*` 绑定到 `writer`, 然后用 `println` 来打印的逻辑。我们简单

注 29: `with-open` 能够保证 `f` 在 `with-open` 结束的时候被关闭, 它跟 Java 7 里面的“try with resources”语法以及 Python 里面的 `with` 语法类似。364 页的“`with-open: finally` 的挽歌”一节有关于 `with-open` 的详细介绍。

地以我们创建的 `Writer` 来调用 `print-logger`, 这会创建一个日志函数。要记住的一点是, 函数本身就是值, 所以我们不需要在上面先定义一个指向函数的引用, 然后再调用它。这里由 `print-logger` 创建的函数被创建之后, 直接调用, 然后丢弃掉了。

来试一下效果:

```
(def log->file (file-logger "messages.log"))
;= #'user/log->file
(log->file "hello")
;= nil

% more messages.log
hello
```

我们能够创建一个打印日志到文件的函数了(当前目录下的 `messages.log` 文件), 而且测试的结果告诉我们日志确实被打印到这个文件里面去了。对于只有 10 行左右的代码来说已经很棒了。还可以定义其他的日志高阶函数, 比如定义一个打印到数据库的, 一个打印到消息队列的, 打印到其他存储介质的等, 这样我们可以根据需要来选择使用。
75>

但是如果我们想把一条日志打印到多个地方呢? 这个逻辑不应该是我们的日志函数所需要关心的。要实现这个, 需要另一个不同的高阶函数, 这个函数它本身不做写日志的事情, 而只是调用其他日志函数来写日志:

```
(defn multi-logger
  [& logger-fns]          ①
  #(doseq [f logger-fns]  ②
    (f %)))
```

- ① `multi-logger` 接受任意个数的日志函数。
- ② 返回的函数会遍历^{注30} 调用的每一个函数来打印日志。

有了这个高阶函数, 就可以很方便地定义打印日志到多个地方的日志函数了:

```
(def log (multi-logger           ①
            (print-logger *out*)
            (file-logger "messages.log")))
;= #'user/log
(log "hello again")
; hello again
;= nil

% more messages.log
hello
hello again
```

注 30: 91 页的“序列不是迭代器”一节详细介绍了 `doseq` 以及在 Clojure 里面如何遍历序列。

- 有了 `multi-logger` 的帮助，我们定义了一个新的“顶级”日志函数，它能把一条日志打印到标准输出，同时也打印到 `messages.log` 文件中去。

是的，现在我们能把日志打印到多个地方去了。

下面我们来看看对于这个迷你日志函数库的最后一个改进。因为我们在所有这些函数之间传递的参数形式是统一的（一个字符串），因此可以定义一个高阶函数来对这个传入的日志进行一些处理，比如最常见的：在日志消息前面加一个时间戳：

示例2-5：在每条日志前面加一个时间戳

```
(defn timestamped-logger
  [logger]
  (#(logger (format "[%1$tY-%1$tm-%1$te %1$tH:%1$tM:%1$tS] %2$s" (java.util.Date.) %))) ●

(def log-timestamped (timestamped-logger
  (multi-logger
    (print-logger *out*)
    (file-logger "messages.log"))))

(log-timestamped "goodbye, now")
; [2011-11-30 08:54:00] goodbye, now
:= nil

% more messages.log
hello
hello again
[2011-11-30 08:54:00] goodbye, now
```

- `timestamped-logger` 返回的函数简单地在每条消息前面加上时间戳，然后再把这个有时间戳的消息传给日志函数去打印日志。`format` 函数利用 Java 的 `String.format` 方法来实现类似 `sprintf` 的效果。

我们还可以想出很多种对日志消息进行转换的办法，比如加上当前的命名空间，加上一些上下文信息（比如应用所在机器的 ip），再或者加上这条日志信息的行号，^{注31} 等等。更显著的是，如果要在比较大的项目中使用这个迷你日志库的话，那这个日志库最迫切的改进则是使得这些函数能接受字符串之外的其他数据结构，这样日志信息本身的内容可以更丰富。比如我们可以用 `map` 来描述一个日志消息，^{注32} 然后可以很轻松地对这个 `map` 进行各种转换，往 `map` 中加入一些附加信息。因为 `map` 是非常结构化的，因此也可以很轻松地写一些函数来对日志消息进行过滤，等等。

^{注 31}：这个可能需要利用宏来获取调用 `logger` 函数的行号；这个行号被记录在 `:line` 元数据中了。254 页“在宏里面打印有用的错误信息”一节有详细介绍。

^{注 32}：当然，我们应该始终保证简单地传入字符串来记日志也是可以的。我们可以隐式地把 “foo” 这样的字符串转换成 `{:message "foo"}`。

纯函数

虽然利用不可变的值进行编程能够消除对数据进行处理时候的很多错误，但是也有很多错误产生的原因是我们在写函数的方式。大多数错误是由于我们的函数产生了副作用，它指的是函数除了会返回一个值还会改变外界环境（比如某个全局变量之类的）的某些属性，或者是它的返回值依赖于外部环境的某个属性。

再回想一下图 2-1，副作用是指函数与外界环境的交互，不管是函数改变外部环境，还是依赖外部环境的一个值。任何一个跟随机数^{注33}相关的函数都是有副作用函数的典型例子，因为它：

1. 依赖于它使用的随机数生成器的状态。^{注34}
2. 因为随机数生成器每次生成的随机数是不一样的，那么函数下一次调用的时候拿到的随机数，跟这次调用的随机数就不一样了，这个函数也就依赖于环境了，所以说它是有副作用的。



根据定义，任何跟 IO 打交道，或者对共享对象进行修改的函数都是有副作用的。

随机数这个例子可能太极端，那我们来看一个现实的例子：^{注35}

```
(defn perform-bank-transfer!
  [from-account to-account amount]
  ...)

(defn authorize-medical-treatment!
  [patient-id treatment-id]
  ...)

(defn launch-missiles!
  [munition-type target-coordinates]
  ...)
```

或者一个更具体的例子，实现一个小功能：传入参数是 Twitter 的用户名，返回这个用户的粉丝数目：

```
(require 'clojure.xml)
```

注 33：Clojure 中通常用 `rand` 和 `rand-int` 来产生随机数。

注 34：在 Clojure 中就是 `java.math.Random` 类的一个实例。

注 35：我们通常把有副作用的函数的名字以感叹号结尾，这样可以提醒代码的使用者以及读者这个函数有副作用。

```

(defn twitter-followers
  [username]
  (-> (str "https://api.twitter.com/1/users/show.xml?screen_name=" username)
       clojure.xml/parse
       :content
       (filter (comp #{:followers_count} :tag))
       first
       :content
       first
       Integer/parseInt))
(twitter-followers "ClojureBook")
:= 106
(twitter-followers "ClojureBook") ①
:= 107

```

78

- ① 在不同时间以相同参数来调用 `twitter-followers` 函数的结果可能会不一样（这个不难理解吧？你可能有新粉丝嘛）。

显而易见，如果一个函数跟随机数有关，那么我们是没有办法测试它的——我们不知道它下一个随机数是多少。同样，对于任何函数，它如果依赖于外部环境，或者会对外部环境产生影响都不好测试，因为要穷举所有可能的边界条件以及失败条件是很困难的。这也正是我们在测试的时候要使用 `mock` 对象的原因了，使用 `mock` 可使外部环境处于一个确定的状态，这样才能确定函数的输出结果。但是不幸的是，你的函数在现实世界的表现跟你 `mock` 时候的表现不会完全一样的——相信我。

跟有副作用的函数相对的是：纯函数。它不依赖于外部的数据源，也不会改变任何外部环境的状态，因而对于相同的参数，始终返回相同的结果。^{注36} 所有数学相关的函数都是纯函数——比如 `+并不依赖于外部环境，也不会对外部环境产生影响，对于相同的参数，始终返回相同的值。其他编写正确的、处理不可变值的函数都是这样的。`

为什么纯函数很有意思？

前面已经列举了一些使用、编写纯函数的好处。下面会列举几个原因，这些使得使用纯函数，再加上一些良好的不可变类型的值、数据结构可以大大简化软件开发的过程。

纯函数更容易理解。回忆一下我们在 54 页的“值与可变对象的比较”一节所提到的不可变的值是如何使得我们发现一个新的不可变的领域。纯函数提供了类似的好处，但是纯函数更多地关注操作，而不仅仅是数据的状态。如果你知道某个函数 `f` 对于参数 `α` 和 `β`，始终返回 `γ`，而且你知道调用 `f` 从来不会去改变数据库或者去写一个文件，或者从 `socket` 读取一些数据，那么你可以在任何环境下调用 `f`，并且知道它的返回值是确定的。

^{注 36：} 这最后一个特征被称为 `幂等性`，跟函数的“纯”类似。比如，一个函数如果始终返回同一个值，但是有副作用——比如会写日志文件，那么我们说这个函数是幂等的，但不是纯的。

纯函数更容易测试。上面谈了那么多之后，这一点应该是显而易见的了。如果你知道你的函数没有副作用，并且它的返回值是完全由它的参数决定的，测试它们将非常简单。你可以明确地定义函数参数的范围，同样也可以严格地定义函数返回值的范围。因为函数的返回值完全由参数决定，mock 将不再需要。这些特点将使你可以全面地测试一个纯函数——如果你想这么做的话——而这对于有副作用的函数来说是基本不可能的。

你可能已经在这样做了——把尽可能多的功能写进可以进行单元测试的方法，而剩下的一些跟状态强相关的功能代码（比可单元测试的代码难测很多）由集成测试和功能测试来覆盖。从这个角度来考虑：纯函数会帮助你缩小这两个领域的代码量，并且使你可以花更多的时间在你喜欢的领域（可单元测试的代码）。

纯函数的结果是可以被缓存的，并且很容易并行化。只包含纯函数的表达式被称为是引用透明的。意思就是，对于任何使用这种表达式的地方，你可以放心地把它用它的返回值代替，而不会改变代码的行为。比如下面的这些表达式都是等价的，因为这些表达式中使用的函数都是纯函数：

```
(+ 1 2) (- 10 7) (count [-1 0 1])
```

这些表达式出现的地方都可以直接把它们用它们的结果 3 来代替，而不会改变代码整体的结果。

从实用的角度来说，这意味着我们可以放心地缓存纯函数的返回值，这样下次再调用它的时候可以直接返回这个返回值而不用重新计算。这个技术叫做：内存化，当一个函数的调用成本很高时，这个技术就能帮上忙了。Clojure 提供了一个简单的实现：`memoize`，把一个函数传给它会返回这个函数的内存化版本：

```
(defn prime?          ❶
  [n]
  (cond
    (= 1 n) false
    (= 2 n) true
    (even? n) false
    :else (-> (range 3 (inc (Math/sqrt n)) 2)
                (filter #(zero? (rem n %)))
                empty?)))

(time (prime? 1125899906842679))           ❷
; "Elapsed time: 2181.014 msecs"
:= true
(let [m-prime? (memoize prime?)]
  (time (m-prime? 1125899906842679))
  (time (m-prime? 1125899906842679)))
; "Elapsed time: 2085.029 msecs"             ❸
```

```
; "Elapsed time: 0.042 msecs"
;= true
```

80

- 首先，我们定义了一个检测给定数字是不是素数的函数（这里通过简单的相除来实现）。
- 对于一个很大的素数，这个检测要花很长时间。
- 在这个函数被内存化之后，第一次调用这个函数花的时间还是很长。但是…
- 后续所有相同参数的调用都会立刻返回，因为对应的返回值被内存化了。

有副作用的函数，就不是引用透明的了，所以也就不能内存化了。比如如果内存化 `rand-int` 函数，会怎么样？^{注37}

```
(repeatedly 10 (partial rand-int 10))
;= (3 0 2 9 8 8 5 7 3 5)
(repeatedly 10 (partial (memoize rand-int) 10))
;= (4 4 4 4 4 4 4 4 4 4)
```

你猜对了，我们的随机数生成方法不能产生随机数了！因为一旦内存化之后，第二次之后的所有调用都不会去真正调用这个函数了（`rand-int` 没有参数，对 `memoize` 来说，所有调用的参数是一样的），所以后续调用返回的都是第一次调用生成的随机数——这可不是我们想要的结果。



`memoize` 是怎么做到的呢？保存所有调用参数和对应返回值的映射——不会被垃圾回收。所以如果一个函数的参数的取值范围很广，或者参数、返回值很占用内存，那么通常会造成“内存泄露”，特别是当它们被用 `def` 幼稚地定义成一个全局的 var。

对于这种问题的解决办法：

1. 不要把它们定义成顶层的函数，把它们定义成一个顶层函数的内部，然后只在需要的时候调用它们。
2. 使用 `core.memoize` (<https://github.com/clojure/core.memoize>)，这是一个内存化函数库，它提供一些不同的内存化策略，包括在一定时间之后失效缓存的内容等。

现实生活中的函数式编程

81

在本章正式讨论函数式编程之前，我们看了一张所有程序的原理图（图 2-1），而且我们讨论过对于“要写一个有用的程序，那么必须要跟那些繁杂、不可靠的外部状态打交道”种种反对意见。这里我们要给你一个稍微修正一点的观点：你可以利用函数式编程的各

注 37：不管是 `rand-int` 自身被内存化，还是 `(partial rand-int 10)` 的结果被内存化，结果都是一样的，到底是为什么就让大家自己想想作为练习了。提示：想想在每种情况下传递给内存化函数的参数各是什么。

种优点，同时又不会降低代码的实用性。

不管你在编写的是什么软件，你都应努力地把你代码独特的地方与那些普通的细枝末节隔离开。你会设计你的领域模型、抽象关键操作、定义核心算法、构建细粒度的可测试的代码模块，所有的这些都是在寻找和定义“不变量”，从而可以圈定软件的范围，而不是无边无界的。

利用这些已有的成果，并且以函数式的方式来做事情会给我们带来巨大的好处，就如同我们整章一直提到的那样。有了这样一个坚实的基础，你可以比以前更自信地处理你的程序与外界环境的交互，因为你知道至少你的应用的核心部分（不直接跟外界打交道的那部分）是稳定可靠的。这使得我们可以把前面的程序原理图做一个细微但是影响重大的调整，如图 2-2 所示。

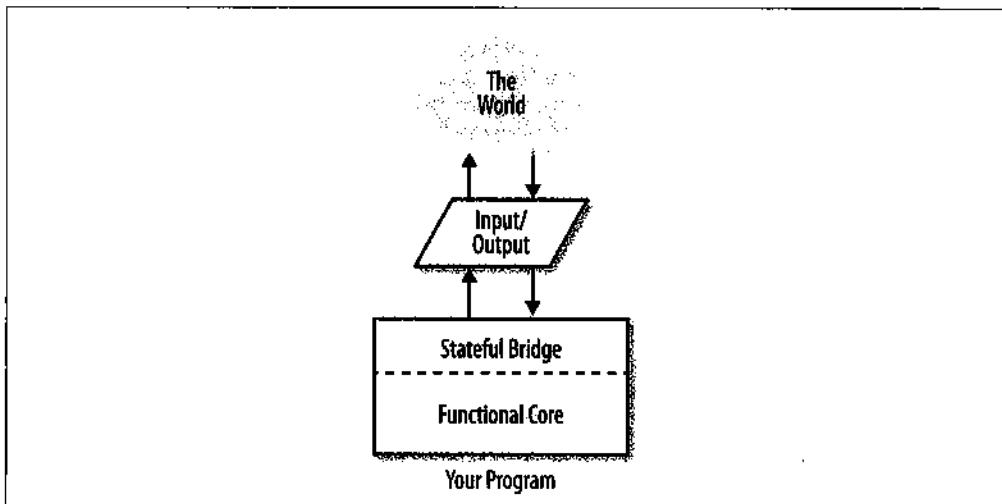


图 2-2：所有函数式程序的原理图

82 Clojure 的大多数吸引人的特性都依赖于使用不可变的数据结构、作为头等公民的函数、可组合的值以及强调最小化副作用的编程原则。就像我们在本章总结的那样，这个的影响是深远的，它使得代码更容易看懂、测试以及组合，同时也使得那些看起来很难的并行化、可靠的并发语义变得很简单。把你的核心应用用函数式编程来实现是一个明智的选择。

集合类与数据结构

map、vector、set 和列表是 Clojure 提供的基本数据结构。你已经看到，Clojure 为它们提供了便利的字面量：

```
'(a b :name 12.5)           ;; 列表  
['a 'b :name 12.5]          ;; vector  
{:name "Chas" :age 31}      ;; map  
#{1 2 3}                     ;; set  
  
{Math/PI "~-3.14"  
 [:composite "key"] 42  
 nil "nothing"}              ;; 又一个 map  
  
#{{{:first-name "chas" :last-name "emerick"}  
  {:first-name "brian" :last-name "carper"}  
  {:first-name "christophe" :last-name "grand"}}}  ;; 包含 map 的 set
```

这些类型的数据结构以及字面量大部分你可能已经很熟悉了，特别是这些字面量，跟 Ruby 和 Python 中的语法是很相似的。不过，Clojure 的数据结构有两个特色：

1. 数据结构首先是依据抽象来用的，而不是依据具体的实现细节来用。
2. 数据结构是不可改变的而且是持久的，两者对 Clojure 风格的函数式编程效率至关重要。

每种数据结构都有自己的特点和惯用模式，这些我们将去探索。而更为重要的是，深刻理解以上两点及其对 Clojure 本身以及 Clojure 数据结构的意义，领会你可以如何设计 Clojure 程序。

抽象优于实现

100 个函数操作 1 种数据结构比 10 个函数操作 10 种数据结构要好。^{注1}

—引自 Alan J. Perlis 为 *Structure and Interpretation of Computer Programs* 一书作的序，

<http://mitpress.mit.edu/sicp/toc/toc.html>

Clojure 的立场是：更好的是那 100 个函数操作 1 种抽象。从某些方面看，Clojure 的“集合类抽象优先于特定实现”足以跟 Python 和 Ruby 的多态操作、Java 的接口相媲美，但有一些微妙的地方使得 Clojure 对抽象的强调更彻底，能产生更大的威力。

在正式介绍之前我们先来看看对 vector 的一些操作：

```
(def v [1 2 3])
:= #'user/v
(conj v 4)
:= [1 2 3 4]
(conj v 5)
:= [1 2 3 4 5]
(seq v)
:= (1 2 3)
```

seq 总是产生一个集合类的顺序视图——称为一个序列，而 conj^{注2} 向给定的集合类增加新的值。目前来看这是相当普通的，不过同样的操作可用于映射：

```
(def m {:a 5 :b 6})
:= #'user/m
(conj m [:c 7])
:= {:a 5, :c 7, :b 6}
(seq m)
:= ([{:a 5} {:b 6}])
```

以及 set：

```
(def s #{1 2 3})
:= #'user/s
(conj s 10)
:= #{1 2 3 10}
(conj s 4)
:= #{1 2 3 4}
(seq s)
:= (1 2 3)
```

注1：更多 Perlis 的警言妙语请访问 <http://www.cs.yale.edu/quotes.html>。

注2：conj 源于英语“conjoin”，意思是连接、结合。

以及列表：

```
(def lst '(1 2 3))
:= #'user/lst
(conj lst 0)
:= (0 1 2 3)
(conj lst 0 -1)
:= (-1 0 1 2 3)
(seq lst)
:= (1 2 3)
```

85

显然，`seq` 和 `conj` 对于它们所操作的集合类类型来说是多态的。`conj` 把一个值附加在 `vector` 后面，或者附加在列表的前面，或者把一个键 - 值对加到一个 `map` 里。如果键已经存在，就替换对应的值；否则向这个 `map` 添加一个新的键 - 值对。`seq` 为 `vector`、`set`、列表提供了符合直觉的顺序视图，并把 `map` 也包括进来，`map` 的顺序视图是包含 `map` 里面所有键 - 值对（以 `[key value]` 形式表示）的一个列表。

Clojure 的精髓是小而易用的编程接口，在接口上再构建辅助函数。从使用者的角度看，“核心”函数和辅助函数是无法区别的，因而没什么会迫使开发者过早地开发新的辅助函数，或者为支持特定操作在接口与类型上做出错误的选择。

例如，`into` 是建立在 `seq` 和 `conj` 之上的，这意味着 `into` 能自动用于任何支持 `seq` 和 `conj` 的值：

```
(into v [4 5])
:= [1 2 3 4 5]
(into m [[:c 7] [:d 8]])
:= {:a 5, :c 7, :b 6, :d 8}
(into #[1 2] [2 3 4 5 3 3 2])
:= #[1 2 3 4 5]
(into [1] {:a 1 :b 2})          ❶
:= [1 [:a 1] [:b 2]]
```

- ❶ 因为 `map` 的 `seq` 是它的键 - 值对的序列，把这些键 - 值对 `conj` 到一个向量会保留这些键 - 值对的结构。

相比之下，Java 里的 `map` 甚至不是 `java.util` 框架里的集合类，Python 依赖具体的数据结构，每种数据结构都有自己的一套密封的操作。Ruby 好一点儿，它的列表和哈希表提供了一个 `.each` 方法，支持命令式迭代，但其他方面维持完全不同的数据结构，有各自的词汇。而 Clojure 则鼓励使用统一的抽象（序列、`protocol`、集合类接口，等等），让你的代码不会依赖特定类型集合的行为。

抽象的困境

Java 里的接口（特定类型的抽象），通常都比较小，但同时又需要提供很多帮助函数来使用这个接口。这些帮助函数经常也放在接口里（从而使得接口更难实现）。在抽象的基类里实现，只是核心方法没有实现。这让代码重用与继承绑在一起。因为 Java 只支持单继承，使用者需要开发一个新类实现两个那样的接口时，就不得不选择一个辅助方法重新实现，因为不能重用来自两个抽象类的已有实现。

因而出现了 Java 语言里主要的抽象工具存在的一个基本困境，让许多开发者进退两难，好比身处古希腊神话里西拉海妖与克里布迪斯大旋涡之间 (https://en.wikipedia.org/wiki/Scylla_and_Charybdis)，被迫在维护多重实现（且代码重用最少）与糟糕的编程接口间选择。

在第 6 章，我们将学习 Clojure 中的一些内容，让你能用于设计自己的抽象和类型，同时坚守“抽象第一”的 Clojure 原则。

构建小而广泛支持的抽象是 Clojure 的核心设计原则，这个原则可以说是久经考验的。同样的设计理念在 HTTP 中也可以看到，众所周知，HTTP 是目前使用最广泛，非常健壮，具有超强可扩展性以及可互操作性的协议。HTTP 的接口就是非常简单的一个小接口，就是这样的一个小接口，通常也只是部分实现。类似的，很多 Clojure 的数据结构都只支持某个接口的一小部分，比如对于 Java 的集合 API，Clojure 的集合只支持只读部分（以保证 Clojure 集合的不可修改特性）。即使一个抽象不是所有的接口都被实现了，这个抽象也是有用的。^{注3}

下面是 Clojure 集合中会实现的几个主要抽象：

- Collection
- Sequence
- Associative
- Indexed
- Stack
- Set
- Sorted

^{注3}：很多语言中都使用的很好的办法是，利用一个 Error 或者一个异常（`NotImplementedException`）来表明某些方法没有实现。

在下面的章节中，我们将对一些操作的语义进行探索，这些操作定义了各个抽象的 API。同时我们也会学习如何基于这些抽象来使用 Clojure 的数据结构，因此也就学会了所有实现这些抽象的具体实现的使用方法。

集合 : collection

87

Clojure 中的所有数据结构都实现了 Collection 抽象。对于 Collection，你可以使用如下这些核心的集合函数：

- 用 `conj` 来添加一个元素到集合。
- 用 `seq` 来获取集合的顺序视图。
- 用 `count` 来获取集合的元素个数。
- 用 `empty` 来获取一个跟所提供的集合类型一样的空集合。
- 用 `=` 来判断两个或者多个集合是否相等。^{注4}

这些函数基于所操作的具体集合表现出多态的行为，换种说法就是，这些函数具体的行为跟所操作的集合的特性有关。

我们前面已经粗略学习过 `seq`，不过会在后面 89 页的“序列”一节会更详细地学习，因为它是通向 Clojure 的另一个有用抽象接口的桥梁。

类似的，我们也学习过 `conj` 操作向量的时候会把元素添加到向量的最后面；操作 `map` 的时候会保证把一个键 - 值对添加到 `map` 里面；操作 `set` 的时候会保证把一个元素添加到集合里面去。`conj` 会保证对于所有的集合类型，它都会高效地把元素添加进去。这个使得 `conj` 的行为初看起来有点奇怪，`conj` 会把元素添加为列表的第一个元素（我们会在第 114 页详细介绍列表）：

```
(conj '(1 2 3) 4)
;= (4 1 2 3)
(into '(1 2 3) [:a :b :c])
;= (:c :b :a 1 2 3)
```

如果 `conj` 不把元素添加到最前面的话，比如添加到列表的最后面，那么就需要遍历这个列表，这对于一个很大的列表来说是很耗时间的操作。所以，`conj` 所保证的并不是它把元素添加到所有集合的最前面或者最后面（况且在 `map` 和 `set` 中元素没有顺序），它保证的是对于所有的集合它都能高效地插入。

`empty`。`empty` 这个函数的作用对于很多人来说不是那么熟悉。在很多情况下，必须要知道一个数据结构的具体类型，然后才能创建一个同类型的集合来进行操作。`empty` 使

注 4： 这里就不讨论相等性了，详细信息请参看 433 页“相等与等值”一节。

我们可以不必知道一个集合的具体类型就能创建出一个同类型的集合。比如，可以写一个函数来交换一个顺序集合中的两个元素的位置：

```
(defn swap-pairs
  [sequential]
  (into (empty sequential)
    (interleave
      (take-nth 2 (drop 1 sequential))
      (take-nth 2 sequential))))
```

(swap-pairs (apply list (range 10)))
;= (8 9 6 7 4 5 2 3 0 1)
(swap-pairs (apply vector (range 10)))
;= [1 0 3 2 5 4 7 6 9 8]

值得注意的是，`swap-pairs` 返回的类型跟我们提供的参数的类型是一样的：传一个列表进去，它返回的就是列表；传入一个向量返回的就是向量。这都要归功于 `into`（内部使用 `conj` 和 `seq`）的多态性，以及 `empty`，使得我们可以返回跟参数类型一样的集合。^{注5}

`empty` 不止对顺序集合作用，比如有一个函数让我们对于 `map` 中的每一个键~值对调用一个指定的函数，那么如果传入的参数是有序的或者是无序的呢？没问题，`empty` 可以返回跟传入参数一样类型的 `map`——在这种情况下，如果传入的 `map` 是有序的，那么返回的 `map` 就是有序的，如果传入的是无序的，那么返回的就是无序的：

```
(defn map-map
  [f m]
  (into (empty m)
    (for [[k v] m]      ①
      [k (f v)])))
```

① `for` 是一个列表推导的形式，跟 Python 中的列表推导是很类似的；`for` 产生的是一个惰性序列，惰性序列中的每一个键值对 `[k (f v)]`，`[k v]` 是对于 `m` 的 `map` 解构。

```
(map-map inc (hash-map :z 5 :c 6 :a 0))
;= {:z 6, :a 1, :c 7}
(map-map inc (sorted-map :z 5 :c 6 :a 0))
;= {:a 1, :c 7, :z 6}
```

有序地进去，有序地出来；无序地进去，无序地出来。也就是说，如果你想的话，可以让你的函数调用者决定函数的返回值类型。

注5：跟这个相比，Java 中的集合帮助函数也可以选择接受比较抽象的接口。比如 `java.util.List`，但是它们的返回值必须是一个具体类型或者返回一个同等抽象的接口。

`count`。`count` 的作用跟你期望的一样：返回集合中的元素个数：

```
(count [1 2 3])
:= 3
(count {:a 1 :b 2 :c 3})
:= 3
(count #{1 2 3})
:= 3
(count '(1 2 3))
:= 3
```

`count` 保证对于所有集合操作耗时都是高效的（序列除外，因为序列的长度可能是未知的，注6 我们后面会介绍）。

`count` 还可以操作一些非 Clojure 集合的 Java 类型，比如：`String`^{注6}、`map`、`set` 以及数组。

序列：Sequence

`Sequence` 接口定义了一个获取并且遍历各种集合的一个顺序视图的一种方法。这个集合可以是另一个集合，也可以是某个函数的一个返回值。在 Clojure 中，序列通常都被叫做“seq”——除了 `collection` 接口提供的一些函数，还提供了一些额外接口：

- `seq` 函数返回给传入参数的一个序列。
- `first`、`rest` 以及 `next` 提供了遍历序列的一个方法。
- `lazy-seq` 创建一个内容是一个表达式结果的惰性序列。

那些可以用 `seq` 来产生有效值的类型称为可序列的类型：

- 所有的 Clojure 集合类型。
- 所有的 Java 集合类型（也就是说，`java.util.*`）。
- 所有的 Java `map`。
- 所有的 `java.lang.CharSequence`，包括 `String`。
- 实现了 `java.lang.Iterable` 的任意类型。^{注7}
- 数组。
- `nil`（或者是 Java 方法返回的 `null`）。
- 任何实现了 `clojure.lang.Seqable` 接口的类型。

注 6： 不错，任何 `java.lang(CharSequence` 类型的实例，包括 `java.lang.StringBuilder`、`java.lang.StringBuffer` 和 `java.nio.CharBuffer`。

注 7： `seq` 可直接作用于 `Iterable` 对象。`iterator-seq` 和 `enumerator-seq` 则可以从 `java.lang.Iterator` 或者 `java.lang.Enumerator` 获得一个序列，这两个函数是独立于 `seq` 的，因为它们是有破坏性的：`Iterator` 或者 `Enumerator` 只能被遍历一次，如果你是从它们上面获取序列的话，这一点是必须遵守的。相比较而言，`seq` 是非破坏性的，因为它对于每次调用都能够提供一个新的 `Iterator` 对象。

要把这些类型一一演示就太繁杂了，我们这里演示一些吧：

```
(seq "Clojure")
:= (\C \l \o \j \u \r \e)
(seq {:a 5 :b 6})
:= ([{:a 5} [:b 6]])
(seq (java.util.ArrayList. (range 5)))
:= (0 1 2 3 4)
(seq (into-array ["Clojure" "Programming"]))
:= ("Clojure" "Programming") ❶
(seq [])
:= nil
(seq nil)
:= nil
```

90 ➤

- `seq` 对于任何 `nil` 或者任何类型的空集合都返回 `nil`。这对于很多场合都很方便，比如我们把它用在条件判断中，它跟 `(not (empty? some-collection))` 是等价的。

注意，许多跟序列打交道的函数都对它们的参数隐式调用 `seq`。比如，要想把 `String` 作为参数传给 `map` 或 `set`，我们不用先自己调用 `seq` 再传给它们：

```
(map str "Clojure")
:= ("C" "l" "o" "j" "u" "r" "e")
(set "Programming")
:= #{\a \g \i \m \n \o \P \r}
```



如果你是利用现有的序列函数来定义自己的操作序列的函数，那么不必对你的参数来调用 `seq`，但是如果你用 `lazy-seq` 的话，那么就要自己调用 `seq` 了。

有很多种方法遍历、处理序列，而且 Clojure 的标准库 `clojure.core` 也提供了很多函数来操作、创建序列。最基本的函数是 `first`、`rest` 和 `next`：

```
(first "Clojure")
:= \C
(rest "Clojure")
:= (\l \o \j \u \r \e)
(next "Clojure")
:= (\l \o \j \u \r \e)
```

`first` 和 `rest` 为什么有用应该很明显了。回忆一下，我们前面提到过可变参数，在 Clojure 中，可变参数就是通过 `rest` 来实现的。^{注8}

而 `rest` 和 `next` 的区别就不是那么明显了：从上面的例子可以看到，对于大多数值，它们的结果是一样的。`next` 和 `rest` 以不同的方式对待空序列和只包含单个值的序列，这

注8：详见38页的“可变参数”一节。

是它们唯一的不同：

示例3-1：rest与next的比较

```
(rest [1])
:= []
(next [1])
:= nil
(rest nil)
:= []
(next nil)
:= nil
```

91

可以看出，如果操作的结果是空的话，那么 `rest` 始终返回一个空序列，但是 `next` 则返回 `nil`，这个对于任何值 `x` 都是成立的：

```
(= (next x)
  (seq (rest x)))
```

这个区别看起来很小，但是它使得我们很容易实现惰性序列。前面稍微提到过 `lazy-seq`。



我们一般都把序列叫做“seq”（也许就是因为生成序列的函数名叫 `seq`），我们在本书的后面都会这么叫。

序列不是迭代器

当你看到这样的代码时：

```
(doseq [x (range 3)]
  (println x))
; 0
; 1
; 2
```

你可能会以为 `doseq` 是从某个迭代器里面把 `x` 拉出来，而这个迭代器是由 `range` 生成的。虽然 `x` 确实是被绑定到 `(range 3)` 的下一个值，但是 `seqs` 都是不可变的、持久的集合——跟 Clojure 中的其他集合一样：

```
(let [r (range 3)
      rst (rest r)]
  (prn (map str rst))
  (prn (map #(+ 100 %) r))
  (prn (conj r -1) (conj rst 42)))
; ("1" "2")
; (100 101 102)
; (-1 0 1 2) (42 1 2)
```

`rst` 这个 `seq` 可以跟它的“父” `seq : r` 一样工作，我们可以用序列和集合的一些函数来同时对它们进行操作。比如说，可以对一个 `seq` 进行 `map`，而不用担心 `map` 会改变这个 `seq` 的父 `seq` 或者子 `seq`。这些都是有状态的迭代器所具有的特征——你没办法简单地对一个 `seq` 做一个快照，然后继续操作这个 `seq` 而不影响这个快照，有状态的迭代器一旦被遍历的话，就不能再遍历了。

92 序列不是列表

第一眼看上去，序列跟列表很像：序列要么是空的，要么包含两个部分，一个作为头的值以及一个本身是序列的一个尾巴，同时列表本身就是自身的序列。^{注9}但是，它们在很多重要的方面都有不同：

- 要计算一个序列的长度是比较耗时的。
- 序列的内容可能是惰性的，而只有在真的要用到序列中值的时候才会去实例化。
- 一个生成序列的函数可能生成一个无限惰性序列，所以也就是不可数的。

作为对比，列表会保存它的长度，所以要获取列表长度的方法是非常高效的。而序列则不能保证这一点，因为这个序列可能是惰性序列，甚至可能是无限的序列。所以获取序列长度的唯一办法是去遍历这个序列。我们可以对比一下：

```
(let [s (range 1e6)]          ❶
    (time (count s)))
; "Elapsed time: 147.661 msecs"
:= 1000000
(let [s (apply list (range 1e6))]      ❷
    (time (count s)))
; "Elapsed time: 0.03 msecs"
:= 1000000
```

- ❶ `range` 返回一个包含一个范围的数字的惰性序列，而这些数字只有在真正需要的时候才会去计算出来。计算包含的元素个数是把一个惰性序列实例化的方法之一，因为你要知道这个序列一共有多少个元素，必须要把这个序列实例化。这里我们把惰性序列实例化的时间也包含在内了。
- ❷ 列表始终保存了它里面的元素个数，因此要获取列表的元素个数，`count` 函数会立即返回的。

创建序列

你可能已经意识到了，我们到这里为止都没有显式地去创建过任何序列。不过既然序列

注9： 虽然是语言的一个实现细节，不过了解一下也无妨：目前，`(identical? some-list (seq some-list))` 始终为 `true`。

只是其他集合的一个顺序视图，那么这其实不奇怪；一般来说，一个序列是从一个集合生成而来，要么是通过 seq 函数直接创建，要么通过其他函数（比如 map）间接调用 seq。但其实有两种方法来直接创建一个 seq：cons 和 list*。^{注 10}

cons 接受两个参数，第一个参数是一个值作为新 seq 的头，第二个参数是一个集合，作为新 seq 的尾巴：◀ 93

```
(cons 0 (range 1 5))  
:= (0 1 2 3 4)
```

你可以这么想：cons 始终把第一个参数值加入到第二个集合参数的头上去，而不管第二个参数具体的集合类型，所以它跟 conj 的作用是不同的：

```
(cons :a [:b :c :d])  
:= (:a :b :c :d)
```

list*^{注 11} 只是一个帮助函数，来简化创建指定多个值来创建 seq 的需求，所以下面两个表达式是等价的：

```
(cons 0 (cons 1 (cons 2 (cons 3 (range 4 10)))))  
:= (0 1 2 3 4 5 6 7 8 9)  
(list* 0 1 2 3 (range 4 10))  
:= (0 1 2 3 4 5 6 7 8 9)
```

cons 和 list* 在编写宏的时候用得最多，要么是用来把一个值添加到列表或者序列的头上，这时候序列和列表是等价的；要么用来构建惰性序列的下一步（我们在下一节会进行介绍）。

惰性序列

在 Clojure 中，一个集合的具体内容可以是惰性生成的，这种情况下集合的元素是一个函数调用的结果，而我们只在真正需要集合里面元素内容的时候才去调用函数去计算一次，而且只算一次。访问一个惰性序列的过程被称为实例化；当一个惰性序列中的元素都被计算出来了，我们说这个序列被完全实例化了。

可以通过宏 lazy-seq 来创建一个惰性序列，这个宏接受任意返回值是一个可序列的值的表达式。下面是一个很傻的例子：

注 10：如果你之前有其他 Lisp 方言的经验的话，那么要注意，Clojure 中的 cons 跟其他方言中的 cons 没有任何关系。同样，Clojure 的列表也不是由一个个 cons 单元组成的。

注 11：有点容易弄混淆的是，list* 不返回一个列表，也就是说，(list? (list* 0 (range 1 5))) 会返回 false——除非你只提供一个值。这在一般情况下是没有问题的，除非你用那种跟具体数据结构相关的函数、谓词来对它进行操作，比如 list?。还是我们之前说的那样，面向抽象编程总是比面向具体类型编程要好。所以任何你想用 list? 的地方，换成 seq? 或者 sequential? 就可以了。

```
(lazy-seq [1 2 3])
;= (1 2 3)
```

这个例子实在是很无趣，因为这个惰性序列是从一个已有的数组里面去“实例化”它的元素。一个有趣点的例子是生成一个包含随机数的序列：

示例3-2：实现一个惰性序列

```
(defn random-ints
  "Returns a lazy seq of random integers in the range [0,limit)."
  [limit]
  (lazy-seq
    (cons (rand-int limit)           ①
          (random-ints limit))))      ②

(take 10 (random-ints 50))
;= (32 37 8 2 22 41 19 27 34 27)
```

- ① 我们返回一个惰性序列，这个惰性序列的头是函数调用 (`rand-int limit`) 结果。
- ② 而这个序列的尾巴本身也是一个惰性序列，这个惰性序列则是对相同函数的 `random-ints` 的递归调用。

生成的结果可能不太特别，但是这个结果中的数字都是惰性地通过调用 `random-ints` 产生的——这里只是把前 10 个元素打出来，因为打印一个序列会迫使整个序列被实例化。^{注12} 让我们稍微修改一下 `random-ints` 的实现，让它在每次一个元素被具体化的时候打印一条信息出来：

```
(defn random-ints
  [limit]
  (lazy-seq
    (println "realizing random number") ①
    (cons (rand-int limit)
          (random-ints limit))))           ②

(def rands (take 10 (random-ints 50)))           ③
;= #'user/rands
(first rands)                                     ④
; realizing random number
;= 39
(nth rands 3)
; realizing random number
; realizing random number
; realizing random number
;= 44
```

注 12：这意味着我们在对那些无限序列或者不是无限但是非常大的序列求值的时候要非常小心，比如使用 (`take 100 infinite-seq`) 来只取序列的前 100 个元素。或者也可以利用 `set!` 来设置 `*print-length*` 参数到一个合理的值（比如 100）来限制打印集合的元素个数。

```

(count rands)
; realizing random number
;= 10
(count rands)
;= 10

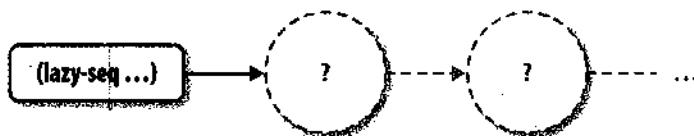
```

❶

❷

- ① 每当 `lazy-seq` 被求值（每次查看 `cons` 产生的序列的头的时候），我们打印一条信息。
- ② 我们定义一个惰性序列，所以它的内容不会被打印出来，但是表达式已经被完全求值了，这个序列是完全惰性的，还没有去产生任何值。 95
- ③ 通过 `first` 函数来返回序列的第一个元素，这会迫使我们调用 `lazy-seq` 时传入的表达式被调用。我们得到了一个随机数，并且对应地打印出了一条信息。
- ④ 如果我们获取惰性序列中间位置的一个元素，那么在这个元素前面的所有元素都会被实例化，这里我们看到，`count` 迫使整个序列被实例化以使得它能够算出序列的大小。
- ⑤ 再去获取一次序列的大小（或者类似地使用 `nth` 访问序列中的一个值）就不需要再去重新计算每个值了；一旦惰性序列中的一个元素被实例化，那么这个元素的值会被保存下来。

当我们第一次去定义这个序列的时候，这个序列的具体元素并不存在：



一旦我们尝试去访问它的第一个值，第一个值就被实例化并且保存下来，以使得后面再需要这个值的时候不用再计算一遍：



它的头是一个具体的值了，但是它的尾巴还是由我们提供给 `lazy-seq` 的那个表达式所定义的，而这个尾巴的值直到我们去访问它的值才会去计算。这一点是 `cons` 和 `list*` 的一个关键优势：它们不会强制对传入的序列（可能是惰性的）求值。这使得这些函数成为构建惰性序列非常有用的帮助函数，我们通常把一个或者多个具体值以及一个惰性序列传给 `cons` 或者 `list*` 来创建一个更大的惰性序列。

`random-ints` 实际上实现得很差，而且过于复杂。同样的功能用 Clojure 的几个标准库函数就可以轻松实现：`random-ints` 已经使用的 `rand-int` 以及 `repeatedly`，`repeatedly` 可以通过调用给定的函数参数来创建一个惰性序列：

96 (repeatedly 10 (partial rand-int 50))
:= (47 19 26 14 18 37 44 13 41 38)



值得强调的一点是，你传给 `lazy-seq` 的表达式可以做任何事情，不只是可以返回一些随机数或者其他一些简单数据。任何纯函数实现的计算都可以作为参数传给它。

注意，只接受一个参数的 `repeatedly` 会返回一个包含无限随机数的惰性序列。在 Clojure 中处理和使用无限长度的序列是很平常的事情，Clojure 的标准库以及社区的一些库中有很多透明处理（可能是无限长度的）惰性序列的函数。比较方便的是，所有这些核心的序列处理函数都返回惰性序列：`map`、`for`、`filter`、`take` 和 `drop`^{注13}。你可以安全地嵌套使用这些函数而不用考虑操作的序列到底是不是惰性的。有了这些工具我们可以把许多问题归结为对一个序列的值的处理，而不管这些值到底是具体的值，还是从消息队列惰性加载进来的数据，或者是一个并行的搜索，^{注14} 或者是利用 `file-seq`、`line-seq` 和 `xml-seq` 从各种数据源进行消费和转化的数据。

如果对于序列中每个元素的实例化都需要进行一些 IO 操作或者计算量很大，那么你就要非常小心地处理了，这个也是 `next` 和 `rest` 很大的一个区别所在。回忆一下我们在示例 3-1 中提到的：`next` 始终返回 `nil` 而不是空的序列。它之所以能够做到这一点是因为它始终会去强制实例化序列尾巴的第一个元素：

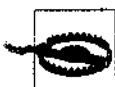
```
(def x (next (random-ints 50)))  
; realizing random number  
; realizing random number
```

相比较而言，`rest` 则始终简单地返回序列的尾巴而已，从而也就避免了去实例化序列的尾巴的第一个元素，也就最大程度的“惰性”了：

注 13： 以及由它们引申出来的一些函数，比如 `take-nth`、`take-while`、`drop-while`、`remove` 等。

注 14： 利用 `pmap` 和惰性序列，我们可以在一个多核的机器上只消耗 MB 级别的内存就可以处理 GB 级别的数据，很帅吧？有关 `pmap` 的更详细的介绍请看 166 页的“简单的并行化”一节。

```
(def x (rest (random-ints 50)))
; realizing random number
```



顺序解构始终使用 `next` 而不是 `rest`。所以如果你在解构一个惰性序列的话，那么始终会实例化它的尾巴的头元素：

```
(let [[x & rest] (random-ints 50)])
; realizing random number
; realizing random number
;= nil
```

97

而在另外一些情况下，你需要完全实例化一个惰性序列。如果你想要保持这个序列的所有元素，那么应该使用 `doall`,^{注15} 如果并不需要惰性序列的具体内容（比如你想要的是惰性序列实例化的时候所产生的副作用），那么应该使用 `dorun`：

```
(dorun (take 5 (random-ints 50)))
; realizing random number
;= nil
```

比如有一个惰性序列的文件（通过 `file-seq` 获得的），并且你希望对这些文件进行一些操作，那么利用 `map` 来对这个惰性序列进行操作是非常方便的，利用 `map` 进行的操作其实没有真正发生，只是保存在惰性序列里面了，然后调用 `dorun` 就可以迫使这些操作实际发生，而且我们对于 `dorun` 的返回值也确实是不关心的。

通常，对序列操作的函数都会在它的文档中说明它的返回值是不是惰性序列或者会不会强制实例化这个惰性序列：

```
(doc iterate)
; -----
; clojure.core/iterate
; ([f x])
;   Returns a lazy sequence of x, (f x), (f (f x)) etc.
;   f must be free of side-effects

(doc reverse)
; -----
; clojure.core/reverse
```

注 15：跟我们前面看到的一样，`count`（以及其他集合函数）的表现是一样的，不过这只是语言的一个实现细节。理论上来说，虽然很少会发生，但是我们要实现一个不用实例化它所有元素就能知道它大小的惰性序列是可能的。

```
; ([coll])
;   Returns a seq of the items in coll in reverse order. Not lazy.
```

显然，`iterate`是惰性的，而`reverse`不是。`iterate`注释上的警告“传入的函数必须没有副作用”需要解释一下。

96 定义惰性序列的代码应该尽量不要有副作用。我们在 76 页“纯函数”一节学习了纯函数的许多好处，这些特征以及副作用代码的缺点在惰性序列中反映得更加突出。因为惰性序列中的元素在它们定义的时候没有实例化，在它们被访问的时候才实例化的，所以如果代码有副作用的话，我们很难跟踪到底是在哪里以及什么时候产生的副作用——有时候甚至不会有副作用——意思就是，你根本不能确定！比如我们在惰性序列中对日志级别进行更改，可能一开始日志级别是`INFO`，在某个时候你实例化了这个惰性序列，日志级别突然变成了`ERROR`，会让人摸不着头脑的。

更糟糕的是，对于惰性序列的实例化有时候会做批量处理，比如一次实例化 5 个元素以优化性能，^{注16}这样如果你的函数是有副作用的话，你就更不知道副作用会在什么时候发生了。

所以你不应该依赖惰性序列元素的实例化来控制程序的执行流程。`Clojure` 的惰性特性跟其他到处使用惰性特性的语言的设计目的是不一样的。在`Clojure` 中只有序列有惰性特性，而其他数据结构则都是立即求值的。惰性序列使得`Clojure` 可以透明地处理那些无法放进内存的大数据、使得对于大数据和小数据可以使用统一的声明、可以用管道的方式来表达处理算法，在这种情况下序列可以看做计算的中间载体，而不是集合。

你会经常在`Clojure` 代码中看到这样的用法：给定一个或者多个数据源，从这个数据源抽出一个序列，处理这个序列，然后再返回一个更加合适的数据结构。在下面这段非常简单的代码中也能看到这种用法：

```
(apply str (remove (set "aeiouy")
                    "vowels are useless! or maybe not..."))
:= "vwls r slss! r mb nt..."
```

这里我们有一个字符串形式的数据源：“`vowels are useless! or maybe not...`”，它会被隐式地转换成一个字符的序列，所有的元音字母被去掉，然后再组装成一个字符串。

头保持

一个经常被`Clojure` 初学者忽视的问题是：一旦惰性序列的一个元素被实例化之后，只

注 16：有些数据结构（比如`vector`）生成“块状”的序列，而且很多序列函数（比如`map` 和 `filter`）也是成块地处理元素（比如对于由`vector` 产生的序列，每个块包含 32 个元素）。

只要你保持了对这个序列的引用，那么这个元素就会一直保持着了。^{注17}这意味着，只要你保持了对序列的一个引用，那么序列中的元素就不能被垃圾回收。这种问题称为“头保持”，会给虚拟机压力从而影响性能，如果序列中的元素占用内存过大的话甚至可能导致内存用尽的错误。

`split-with`是一个这样的函数，你给它一个谓词函数，一个可序列的值，它返回两个惰性序列，第一个是满足这个谓词的惰性序列，第二个是不满足这个谓词的惰性序列：

```
(split-with neg? (range -5 5))
;= [(-5 -4 -3 -2 -1) (0 1 2 3 4)]
```

想想下面这个例子，如果我们知道 `split-with` 返回的第一个惰性序列会很小，但是第二个惰性序列会很大。那么如果保持第一个序列的引用的话，那么整个序列（由 `range` 调用创建的这个序列）里面的元素就都会被保持住——即使你是以惰性的方式来处理第二个序列的。如果要处理的序列很大的话，那么这个头保持将会导致内存用尽的问题：

```
(let [[t d] (split-with #(< % 12) (range 1e8))]
  [(count d) (count t)])
;= #<OutOfMemoryError java.lang.OutOfMemoryError: Java heap space>
```

把上面的 `count` 调用次序调换一下就能解决这个问题了：

```
(let [[t d] (split-with #(< % 12) (range 1e8))]
  [(count t) (count d)])
;= [12 99999988]
```

因为最后一个对 `t` 的引用出现在对 `d` 的处理之前，没有对 `range` 调用所产生的序列的头的保持，所以也就没有内存问题了。

向 `map` 或者 `set` 中插入元素、`=` 函数以及 `count` 函数都是“头保持”问题的诱因，^{注18} 因为它们都会强制对惰性序列进行完全实例化。

Associative

关系型数据结构 *Associative* 接口所抽象的是把一个 `key` 和一个 `value` 关联起来的数据结构。这个接口包括 4 个操作：

- `assoc`, 它向集合中添加一个新的 `key` 到 `value` 的映射。
- `dissoc`, 从集合中移除一个从指定 `key` 到 `value` 的映射。
- `get`, 从集合中找出指定 `key` 的 `value`。

注17：“这个跟 Python 中的 generators 和 Ruby 里面的 Enumerator 形成鲜明的对比，虽然它们也能惰性地生成一个可能是无限的序列，但是如果你想保留之前已经实例化的值，它们并不提供一致的方法给你使用。”

注18：“或者，过早实例化，但是结果是一样的，同样会抛出内存不足的异常。”

- `contains?`, 是一个谓词, 当这个集合包含指定的 key 的时候它返回 true, 否则返回 false。

最正宗的关系型数据结构是 map, 它同时也是 Clojure 提供的最有用的数据结构, 你很快就会喜欢上这个 map 的。

100> 前面已经看到过, 当我们用集合的核心函数 `conj` 和 `seq` 来处理 map 的时候, map 被看成包含一个个键值对的集合, 但是关系型的函数处理起 map 来显得更加自然。

```
(def m {:a 1, :b 2, :c 3})
;= #'user/m
(get m :b)
;= 2
(get m :d)
;= nil
(get m :d "not-found")
;= "not-found"
(assoc m :d 4)
;= {:a 1, :b 2, :c 3, :d 4}
(dissoc m :b)
;= {:a 1, :c 3}
```

你还可以使用 `assoc` 和 `dissoc` 来很方便地一次添加 / 去除多个键值对 :

```
(assoc m
:x 4
:y 5
:z 6)
;= {:z 6, :y 5, :x 4, :a 1, :c 3, :b 2}
(dissoc m :a :c)
;= {:b 2}
```

虽然这些函数通常是用来操作 map 的, 但是 `get` 和 `assoc` 也支持对 vector 进行操作。虽然一开始看起来有点怪, 但是 map 和 vector 都可以看做是关系型集合, 只不过对 vector 来说, 这个“关系”中的 key 是数组下标 :

```
(def v [1 2 3])
;= #'user/v
(get v 1)
;= 2
①
(get v 10)
;= nil
(get v 10 "not-found")
;= "not-found"
(assoc v
1 4
0 -12)
```

```
2 :p)
:= [-12 4 :p]
```

❶ v 把下标 1 和数字 2 “联系” 在一起。

❷ 可以使用 assoc 来“更新”数组中某个下标对应的值；同样，更新接口是关系型数据结构的接口，但是对于不同的数据结构有着不同的行为。可以看出，这里用 assoc 来操作一个 vector 在调用结构上跟前面用 assoc 来操作一个 map 是一样的。

我们可以用 assoc 来实现和 conj 一样的效果：把一个值添加到 vector 的最后。◀ 101

的麻烦之处在于，你需要显式地告诉 vector 新的下标是多少：

```
(assoc v 3 10)
:= [1 2 3 10]
```

最后，get 还能够操作 set，如果一个 key 在 set 中存在的话，那么会返回它，否则返回 nil：

```
(get #{1 2 3} 2)
:= 2
(get #{1 2 3} 4)
:= nil
(get #{1 2 3} 4 "not-found")
:= "not-found"
```

虽然 set 本身没有键值对的语义，不过当我们用 get 来操作一个 set 的时候，它的返回值意味着 set 是它里面值到值本身的映射。这看起来可能有点奇怪，但是它使得 set 可以满足 get 的语义，同时也跟条件判断里面的典型用法保持一致：

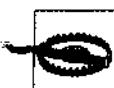
```
(when (get #{1 2 3} 2)
  (println "it contains '2'!"))
; it contains '2'!
```

 Clojure 中的 map 和 set 其实本身就直接支持 get 语义，它比使用 get 函数更方便。我们在 111 页的“集合本身也是函数”一节会介绍如何简洁地从集合中获取数据而不用 get 函数。

contains? contains? 是一个谓词，它的作用是检测集合中是否包含指定的 key。

```
(contains? [1 2 3] 0)
:= true
(contains? {:a 5 :b 6} :b)
:= true
(contains? {:a 5 :b 6} 42)
:= false
```

```
(contains? #{1 2 3} 1)
:= true
```



Clojure 初学者一个常见的错误是：他们觉得 `contains?` 始终会查询一个集合是否包含某个值，那么如果想确定一个 `vector` 中是否包含一个数值，应该也是可以的，这个误解会引起一些很容易误解的结果：

```
(contains? [1 2 3] 3)
:= false
(contains? [1 2 3] 2)
:= true
(contains? [1 2 3] 0)
:= true
```

当然，上面的结果都是对的，因为 `contains?` 检查的是集合中是否包含某个 key——在这种情况下，这个 key 是 `vector` 的下标 3、2 以及 0。如果要检查一个集合中是否包含某个值，一般使用 113 页“集合、key 以及高阶函数”一节中介绍的 `some` 函数。

`get` 和 `contains?` 非常通用：它们可以很高效地操作 `vector`、`map`、`set`、Java 里面的 `map` 字符串以及 Java 数组：

```
(get "Clojure" 3)
:= \j
(contains? (java.util.HashMap.) "not-there")
:= false
102 (get (into-array [1 2 3]) 0)
:= 1
```

小心 nil

如果集合里面不包含某个 key，并且你调用的时候也没提供默认值，那么 `get` 会返回 `nil`。但是这个 key 对应的 value 也可能正好就是 `nil`，这是完全没有问题的：

```
(get {:ethel nil} :lucy)
:= nil
(get {:ethel nil} :ethel)
:= nil
```

那我们到底怎么知道这个 `nil` 值指的是集合里面不包含这个 key 还是说这个 key 对应的值就是 `nil` 呢？

我们可以使用 `contains?`，但是这样就需要对 `map` 进行两次查询了：先用 `contains?` 来检测 `map` 里面到底包不包含某个 key，然后用其他查询函数来获取这个值。你也许会想，我们可以直接使用 `get`，但是需要给它提供一个“比较特殊”的默认值，当返回的是这个“比

较特殊”的默认值的时候，就认为这个 map 不包含这个 key，但问题是这个“比较特殊”的 key 很容易被当做一个正常的值被加入 map，然后也就有了和 nil 一样的问题。

我们应该使用 find。find 跟 get 很类似，只是它返回的不是 key 所对应的 value，而是返回一个键值对——如果 map 里面包含指定的 key 的话，或者返回 nil 如果不包含这个 key 的话。

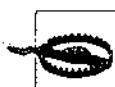
```
(find {:ethel nil} :lucy)
;= nil
(find {:ethel nil} :ethel)
;= [:ethel nil]
```

而且 find 也可以很容易跟解构形式如 if-let（或者 when-let）一起使用：

```
(if-let [e (find {:a 5 :b 6} :a)]
  (format "found %s => %s" (key e) (val e))
  "not found")
;= "found :a => 5"
(if-let [[k v] (find {:a 5 :b 6} :a)]
  (format "found %s => %s" k v)
  "not found")
;= "found :a => 5"
```

103

当然，如果你只是想检查集合中是不是包含指定的 key，那么直接使用 contains? 就好了。



处理 false 也要小心点

当被使用在条件判断中的时候，false 在关系型集合中和 nil 有几乎一样的问题，解决方法跟 nil 一样，大家要小心。

索引集合：Indexed

到目前为止，我们已经讨论过 vector 了，但是一直回避讨论怎么直接获取 vector 中的第 n 个值，或者直接修改 vector 里面的第 n 个值，原因是：下标是新的指针。^{注 19}

老实说，字符串、数组或者其他顺序集合里面的下标对于实现一个算法来说很少是必需的。^{注 20} 在很多情况下，如果你的代码涉及数组下标了，那么就意味着你的代码过于复杂了：你要进行下标计算，要做数组越界检查等。除非是一些很特殊的情况，一般来说，如果

注 19： 或者这么说，IndexOutOfBoundsException 相当于 Clojure 中的 core dump。

注 20： 所有的规则都是有例外的。如果你在编写一些例如 Boyer-Moore 字符串查找之类的亚线性算法，那么确实需要下标。在用 Clojure 去实现一个指定的算法时，我们要想想为了实现这个我们真的需要下标吗？很多文章、教材在例子里面到处使用数组和下标来表示算法，只是因为它们对于所有语言都比较通用，大多数语言里面都有数组、下标的概念。但是这并不意味着 Clojure 里面没有更干净利落、高效的方式来完成同样的事情。

你的代码依赖下标了，那么说明你的代码有点问题了。

虽然这么说，如果有些情况实在要用下标，那么 `Indexed` 接口就提供了操作下标的各种函数。这个接口里面只有一个函数：`nth`，它跟前面说的 `get` 类似。不同之处在于，它们对于越界的下标的处理：`nth` 会抛出异常，而 `get` 则是返回 `nil`：

示例3-3：比较`nth` 和 `get` 在 `vector` 上的使用

```
(nth [:a :b :c] 2)
;= :c
(get [:a :b :c] 2)
;= :c
(nth [:a :b :c] 3)
;= java.lang.IndexOutOfBoundsException
(get [:a :b :c] 3)
;= nil
104 (nth [:a :b :c] -1)
;= java.lang.IndexOutOfBoundsException
(get [:a :b :c] -1)
;= nil
```

除了这一点，其他都是一样的：

```
(nth [:a :b :c] -1 :not-found)
;= :not-found
(get [:a :b :c] -1 :not-found)
;= :not-found
```

`nth` 和 `get` 的意义是很不一样的。首先，`nth` 只能接受数字作为查询的 key（下标），它可以作用于很多有“下标”概念的值：`vector`、列表、序列、Java 数组、Java 列表、字符串、正则表达式的匹配数组等，而 `get` 则更通用：它可以操作任何关系型的值，在前面已经看到，处理 `vector` 的时候，它可以把下标当做 key。

`nth` 和 `get` 的另一个重大区别是，`get` 对错误更容忍。我们在前面已经看到对于一个越界的下标 `get` 是返回 `nil` 而不是抛出一个异常。`get` 甚至可以更容忍：当你传给它一个不支持的参数类型，它也是返回 `nil` 而不是抛出异常，`nth` 则直接抛出异常。

```
(get 42 0)
;= nil
(nth 42 0)
;= java.lang.UnsupportedOperationException: nth not supported on this type: Long
```



Clojure 的 vectors 本身就直接支持 `nth` 的语义，比调用 `nth` 函数更方便。111 页的“集合本身也是函数”一节会详细介绍如果在不使用 `nth` 的情况下，如何简洁地访问集合里面的元素。

栈 : stack

栈是支持经典的后进先出（LIFO）语义的集合。也就是说，你最后加进去的元素会被最先取出来。Clojure 并没有一个独立的“栈”集合，但是它通过三个操作来支持这种语义：

- 用 `conj` 来把一个值添加到栈中去（重用集合的通用操作）。
- 用 `pop` 来获取栈顶的值，并且移除这个值。
- 用 `peek` 来获取栈顶的值，但是不去除这个值。

列表和 `vector` 都可以当做栈来用（见示例 3-4 和 3-5），在两种集合中，栈顶都是 `conj` 可以高效添加元素的另外一端。

示例3-4：把列表当做一个栈

<105

```
(conj '() 1)
;= (1)
(conj '(2 1) 3)
;= (3 2 1)
(peek '(3 2 1))
;= 3
(pop '(3 2 1))
;= (2 1)
(pop '(1))
;= ()
```

示例3-5：把vector当做一个栈

```
(conj [] 1)
;= [1]
(conj [1 2] 3)
;= [1 2 3]
(peek [1 2 3])
;= 3
(pop [1 2 3])
;= [1 2]
(pop [1])
;= []
```

对一个空栈调用 `pop` 会出错。

set

我们前面在讨论关系型抽象的时候已经讨论过 `set`，当时我们把 `set` 看做一种退化的 `map`，里面的元素自己映射到自己：

```
(get #{1 2 3} 2)
;= 2
(get #{1 2 3} 4)
;= nil
```

```
(get #{1 2 3} 4 "not-found")
;= "not-found"
```

这里需要补充的一点是，set 接口需要 disj 来从 set 里面移除一个元素：

```
(disj #{1 2 3} 3 1)
;= #{2}
```

虽然 set 接口本身是很轻量的，我们可以重用最通用的集合函数以及关系型数据结构的函数来对集合进行绝大多数的操作，但是我还是建议你熟悉一下 Clojure 的标准库 `clojure.set`。`clojure.set` 提供了一些函数使我们可以很方便地实现一些对 set 进行操作的高阶函数，比如 `subset?`、`superset?`、`union`、`intersection`、`project` 等。

106 有序集合：sorted

实现 sorted 抽象的集合保证集合内的值被以特定的顺序保存，这个顺序可以通过一个谓词或者一个特定的 comparator 接口来定义。这使得你可以高效地、正序（或者反序）获取集合或者集合的一部分。这个接口包含以下函数：

- rseq 函数可以在常量时间内反序地返回一个集合的元素。
- subseq 可返回一个集合的某一个区间的元素的序列。
- rsubseq 跟 subseq 类似，但是返回的元素是反序的。

只有 map 和 set 实现了 sorted 接口。没有字面量来表示 sorted 的集合，要创建 sorted 集合可以用 `sorted-map` 和 `sorted-set` 来创建有序的 map 和 set。如果要提供你自己的谓词或者比较器来定义排序规则的话，那么使用 `:sorted-map-by` 和 `:sorted-set-by`。

给定一个有序集合，你可以使用 sorted 抽象定义的那几个函数来操作它：

```
(def sm (sorted-map :z 5 :x 9 :y 0 :b 2 :a 3 :c 4))
;= #'user/sm
sm
;= {:a 3, :b 2, :c 4, :x 9, :y 0, :z 5}
(rseq sm)                                     ❶
;= ([:z 5] [:y 0] [:x 9] [:c 4] [:b 2] [:a 3])
(subseq sm <= :c)                           ❷
;= ([:a 3] [:b 2] [:c 4])
(subseq sm > :b <= :y)                      ❸
;= ([:c 4] [:x 9] [:y 0])
(rsubseq sm > :b <= :y)                     ❹
;= ([:y 0] [:x 9] [:c 4])
```

❶ rseq 会在常量时间内返回 sm 的一个反向序列。

❷ 这里我们查询 sm 里面排在 :c 前面（包括 :c 本身）的所有元素。

❸ 这个查询返回集合里面所有排在 :b 后面并且排在 :y 前面(包括 :y 本身)的所有元素。

❹ rsubseq 跟 subseq 的效果类似，只是返回的结果是反序的。

因为 sm 本身是有序的，为获得相同的结果，在 sm 上调用这些函数比在普通 seq 上调用这些函数要快很多(比如，使用 filter、take-while)。比如，rseq 保证能在常量时间内返回，而 reverse 则需要线性的时间。^{注 21}

compare 函数定义默认排序：正序，它支持所有的 Clojure 标量及顺序集合，它会按照字典排序法来在每一层对元素进行排序：^{注 22} ◀107

```
(compare 2 2)
;= 0
(compare "ab" "abc")
;= -1
(compare ["a" "b" "c"] ["a" "b"])
;= 1
(compare ["a" 2] ["a" 2 0])
;= -1
```

实际上，compare 不止支持字符串、数字以及顺序集合：它支持任何实现了 java.lang.Comparable 接口的值，包括布尔值、关键字、符号以及所有实现了这个接口的 Java 类。compare 是很强大的函数，但是也只是默认的比较器。(你可以定义你自己的。)

使用比较器和谓词来定义排序规则

比较器是一个接受两个参数的函数，如果第一个参数大于第二个参数，那么返回一个正数；如果第一个参数小于第二个参数则返回负数，如果相等则返回 0。

Clojure 里面所有的函数都实现了 java.util.Comparator 接口，所以它们都是潜在的比较器——不过很明显，并不是所有的函数都是设计来作为比较器的。你并不需要一定实现 Comparator 接口才能实现一个比较器——任何一个两参数的谓词都可以。

不用实现一个特定接口就能定义一个比较器也意味着我们可以很容易地定义多层排序：先按照一个规则排序，在这个基础上再按照另外一个规则排序。只要定义一个高阶函数就可以了。比较函数可以直接传给有序集合的工厂函数，如 sort 和 sort-by：^{注 23}

```
(sort < (repeatedly 10 #(rand-int 100)))
;= (12 16 22 23 41 42 61 63 83 87)
```

注 21： 其实我们这里撒了个小谎，从技术上来说，rseq 也是另外一个小抽象 reversible 的一部分——实现它的数据结构都是可反序的。我们这里要提这个是因为除了有序集合，只有 vector 是可以反序的(所以 rseq 也可以使用在 vector 上)。

注 22： 意思就是说，一个包含 vector 的 vector 可以被可靠地排序。

注 23： 或者任何期望一个 java.util.Comparator 作为参数的 Java API。

```
(sort-by first > (map-indexed vector "Clojure"))
;= ([6 \e] [5 \r] [4 \u] [3 \j] [2 \o] [1 \l] [0 \C])
```

Clojure 是如何把一个谓词变成一个比较器的？

初看可能很奇怪的是，像 `<` 这样的谓词怎么能作为比较器呢？谓词只返回 `true/false`，而比较器需要返回三种结果（`-1`、`0` 或 `1`）。

其实方法很简单：比如我们要比较 `a` 和 `b`，先按顺序来调用谓词：`(predicate-fn a b)`，如果返回 `true`，那么我们直接返回 `-1`，否则以反序调用这个谓词 `(predicate-fn b a)`，如果返回值是 `true` 的话，那么直接返回 `1`，否则返回 `0`。

利用这个原理，`comparator` 函数可以显式地把一个两参谓词转换成一个比较器函数：

```
((comparator <) 1 4)
;= -1
((comparator <) 4 1)
;= 1
((comparator <) 4 4)
;= 0
```

但是我们很少会这么干，因为 Clojure 里面接受比较器作为参数的函数都默认做这个转换，而且两参函数已经实现了 `java.util.Comparator` 接口。

所以，`sorted-map` 和 `sorted-set` 通过 `compare` 来定义默认规则进行排序的 `map` 和 `set`。而 `sorted-map-by` 和 `sorted-set-by` 接受一个比较器（任何两参的谓词函数也可以）来定义排序规则。而除了 `compare` 之外，你能传给有序集合的最简单的比较器可能就是 `(comp - compare)` 了，正好定义 `compare` 的反序：

```
(sorted-map-by compare :z 5 :x 9 :y 0 :b 2 :a 3 :c 4)
;= {:a 3, :b 2, :c 4, :x 9, :y 0, :z 5}
(sorted-map-by (comp - compare) :z 5 :x 9 :y 0 :b 2 :a 3 :c 4)
;= {:z 5, :y 0, :x 9, :c 4, :b 2, :a 3}
```

需要注意的是，排序规则在有序 `map` 和有序 `set` 里面同时也定义了两个元素是否相等。这个特点有时候会产生逻辑正确但是让人很吃惊的结果。比如，我们有一个函数来计算参数是 10 的几次方：

```
(defn magnitude
  [x]
  (-> x Math/log10 Math/floor))
:= #'user/magnitude
(magnitude 100)
:= 2.0
(magnitude 100000)
:= 5.0
```

看起来很简单，现在我们利用 `magnitude` 这个函数来创建一个谓词函数：

```
(defn compare-magnitude
  [a b]
  (- (magnitude a) (magnitude b)))

((comparator compare-magnitude) 10 10000)
:= -1
((comparator compare-magnitude) 100 10)
:= 1
((comparator compare-magnitude) 10 75)
:= 0
```

109

当我们把这个谓词函数用在一个有序集合里面的时候，有趣的事情就发生了：

```
(sorted-set-by compare-magnitude 10 1000 500)      ①
:= #{10 500 1000}
(conj *1 600)                                     ②
:= #{10 500 1000}
(disj *1 750)                                     ③
:= #{10 1000}
(contains? *1 1239)                                ④
:= true
```

- ① 因为 10、1000 和 500 分别是 10 的不同次方，所以它们作为不同元素被保留在集合里面了。
- ② 当我们尝试把 600 加入集合的时候，集合没有发生变化；这是因为 500 和 600 都是 10 的二次方，因此对于这个比较器来说，500 和 600 是相等的，既然 500 已经在集合里面，那么当然就不会把 600 也加进去了。
- ③ 因为对比较器来说，750 和 500 是等价的，因此我们要删除 750 的时候把 500 给删掉了。
- ④ 类似的，1239 跟 1000 等价，因此我们用 `contains?` 来检测集合是否包含 1239 的时候它返回 `true` 了。

有时候比较器的行为是你所预期的，而有时候则可能不是。需要记住的一点是，比较器的语义我们可以完全控制，因此虽然使用已有的谓词作为比较器很方便，有时候你也

可以选择直接返回正数、负数后者 0 来使得比较器的相等性更符合我们的期望。可以对 compare-magnitude 进行重写以使得只在两个数字真的相等时才认为它们两个相等（通过在两个数字的 magnitude 相等时把逻辑代理给 compare 函数实现）：

```
(defn compare-magnitude
  [a b]
  (let [diff (- (magnitude a) (magnitude b))]
    (if (zero? diff)
        (compare a b)
        diff)))

(sorted-set-by compare-magnitude 10 1000 500)
;= #{10 500 1000}
(conj *1 600)
;= #{10 500 600 1000}
(disj *1 750)
;= #{10 500 600 1000}
```

现在我们集合中的值按照 10 的量级排列了，同时那些只依赖数字相等性的操作（比如 conj 和 disj）的结果也跟我们期望的一样了。

110 ➤ subseq 和 rsubseq 也能继续按照比较器所定义的顺序正常截取集合的一段（正序或者反序）：

```
(sorted-set-by compare-magnitude 10 1000 500 670 1239)
;= #{10 500 670 1000 1239}
(def ss *1)
;= #'user/ss
(subseq ss > 500)
;= (670 1000 1239)
(subseq ss > 500 <= 1000)
;= (670 1000)
(rsubseq ss > 500 <= 1000)
;= (1000 670)
```



我们通过 <、<=、> 和 >= 来指定需要的元素的区间，其他的比较器也是可以的。

这些函数的一个有意思的用法是实现线性牛顿插值：

```
(defn interpolate
  "Takes a collection of points (as [x y] tuples), returning a function
  which is a linear interpolation between those points."
  [points]
  (let [results (into (sorted-map) (map vec points))])
```

❶

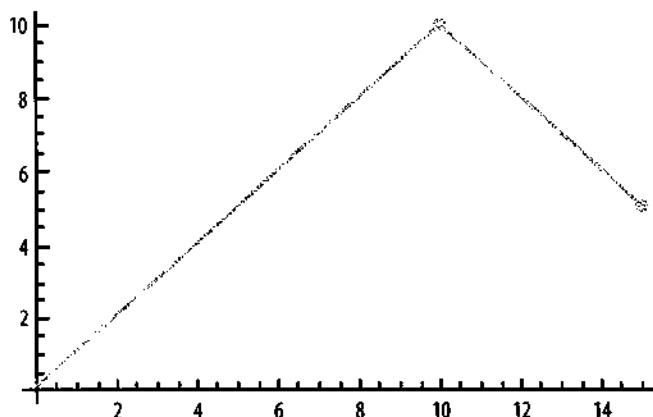
```

(fn [x]
  (let [[xa ya] (first (rsubseq results <= x))           ❶
        [xb yb] (first (subseq results > x))])
    (if (and xa xb)                                         ❷
        (/ (+ (* ya (- xb x)) (* yb (- x xa)))
            (- xb xa))
        (or ya yb))))))

```

- ❶ (`map vec points`) 确保每个点都是用 vector 表示的，从而可以加入 `map`。
- ❷ 这行以及下面一行我们在已知的点里面找到跟 `x` 最近的两个点。
- ❸ 如果附近没有邻点——只要 `xa` 和 `xb` 中有一个为 `nil`，那么我们返回 `(or ya yb)`，这也是我们唯一知道的值。
- ❹ 对于一般情况的线性牛顿插值。

让我们来测试一下。比如有三个已知点：`[0 0]`、`[10 10]` 和 `[15 5]`：



对于已知的 `x` 坐标，我们可以找到对应符合要求的 `y` 坐标：

◀111

```

(def f (interpolate [[0 0] [10 10] [15 5]]))
;= #'user/f
(map f [2 10 12])
;= (2 10 8)

```

完美啊！

访问集合元素的简洁方式

访问集合里面的元素可能是对集合进行的最常见的操作了，特别是对那些关系型数据结构来说。虽然是这么说，但是如果要一直用 `get` 或者 `nth` 来访问集合元素也是很冗长而

无聊的。谢天谢地，Clojure 的集合以及大多数作为集合里面元素的类型本身也是函数，它们的语义跟 `get` 或者 `nth` 是一样的。

集合本身也是函数。很简单，Clojure 里面的集合都可以当做函数来使用，这个函数的作用就是从自己里面查出指定 key 或者下标所对应的元素。所以

```
(get [:a :b :c] 2)
;= :c
(get {:a 5 :b 6} :b)
;= 6
(get {:a 5 :b 6} :c 7)
;= 7
(get #{1 2 3} 3)
;= 3
```

和下面这些更简洁的表达式是等价的：

```
([:a :b :c] 2)
;= :c
({:a 5 :b 6} :b)
;= 6
({:a 5 :b 6} :c 7)
;= 7
(#{1 2 3} 3)
;= 3
```

112 >

在这些用例中，集合都被放在函数位置上，所以它们被调用了，调用的参数就是对应的 key 或者下标。`map` 跟 `get` 类似，还接受第二个参数——如果查找没找到，则返回这个默认值。`vector` 和 `set` 都只接受一个参数的调用形式——不支持传入默认值。而传递给 `vector` 的数组下标必须要在 `vector` 的下标的范围内——就像 `nth` 一样：

```
([:a :b :c] -1)
;= #<IndexOutOfBoundsException java.lang.IndexOutOfBoundsException>
```

集合的 key（通常）也是函数。类似的，最常见的 key 类型——关键字以及符号——也是函数，语义就是到集合里面去把自己或者自己所对应的 value 找出来，所以这些：

```
(get {:a 5 :b 6} :b)
;= 6
(get {:a 5 :b 6} :c 7)
;= 7
(get #{:a :b :c} :d)
;= nil
```

跟下面这些更简洁的表达方式的语义是一模一样的：

```
(:b {:a 5 :b 6})
```

```
;= 6
(:c {:a 5 :b 6} 7)
;= 7
(:d #{:a :b :c})
;= nil
```

因为这个值是在函数位置上，所以数字下标是不能这么用的，因此 `vector` 的查找不能用这种方式。

习惯用法

太好了，我们有两种更简洁的访问集合元素的办法。这真的不错，但是这两种办法分别在什么情况下使用比较好呢？什么时候该把集合本身作为查找函数，什么时候应该把关键字或者符号作为查找函数呢？

这很大程度上要取决于个人的喜好了，但是通常，我们推荐把关键字或者符号作为查找函数来使用。最直接的好处是，这种方式可以避免 `NullPointerException`，因为关键字、符号通常都是字面量也就不可能为 `null`，比如：

```
(defn get-foo
  [map]
  (:foo map))
;= #'user/get-foo
(get-foo nil)
;= nil
(defn get-bar
  [map]
  (map :bar))
;= #'user/get-bar
(get-bar nil)
;= #<NullPointerException java.lang.NullPointerException>
```

113

而且，`(coll :foo)` 这个形式假定 `coll` 这个集合是一个函数。这个假定对于绝大多数的 Clojure 数据结构是成立的，但是对于有些集合就不成立了——比如列表，而且重要的是，实现了 Clojure 集合接口的数据结构不一定是函数，这使得使用 `(:foo coll)` 这种形式更好，因为 `:foo` 是一个字面量，它始终是函数，而且绝不可能为 `nil`，而 `coll` 则既可能不是函数，也可能为 `nil`。

当然，如果一个集合的 `key` 不是关键字或者符号类型的，那么你只能使用集合本身或者 `get` 或 `nth` 来作为查找函数了。

集合、`key` 以及高阶函数

因为关键字、符号以及很多的集合都是函数，使用它们作为高阶函数的输入参数是很常见同时也是非常方便的做法。比如，我们要获取所有客户的名字，不需要定义任何函数，

也不需要显式地使用 get :

```
(map :name [{:age 21 :name "David"}  
           {:gender :f :name "Suzanne"}  
           {:name "Sara" :location "NYC"}])  
:= ("David" "Suzanne" "Sara")
```

some 函数在一个序列里面搜索第一个能够符合指定谓词的元素，把它和 set 一起使用非常常见：

```
(some #{1 3 7} [0 2 4 5 6])  
:= nil  
(some #{1 3 7} [0 2 3 4 5 6])  
:= 3
```

这使得 some 非常适合用在条件判断中来判断集合里面是否包含某个元素。一种更通用的方式是使用 filter，它返回一个惰性序列，惰性序列的内容就是那些满足指定谓词的元素。同样的道理，我们可以直接把集合本身或者关键字或者符号作为谓词来使用：

```
(filter :age [{:age 21 :name "David"}  
              {:gender :f :name "Suzanne"}  
              {:name "Sara" :location "NYC"}])  
:= ({:age 21, :name "David"})  
  
(filter (comp (partial <= 25) :age) [{:age 21 :name "David"}  
                                         {:gender :f :name "Suzanne" :age 20}  
                                         {:name "Sara" :location "NYC" :age 34}])  
:= ({:age 34, :name "Sara", :location "NYC"})
```

114

remove 的作用和 filter 的作用相反，它把符合谓词的元素从集合中去掉，返回所有不符合谓词要求的元素：相当于 (filter (complement f) collection)。



小心 nil (再一次)

使用 set 来测试集合中是否包含某个元素是很方便的，但是不要忘记，如果要检测的元素是 nil 或者是 false 的话，那么结果可能就跟你的预期不一样了，因为这两个值在逻辑意义上就是 false：

```
(remove #{5 7} (cons false (range 10)))  
:= (false 0 1 2 3 4 6 8 9)  
(remove #{5 7 false} (cons false (range 10)))  
:= (false 0 1 2 3 4 6 8 9)
```

所以，当你不确定你的集合里面是否会包含 nil 或者 false 元素的时候，那么最好使用 contains? 而不是 get 之类的直接调用：

```
(remove (partial contains? #{5 7 false})) (cons false (range 10))  
:= (0 1 2 3 4 6 8 9)
```

数据结构的类型

Clojure 提供了一些具体的数据结构来供我们使用，这每一种具体的数据结构都实现了一个或者多个抽象。对于这些具体数据结构，很多我们前面已经用过了，跟我们想的一样，它们的行为和语义都是由那些它们实现的抽象所定义的。

这里我们会快速地过一遍每种具体数据结构类型的一些实现细节——看看它们各自独特的地方，这种独特之处大多数在于集合的构建方式。

列表：List

列表是 Clojure 中最简单的数据结构。它们最简单同时也最典型的用法是表示 Clojure 里面的函数调用，这个我们在第 7 页的“表达式、操作符、语法以及优先级”一节已经解释过了，所以对于列表，你可能绝大多数情况下都是用来做函数调用，直接把它作为数据结构来保存应用数据的场景应该不多。^{注 24}

Clojure 的列表是单向链表，因此只对链头支持高效的访问和“修改”操作——使用 `conj` 把一个新元素加到列表上去，使用 `pop` 或者序列操作符 `rest` 来获取不包含链头的子列表。因为是列表，所以它不支持高效的随机访问操作；所以对一个列表调用 `nth` 的时间开销是线性的而不是常量的（`vector` 和数组中随机访问元素的时间开销是常量级别的），而且列表根本就不支持 `get`，因为在列表上无法达到 `get` 亚线性的性能要求。

115

值得一提的是，列表是它们自身的序列，所以对一个列表调用 `seq` 会始终返回这个列表本身而不是对于这个列表的顺序视图。

我们前面只见过用来表示函数调用的列表字面量，真正用来存储数据的列表是这么用的：

```
'(1 2 3)  
:= (1 2 3)
```

如果不用引号把这个列表括起来，那么它就变成了对于函数 `1` 的调用，因为 `1` 根本不是函数或者宏，因此肯定会失败。^{注 25} 这个引号的一个副作用是，这个列表字面量里面的所有元素都不会被求值：

```
'(1 2 (+ 1 2))  
:= (1 2 (+ 1 2))
```

在这种场景下，大多数人会直接使用一个 `vector` 字面量，`vector` 字面量中的所有表达式

注 24：这跟其他 Lisp 方言有点不一样，其他方言里面的 `list` 和 `cons cell` 是核心对象。Clojure 则使用了更丰富的数据结构——比如 `map`、`set`、`vector` 以及 `list` 的“表弟”：序列，序列在很大程度上跟 `list` 是类似的，这也使得我们几乎不会在代码里面直接用到列表。

注 25：注意，一个空列表是不需要引号括起来的，因为没有第一个元素可以组成一个可以“调用”的值。所以我们可以合法地使用 `()` 来表示一个空列表。

都会被求值。但是如果你真的需要一个列表，其他任何数据结构都满足不了你的话，^{注26}那么可以使用 `list`：

```
(list 1 2 (+ 1 2))  
:= (1 2 3)
```

`list` 接受任意数量的参数，而每一个参数会成为返回的列表的一个元素。

最后，你可以使用谓词 `list?` 来测试一个集合是不是列表。

vector

`vector` 是一种顺序数据结构，它支持高效的随机访问和更改语义，跟我们熟悉的 Java 的 `ArrayList`、Python 的列表以及 Ruby 的数组是类似的。`vector` 也是非常通用的，它实现了我们前面看过的 `associative`、`indexed` 以及 `stack` 抽象。

除了我们熟悉的 `vector` 字面量，`vector` 还可以通过 `vector` 和 `vec` 函数来创建：

```
(vector 1 2 3)  
:= [1 2 3]  
(vec (range 5))  
:= [0 1 2 3 4]
```

`vector` 和 `list` 函数的作用类似，而 `vec` 函数只接受一个参数，它的作用是把这个传入的集合转换成一个新的 `vector`。当我们手中有的是一个数组、列表、序列或者其他可序列的值，但是要对这个值进行进一步处理的话，可能 `vector` 更合适，那么我们就可以使用这个函数进行转换。

`vector?` 谓词跟 `list?` 也是类似的，它可以用来测试一个值是不是 `vector`。

把 `vector` 当做元组来使用

元组 (`Tuple`) 是 `vector` 的最常见的使用场景。任何时候你想把多个值绑定在一起处理——比如你想从一个函数中返回多个值——那时候你就可以把这多个值放进一个 `vector`：

```
(defn euclidian-division ❶  
  [x y]  
  [(quot x y) (rem x y)])  
  
(euclidian-division 42 8)  
:= [5 2]
```

❶ 如果你想更简洁一点的话，`(juxt quot rem)` 返回的结果跟我们实现的结果是一样的。

注 26：这种情况最常见的例子是编写宏，我们会在第 5 章详细介绍。

这个跟 Clojure 到处都在使用的解构机制（28 页“解构（let, 第 2 篇）”一节有详细介绍）可以很好地配合，使得我们可以很方便地从一个 vector 里面取出某个具体的元素：

```
(let [[q r] (euclidian-division 53 7)]
  (str "53/7 = " q " * 7 + " r))
;= "53/7 = 7 * 7 + 4"
```

虽然 vector 跟 tuple 一样易用且富有诱惑力，你需要记住的是，它终究是一种不适合暴露到系统外部的数据结构：对它的使用最好保持在你的类库或者组件的内部，而不要在公共 API 上这么用。这么说有两方面原因：

- tuple 本身是不能自我注解的。看到一个 tuple 你不知道 tuple 里面的每个值是什么含义，必须要回忆或者回到 tuple 赋值的地方才能知道每个字段是什么意义。
- tuple 是不灵活的。要构建一个 tuple，你需要提供 tuple 的所有字段，即使 tuple 的某个字段对于你的场景来说是没有意义的，你还是要提供。而且你只能往一个 tuple 的尾部添加元素。

map 不受这两个限制，所以如果你要用在那种公共 API 的参数或返回值上面，那么 map 都比 vector 更适合。

当然，任何规则都是有例外的，这个规则也是。如果在某个场景下，tuple 的字段的含义是非常明确的，那么使用 vector 是完全没有问题的。比如坐标、几何图形中的边等——这些情况下完全可以使用 vector：

```
(def point-3d [42 26 -7])

(def travel-legs [[["LYS" "FRA"] ["FRA" "PHL"] ["PHL" "RDU"]]])
```

在后一种情况下，我们使用了 vector 的两种用法：外层 vector 只是一种更好的列表，而 117 内层的 vector 则是作为 tuple——表示我们的航班从哪里飞到哪里：[from to]。

set

作为一个具体类型的数据结构的 set 其实没有太多要说的——我们前面在介绍 associative 和 set 抽象的时候已经介绍过了。跟 Clojure 的其他数据结构一样，set 有自己的字面量表示（我们前面也已经看过了）：

```
#{}{1 2 3}
;= #{}{1 2 3}
#{}{1 2 3 3}    ①
;= #<IllegalArgumentException java.lang.IllegalArgumentException:
;=   Duplicate key: 3>
```

① 因此根据 set 的定义，它是不能包含重复的元素的，因此这里抛出了异常。

`hash-set` 函数可以用来创建包含任意数量元素的无序 set：

```
(hash-set :a :b :c :d)
:= #{:a :c :b :d}
```

最后，你可以利用 `set` 函数来把其他类型的集合转换成一个 set：

```
(set [1 6 1 8 3 7 7])
:= #{1 3 6 7 8}
```

这个函数可以作用于任何可序列的值，而且因为 `set` 本身又是函数，因此可以利用它写出非常简洁的代码：

```
(apply str (remove (set "aeiouy") "vowels are useless"))
:= "vwls r slss"
(defn numeric? [s] (every? (set "0123456789") s))
:= #'user/numeric?
(numeric? "123")
:= true
(numeric? "42b")
:= false
```

我们也可以定义有序的 set，这个我们在 106 页的“有序集合：sorted”一节已经介绍过了。

map

除了我们已经很熟悉的 map 字面量：

```
{:a 5 :b 6}
:= {:a 5, :b 6}
{:a 5 :a 5}          ●
:= #<IllegalArgumentException java.lang.IllegalArgumentException:
:= Duplicate key: :a>
```

❶ 跟 `set` 一样，`map` 里面的 key 必须保证唯一，上面例子里面提供了重复的 key，因此抛出了异常。

我们还可以通过 `hash-map` 函数来创建无序的 map，它接受任何一个键值对。这个函数通常被用来和 `apply` 函数一起使用：当你有一个集合的 key/value 的对子，你想把它们放进一个 map，那么你可以这么用：

```
(hash-map :a 5 :b 6)
:= {:a 5, :b 6}
(apply hash-map [:a 5 :b 6])
:= {:a 5, :b 6}
```

也可以创建有序的 map，这个我们在 106 页的“有序集合 sorted”一节也介绍过了。

`keys` 和 `vals`。虽然是针对 map 的，这些函数可以很方便地返回一个 map 的所有的键和值：

```
(keys m)
:= (:a :b :c)
(vals m)
:= (1 2 3)
```

这两个函数本质上其实只是从 map 上面获取它的键值对的序列，然后再对序列的元素应用 `key` 或者 `val` 函数：

```
(map key m)
:= (:a :c :b)
(map val m)
:= (1 3 2)
```

map 作为临时的 struct

因为 map 里面的值可以是任何类型，因此我们经常把 map 作为一种简单灵活的数据模型来使用，而且把关键字作为 map 的 `key` 来标示 map 里面的字段（也被称为槽）。

```
(def playlist
  [{:title "Elephant", :artist "The White Stripes", :year 2003}
   {:title "Helioself", :artist "Papas Fritas", :year 1997}
   {:title "Stories from the City, Stories from the Sea",
    :artist "PJ Harvey", :year 2000}
   {:title "Buildings and Grounds", :artist "Papas Fritas", :year 2000}
   {:title "Zen Rodeo", :artist "Mardi Gras BB", :year 2002}])
```

Clojure 中的建模通常是从 map 开始的。特别是当你不确定你的模型中到底由哪些字段组成的时候，map 允许你不需要定义一个严格的模型就开始编写代码逻辑。

当你使用 map（或者其他任何实现了 Clojure 那些抽象数据接口的数据结构），那么有关 map 的函数就都可以派上用场了。比如，假定我们使用关键字作为 map 的 `key`，那么要查询 map 里面的一些聚合信息就变得非常简单了：

```
(map :title playlist)
:= ("Elephant" "Helioself" "Stories from the City, Stories from the Sea"
   "Buildings and Grounds" "Zen Rodeo")
```

类似的，像我们在 32 页的“map 解构”一节介绍的解构可以帮助我们以更简洁的方式访问 map 里面的元素，而不用编写重复而冗长的 `(:slot data)` 函数：

```
(defn summarize [{:keys [title artist year]}]
  (str title " / " artist " / " year))
```

Clojure 试图为我们避免不成熟的过度设计——你可以很轻松地把你的模型从 map 升级成一个真正意义上的模型——如果你需要的话。所以，使用 map 作为模型并不意味着

将来换成对象的时候会需要大幅地更改代码。只要你是面向 Clojure 的集合抽象编程的，而不是面向集合的具体实现编程的，就可以很轻松地把一个基于 map 的模型换成一个由 `defrecord` 定义的模型。`defrecord` 定义的类型都实现了 `associative` 接口。

我们会在 270 页“[定义你自己的类型](#)”一节详细介绍 `defrecord`, 并且会在 227 页的“[什么时候使用普通 map, 什么时候使用记录类型](#)”一节和 `map` 做一些对比。

map 的其他用途

map 通常也被用来保存总结信息、索引信息或者对应关系，想象一下数据库中的索引和视图，map 跟它们类似。

比如，group-by 函数可以用来很方便地根据一个 key 函数把一个集合分成多组：

```
(group-by #(rem % 3) (range 10))  
:= {0 [0 3 6 9], 1 [1 4 7], 2 [2 5 8]}
```

这里我们看到，把有相同 key 值的元素组合到同一组里面去了。继续使用前面的 playlist 数据，我们可以轻松地给专辑信息以 artist 字段建立一个索引：

```
(group-by :artist playlist)
:= {"Papas Fritas"[{:title "Helioself", :artist "Papas Fritas", :year 1997}
 :=                      {:title "Buildings and Grounds", :artist "Papas Fritas"}]
:= ...]
```

要在两个“列”上创建索引也很简单：(group-by (juxt :col1 :col2) data)。

有时候我们需要的不是把元素分组，而是要算出每组的聚合信息。我们可以先用 group-by 函数来分组，然后再对每个元素进行处理来计算聚合信息：

```
(into {} (for [[k v] (group-by key-fn coll)]
    [k (summarize v)])))
```

这里 `key-fn` 和 `summarize` 是需要你自己实现的函数。但是如果你的集合很大的话，这么写代码会很笨重，就像如果从数据库查出的数据量很大的时候，处理这些数据也会很笨重。在这种情况下，我们可能需要利用 `group-by` 和 `reduce` 来定义自己的一个函数。`reduce-by` 可以用来对任意种类的数据计算聚合数据。跟 SQL 语句里面的 `SELECT ... GROUP BY ...` 作用差不多：

```
(defn reduce-by
  [key-fn f init coll]
  (reduce (fn [summaries x]
            (let [k (key-fn x)]
              (assoc summaries k (f (summaries k init) x))))
          {} coll))
```



x、xs 并不是很糟糕的命名

Clojure 代码中有很多很简短的名字，比如 `x` 和 `xs`，初学者可能会觉得这种命名很容易混淆，不知道它们表示的到底是什么值。这种命名方式其实已经被定义在 Clojure 的命名规范 (<http://dev.clojure.org/display/design/Library+Coding+Standards>) 里面了。实际上，当你使用 `x` 作为值的名字时，你的意思是说你的代码很通用，而且 `x` 的类型对于代码阅读者来说应该是很明显的，所以完全没必要把它命名成 `person` 之类的名字。类似的，包含 `x` 类型的值的集合就可以命名成 `xs`。你的代码越通用，你的变量名就越不需要很具体。

假设有一些订单数据，我们利用 `map` 来表示：

```
(def orders
  [{:product "Clock", :customer "Wile Coyote", :qty 6, :total 300}
   {:product "Dynamite", :customer "Wile Coyote", :qty 20, :total 5000}
   {:product "Shotgun", :customer "Elmer Fudd", :qty 2, :total 800}
   {:product "Shells", :customer "Elmer Fudd", :qty 4, :total 100}
   {:product "Hole", :customer "Wile Coyote", :qty 1, :total 1000}
   {:product "Anvil", :customer "Elmer Fudd", :qty 2, :total 300}
   {:product "Anvil", :customer "Wile Coyote", :qty 6, :total 900}])
```

利用 `reduce-by`，可以很方便地算出每个人的订单总金额：

```
(reduce-by :customer #(+ %1 (:total %2)) 0 orders)
;= {"Elmer Fudd" 1200, "Wile Coyote" 7200}
```

类似的，我们可以计算出订购了每种商品的所有客户的名字：

```
(reduce-by :product #(conj %1 (:customer %2)) #{ } orders)
;= {"Anvil" #{"Wile Coyote" "Elmer Fudd"},
;= "Hole" #{"Wile Coyote"},
;= "Shells" #{"Elmer Fudd"},
;= "Shotgun" #{"Elmer Fudd"},
;= "Dynamite" #{"Wile Coyote"},
;= "Clock" #{"Wile Coyote"})
```

那如果我们需要两个维度的分组呢？比如想要查看每个客户在每种商品上面花了多少钱，只需要返回一个包含两个值的 `vector` 作为 `key` 函数。有很多种方法来编写这个函数：

```
(fn [order]
  [(:customer order) (:product order)])

#(vector (:customer %) (:product %))

(fn [{:keys [customer product]}]
  [customer product])
(juxt :customer :product)
```

这里使用最清晰、最简单的方式：

```
(reduce-by (juxt :customer :product)
  #(+ %1 (:total %2)) 0 orders)
:= {[["Wile Coyote" "Anvil"] 900,
:= ["Elmer Fudd" "Anvil"] 300,
:= ["Wile Coyote" "Hole"] 1000,
:= ["Elmer Fudd" "Shells"] 100,
:= ["Elmer Fudd" "Shotgun"] 800,
:= ["Wile Coyote" "Dynamite"] 5000,
:= ["Wile Coyote" "Clock"] 300}
```

这个结果跟大家的期望可能有点差距——大家想要的可能是一个包含 map 的 map。这个问题其实跟 reduce-by 实现有关，它假定 map 是一维的。要得到一个包含 map 的 map，要么修改 reduce-by 实现，要么自己把结果加工成一个包含 map 的 map。

要让 reduce-by 能返回一个嵌套的 map 其实很简单，只要把调用 assoc 以及隐式调用 get(当把 map 作为一个函数来使用的时候)的地方全部换成 assoc-in 和 get-in 就行了：

```
(defn reduce-by-in
  [keys-fn f init coll]
  (reduce (fn [summaries x]
    (let [ks (keys-fn x)]
      (assoc-in summaries ks
        (f (get-in summaries ks init) x))))
  {} coll))
```

现在的结果跟期望的一样的，我们得到了一个嵌套的 map 了：

```
(reduce-by-in (juxt :customer :product)
  #(+ %1 (:total %2)) 0 orders)
:= {"Elmer Fudd" {"Anvil" 300,
:=                 "Shells" 100,
:=                 "Shotgun" 800},
:= "Wile Coyote" {"Anvil" 900,
:=                 "Hole" 1000,
:=                 "Dynamite" 5000,
:=                 "Clock" 300}}
```

第二个办法是我们得到结果后自己对这个结果做转换：

```
(def flat-breakup
  {[["Wile Coyote" "Anvil"] 900,
  ["Elmer Fudd" "Anvil"] 300,
  ["Wile Coyote" "Hole"] 1000,
  ["Elmer Fudd" "Shells"] 100,
  ["Elmer Fudd" "Shotgun"] 800,
  ["Wile Coyote" "Dynamite"] 5000,
  ["Wile Coyote" "Clock"] 300})
```

我们还是会使用 assoc-in 函数：

```
(reduce #(apply assoc-in %1 %2) {} flat-breakup)
:= {"Elmer Fudd" {"Shells" 100,
                  "Anvil" 300,
                  "Shotgun" 800},
    := "Wile Coyote" {"Hole" 1000,
                      "Dynamite" 5000,
                      "Clock" 300,
                      "Anvil" 900})}
```

由 flat-breakup 提供的序列中的每个值都被当做 map 的一个元素 (entry)，比如 `[["Wile Coyote" "Anvil"] 900]`。因此，当 reduce 函数调用 apply 函数在对 map 的每个元素进行处理的时候，对每个元素调用的函数其实是 assoc-in——比如，`(assoc-in {} ["Wile Coyote" "Anvil"] 900)`——我们使用 `["Wile Coyote" "Anvil"]` 来定义 map 的结构，使用 900 来定义它的值。

不可变性和持久性

我们已经学习了 Clojure 集合的各种特性和各种抽象接口。我们没有强调的它们的两个重要特征是：它们是不可变的，而且是持久的。

我们在第 2 章介绍过不可变性，而且也学习了这些不可变的实体所提供的值语义能够如何地简化我们的编程生活。但是你可能会有点隐隐的担心。比如，看下面这个例子：

```
(+ 1 2)
:= 3
```

这里的 3 是跟 + 的参数完全独立的一个值。这个把两个数字相加的动作不会去修改任何一个参数的值，使得参数变成另外一个数字。这种风格跟其他那种鼓励不受限制对数据进行修改有着很大的区别，比如看下面的 Python 代码：

```
>>> lst = []
>>> lst.append(0)
>>> lst
[0]
```

append 真的修改了 lst 对象。这意味着很多事情，包括一些不好的事情，但是你可以肯定的是这些操作都很高效，几乎与所要操作的集合大小没有关系。而另一方面这种随意修改又是可能导致问题的：

```
(def v (vec (range 1e6))) ❶
:= #'user/v
(count v)
:= 1000000
```

```
(def v2 (conj v 1e6))      ❶
:= #'user/v2
(count v2)
:= 1000001
(count v)
:= 1000000
```

- ❶ 首先我们定义一个 vector，它包含从 0 到 1e6 的数字：一百万个元素，一个很大的集合。
- ❷ 使用 conj 来向这个 vector 添加一个新元素，这样它就包含 1 000 001 个元素了。
- ❸ 而我们前面也说了，Clojure 的所有数据结构是不可修改的，因此 v 的内容并没有发生变化。

这里的 v2 是一个完全独立的数据结构。你可能会想：“这肯定会有性能问题，因为看起来 conj（以及其他任何对 Clojure 数据结构进行操作的函数）创建了它所操作的集合的完全拷贝！”

不过，实际情况不是这样的。

持久性与结构共享

对于 Clojure 不可变数据结构的操作是非常高效的，通常跟 Java 里面对应的操作一样高效。这是因为 Clojure 数据结构是持久的，这种技术使得你做过修改的集合跟原来的集合共享内部数据存储，而同时又保证从这个源集合产生出来的集合一样的高效率。



“持久性”的语义

很显然，我们这里说的“持久性”并不是对于数据、对象的序列化、存储等。这里的含义最早是在 Okasaki 的 *Purely Functional Data Structures* (<http://www.amazon.com/dp/0521663504>) 里面提出的，在这本书中，它描述了一种能够使不可变数据结构的所有修改版本的效率一直保持的一种技术。

Purely Functional Data Structures 是函数式编程领域影响深远的创新，Clojure 以及现在市面上的很多函数式语言都是基于这个技术的。大家如果想更深入地了解函数式编程、函数式编程中的数据结构设计，那么一定要读一读这本书。

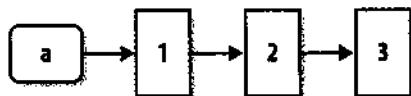
- 124> 为了实现这个持久性而又不牺牲性能，Clojure 的数据结构实现了一种结构共享的技术。意思是说，对于任何一个操作，Clojure 都不会去做一个深度拷贝；只有受影响的那部分会被添加 / 删除，而不变的那些元素还是共享老集合里面的那些。

持久性的图解：列表

对于这种机制最简单的示例是操作一个列表，通过展示对列表的操作会怎样影响底层存储，我们来看看内部是怎么共享结构的：

```
(def a (list 1 2 3))
```

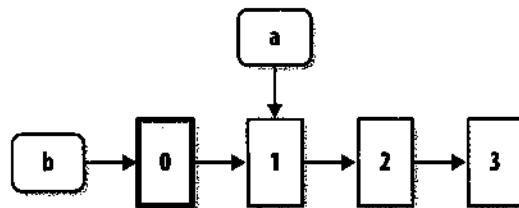
我们前面提到，Clojure中的列表是单向的链表，可以这样表示一个列表：



下面我们利用conj函数来把一个新值添加到这个列表上，记住，conj操作列表的时候始终是把元素添加到列表的头上：

```
(def b (conj a 0))
;= #'user/b
b
;= (0 1 2 3)
```

可以这样来表示这个操作：



conj确实创建了一个新的列表，而且0是它的第一个值，但是它完全重用了a的所有元素，显然，这是一个非常高效的操作：

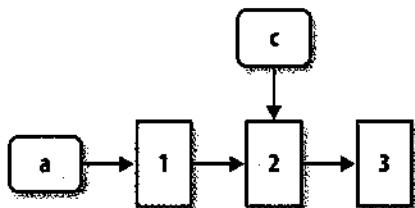
- 没有做任何数据的拷贝。
- 老列表a还是保持原样，而且我们仍然对它进行访问，而且还可以对它进行操作。
- 这个新的列表b共享了a的整个结构，只是添加了一个值而已。

就像我们前面说过的一样，conj会保证它的操作始终在常量时间内完成：往一个3个元素的列表上添加一个元素应该跟往一个百万元素的列表上添加一个元素花费相同的时间，这个对于列表来说是没问题的。所以a的这种特征在它的修改版b上面保持了。

那对于其他类型的操作呢？Clojure的列表不支持随机访问（跟其他列表一样），所以我

我们知道，我们无法以常量时间获取 `nth` 对列表操作的结果——只能是线性时间。但是我们可以很高效地把链头上的那个元素去掉：

```
(def c (rest a))
:= #'user/c
c
:= (2 3)
```



列表的 `rest` 操作也是一个类似的常量时间操作，多亏 Clojure 的结构共享技术使得对列表的修改版都能保持这个特性。`rest注27` 会在常量时间内返回列表的尾巴，不管这个列表的长度是多少，而且对于这个列表的修改版，这种特性还是能够保持的。

持久性图示：map（以及 vector 和 set）

所以对于一个列表的操作可以非常高效。那么对于 Clojure 编程中用得最多的数据结构呢？`map`、`vector`、`set`？其实内部实现上它们使用的技术是一样的，^{注28} 不过内部结构比列表要更复杂。

我们这里以 `map` 为例来看一下：

```
(def a {:a 5 :b 6 :c 7 :d 8})
```

跟绝大多数其他语言中的 `map` 一样，Clojure 中的 `map` 也是实现成树状的，而 `map` 中的具体的值是保存在树结构的叶子节点上。

126 来更新一下我们的 `map`：

```
(def b (assoc a :c 0))
:= #'user/b
b
:= {:a 5, :c 0, :b 6, :d 8}
```

注 27： 其实，这里我们可以简单地使用 `pop`。

注 28： 它们内部有很多细微的差别，但是并不妨碍我们把它们放在一起讨论它们的持久性语义。需要注意的是，这一节中我们展示的 `map` 的数据结构只是为了讨论这个语义的需要。真正实现的结构不一定跟这里介绍的完全一样。

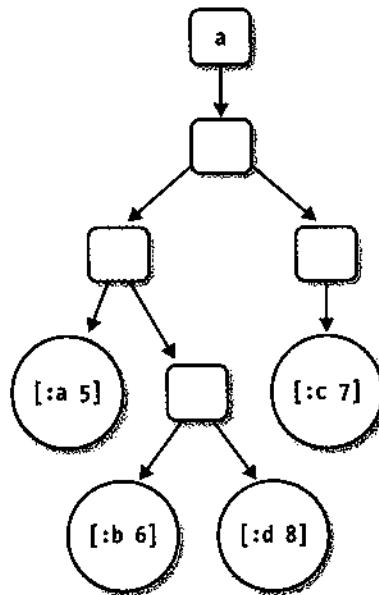


图3-1: map {:a 5 :b 6 :c 7 :d 8} 的内部表示²⁹

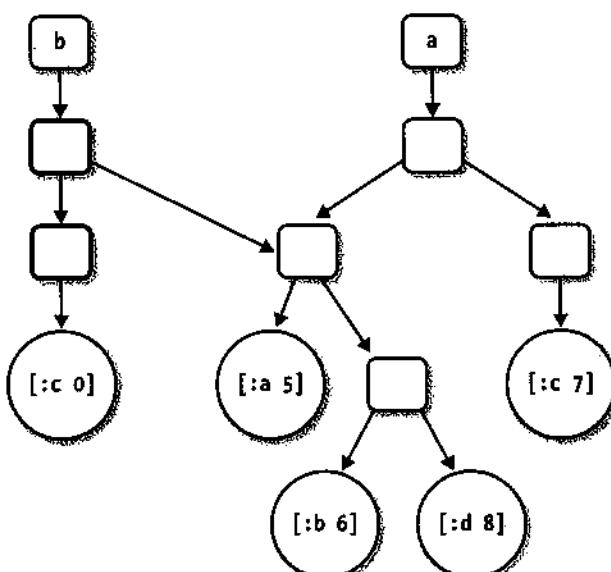


图3-2: (assoc a :c 0) 的效果

注 29：注意，我们这里介绍的是“不具有可扩展性的”，这里只是为了方便讨论结构共享与持久性。

127 跟前面用 `conj` 来把一个元素添加到列表中不同，对于 `map`，我们使用 `assoc`。新产生的 `map` 跟老 `map` 共享了很大一部分的结构。包含 `:a`、`:b` 和 `:d` 的子树在新 `map` 里面都被重用了，唯一的内存分配是我们新添加的数据的叶子节点以及从这个叶子节点链接到根节点的所有内部节点。

而从 `map` 中移除一个元素跟往 `map` 中添加一个元素在结构共享方面也是差不多的：

```
(def c (dissoc a :d))
:= #'user/c
c
:= {[:a 5], [:c 7], [:b 6]}
```

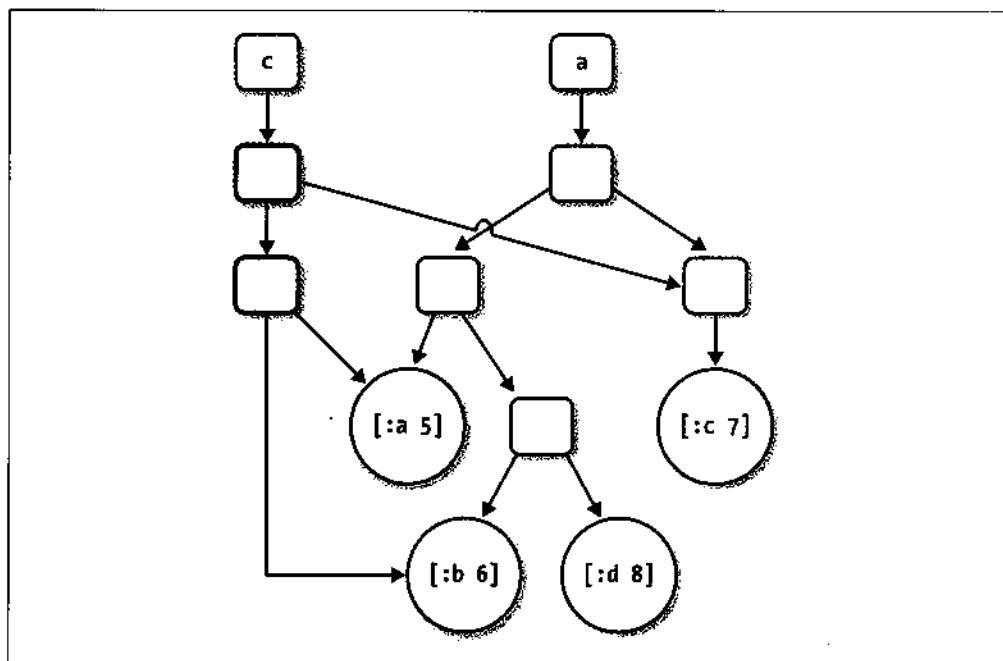


图3-3: (`dissoc a :d`) 的效果

128 跟添加一个元素一样，这个树的绝大部分都被重用了，所以新 `map` 跟旧 `map` 共享了大部分树结构。

树的森林

为了提供持久性的语义，Clojure 中几乎所有的数据结构都是通过树来实现的，包括 hashmap 和 hashset，有序 map 和有序 set，vector 等。到底是什么种类的树可能差别很大：hashmap 使用的是 hash array mapped trie 的持久化版本，^{注 30} 而 vector 使用的则是一个叫做 array mapped hash trie 的更进一步的变种，^{注 31} hashset 则是基于 hashmap 来实现的。而有序 set 和有序 map 则使用的是一种红黑树的持久化版本，以保证排序的性能。

在所有情况下，Clojure 的树实现使得你期望高效的操作（比如给 vector 添加一个元素，给 set 添加一个元素，给 map 添加一个键值对，以及从这些集合中查询元素）都很高效。以大 O 形式来表示的话，这些操作要么是 $O(\log_2 n)$ （对于无序集合）或者 $O(\log_2 n)$ （对于有序集合），这里 n 是集合中的元素个数。这意味着什么？对于不可变数据结构的所有操作与对于可变集合的所有操作都一样快，或者非常接近，同时又能够让我们享有不可变值语义所带来的众多好处。

129

有趣的是，现在很多系统都在使用这种类似的实现策略：不可变的持久的树。除了 Clojure 还有 Git、CouchDB 等。关于这些系统的一些相似点在这篇文章里面有介绍 <http://eclipsesource.com/blogs/2009/12/13/persistent-trees-in-git-clojure-and-couchdb-data-structure-convergence>。

显著效果

虽然看起来好像我们又深入到 Clojure 数据结构具体的实现细节了（这些细节确实很有趣、很吸引人），这些实现细节确实给我们带来一些显著的好处，因为它们都是持久的而且不可变的。我们已经在第 2 章详细介绍了使用值能够如何简化我们的代码，消除由于不加控制地对数据结构的修改所带来的一大类的 bug。这里再介绍一些其他好处。

简化多线程编程。Clojure 的引用类型我们将在 170 页的“Clojure 的引用类型”一节介绍，如果没有有用的、高效的、不可变的值可以让它们来保持，它们也就没有存在的意义。因为这些引用类型定义的是对于一块数据在一段时间内变化的语义，所以如果你将一个本身是可以改变的数据放在引用类型里面是没有意义的。因为在任何一个时间点任何其他的代码都有可能对这些可变集合进行修改，因此也就埋没了我们使用引用类型的意义，

注 30：<http://blog.higher-order.net/2009/09/08/understanding-clojures-persistenthashmap-deftwice> 讲解了 Clojure 的 PersistentHashMap 类中对于 hash array mapped trie 的实现。

注 31：<http://blog.higher-order.net/2009/02/01/understanding-clojures-persistentvector-implementation> 有对 Clojure 的 PersistentVector 类的详细介绍。

而对于这个共享数据的并发访问和修改的协调任务又落到开发人员自己身上了。

轻松的版本化。在很多系统中的一个需求是如何保持一块数据的多个版本。如果这个数据结构是可以改变的话，那么这个需求不是那么容易实现的。你怎么复制一个数据以保证后续对它的修改不会影响它的历史版本？你怎么正确地回退到某个历史版本？

不可变集合使得实现这样的需求不可思议地简单：

```
(def version1 {:name "Chas" :info {:age 31}})
:= #'user/version1
(def version2 (update-in version1 [:info :age] + 3))          ❶
:= #'user/version2
version1
:= {:info {:age 31}, :name "Chas"}
version2
:= {:info {:age 34}, :name "Chas"}
```

- 130 ❶ update-in 函数更新关系型数据结构（有可能是嵌套的）中指定的字段（由 vector 中的参数指明更新哪个字段），怎么更新呢，通过对它执行某个函数（我们这里是 +），同时指定任意多个需要的参数（这里是 3）。

对于 Clojure 集合的每一次“更新”都不会影响原来的集合，而是产生了一个新的版本。每一个版本都可以被独立地使用、修改和保存；但是你从来不需要自己去做这些烦琐的事情：生成新版本、保留旧版本等，因为这些都是集合的实现帮我们做了的事情，我们什么也不需要做。

易变集合

易变集合是持久化集合的对应物：持久化集合保证一个值的历史版本的完整性，而易变集合不做这个保证。在对一个易变集合做修改后，对于旧版本易变集合的任何引用都不能再使用了；这个旧版本的集合可能是对的，可能包含了新的值，也可能已经被垃圾回收了。

这个初看起来像是一个很傻的设计，但是，如果一个数据结构被修改了，但是没有人访问这个旧集合，那又有什么关系呢？

跟 Clojure 中任何其他东西形成鲜明对比的是，易变集合是可修改的：

```
(def x (transient []))      ❶
:= #'user/x
(def y (conj! x 1))         ❷
:= #'user/y
(count y)                  ❸
:= 1
```

```
(count x)          ❶
:= 1
```

- ❶ 我们基于一个持久 vector 创建了一个易变的 vector。
- ❷ 我们用 `conj!` 向易变集合添加了一个新值，它是 `conj` 的“易变”版本……
- ❸ 跟它的持久化版本一样，它也是返回一个指向新易变 vector 的一个引用，但是……
- ❹ 旧的易变集合内容也发生改变了！^{注32}

那么，我们上面谈到那么多不可变性、值语义给我们带来的众多好处，那么为什么还要设计这么一个可变的易变集合呢？Clojure 的持久化数据结构有一些非常吸引人的特性，而且它们所提供的性能也是非常高的。但是，还是有一些不可避免的、无法忽视的场景是持久性集合所不适合的：简单来说，持久性数据结构所保证的值语义对于每一个更新操作都需要进行内存分配，而这个内存分配确实会有一些开销。虽然对于单个操作来说这个开销可以忽略，但是如果进行大量的这种操作，比如要把成千上万个值用 `conj` 添加到一个集合中去，这些开销经过累积之后就相当可观了。◀131

所以易变集合只是设计来对这种场景进行优化的。在有些场景下，易变集合可以减少甚至消除这种累积的对象分配，因此也就最小化了垃圾收集时间，从而也就降低了整体的集合构造时间。它对于那种在一个循环中不断创建对象是一种很方便的优化手段。^{注33}

我们所说的到底是什么样的场景呢？好吧，来看一个之前用过的 Clojure 核心函数 `into`，它接受一个集合以及一些序列的值作为参数，然后把序列中的值都通过 `conj` 添加到集合中去：

```
(into #{} (range 5))
:= #{0 1 2 3 4}
```

要是让我们自己来实现，应该怎么做呢？作为我们尝试的第一个版本，很简单：

```
(defn naive-into
  [coll source]
  (reduce conj coll source))

(= (into #{} (range 500))
   (naive-into #{} (range 500)))
:= true
```

太棒了，我们有了自己实现的 `into` 了，而且它能够接受任意的持久化集合作为参数。但

注 32：就像我们在这一节开头说过的那样，易变集合的旧版本一定不可以再使用，我们这里只是想告诉大家易变集合是可变的。

注 33：Clojure 程序如果发现性能问题，我们一般首先要查看的就是内存分配是不是有问题。

是性能跟 Clojure 核心库中的 `into` 相比怎么样呢？

```
(time (do (into #{} (range 1e6))
           nil))
; "Elapsed time: 1756.696 msecs"
(time (do (naive-into #{} (range 1e6))
           nil))
; "Elapsed time: 3394.684 msecs"
```

- ❶ 我们不想把一百万个数字打印到 REPL 上，因此我们把代码包在一个 `do` 形式里面，然后让它返回 `nil`。

啊！`naive-into` 要慢两倍！原因在于 `into` 在任何可以使用易变集合的地方都使用了易变集合，^{注34} 也就是说，当“目标”集合是 `vector` 或者是无序的 `map` 或者 `set` 的时候使用易变集合，因为它们是目前有易变版本的几种集合。

132 我们也可以这么实现：

```
(defn faster-into
  [coll source]
  (persistent! (reduce conj! (transient coll) source)))
```

在应用 `reduce` 之前，先把我们的初始集合变成一个易变集合。然后在 `reduce` 的每一步，使用 `conj!`——`conj` 的易变版本，来把 `source` 中的下一个值添加到易变集合中去。`reduce` 返回这个新的填充了值的易变集合，然后我们再用 `persistent!` 把它持久化。结果怎么样呢？

```
(time (do (faster-into #{} (range 1e6))
           nil))
; "Elapsed time: 1639.156 msecs"
```

是的，现在性能已经和 `into` 相当了。我们先停一下来思考一个问题：快始终是更好的，但是我们是不是失去了持久的、不可变的集合所给我们带来的所有好处呢？简单来说就是，没有。虽然我们在整个 `faster-into` 函数中确实是在对一个可变集合进行操作，但是可变集合从来就没有溢出使用它的函数的范围！这意味着 `faster-into` 和 `naive-into` 有着完全一样的语义——持久性集合入和持久性集合出。所以用户可以享有持久性集合带来的所有好处，同时又可以获得随意修改的性能。



记住，只有 `vector` 以及无序的 `map` 和无序 `set` 有易变版本。因此如果你传一个有序集合给 `faster-into` 的话，那么它会抛出一个异常。

遗憾的是，没有一个标准的谓词可以用来检测一个集合是否有易变版本。你需要

注 34： 其他 Clojure 核心库函数在可能的地方也都使用易变集合，比如 `frequencies` 和 `group-by`。

检测一个集合是不是 `clojure.lang.IEditableCollection` 的实例，Clojure 用这个集合来标明一个集合可以产生可变版本。我们可以编写下面这个谓词 `transient-capable?` 来做检测：

```
(defn transient-capable?
  "Returns true if a transient can be obtained for the given collection,
  i.e. tests if `(transient coll)` will succeed."
  [coll]
  (instance? clojure.lang.IEditableCollection coll))
```

既然我们已经知道了易变集合的优点以及应该限制它的使用范围，下面来看一下易变集合的一些细节机理。

从这个函数名就可以看出来，被用来产生易变集合的持久性集合是不会被影响的：

```
(def v [1 2])
:= #'user/v
(def tv (transient v))
:= #'user/tv
(conj v 3)
:= [1 2 3]
```

而另一方面，用 `persistent!` 来把一个易变集合变成持久性集合会使得那个易变集合不可用：^{注35} ◀133

```
(persistent! tv)
:= [1 2]
(get tv 0)
:= #<IllegalAccessError java.lang.IllegalAccessError:
:= Transient used after persistent! call>
```

`transient` 和 `persistent!` 都会在常量时间内返回。

易变集合支持很多它的持久化版本所支持的访问函数，但是并不是全部：

```
(nth (transient [1 2]) 1)
:= 2
(get (transient {:a 1 :b 2}) :a)
:= 1
((transient {:a 1 :b 2}) :a)      •
:= 1
((transient [1 2]) 1)
:= 2
(find (transient {:a 1 :b 2}) :a)
:= #<CompilerException java.lang.ClassCastException:
```

注 35：`persistent!` 名字里的感叹号提醒用户这个函数是一个破坏性操作。

```
;= clojure.lang.PersistentArrayMap$TransientArrayMap
;= cannot be cast to java.util.Map (NO_SOURCE_FILE:0)>
```

❶ 易变集合也是函数。

一个值得一提的例外是 `seq`，易变集合不支持这个，原因是一个序列可能会比它的数据源活得时间更长，而易变集合是可变的，因此它不适合作为序列的数据源。

但是，所有持久性集合所支持的更新函数，易变集合都是不支持的。易变集合有它们自己的更新函数，它们的名字跟持久性的版本一样，只是在最后有个感叹号，而作用也是类似的：`conj!`、`assoc!`、`dissoc!`、`disj!` 和 `pop!`。

一旦你在易变集合上面使用了上面列出的任意一个函数，那个集合就再也不可用了一一即使是读取操作也不行。就像你始终要使用 `conj`、`assoc` 和 `disj` 的结果一样，为了这些操作的效果生效，你必须使用这些函数的返回值。也就是说，你不能始终保留对易变集合的原始引用，否则就会产生一些错误的结果：

```
(let [tm (transient {})]
  (doseq [x (range 100)]
    (assoc! tm x 0))           ❶
  (persistent! tm))
;= {0 0, 1 0, 2 0, 3 0, 4 0, 5 0, 6 0, 7 0}
```

- 134 ❶ 虽然 `assoc!` 添加了 100 个记录到易变 map 中去，但是几乎所有的记录都丢了，因为这里没有保留 `assoc!` 的返回值。

易变集合跟它的持久性版本使用是类似的，只是被修改过的持久化集合不再可用。最简单而又最安全的办法就是写一些普通的 Clojure 代码，然后添加一些必要的 `transient` 和 `persistent!` 调用，然后给所有的修改操作函数后面加上感叹号。就像我们在 `faster-into` 里面看到的一样，`reduce` 函数可能是可以最简单地改造成使用易变函数的操作。

因为易变集合只是一种优化的手段，因此使用的时候要非常慎重而且控制它的使用范围，通常限制在单个函数（或者一个库里面的几个相关的私有函数之间）。这个最佳实践其实在实现层面上被强制了——易变集合在实现的时候提供了一个并发保护——易变集合只能在创建这个集合的线程内操作。我们会在第 4 章系统介绍并发编程，为了理解下面的例子，这里大家只需要知道下面的 `future` 函数会使得 `get` 调用被放到一个单独的线程去执行：

```
(let [t (transient {})]
  @(future (get t :a)))
;= #<IllegalAccessError java.lang.IllegalAccessException:
;= Transient used by non-owner thread>
```

易变性可能是一个非常有效的优化手段，但是并不是没有代价的：你需要重新规划你的代码，因为易变集合的使用是有限制的，而且 Clojure 核心库对易变集合的支持也是有限的。跟任何优化手段一样，你应该分析一下它对你的代码来说是不是最好的办法以及一旦使用了这个手段，会不会有效果。

易变集合不能组合。易变集合不能组合，`persistent!` 不会遍历你创建的嵌套易变集合，因此对最上层调用 `persistent!`，对于内嵌的子集合是没有效果的：

```
(persistent! (transient [(transient {})]))  
;= [#<TransientArrayList clojure.lang.PersistentArrayList$TransientArrayList@  
b57b39f>]
```

在任何情况下，因为易变集合是可修改的，它们没有值语义，因此不能这么比较：

```
(= (transient [1 2]) (transient [1 2]))  
;= false
```

这也应该使我们想到易变集合不应该和值语义混起来使用。它应该只作为一个本地的优化策略，不应该暴露给用户。

元数据

135

元数据是有关数据的数据，元数据在其他语言中有很多不同的名字并且也有各种不同的表现形式：

- 类型声明和访问控制修饰符（比如 `private`、`protected` 等）是跟它们相关联的值、变量以及函数的元数据。
- Java 注解是类、方法以及方法参数等的元数据。

虽然 Clojure 中的元数据也是使用在这些场景下，^{注36} 但是 Clojure 中的元数据更通用，你可以用自己的应用中、用在你自己的数据上。

元数据可以被关联到任何 Clojure 数据结构、序列、记录、符号以及引用类型，而且始终是以一个 `map` 的形式。Clojure 还提供了一个 `reader` 字面量来更方便地把元数据关联到值：

```
(def a ^{:created (System/currentTimeMillis)}  
      [1 2 3])  
;= #'user/a  
(meta a)  
;= {:created 1322065198169}
```

注 36：366 页的“为了效率进行类型提示”一节介绍了如何指定类型信息给 Clojure 的编译器；198 页的“vars”介绍了如何给 var 添加元数据来指定它的访问策略和并发语义；381 页的“注解”介绍了如何利用元数据来给 Clojure 的类型指定 Java 的注解（Annotation）。

为了方便，如果要指定的元数据的 key 是关键字而值是布尔值 true，那么我们以一种简单的方式来指定，而且可以把多个这种元数据并排指定：

```
(meta ^{:private [1 2 3]})  
:= {:private true}  
(meta ^{:private ^{:dynamic [1 2 3]}})  
:= {:dynamic true, :private true}
```

对于更新一个给定值的元数据，Clojure 提供了 with-meta 和 vary-meta：

```
(def b (with-meta a (assoc (meta a)  
                           :modified (System/currentTimeMillis))))  
:= #'user/b  
(meta b)  
:= {:modified 1322065210115, :created 1322065198169}  
(def b (vary-meta a assoc :modified (System/currentTimeMillis)))  
:= #'user/b  
(meta b)  
:= {:modified 1322065229972, :created 1322065198169}
```

 **136** with-meta 是把一个值的元数据完全替换成给定的元数据，而 vary-meta 通过给定的更新函数（这里是 assoc）以及需要的参数对值当前的元数据进行更新。

需要记住的是，元数据是关于数据的数据。换句话说，你改变一个值的元数据并不会改变这个值打印 (println) 出来的形式，也不会影响这个值跟别的值的相等性（或者不等性）。

```
(= a b)  
:= true  
a  
:= [1 2 3]  
b  
:= [1 2 3]  
(= ^{:a 5} 'any-value  
    ^{:b 5} 'any-value)  
:= true
```

当然，一个有元数据的值跟没有元数据的值一样是不可变的，所以像对数据结构进行修改的操作返回的新的数据结构会保留原值的元数据：

```
(meta (conj a 500))  
:= {:created 1319481540825}
```

这比其他语言中应用层面的元数据要有用很多，处理应用层面的元数据的话，我们必须非常小心地把应用元数据跟“实际”数据分开，如果不小心把它跟实际数据混在一起的话会影响实际数据的相等性或者更新的值里面没有带旧值的元数据。上面我们已经看到数据的创建时间、修改时间可以放在元数据里面；还可以通过元数据来跟踪数据的不同

来源，是从数据库来的？还是从缓存来的？我们可以给从数据库出来的数据加上一个元数据，给从缓存出来的数据加上一个不同的元数据，这样不会影响数据本身的相等性，但是我们可以对不同来源的数据做不同的处理了。

用 Clojure 的集合来小试牛刀

为什么数据结构那么重要？Frederick Brooks 在他的《人月神话》一书中说到：

如果你给我展示你的流程图而不展示你的数据库表结构，我可能还是不理解你的系统。而如果你给我展示你的表结构，那通常就不需要你的流程图了；看了你的表结构，你的系统的功能应该就很明显了。

20 年之后，Eric Raymond 总结了一个现代的版本：

如果你给我展示你的代码而不展示你的数据结构，我可能还是不理解你的系统。而如果你给我展示你的数据结构，那通常就不需要看你的代码了；看了你的数据结构，你的系统的功能应该就很明显了。

组织数据、数据建模的方式决定了最终代码的样子，因此如果还是用那种旧思维利用数组、专门的对象是写不出好的函数式的 Clojure 代码的。◀ 137

在 Clojure 中要进行好的数据建模，要把精力集中在值（特别是可组合的值）、自然标识符以及 set 和 map 上面。我们要适应的思路跟关系型^{注 37} 建模类似。但是一个值里面到底包含什么样的内容是上下文相关的：还好我们有可组合的值，它是结构化的、可嵌套的，因此到底在值里面维护怎样的数据关系是完全由开发者来定义的。

标识符和循环引用

自然标识符是相对人工虚构的标识符来说的，后者通常来说是字符串或者自动生成的数字，通常是数据库的自增字段或者大多数面向对象语言中的对象引用。因此根据定义来说，人工虚构的标识符就已经引入了一种不必要的复杂性。

如果没有这个不必要的人工标识符这个中间产物的话，我们就没有必要维护对象跟它们的标识符之间的对应关系：那么对象之间要么是不同的——作为值它们本身已经可以标识它们自己的独特性了；要么它们就是相同的。作为一个额外的好处，去除人工虚拟的标识符之后，不同的线程甚至进程不需要相互通信以获取一个唯一的人工虚拟标识符了，因为你的数据本身就是它自身的自然标识符。

到底自然标识符好还是人工虚构标识符好的辩论并不是什么新话题，但是在设计数据库

注 37：clojure.set 这个命名空间是关系代数操作符的一个实现。

的表结构时还是要考虑这个问题。关于人工虚构的标识符的争论主要集中在两点：一个进程创建的数据会比这个进程本身（以及这个进程中关于这个数据所定的一些规则）的寿命要长，这使得这个数据很难重构。这其实也意味着人工虚拟标识符在创建它的进程内是没用的，而只在整个应用层面才有用。

关于不必要的人工虚拟标识符的典型例子是由解析器或者正则表达式引擎产生的各种不同的状态。解析器或者正则表达式引擎通常会给这些状态分配数字的标识符来标识它们，而在 Clojure 中，你可以组装一个非常复杂的值，这个值本身就可以标识它自己。然后要把一个状态转换成另一个状态，只需要简单地对它应用一个函数就好了，只需要把这个实际的值作为参数就好了。跟使用人工虚拟标识符相比，我们不再需要传一个状态跟数字之间的对应关系给这个函数了：数据本身已经足够标识它自己了。

如果为了追求更好的性能，你在后面仍然可以给你的状态分配数字标识符，但是有两点要特别说明：

1. 你可以不这么做。

2. 你可以在一个独立的步骤对状态进行数字标识，清楚地隔离关注点和复杂性。这意味着如果需要的话，你可以维护多套人工虚拟标识符，如果一份数据会被多个系统操作的情况下，这个通常是需要的。

这里唯一的禁忌是不要有循环引用：引入循环引用的唯一合理的方法是通过人工虚拟标识符和一个单独的间接层。而没有循环引用也意味着没有反向引用。

循环引用、值和标识符。假定我们有一个不可变的树节点（比如它是一个 DOM 节点），它有两个属性：`:parent-node` 和 `:children`。现在如果我们要对这个节点进行“修改”（创建一个新版本），这两个属性会怎么样？如果你不更新 `:parent-node` 和 `:children` 的值，那么通过这个旧的 `:parent-node` 和 `:children` 来回这么一找，最后找到的将是这个节点的旧版本而不是新版本！这是非常有问题的啊，它迫使你在修改一个节点的时候必须同步修改这个节点的父亲节点和子孙节点。

168 页的“状态和标识”一节有介绍。但是这么一来的话，我们要操作的就不再是不可变的值了，我们需要去编写一些特殊的逻辑来对这些可变的引用进行操作。欢迎回到随意修改对象状态、十分复杂的境地。

大多数时候，循环引用（又称为反向引用）会被内建到数据结构里面以满足我们对数据结构进行遍历的需求。这种场景由 zippers 来搞定，我们会在 151 页的“遍历、更新和 Zipper”一节来介绍。这一节剩下的内容我们会介绍通过引入一个间接层来引入循环引用。

这个间接层可以是普通的引用类型或者标识符（用来从一个索引里面来查找）。这个通

常感觉起来可能非常繁琐，但是这么处理会迫使你去问你自己非常重要的两个问题：我的状态中哪些数据是纯值？哪些数据应该有自己的人工虚拟标识符？这些人工虚拟标识符标识的是什么？^{注38} 这些问题及其答案在随后把你的数据分享到你的应用之外（要么是把这些数据发送到互联网、存到磁盘、存到数据库，或者暴露成一个服务）的时候会节省你很多时间。

思维方式的改变：从命令式到函数式

我们前面已经讨论过用序列来代替数字型的下标了。但是，除了作为一个比较漂亮的抽象层，序列本身还是线性的，而且序列本身并不会帮助你以函数式的思维来思考问题以及如何有效地使用 Clojure 的数据结构来解决问题。你需要转换思维来面向值的编程，使用它们来更好地使用 Clojure。

自己动手实现经典的游戏：Conway's Game of Life

139

Conway's Game of Life (https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life) 似乎是一种专门为数组而设计的算法。我们这里用 Clojure 重新实现一下它的规则。首先以一种传统的思路——整个板子是一个包含 vector 的 vector，板子里面的每个格子的状态要么是 :on，要么是 nil——然后再以一种更 Clojure 的风格来实现，消除下标的复杂性（以及限制）。

```
(defn empty-board
  "Creates a rectangular empty board of the specified width
  and height."
  [w h]
  (vec (repeat w (vec (repeat h nil)))))
```

现在我们已经有一个空的板子了，需要加一些活的格子上去：

```
(defn populate
  "Turns :on each of the cells specified as [y, x] coordinates."
  [board living-cells]
  (reduce (fn [board coordinates]
            (assoc-in board coordinates :on))
          board
          living-cells))
(def glider (populate (empty-board 6 6) #{{[2 0] [2 1] [2 2] [1 2] [0 1]})))
(pprint glider)
; [[nil :on nil nil nil]
;  [nil nil :on nil nil]
;  [:on :on :on nil nil]]
```

注 38：这里有关识别出系统中的实体以及它们的界线其实是一种领域驱动设计的思路，Eric Evans 编写的 *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley Professional) 一书中详细讨论了这个。

```
; [nil nil nil nil nil nil]  
; [nil nil nil nil nil nil]  
; [nil nil nil nil nil nil]
```

现在到了我们要实现的核心部分了：一个 `indexed-step` 函数，它接受一个板子的状态作为输入参数，然后根据游戏的规则返回板子的下一个状态：

示例3-6：`indexed-step`函数的实现以及一些帮助方法

```
(defn neighbours  
  [[x y]]  
  (for [dx [-1 0 1] dy [-1 0 1] :when (not= 0 dx dy)]  
    [(+ dx x) (+ dy y)]))  
  
(defn count-neighbours  
  [board loc]  
  (count (filter #(get-in board %) (neighbours loc)))) ❶  
  
(defn indexed-step  
  "Yields the next state of the board, using indices to determine neighbors,  
  liveness, etc."  
  [board]  
  (let [w (count board)  
        h (count (first board))]  
    (loop [new-board board x 0 y 0]  
      (cond  
        (>= x w) new-board  
        (>= y h) (recur new-board (inc x) 0)  
        :else  
          (let [new-liveness  
                (case (count-neighbours board [x y])  
                  2 (get-in board [x y])  
                  3 :on  
                  nil)]  
            (recur (assoc-in new-board [x y] new-liveness) x (inc y)))))))  
140→
```

❶注意，因为 `count-neighbours` 使用 `get-in`——它是基于 `get` 的，就像我们已经看到的，对于那些不存在的下标它不会抛出异常，而是返回 `nil`。

我们来看看它工作得怎么样：

```
(-> (iterate indexed-step glider) (nth 8) pprint)  
; [[nil nil nil nil nil nil]  
;  [nil nil nil nil nil nil]  
;  [nil nil nil :on nil nil]  
;  [nil nil nil nil :on nil]  
;  [nil nil :on :on :on nil]  
;  [nil nil nil nil nil nil]]
```

现在我们已经有一个可以正常工作的 Conway's Game of Life 游戏的实现了，而且看起来

还不错哦！

下面来看看如何对这个实现进行重构以避免使用下标。第一步是去除手动的遍历，每一个 loop 可以替换成 reduce：

```
(defn indexed-step2
  [board]
  (let [w (count board)
        h (count (first board))]
    (reduce
      (fn [new-board x]
        (reduce
          (fn [new-board y]
            (let [new-liveness
                  (case (count-neighbours board [x y])
                    2 (get-in board [x y]))
                  3 :on
                  nil)]
              (assoc-in new-board [x y] new-liveness)))
          new-board (range h)))
      board (range w))))
```

<141

嵌套的 reduce 总是可以整合成一个 reduce 以使代码更简洁：

```
(defn indexed-step3
  [board]
  (let [w (count board)
        h (count (first board))]
    (reduce
      (fn [new-board [x y]]
        (let [new-liveness
              (case (count-neighbours board [x y])
                2 (get-in board [x y]))
                3 :on
                nil)]
            (assoc-in new-board [x y] new-liveness)))
      board (for [x (range h) y (range w)] [x y]))))
```

现在我们有了一个无 loop、但是还是使用下标的实现版本了。

虽然已经说过很多遍，我们要使用序列来替换下标，但是这里使用的函数 count-neighbours 和 neighbours 都很大程度上依赖于下标。我们如何在不依赖下标的情况下用序列表达一个格子的“邻居”的概念呢？

如果要表示的“邻居”是一维的，那么应该很简单，直接使用 partition 函数就可以了：

```
(partition 3 1 (range 5))
;= ((0 1 2) (1 2 3) (2 3 4))
```

这里，partition 产生的结果可以看成是 1、2 和 3 以及它们的邻居。这个函数的唯一问题是它只会为那些有足够的邻居的元素产生邻居数据：比如这里就没有产生 0 和 4 的邻居数据！可以通过在这个集合中添加一些填充值来修正这个问题：

```
(partition 3 1 (concat [nil] (range 5) [nil]))  
;= ((nil 0 1) (0 1 2) (1 2 3) (2 3 4) (3 4 nil))
```

这下我们可以写出一个可以工作的计算邻居数据的 window 函数了：

```
(defn window  
  "Returns a lazy sequence of 3-item windows centered around each item of coll."  
  [coll]  
  (partition 3 1 (concat [nil] coll [nil])))
```

那我们怎么把这个一维的函数转换成支持二维的呢？这里的窍门是，当把 window 函数应用到一个有 n 行的集合的时候，我们得到 n 个有 3 行的元组，而每个 3 行的元组（长度为 m）可以转换成包含 m 元组的一个序列；这种方式其实就叫变换。然后再次把这个 window 函数作用到一个包含元组的元组，我们可以很快地给每个格子创建一个 3×3 邻居。

来看一下代码：

```
(defn cell-block  
  "Creates a sequences of 3x3 windows from a triple of 3 sequences."  
  [[[left mid right]]]  
  (window (map vector  
    (or left (repeat nil)) mid (or right (repeat nil)))))
```

142 > 这里的两个 or 形式是用把 window 产生的 nil 填充转换成一个序列的 nil，因为 map 函数会在它的任何一个参数序列为空的时候结束。^{注 39} 我们可以通过让 window 接受一个额外的 pad 参数来简化代码：

```
(defn window  
  "Returns a lazy sequence of 3-item windows centered  
  around each item of coll, padded as necessary with  
  pad or nil."  
  ([[coll] (window nil coll)])  
  ([[pad coll]  
   (partition 3 1 (concat [pad] coll [pad]))))  
  
(defn cell-block  
  "Creates a sequences of 3x3 windows from a triple of 3 sequences."  
  [[[left mid right]]]  
  (window (map vector left mid right)))
```

注 39：换句话说，map 的结果跟它的最短的参数一样长。所以只要有一个参数是 nil 或者为空，那么 map 就会返回一个空序列。

我们需要对一个板子的中间格子计算它们的生命力，这个也需要把它重构出来，这一次我们使用解构来精确地获取这个中间的格子：

```
(defn liveness
  "Returns the liveness (nil or :on) of the center cell for
  the next step."
  [block]
  (let [[[_ _ center _] _] block]
    (case (- (count (filter #{:on} (apply concat block))))
      (if (= :on center) 1 0))
      2 center
      3 :on
      nil)))
```

这样，可以利用这些不依赖下标的帮助函数来重构 indexed-step 函数了：

```
(defn- step-row
  "Yields the next state of the center row."
  [rows-triple]
  (vec (map liveness (cell-block rows-triple)))))

(defn index-free-step
  "Yields the next state of the board."
  [board]
  (vec (map step-row (window (repeat nil) board))))
```

虽然 index-free-step 所依赖的帮助函数完全不依赖下标，但它的作用跟 indexed-step 是完全等价的：

```
(= (nth (iterate indexed-step glider) 8)
   (nth (iterate index-free-step glider) 8))
  := true
```

这整个的实现过程每一步都是很简单的，但是从整体来看，从“命令式”的解决方案转换到一个“函数式”的解决方案对于初学者来说不是那么容易的。 ◀143

更进一步。到目前为止的实现方法的一个问题是，我们始终还是没有完全摆脱最开始那种实现思维。但是实际上是有更优雅的实现方式的。要达到这种效果，让我们来做一次深呼吸，后退一步来好好看看这个游戏的规则。

在游戏的每一步，会发生如下的转换：

- 任何只有少于两个活邻居的格子会死掉，因为邻居太少。
- 任何有两个或者三个活邻居的格子可以活到下一轮。
- 任何多余三个活邻居的格子会死掉，因为太拥挤了。
- 任何有正好三个活邻居的死格子可以变成一个活格子，这可以看做一种再生机制。

这个对于游戏规则的表述没有提到行、列、数组下标什么的。它提到的仅仅是格子和邻居；或者更准确一点，它讨论的是活着的格子、邻居以及活格子旁边的死格子。因此这里的两个主要概念是活格子和邻居：死格子是活格子的死邻居，因此死格子可以从邻居和活格子两个概念引申出来。

如果我们围绕这两个概念来建模，那么这里唯一的状态就是所有活着的格子。在每次生成板子的下一个状态的时候，只要首先计算所有活格子的邻居，然后数一下每个邻居格子出现的次数（每个格子出现的次数其实就是它旁边的活格子的个数）。

我们把上面的表述翻译成 Clojure 代码，得到的将是这样的：

示例3-7：Conway's Game of Life的一个更优雅的实现

```
(defn step
  "Yields the next state of the world"
  [cells]
  (set (for [[loc n] (frequencies (mapcat neighbours cells))]
            :when (or (= n 3) (and (= n 2) (cells loc))))
            loc)))
```

搞定了！这里只依赖了前面的 `neighbours` 帮助函数；没有下标，没有包含 `vector` 的 `vector`，我们也没有去限定这个板子的大小，而且对于这个板子的表示也是稀疏的——只显式记录了所有活格子的坐标。^{注40}

来试试我们的新 `step` 函数，这个游戏的状态已经不是一个板子了——只是一些活着的格子的集合——但是我们可以重用 `populate` 函数来产生一个板子：

```
[144] ->> (iterate step #{{2 0} {2 1} {2 2} {1 2} {0 1}}) ●
      (drop 8)
      first
      (populate (empty-board 6 6))
      pprint)
; {[nil nil nil nil nil nil]
;  [nil nil nil nil nil nil]
;  [nil nil nil :on nil nil]
;  [nil nil nil nil :on nil]
;  [nil nil :on :on :on nil]
;  [nil nil nil nil nil nil]}
```

❶首先产生一个跟之前一样的板子，我们可以看到它跟我们期望的一样，每 4 步漂移一个单元。

这里有趣的一点是，示例 3-7 里面的 `step` 函数使用了我们在传统思路（示例 3-6）里面为 `indexed-step` 函数而设计的帮助函数 `neighbours`。但是在这里，这个方案的上下文中，

^{注 40：} 这里是和数组处理语言，比如 APL 或者 J 来看齐的，虽然底层的实现可能完全不一样。

`neighbours` 所处理的并不是某个具体数据结构的下标，而是坐标，这里的 [x y] 坐标对对 `step` 来说是完全透明的，而在之前的实现版本中，这个坐标是由 `indexed-step` 自己产生的。

因此，现在 `neighbours` 函数是整个算法中唯一关心格子标识符的部分。也就是说，这个函数定义了格子的拓扑结构。通过对 `neighbours` 函数进行调整，我们可以让它支持有限的板子、环形板子、六边形的板子、N 维的板子等。我们不用去改变 `step` 的实现。通过排除命令式的思维方式，我们得到的是一个更好的、清楚地隔离关注点的，一个更清晰的、跟原始问题的描述更接近的解决方案，而且这个解决方案也更通用。

我们只要稍微做一些改动，就能让这个 `step` 更通用：设计成一个高阶函数 `stepper`，这个函数将成为创建 `step` 函数的工厂函数。

```
(defn stepper
  "Returns a step function for Life-like cell automata.
  neighbours takes a location and return a sequential collection
  of locations. survive? and birth? are predicates on the number
  of living neighbours."
  [neighbours birth? survive?]
  (fn [cells]
    (set (for [[loc n] (frequencies (mapcat neighbours cells))]
              :when (if (cells loc) (survive? n) (birth? n)))
          loc))))
```

我们的 `step` 实现跟 (`stepper neighbours #{} #{} {2 3}`) 返回的函数是等价的。这个 `stepper` 高阶函数可以定义不同的生存策略以及格子的拓扑结构。比如对于 H.B2/S34 这样的自动生命生成游戏（六边形的板子，有 2 个、3 个、4 个活格子邻居的格子变成活格子）可以简单地实现成：

```
(defn hex-neighbours
  [[x y]]
  (for [dx [-1 0 1] dy (if (zero? dx) [-2 2] [-1 1])]
      [(+ dx x) (+ dy y)]))

(def hex-step (stepper hex-neighbours #{} #{} {2 3 4}))
```

```
;= ; this configuration is an oscillator of period 4
(hex-step #{{[0 0] [1 1] [1 3] [0 4]}})
;= #{{[1 -1] [2 2] [1 5]}}
(hex-step *1)
;= #{{[1 1] [2 4] [1 3] [2 0]}}
(hex-step *1)
;= #{{[1 -1] [0 2] [1 5]}}
(hex-step *1)
;= #{{[0 0] [1 1] [1 3] [0 4]}}
```

145

所以上面那个只有四行的 `stepper` 函数是一个通用的自动生命类游戏的工厂。但这不是因为我们去掉了下标（因为基于序列的实现也是不通用的），而是因为我们把数据结构替换成 `set`、自然标识符，`map`（那个出现频次的 `map`）。因为我们的关注点在 `set` 和自然标识符上面，所以这个解决方案可以称为是“关系型”的。



除了一个使用 `set` 和 `map`，一个使用具体的下标这个区别之外，`step` 和 `indexed-step` 还有一个区别：后者是作用在一个有限的矩形板子上的，而前者是作用在一个无限的板子上的。但是我们可以轻松地使用 `stepper` 来重新实现 `index-step` 函数，这里假定 `w` 和 `h` 已经全局绑定到要创建的有限板子的宽和高了：

```
(stepper #(filter (fn [[i j]]) (and (< -1 i w) (< -1 j h)))
           (neighbours %) #{2 3} #{3}))
```

迷宫生成

让我们再来看另外一个例子：Wilson 的迷宫生成算法。^{注41}

Wilson 的算法是一个雕刻算法：它从一个全是墙，没有路径的“迷宫”开始，然后通过从这个迷宫中移除一些墙来雕刻出来一个迷宫。它的原则是：

1. 随机选择一个坐标并且把它标记成已经访问过的。
2. 随机选择一个还没有访问过的坐标，如果没有的话，那么返回这个迷宫。
3. 从这个新选的坐标开始，在这个迷宫里面做一个随机的遍历，直到碰到一个已经访问过的坐标——如果在一次随机遍历过程中经过一个坐标多次，那么记住每次你离开这个坐标的方向。
4. 把这个随机遍历经过的所有坐标标记为已经访问过的，然后根据每个坐标的最后“离开方向”把这个路径上的墙都拆掉。
5. 从第 2 步开始重复。

146 >

一般来说，我们会以一个矩阵来表示一个迷宫，而矩阵里面的每个元素是一个 `bitset`，这个 `bitset` 表示的是这个坐标的哪些墙还在。聪明的读者可能已经注意到了，对于相邻坐标的那堵公共的墙，它在两个坐标里面被保存了两遍。

Willson 算法还进一步需要你记住每个坐标的离开方向，这也为你下一步要把每一个坐标的状态记入 `bitset` 的时候添加了额外的复杂性。有很多原因导致 Willson 算法被认为是实现起来比较复杂的，但是有了 Clojure 以及“关系型”建模的帮助，我们可以最小化甚至彻底干掉这个复杂性——利用 `set`、`map` 以及自然标识符！

注 41：如果你对迷宫生成器感兴趣的话，Jamis Buck 写了一些非常不错的 blog：<http://weblog.jamisbuck.org/2011/2/7/maze-generation-algorithm-recap>。

如果观察一下这个算法的梗概，会发现这些实体：坐标、访问过的状态、这个迷宫本身、随机遍历、离开方向。让我们看看如何在 Clojure 里面表示这些概念。

很显然，可以使用一个 vector 来表示坐标：[x y]。

如果在坐标中只保存它的坐标信息，我们怎么保存那些访问过的状态呢？答案很简单：这个状态是在坐标外保存的，一个坐标只是一个（自然）标识符。因此我们会保存一个被访问过的坐标 set。

迷宫本身是由墙组成的，而一堵墙处于两个坐标之间，因此一堵墙可以用两个坐标对来表示：比如 [[0 0] [1 0]] 表示的是坐标 [0 0] 和坐标 [1 0] 之间的那堵墙。这里的问题是，[[1 0] [0 0]] 表示的是同一堵墙。因此我们应该使用一种不关心元素顺序的一个集合来保存——set。这样，#[[0 0] [1 0]] 就是一堵墙的一个唯一的自然标识符了。一个迷宫是一系列的墙，因此可以很自然地保存成一个墙的 set！^{注42}

怎么来表示一个“遍历”呢？很简单，遍历其实就是坐标的序列，这里我们说的序列是它的广义的意思：任何顺序性的集合都可以，包括 Clojure 中的序列，vector 或者 list 都是可以的。

最后，离开方向：离开方向其实是从离开的坐标到它新进入的坐标，因此一个方向其实也是两个坐标的对子：[from to]——只是这里坐标的顺序是重要的，因此我们这里使用 vector。

既然已经定义了我们的数据结构，下面可以开始编码了：

```
(defn maze
  "Returns a random maze carved out of walls; walls is a set of
  2-item sets #{a b} where a and b are locations.
  The returned maze is a set of the remaining walls."
  [walls]
  (let [paths (reduce (fn [index [a b]]
                        (merge-with into index {a [b] b [a]}))
                      {} (map seq walls))
                      start-loc (rand-nth (keys paths)))]
    (loop [walls walls
           unvisited (disj (set (keys paths)) start-loc)]
      (if-let [loc (when-let [s (seq unvisited)] (rand-nth s))]
          (let [walk (iterate (comp rand-nth paths) loc)
                steps (zipmap (take-while unvisited walk) (next walk)))
            (recur (reduce disj walls (map set steps))
                  (reduce disj unvisited (keys steps))))
          walls)))
```

注 42：然后一个迷宫就是一个坐标（每个坐标包含两个信息）的集合。不要让数据结构弄晕你：不要看这个数据结构嵌了这么多层，每次看一层就好了。

- ❶ paths 是从各个坐标到相邻坐标（一个 vector）的一个索引（map），看一下❸）。
- ❷ (map seq walls) 把墙转换成序列，因此我们可以使用解构 [a b]。^{注 43}
- ❸ 通过创建 (keys path) 包含了所有的坐标，因此，rand-nth 返回一个随机的遍历起点。
- ❹ 我们这里维护的是没有访问过的坐标的集合而不是访问过的坐标的集合，因为维护访问过的坐标的集合的话，代码写起来会比较复杂；看❻ 和❼。
- ❺ 这里对 seq 的调用有两个目的：保证这个 set 不为空，同时提供一个顺序的视图以使用 rand-nth 函数。如果我们使用访问过的坐标而不是没有访问过的坐标，那么 (seq unvisited) 则要替换成 (seq (remove visited (keys paths)))。
- ❻ (iterate (comp rand-nth paths) loc) 产生一个无限步的遍历：它以一个坐标为起点，从 paths 中获取它的相邻坐标，然后用 rand-nth 来随机选取一个。如果 paths 返回的是一个 set 而不是一个顺序集合（比如 vector），那么我们可能就要写成这样了：(comp rand-nth seq paths)。
- ❼ (take-while unvisited walk) 是随机遍历的一部分，直到碰到一个访问过的坐标（但是不包括它）。如果我们保持的数据是访问过的坐标，(take-while unvisited walk) 要改成 (take-while (complement visited) walk)。
- ❽ (next walk) 是无限的，但是 (take-while unvisited walk) 不是，因此，zipmap 只会处理 (next walk) 的前 n 个元素（这里 n 是 (count (take-while unvisited walk)))。因此 (next walk) 的前 n 个元素也就是一次随机的遍历——除掉开始坐标和第一个访问过的坐标。因此这两个序列是一一对应的，每个键值对将会组成一个方向。从这些键值对来创建一个 map 的话会只保留对于一个 key 的最后的方向，因此也就是最后的离开方向。
- ❾ 这个结果 map 的每一条记录都是一个随机遍历上经过的每个坐标的“最后离开方向”。(map set steps) 把这些方向（map 的元素）转换成我们要从当前“迷宫”里面去除的墙（set）。

要测试这个实现的话，我们还需要两个工具函数：grid——用来创建一个全都是墙的迷宫；draw——用来渲染生成好的迷宫（我们这里使用 Swing 的 JFrame）：

```
(defn grid
  [w h]
  (set (concat
    (for [i (range (dec w)) j (range h)] #{{[i j] [(inc i) j]}})
```

注 43： [& [a b]] 可以直接对 set 进行解构，但是比较少见。

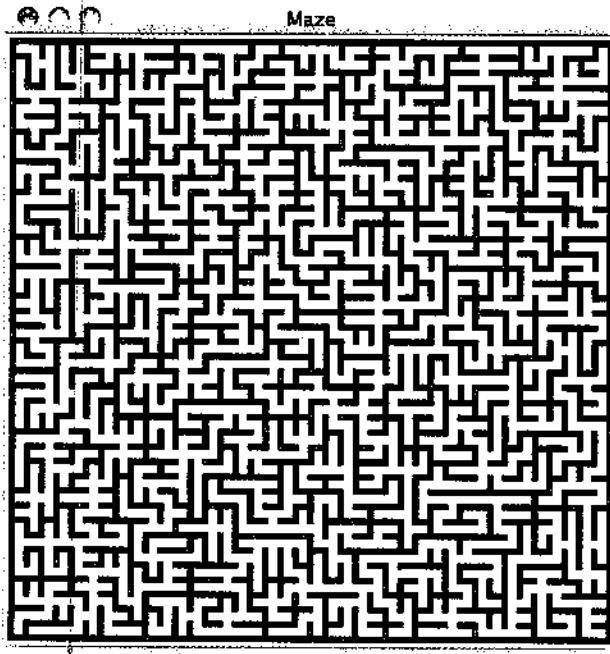
```

(for [i (range w) j (range (dec h))] #{{[i j] [i (inc j)]}}))

(defn draw
  [w h maze]
  (doto (javax.swing.JFrame. "Maze")
    (.setContentPane
      (doto (proxy [javax.swing.JPanel] []
        (paintComponent [^java.awt.Graphics g]
          (let [g (doto ^java.awt.Graphics2D (.create g)
            (.scale 10 10)
            (.translate 1.5 1.5)
            (.setStroke (java.awt.BasicStroke.0.4)))])
        (.drawRect g -1 -1 w h)
        (doseq [[[xa ya] [xb yb]] (map sort maze)]
          (let [[xc yc] (if (= xa xb)
            [(dec xa) ya]
            [xa (dec ya)])]
            (.drawLine g xa ya xc yc))))))
      (.setPreferredSize (java.awt.Dimension.
        (* 10 (inc w)) (* 10 (inc h))))))
    .pack
    (.setVisible true)))

(draw 40 40 (maze (grid 40 40)))

```



真正的 Wilson 算法

实际上，我们在这里撒了个小谎：这里实现的并不是一个真正的 Wilson 算法。

在我们的代码里面，当随机遍历碰到一个已经在我们的迷宫中的坐标的时候，我们把这个随机遍历中经过的所有坐标都加入到这个迷宫里面去了，而不仅仅是这个子路径：从起点到终点。显然我们的算法比 Wilson 算法要快，因为它每次添加的坐标更多。但是 Wilson 的卖点是每一种迷宫生成出来的几率是一样的。

Empirical 度量（通过查看 Wilson 算法和我们算法生成的迷宫的分布程度）可以看出这个特性对于我们的算法仍然成立，但是我们并没有正式地去证明这个东西，而且也没有去计算它的时间复杂性。这个就留给读者来做练习吧。^{注 44}

我们这个“小说”后面的有趣的故事是：这种写法用 Clojure 太容易写了，我们不得不这么写。

一个真正的 Wilson 算法的实现并没有太大的不同，只是多加两行代码：

```
(defn wmaze
  "The original Wilson's algorithm."
  [walls]
  (let [paths (reduce (fn [index [a b]]
    (merge-with into index {a [b] b [a]}))
    {} (map seq walls))
    start-loc (rand-nth (keys paths))]
    (loop [walls walls unvisited (disj (set (keys paths)) start-loc)]
      (if-let [loc (when-let [s (seq unvisited)] (rand-nth s))]
        (let [walk (iterate (comp rand-nth paths) loc)
              steps (zipmap (take-while unvisited walk) (next walk))
              walk (take-while identity (iterate steps loc))
              steps (zipmap walk (next walk))]
          (recur (reduce disj walls (map set steps))
            (reduce disj unvisited (keys steps))))
        walls))))
```

❶ 这里只追溯随机遍历的最近的分支，从 loc 开始。

❷ 把这个路径变成一个 map，每个元素是一个 [from-loc to-loc] 键值对。

注 44：但是我们对这个答案感兴趣！

正式地说：Wilson 的算法生成随机跨越树的图。在最原始的论文里面^{注45} 描述算法的伪代码——虽然是命令式的——使用的是 set 和 map，但是它依赖人工虚拟的标识符（每个坐标的编号）。这个实现仍然是很不错的而且很清晰，但是已经被大多数实现迷宫生成器的程序员忘掉了。

作为一个额外的好处，你可能已经注意到了，跟前面例子中的 step 一样，maze 是不关心你的一个坐标是怎么表示的。^{注46} 因此 maze 可以处理任何拓扑的迷宫：六边形的、N 维的等。作为这种通用性的一个示例，下面的代码生成一个六边形的迷宫：

```
(defn hex-grid
  [w h]
  (let [vertices (set (for [y (range h) x (range (if (odd? y) 1 0)(* 2 w) 2)]
                        [x y]))]
    deltas [[2 0] [1 1] [-1 1]])
    (set (for [v vertices d deltas f [+ -]]
            :let [w (vertices (map f v d))])
          :when w) #{v w})))

(defn- hex-outer-walls
  [w h]
  (let [vertices (set (for [y (range h) x (range (if (odd? y) 1 0)(* 2 w) 2))
                        [x y]))]
    deltas [[2 0] [1 1] [-1 1]])
    (set (for [v vertices d deltas f [+ -]]
            :let [w (map f v d)])
          :when (not (vertices w))) #{v (vec w)}))

(defn hex-draw
  [w h maze]
  (doto (javax.swing.JFrame. "Maze")
    (.setContentPane
      (doto (proxy [javax.swing.JPanel] []
                    (paintComponent [^java.awt.Graphics g]
                      (let [maze (into maze (hex-outer-walls w h))]
                        g (doto ^java.awt.Graphics2D (.create g)
                          (.scale 10 10)
                          (.translate 1.5 1.5)
                          (.setStroke (java.awt.BasicStroke. 0.4
                            java.awt.BasicStroke/CAP_ROUND
                            java.awt.BasicStroke/JOIN_MITER)))
                        draw-line (fn [[[xa ya] [xb yb]]]
```

150

注 45：参见 David Bruce Wilson 的 *Generating Random Spanning Trees More Quickly than the Cover Time*：
<http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.8598>。

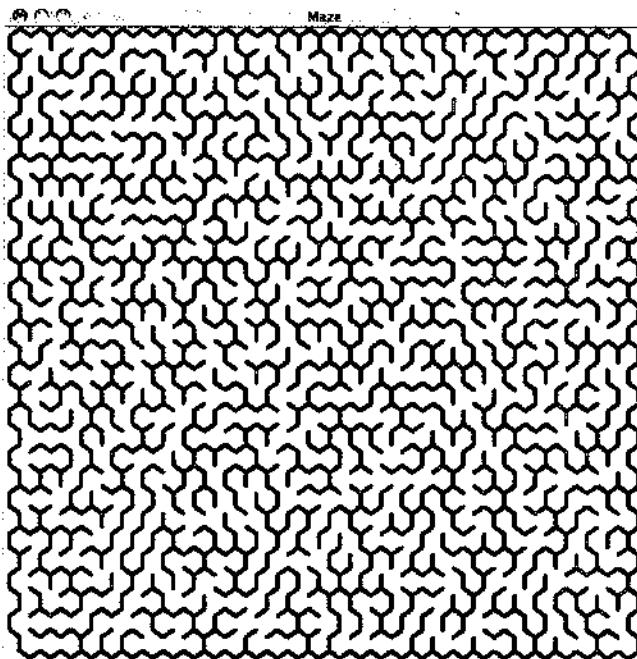
注 46：只要它不是 nil 或者 false。

```

        (.draw g
          (java.awt.geom.Line2D$Double.
            xa (* 2 ya) xb (* 2 yb))))]
  (doseq [[[xa ya] [xb yb]] (map sort maze)]
    (draw-line
      (cond
        (= ya yb) [[(inc xa) (+ ya 0.4)] [(inc xa)(- ya 0.4)]]
        (< ya yb) [[(inc xa) (+ ya 0.4)] [xa (+ya 0.6)]]
        :else [[(inc xa) (- ya 0.4)] [xa (- ya 0.6)])))))))
  (.setPreferredSize (java.awt.Dimension.
    (* 20 (inc w)) (* 20 (+ 0.5 h))))))
.pack
(.setVisible true))

(hex-draw 40 40 (maze (hex-grid 40 40)))

```



用 Java、Python 或者 Ruby 来实现 `maze` 也是可行的，但是写出来的代码会很脆弱，因为它需要依赖可变的对象作为值；^{注47} 对于这么小的例子来说，可能问题还不是很大，但是如果是一个大点的项目的话，那么开发者就必须要遵守一些规则来避免由于依赖可变的对象而带来的问题了。或者你也可以以积极防御的风格来编写代码，对于输入的数据在进行深度拷贝后再做处理，但是这样的话，性能会下降很厉害。

注 47：这个问题可以通过 Ruby 中的 `freeze` 或者 Java 中的不可修改的集合来稍微缓解下。

一般来说，当你发现写代码的时候很痛苦很难受的话，那么就意味你没有用对这门语言。这时候重新设计一下你的数据结构，应该就能解决问题。就像 Brooks 说的那样，具体的代码怎么编写是由你的数据建模方式决定的；同样的道理，要编写好的 Clojure 代码也需要你正确地使用好的数据结构，而这些“好的数据结构”通常是自然标识符、set 以及 map。

遍历、更新以及 Zipper

因为不可变性就意味着反向引用不可用，你不能依赖反向引用来遍历这个树。对于这个问题典型的解决办法是 zipper，在 `clojure.zip` 命名空间里面有它的一个实现。

zipper 跟希腊神话里面的阿里阿德涅之剑是不一样的，阿里阿德涅杀死了人身牛头怪物弥诺陶洛斯之后，帮助提修斯逃离了迷宫：^{注48} 从本质上来说，它就是一个堆栈，包含了我们要遍历的所有节点，以及这些节点的子节点。因此它实际上是一种游标，它是一种遍历机制，也是一种对树进行修改的机制。你可以移动一个 zipper，查看当前的节点以及坐标，然后对当前节点进行更新，再通过你的遍历路径来得到一个新树。跟 Clojure 的集合一样，zipper 也是持久的而且不可变的，因此一个 zipper 进行移动或者更新的话，会返回一个新的 zipper，并且原来的 zipper 不会发生改变。

◀ 152

操作 zipper

`clojure.zip` 提供了一个很通用的生成 zipper 的工厂函数以及三个更具体的工厂函数：用 `seq-zip` 来处理嵌套的序列，用 `vector-zip` 来处理嵌套的 vector，用 `xml-zip` 来处理由 `clojure.xml` 表示的 xml 文档。我们稍后会看一下如何自定义一个 zipper，现在我们的例子会使用由 `vector-zip` 创建的 zipper。

移动一个 zipper 的基本函数包括 `up`（向树的根部移动）、`down`（向树的叶子移动）以及 `left` 和 `right`（向相邻节点移动）。如果我们做一个深度优先的遍历，那么还可以使用 `prev`、`next`、`leftmost` 以及 `rightmost` 来把游标移动到树的最左边或者最右边（但是如果 zipper 已经在最边上了，那么就不会移动了）。

`clojure.zip` 命名空间还提供了下面这些函数来对当前节点进行检测：`node`、`branch?`、`children`、`lefts`、`rights` 以及 `root`，它们的作用分别是返回当前节点；当前节点是不是一个分支节点？^{注49} 当前节点的子节点（适用于分支节点）；当前节点所有左边兄弟节点以及右边兄弟节点。`root` 在利用 zipper 对一个树进行修改的时候非常关键，因为它是获取一个更新的树的唯一的办法。

注 48： <https://en.wikipedia.org/wiki/Ariadne> 有关于这个神话的介绍。

注 49： 一个分支节点是一个可能有子节点的节点；一个分支可以没有子节点。

```
(require '[clojure.zip :as z])

(def v {[1 2 [3 4]] [5 6]})
:= #'user/v
(-> v z/vector-zip z/node)
:= {[1 2 [3 4]] [5 6]}
(-> v z/vector-zip z/down z/node)
:= [1 2 [3 4]]
(-> v z/vector-zip z/down z/right z/node)
:= [5 6]
```

你可以利用 `remove` 来删除当前节点，利用 `replace` 来把当前节点替换成一个新节点，利用 `insert` 来插入一个子节点到当前节点（插在第一个位置），利用 `append` 来插入一个子节点到当前节点（插在最后面）。更常见的情况是，你需要 `edit` 一个节点。`edit` 函数利用统一的更新模型：除了要更新的 `zipper` 之外，它接受一个函数以及一些需要的参数，当前节点会被这个函数调用这些参数的结果所替换。

163 还可以基于你现在 `zipper` 的位置来利用 `make-node` 来创建一个新的节点。但是这个节点不会被加到我们的树结构里面，如果想把它加入树结构的话，那么调用上面介绍的那些函数。

```
(-> v z/vector-zip z/down z/right (z/replace 56) z/node)
:= 56
(-> v z/vector-zip z/down z/right (z/replace 56) z/root) ❶
:= {[1 2 [3 4]] 56}
(-> v z/vector-zip z/down z/right z/remove z/node) ❷
:= 4
(-> v z/vector-zip z/down z/right z/remove z/root)
:= {[1 2 [3 4]]}
(-> v z/vector-zip z/down z/right (z/edit * 42) z/root) ❸
:= {[1 84 [3 4]] [5 6]}
```

❶ `z/vector-zip` 和 `z/root` 一起配合构建了一个“事务”的边界。

❷ `z/remove` 把 `zipper` 移到深度优先遍历的前一个位置。

❸ 这里的 2 节点被乘以了 42。

这些几乎是你所需要知道的^{注 50} 关于 `zipper` 的所有 API。我们现在可以学学如何创建一个自定义的 `zipper`，并看看怎么把它用到我们的迷宫的例子中。

自定义 zipper

通用的 `zipper` 一般是利用 `zipper` 函数来创建的，它接受三个函数，以及要操作的树的根

注 50： 我们没有提到 `end?` 谓词，它可以用来测试一个深度优先遍历有没有遍历到底部。

节点作为参数：

- 一个谓词函数要判断一个节点是否可以有子节点。
- 一个函数你给它一个分支节点，它返回这个分支节点的子节点的序列。
- 一个函数你给它一个存在的节点以及一个序列的子节点，它给你返回一个分支节点。

我们来实现一个操作自定义数据的 zipper：用 vector 来表示 HTML 元素。vector 里面的第一个元素是 tag 的名字，第二个可以是一个包含所有属性的 map，剩下的是这个 tag 的子节点，我们把文本节点以字符串表示。因此，[:h1 "zipper"] 和 [:a {:href "http://clojure.org/" "Clojure"}] 都是合法的节点，这个数据例子已经足够让我们实现一个有趣的 zipper 了：

```
(defn html-zip [root]
  (z/zipper
    vector?
    (fn [[tagname & xs]]
      (if (map? (first xs)) (next xs) xs))
    (fn [[tagname & xs] children]
      (into (if (map? (first xs)) [tagname (first xs)] [tagname])
            children))
    root))
```

在通用的 zipper 函数基础之上，我们可以创建一些领域特定的函数。比如，wrap 来使得 ◀154 函数更适合用户使用：

```
(defn wrap
  "Wraps the current node in the specified tag and attributes."
  ([loc tag]
   (z/edit loc #(vector tag %)))
  ([loc tag attrs]
   (z/edit loc #(vector tag attrs %))))
(def h [:body [:h1 "Clojure"]
         [:p "What a wonderful language!"]])
 ;;= #'user/h
 (-> h html-zip z/down z/right z/down (wrap :b) z/root)
 ;;= [:body [:h1 "Clojure"] [:p [:b "What a wonderful language!"]]]
```

给嵌套的数据结构创建一个自定义的 zipper 并不复杂。但是我们也可以给这种嵌套的数据结构创建一个只读的^{注51}、不嵌套的 zipper —— 比如我们在 145 页“迷宫生成”一节介绍的迷宫。在这种情况下，问题的关键是传给 zipper 函数的参数是我们要处理的数据结构的闭包，而节点则是这个数据结构中数据的自然标识符。

注 51：read-write 理论上是可行的，但是会更复杂。

阿里阿德涅 zipper

说到迷宫和史诗般的冒险，让我们利用 zipper 作为阿里阿德涅的线索来帮助提修斯找到迷宫的出口如何？

首先，我们需要一个迷宫：

```
(def labyrinth (maze (grid 10 10)))
```

但是这个迷宫都是墙，而我们只对迷宫中的通道感兴趣——也就是那些缺失的墙：

```
(def labyrinth (let [g (grid 10 10)] (reduce disj g (maze g))))
```

让我们来定义一些角色：

```
(def theseus (rand-nth (distinct (apply concat labyrinth))))
(def minotaur (rand-nth (distinct (apply concat labyrinth)))))
```

在这里，`[(rand-int 10) (rand-int 10)]` 其实就够了，但是它会使得定义 `labyrinth` 的过程跟角色的坐标捆绑在一起。

第一眼看的时候可能觉得我们用一个地点坐标就可以确定提修斯在寻找牛头人过程中的坐标。但是，其实是不够的：我们需要记住提修斯是从哪里过来的，以知道它过来的方向。因此需要的是一个方向：要用两个坐标来表示。

因此，我们的阿里阿德涅 zipper 将会以方向作为节点：

```
155 > (defn ariadne-zip
  [labyrinth loc]
  (let [paths (reduce (fn [index [a b]]
    (merge-with into index {a [b] b [a]}))
    {} (map seq labyrinth))
    children (fn [[from to]]
      (seq (for [loc (paths to)          •
                :when (not= loc from)]
        [to loc])))
    (z/zipper (constantly true)          •
              children
              nil                      •
              [nil loc])))])
```

- ❶ zipper 文档中提到我们传入的 `children` 函数必须要返回一个 `seq` 类型，任何其他的顺序类型都不行，因此我们要对返回值调用 `seq` 函数。
- ❷ 我们使用 `(constantly true)` 来作为分支谓词，因为所有的坐标都可能通向另外其他的坐标。

③ 我们传入 nil 作为节点的工厂函数，因为这个 zipper 是完全只读遍历的：它不能进行任何更新操作。

❶ [nil loc] 是初始方向，是提修斯搜索的起点。

现在只要对这个迷宫进行一个深度优先遍历就能找到牛头人了：

```
(->> theseus
  (ariadne-zip labyrinth)
  (iterate z/next)
  (filter # (= minotaur (second (z/node %))))
  first z/path
  (map second))
([3 9] [4 9] [4 8] [4 7] [4 6] [5 7] [5 6] [5 5] [5 4]
 [5 8] [6 8] [6 7] [6 6] [6 5] [7 6] [8 6] [9 6] [9 5]
 [9 4] [9 3] [9 2] [9 1] [9 0] [8 2] [8 1] [8 0] [7 0]
 [6 0] [7 1] [7 2] [6 2] [6 1] [5 1] [4 1] [4 0] [5 0]
 [3 0] [4 2] [5 2] [3 2] [3 3] [4 3] [4 4] [4 5] [3 5])
```

一旦牛头人被干掉，提修斯就可以找到回到它起点的路径！

我们可以图形化这个故事，第一步是修改 draw 函数来接受一个额外的参数：从提修斯到牛头人的路径。

```
(defn draw
  [w h maze path]
  (doto (javax.swing.JFrame. "Maze")
    (.setContentPane
      (doto (proxy [javax.swing.JPanel] []
        (paintComponent [^java.awt.Graphics g]
          (let [g (doto ^java.awt.Graphics2D (.create g)
            (.scale 10 10)
            (.translate 1.5 1.5)
            (.setStroke (java.awt.BasicStroke. 0.4)))
            (.drawRect g -1 -1 w h)
            (doseq [[[xa ya] [xb yb]] (map sort maze)]
              (let [[xc yc] (if (= xa xb)
                [(dec xa) ya]
                [xa (dec ya)])]
                (.drawLine g xa ya xc yc)))
            (.translate g -0.5 -0.5)
            (.setColor g java.awt.Color/RED)
            (doseq [[[xa ya] [xb yb]] path]
              (.drawLine g xa ya xb yb))))))
        (.setPreferredSize (java.awt.Dimension.
          (* 10 (inc w)) (* 10 (inc h))))))
      .pack
      (.setVisible true))))
```

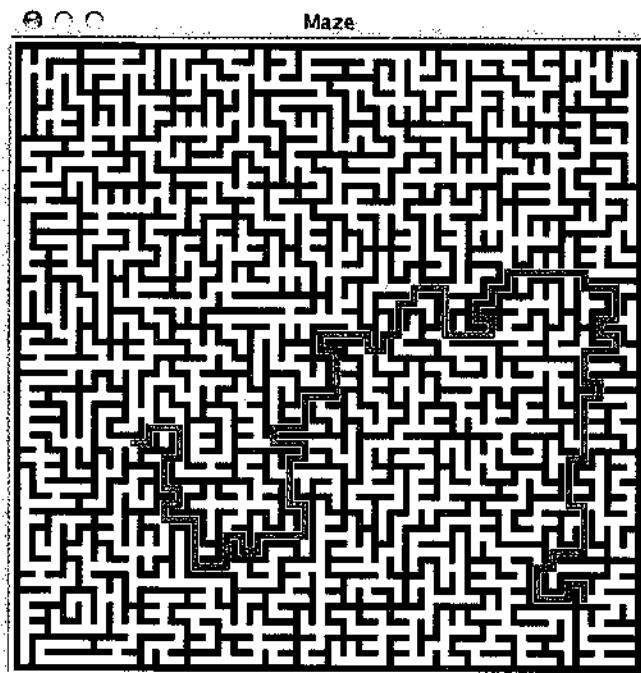
156

- ❶ 这里的这个 `path` 中保存的是一个个坐标对，所以它跟前面计算出来的只返回一个个坐标是不同的——(`map second`) 的功劳。

现在我们可以来讲述这个神话故事了，以 Clojure 的方式来描述并图形化：

```
(let [w 40, h 40
      grid (grid w h)
      walls (maze grid)
      labyrinth (reduce disj grid walls)
      places (distinct (apply concat labyrinth))
      theseus (rand-nth places)
      minotaur (rand-nth places)
      path (-> theseus
              (ariadne-zip labyrinth)
              (iterate z/next)
              (filter #(= minotaur (first (z/node %))))
              first z/path rest)]  
    ❷
  (draw w h walls path))
```

- ❶ 这里用 `rest` 代替了 (`map second`)，因为路径的第一个坐标跟其他的坐标不一样：[`nil theseus`]，因为它是起点，它前面没有别的坐标。



总结

◀ 157

Clojure 的集合抽象以及具体的数据结构的实现是整个语言的核心，它比 Clojure 的其他任何特性都更能代表这门语言的能力、特征以及世界观。理解这些集合的语义、原理以及习惯用法能够帮你从 Clojure 汲取最多有关函数式编程的营养，使你可以更好地理解 Clojure 中其他依赖集合的部分。

多线程和并发

编写多线程的程序是很多程序员要面对的难题。因为多线程程序的行为很难理解，执行的结果也是不确定的；对于同样的输入有可能会产生完全不同的输出，如果代码的执行顺序定义得有问题的话还可能导致竞争条件和死锁。而这些问题是很難检测到的，而且也是不容易解决的。

大多数的语言并没有给我们太多工具来驾驭多线程，大多数提供的只是线程和锁，而只使用这两个工具要编写出一个正确运行的多线程程序是很难的。锁应该以什么样的顺序获取和释放？一个读线程是否需要对一个资源加锁以防止其他线程来写这个资源？对于一个依赖锁的程序如何对它进行全面测试？这种复杂性很快就会失控，最后你只能去调试一个个竞争条件，而且这些竞争条件只在线上环境出现，这些死锁只在这台机器上会发生，而在那台机器上完全没有问题。

想象一下我们手里的这些工具是多么的底层吧，如果总是依赖这些线程啊、锁啊，以及那些苍白的关键字来对付多线程的复杂性的话，这种处境跟我们这些年一直极力倡导的更高效地开发、开发错误更少的软件努力相背离。Clojure 对于这个问题的提供手段是：

1. 就像我们在第 2 章里面讨论过的，尽量减少程序里面可变状态的使用，利用不可变的值、集合，以及它们所提供的值语义和高效的操作。
2. 当你确实需要在多线程的情况下对状态进行修改，那么对这些可变的状态进行隔离并且限制对这些可变值的修改方法。这些是 Clojure 中“引用类型”的基础，我们紧接着会进行讨论。
3. 如果实在是没有别的选择——你也愿意舍弃 Clojure 中的引用类型所提供的语义保证的好处，那么可以退一步去使用纯的锁、线程以及 Java 所提供的高质量的并发 API。

Clojure 没有让并发编程变得非常简单的银弹，但它确实提供了一些新颖且久经考验的工具，可以让并发编程更容易、写出来的代码更可靠。

计算在时间和空间内的转换

Clojure 提供了一些实体：`delay`、`future` 以及 `promise`，它们各自包装了一种控制并发计算在何时以及如何进行的用例。虽然只有 `future` 是唯一一个专注在并发上面的，但这三种实体都经常被用来帮助实现具体的并发语义。

delay

`delay` 是一种让一些代码延迟执行的机制，代码只会在显式地调用 `deref` 的时候执行：

```
(def d (delay (println "Running...")  
              :done!))  
 ;;= #'user/d  
(deref d)  
 ; Running...  
 ;;= :done!
```



`deref` 是由接口 `clojure.lang.IDeref` 定义的，任何实现这个接口的对象会扮演一个值的容器的角色。你可以通过调用 `deref` 来对它解引用以获取它的值，或者使用它的语法糖：`@`。^{注1} Clojure 中很多实体是可以解引用的，包括 `delay`、`future`、`promise` 以及所有的引用类型：`atom`、`ref`、`agent` 以及 `var`。我们会在本章对它们进行全面的介绍。

对于某些事情我们当然可以直接使用函数就可以实现了：

```
(def a-fn (fn []  
              (println "Running...")  
              :done!))  
 ;;= #'user/a-fn  
(a-fn)  
 ; Running...  
 ;;= :done!
```

但是，`delay` 提供了一些引人注目的优势。

161 delay 对于它所包含的代码只执行一次，然后把返回值缓存起来。后面所有使用 `deref` 对它进行的访问都是直接返回的，而不用再去执行它所包含的代码：^{注2}

注1： 我们几乎总是鼓励使用 `@foo` 而不是 `(deref foo)`，除非你需要在高阶函数里面使用解引用 `deref`（比如对一个序列里面的所有 `delay` 进行解引用），或者你想在解引用的时候指定一个超时，超时特性只有 `deref` 函数才提供。

注2： 而且，因此也不会由于多次执行 `delay` 所包含的代码而导致副作用。

```
@d  
:= :done!
```

这意味着在第一次对一个 delay 进行解引用的时候可以多个线程同时解引用，所有的线程都会被阻塞住直到 delay 所包含的代码被求值（只求值一次！），之后所有的线程就可以访问这个值了。

当你想要提供一个计算比较昂贵或者是一个非必需的数据，那就可以使用 delay 来作为一个优化的手段，这样，数据的使用者可以根据需要来获取这个数据。

示例4-1：通过delay来提供可选择的计算

```
(defn get-document  
  [id]  
  ; ... do some work to retrieve the identified document's metadata  
  ;...  
  {:url "http://www.mozilla.org/about/manifesto.en.html"  
   :title "The Mozilla Manifesto"  
   :mime "text/html"  
   :content (delay (slurp "http://www.mozilla.org/about/manifesto.en.html")))}❶  
  ;;= #'user/get-document  
(def d (get-document "some-id"))  
  ;;= #'user/d  
d  
  := {:url "http://www.mozilla.org/about/manifesto.en.html",  
   := :title "The Mozilla Manifesto",  
   := :mime "text/html",  
   := :content #<Delay@2efb541d: :pending>}❷
```

❶ 可以使用 delay 来非常方便地延迟一些昂贵的操作或者延迟获取一些非必需的数据。

❷ 被 delay 所包含的代码直到我们（或者代码的调用者）解引用它的值时才会被求值。

我们程序的某些部分可能只需要文档的基本信息，而根本用不到文档的内容。而另一方面，程序的另外一些部分则一定需要文档的内容才能工作，还有一些部分可能会使用文档的内容：如果文档的内容已经取到了的话。后面的这个用例可以实现是因为我们可以通过 realized? 函数来检测 delay 的内容是否已经获取到了：

```
(realized? (:content d))  
  := false  
 @(:content d)  
  := "<!DOCTYPE html><html>..."  
(realized? (:content d))  
  := true
```



注意，`realized?`也可以用在 `future`、`promise` 以及惰性序列上。

`realized?` 允许你检测 `delay` 的值有没有获取到，如果没有的话，可以选择不去获取这个值，因为这么做可能代价比较大，而你的代码可以暂时不用这个值。

future

在深入讨论更复杂的主题比如引用类型之前，Clojure 程序员经常会问：“我怎么启动一个新的线程来跑一些代码？”如果你确实需要这么做，可以使用 JVM 的原生线程（224 页的“使用 Java 的并发原语”一节有介绍），但是 Clojure 提供了更优雅、更易用的选择——`future`。

Clojure 的 `future` 会在另外一个线程里面执行它所包含的代码：^{注3}

```
(def long-calculation (future (apply + (range 1e8))))
;= #'user/long-calculation
```

对于 `future` 的调用会立即返回，它允许当前线程（比如你的 REPL）可以继续执行。线程执行的结果会保存在 `future` 里面，可以通过解引用来获取这个值：

```
@long-calculation
;= 4999999950000000
```

跟 `delay` 一样，如果在 `future` 的执行还没有完成的时候去解引用它的话，会阻塞当前的线程，因此下面的这个表达式会阻塞我们的 REPL 5 秒钟：

```
@(future (Thread/sleep 5000) :done!)
;= :done!
```

另外一个跟 `delay` 一样的点是，`future` 也会保存它所包含的代码的返回值，因此后续通过 `deref` 对 `future` 的访问会直接返回这个已经计算好的值。

跟 `delay` 不一样的是，在解引用一个 `future` 的时候，你可以指定一个超时时间以及一个“超时值”，“超时值”会在解引用超时的时候返回：^{注4}

```
(deref (future (Thread/sleep 5000) :done!)
      1000
      :impatient!)
;= :impatient!
```

注 3：Clojure 还提供了一个 `future-call` 函数，你可以用它来在另外一个线程中调用一个无参函数。

注 4：@ 语法糖不具备这个特性。

future 通常被用来简化一个程序的并发部分。比如假设我们知道示例 4-1 里面的 `get-document` 函数的所有用户都需要 `:content` 的值，如何去获取它呢？第一个想法也许就是在调用 `get-document` 的时候同步获取这个值，但是这会使得函数的每个调用者都要阻塞在上面，即使有的调用者并不马上需要这个值。更好的办法是使用一个 `future` 来异步获取 `:content` 的值，这使得我们在另外一个线程里面马上去获取这个值，而我们的调用者可以马上返回而不用阻塞在这个 I/O 上面。后面我们去获取 `:content` 的值的时候，即使这个值还没有获取到，但是调用者的阻塞时间肯定会短一点了（甚至可能已经获取回来了，因此不用等）。

```
(defn get-document
  [id]
  ; ... do some work to retrieve the identified document's metadata...
  {:url "http://www.mozilla.org/about/manifesto.en.html"
   :title "The Mozilla Manifesto"
   :mime "text/html"
   :content (future (slurp "http://www.mozilla.org/about/manifesto.en.html"))}) ❶
```

❶ 这里对于示例 4-1 的唯一改变是把 `delay` 换成了 `future`。

这并不需要我们对客户端代码做任何修改（因为客户端代码关心的只是解引用 `:content` 的值，但是到底解引用的是 `delay` 还是 `future` 它并不关心），但是如果调用者始终需要这个数据的话，那么这个小小的代码改变将会带来很大的吞吐量提升。

`future` 跟我们手动启动一个线程来执行代码相比有很多优势：

1. `future` 使用的是跟 `agent`（可能是阻塞的）共享的一个线程池里面的线程（我们会在 209 页的“agent”一节详细讨论）。这种在一个池子里面共享资源使得 `future` 比自己创建原生线程更高效。
2. 使用 `future` 比创建并启动一个原生的线程更简洁。
3. Clojure 的 `future`（`future` 函数的返回值）是类 `java.util.concurrent.Future` 的对象，因此更容易跟相关的 Java API 交互。

promise

`promise` 跟 `delay` 和 `future` 有很多相似之处：一个 `promise` 可以被解引用，解引用的时候可以传入一个超时的参数，解引用的时候如果这个 `promise` 的值还没有，那么解引用的代码会阻塞直到这个值准备好。但是，`promise` 跟 `delay` 和 `future` 还是有着本质的区别，因为 `promise` 在创建的时候并不会指定任何代码或者函数来最终产生出它们的值：

```
(def p (promise)
  := #'user/p)
```

164 ➤ promise 最开始的时候是一个空的容器；当程序运行的某个时间点，这个 promise 会被填充入一个值——通过 deliver 函数：

```
(realized? p)
:= false
(deliver p 42)
:= #<core$promise$reify_1707@3f0ba812: 42>
(realized? p)
:= true
@p
:= 42
```

因此，promise 跟一个一次性的、单值的管道类似：数据在管道的一端通过 deliver 函数传入，然后在管道的另外一端通过 deref 函数来获取值。这种东西有时候被称为“数据流变量”，而且是进行“声明式并发编程”的基础构件。这是多个并发进程之间显式的定义关系，从而使得这个整个程序的陈述式结果会根据需要进行计算——一旦输入数据准备好了，那么并发的结果就是确定的。下面这个简单的例子会牵涉到三个 promise：

```
(def a (promise))
(def b (promise))
(def c (promise))
```

我们通过下面的这个 future 来定义三个 promise 之间的关系。这个 future 先获取 promise:a、b 的值，计算它们的和，然后再 deliver 给 promise C。

```
(future
  (deliver c (+ @a @b))
  (println "Delivery complete!"))
```

在这种情况下，c 的值直到 a 和 b 的值都有了才会被传入（也就是说 realized?），直到那个时候，这个 future 对于 c 的 deliver 操作会阻塞在 a 和 b 上。注意，在这种情况下，如果你不加超时地去解引用 c 的话，会使你的 REPL 无限地阻塞。

在数据流编程的大多数场景下，其他的工作线程最终会计算出要传递给 a 和 b 的值。我们可以通过 REPL 来模拟给它们传递值：^{注5}一旦 a 和 b 被传递了值，future 的阻塞就会被解除，因此也会使得 c 的值被传递：

```
(deliver a 15)
:= #<core$promise$reify_5727@56278e83: 15>
(deliver b 16)
; Delivery complete!
:= #<core$promise$reify_5727@47ef7de4: 16>
@c
:= 31
```

注5：从技术上来说，确实是在另一个线程里面！



这意味着 (`deliver p @p`) 会无限阻塞，这里 `p` 是一个 promise。

但是，被这样阻塞的 promise 并不是被彻底锁住了，我们可以手动给它们解锁：

```
(def a (promise))
(def b (promise))
(future (deliver a @b))         ❶
(future (deliver b @a))         ❷
(realized? a)                  ❸
:= false
(realized? b)
:= false
(deliver a 42)                 ❹
:= #<core$promise$reify__5727@6156f1b0: 42>
@a
:= 42
@b
:= 42
```

❶ 这里使用 `future` 以让它们不要阻塞住 REPL。

❷ `a` 和 `b` 的值还没有被传递。

❸ 给 `a` 传递一个值会使得我们被阻塞住的操作恢复——显然 (`deliver a @b`) 会失败 (返回 `nil`)，但是 (`deliver b @a`) 会正确执行。

对于 promise 的一个最直接的应用是让基于回调的 API 以同步的方式执行。假定你有一个函数，这个函数接受另外一个函数作为回调：

```
(defn call-service
  [arg1 arg2 callback-fn]
  ; ...perform service call, eventually invoking callback-fn with results...
  (future (callback-fn (+ arg1 arg2) (- arg1 arg2))))
```

如果我们想在这个函数体里面马上使用这个回调函数的返回值的话，可能需要利用很多不同（不那么让人满意的）的手段来等待这个回调函数执行完毕。更好的办法是利用 promise 在调用 `deref` 时候的阻塞特性来强制实现同步语义。假定你所感兴趣的所有的函数都把它们的最后一个参数作为回调函数，那么就可以实现一个通用的高阶函数：

```
(defn sync-fn
  [async-fn]
  (fn [& args]
    (let [result (promise)]
      (apply async-fn (conj (vec args) #(deliver result %&)))
      @result)))
```

```
((sync-fn call-service) 8 7)
;= (15 1)
```

简单的并行化

我们会介绍 Clojure 里面的所有并发组件，其中之一就是 agent——可以利用它来很高效地完成并行化计算任务。但是有时候你会发现你想以尽量简单的代码来并行化一些操作。

并行 vs. 并发

并行和并发听起来很像是一样的意思，其实不然，让我们来区分一下。

并发是对于多个、通常会交互执行的、对于一些共享资源进行访问和修改的线程的协调。

并行也会牵涉到状态，但是通常是相反的。它被当做是一种优化的手段来更高效地利用我们所拥有的资源（通常是计算能力，但是有时候也会是一些资源，比如网络带宽）来提高某个操作执行的性能，程序越并行化，那么代码互相独立、互不干扰地执行的（不访问、修改一些共享的状态）的时间就越长，线程之间协调所带来的开销也就越小。不像并发的线程之间会交互执行，平行的多个线程、进程之间是完全独立地执行，有时是在一个 CPU 的多个核上执行，有时则完全是在不同的物理机器上执行。

Clojure 的序列抽象的灵活性^{注6}使得我们可以把很多问题简单地抽象成对于序列的处理。比如，假定有一个函数利用正则表达式从一个字符串中抽取出其中包含的电话号码：

```
(defn phone-numbers
  [string]
  (re-seq #"(\\d{3})[\\.\\-]?\\d{3}[\\.\\-]?\\d{4}" string))
;= #'user/phone-numbers
(phone-numbers "Sunil: 617.555.2937, Betty: 508.555.2218")
;= ([["617.555.2937" "617" "555" "2937"] ["508.555.2218" "508" "555" "2218"]])
```

这段代码非常简单，我们把它应用到任何字符串的序列上去，它都会非常简单、快速、高效。这些字符串的序列可以是我们利用 slurp 和 file-seq 从磁盘读出来的，或者是从消息队列里面读取出来的消息，或者是从数据库里面读出来的一段文本。为了让例子保持简单，我们这里创建 100 个空的字符串，每个字符串的大小大约为 1MB，电话号码则

^{注6}：我们在 89 页的“序列”一节已经讨论过了。

在每个字符串的末尾：

```
(def files (repeat 100
  (apply str
    (concat (repeat 1000000 \space)
      "Sunil: 617.555.2937, Betty: 508.555.2218"))))
```

167

让我们来看一下，要从这些“文件”里面把所有的电话号码找出来需要花多长时间：

```
(time (dorun (map phone-numbers files))) ①
; "Elapsed time: 2460.848 msecs"
```

① 我们这里使用 `dorun` 来彻底实例化由 `map` 所产生的惰性序列，并且不保持实例化所产生的值，因为我们不想让所有的电话号码都打印到 REPL 上去。

不过，这里其实是可以并行的，而且非常简单。`map` 函数有一个并行的版本：`pmap`——它会以并行的方式用多个线程来把一个函数应用到一个序列的元素上去，而且跟 `map` 一样，它会返回一个惰性序列：

```
(time (dorun (pmap phone-numbers files)))
; "Elapsed time: 1277.973 msecs"
```

在一个双核机器上运行这段代码的话，它的效率差不多是前面 `map` 版本的两倍；对于这个特定的程序和数据集来说，如果你在一个四核的机器上运行的话，性能差不多可以达到 4 倍。而我们唯一的修改只是改了一个字母！虽然这看起来很神奇，其实不然：`pmap` 内部只是简单地使用了一些 `future`——`future` 的个数就是机器的 CPU 核心的个数——来并行地调用 `phone-numbers` 来处理字符串序列中的每一个字符串。

这种优化方式对于很多操作是可行的，但是还是需要谨慎地使用 `pmap`。使用它是有一些额外开销的。如果被并行的操作本身执行的时间不是很长，那将会使得线程之间协调的开销相对程序真正运行时间来说相当可观，这会使得你使用 `pmap` 的程序反而比使用普通 `map` 的版本更慢：

```
(def files (repeat 100000
  (apply str
    (concat (repeat 1000 \space)
      "Sunil: 617.555.2937, Betty: 508.555.2218"))))

(time (dorun (map phone-numbers files)))
; "Elapsed time: 2649.807 msecs"
(time (dorun (pmap phone-numbers files)))
; "Elapsed time: 2772.794 msecs"
```

这里我们唯一的改变是对上面的数据做了一些修改：在这个例子中，每个字符串只包含大概 1KB 的数据。虽然总的数据量没有变化（这里“文件”更多了），但是这里线程间协调的开销相对于程序真正运行的时间来说变得很大。由于这样的开销，通常在一个 N

核的机器上面使用 `pmap` 并不能得到 N 倍的性能提升。这里的经验教训很清楚：当你的计算任务本身是可以并行的时候才使用 `pmap`，并且序列里面的每个值处理所需要的时间比线程之间协调所需要的时间来说要大很多。如果你在不适合的地方使用 `pmap` 的话，结果可能适得其反。

对于这样的场景，通常有一个缓解的方案，可以把一些小的计算任务组合成一个较大的计算单元，然后再实施并行化的优化手段。在上面的这个例子中，每个计算单元的文本只有 1KB，但是可以通过组合把每个计算单元的数据量提升，让 `pmap` 处理的每个单元变成 250 个 1KB 的字符串，从而提升每个 `future` 的工作量，减少线程之间并行化的开销：

```
(time (-> files
  (partition-all 250)
  (pmap (fn [chunk] (doall (map phone-numbers chunk)))) ❶
  (apply concat)
  (dorun))
; "Elapsed time: 1465.138 msecs"
```

❶ `map` 会返回一个惰性序列，因此使用 `doall` 来强制惰性序列里面的元素在传递给 `pmap` 之前实例化。否则，`phone-numbers` 函数将根本不会在并行的线程中被调用，而是在最后使用这个惰性序列的线程中被调用，也就没有达到我们要并行化的目的。

通过改变计算单元的大小，我们又重新获得了并行化的好处，虽然对于每个具体字符串的处理复杂度已经简单很多了。

`Clojure` 里面还提供了另外两个基于 `pmap` 的并行化手段：`pcalls` 和 `pvalues`。前者接受任意数量无参函数作为参数，返回一个包含它们的返回值的惰性序列；后一个则是一个宏，做的事情是一样的，但是它接受的参数是任意数量的表达式。

状态和标识

在 `Clojure` 中，“状态”和“标识”有着本质的区别。这两个概念在很多语言里面都被当做同一个概念处理，比如说：

```
class Person {
    public String name;
    public int age;
    public boolean wearsGlasses;

    public Person (String name, int age, boolean wearsGlasses) {
        this.name = name;
        this.age = age;
        this.wearsGlasses = wearsGlasses;
    }
}
```

```
}Person sarah = new Person("Sarah", 25, false);
```

看起来没有什么不对吧？只是一个有着一些字段的普通 Java 类^{注7}，我们可以创建这个类的一些实例。实际上，这里有很多问题。

169

我们这里创建了一个表示 "Sarah" 的 Person 对象，她 25 岁。随着时间的推移，Sarah 会有很多的状态：作为一个小孩的 Sarah，作为一个少年的 Sarah，作为一个成人的 Sarah。在每一个时间点——比如说上个星期二上午 11 点零 7 分，Sarah 有一个状态，而且任何一个状态是不会发生改变的。要修改 Sarah 的任何一个状态都是没有意义的。她上个星期二的状态不会在星期三发生变化；她的状态可能随着时间的不同而不同，但是过去时间的状态是不可能发生任何改变了一——成为历史了。

但不幸的是，我们这里的 Person 类以及底层的引用（实际上就是指针）并不适合用来表示即使是这么简单的一个东西——但是却是很基础的一个概念。假定现在 Sarah 要变成 26 岁了，我们唯一的办法是对表示 Sarah 25 岁的那个对象进行修改：^{注8}

```
sarah.age++;
```

更糟糕的是，如果需要一次性对 Sarah 的多个状态字段进行修改会怎么样？

```
sarah.age++;
sarah.wearsGlasses = true;
```

在这两行代码执行之间的任意一个时间点，Sarah 的年龄递增了，但是她还没有戴眼镜。对于这么一段时间内（技术上来说，由于现代处理器的架构以及语言运行时的操作方式，这段时间到底有多长是不确定的），Sarah 处在一个不一致的状态内，而这个状态对于对象模型来说可能在语义上都是不可能的。就是这种事情造成了所谓的“竞争条件”，也是我们在其他语言中要使用那很容易出错的锁机制的动机。

甚至可以把这个用来表示 sarah 的对象指向另外一个完全不同的人：

```
sarah.name = "John";
```

这是很麻烦的。这里的这个 sarah 对象并不表示 Sarah 的某个时间点的状态，而且也不标识 Sarah 这个人。而是状态和标识两个概念的不良混合物。我们不能对 Person 的状态做任何放心的依赖，因为它的实例的状态在任何一个时间都会发生变化（只要有多个线程就会对这个对象进行访问和修改），不是说很容易把一个对象的状态搞得不一致，这就是它默认的行为。

注 7：这里的讨论不单指 Java。实际上绝大多数——几乎所有的语言都把状态和标识混在一起，包括 Ruby、Python、C#、Perl、PHP 等。

注 8：不要纠结在没有字段访问的函数 (getter, setter) 上面；你到底是直接使用字段，还是通过 getter、setter 来使用这个字段不会对语义有任何影响。

170 > Closure 的方式。其实我们真正想要的是一个表示 Sarah 这个人的标识，不是她在某个特定时间点的状态，而是在整个时间序列中，表示她的一个逻辑实体。更进一步说，我们希望标识在任何时间点上都有一个特定的状态，而每个状态的变更都不会对已有的状态造成影响。再回想一下我们在 52 页“谈谈值的重要性”一节提到的可变的对象和不可变的值之间的对比，这种对于标识和状态的区分会给我们带来很多好处，而且语义上也更通顺。总之，我们除了想保证一个实体的内部状态始终是一致的（通过不可变的值来保证），还想很简单地、安全地引用 Sarah 在上星期二或者去年的状态。

跟大多数语言中的对象不一样，Closure 中的数据结构是不可变的。这使得它们是表示状态的理想数据结构：

```
(def sarah {:name "Sarah" :age 25 :wears-glasses? false})  
= #'user/sarah
```

保存在 `sarah` 这个 var 里面的 map 是 Sarah 在某个特定时间点的状态。因为这个 map 是不可变的，我们可以放心：任何保持了对这个 map 的引用的代码都可以安全地使用它而不用担心有人会对这个 map 进行任何修改。var 本身是 Closure 提供的引用类型之一。从本质上来说，它是一种有明确定义的并发特性以及修改语义的、可以包含任何值的一个容器，Closure 中用它来表示稳定的标识。所以我们可以这么说，Sarah 是由 var `sarah` 来标识的，而 Sarah 的状态可能随着时间的推移而发生变化。

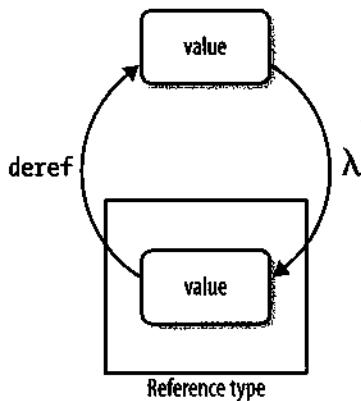
这里我们简单介绍了一下 Closure 是如何处理状态和标识的，以及它们随着时间的变化而表现出来的完全不同的语义，这一点值得关注。^{注9}这一章的其余部分我们会专注于介绍 Closure 如何区分状态和标识的技术细节。从大的方面来说，我们将会介绍 Closure 的 4 种引用类型，每一种实现的是什么样的语义以及状态如何随着时间变化。加上 Closure 对于不可变值的强调，这些引用类型以及它们的语义使得我们可以设计并发程序，并且让这个并发程序最大限度地利用已有硬件的最大计算能力，同时还能够避免使用线程和锁可能会带来的一系列 bug。

Closure 的引用类型

在 Closure 中有 4 种引用类型可以用来表示标识：`var`、`ref`、`agent` 以及 `atom`。所有引用类型都有各自不同的地方，这里我们先讨论一下它们的共通点。

171 > 从最基本的层面上来说，引用类型都是包含其他值的容器，容器里面的值可以利用某些函数进行修改（不同类型使用的修改函数也不一样）：

注9：Rich Hickey 在 2009 年作了一个有关标识、状态、时间的演讲，并且介绍了它们如何影响 Closure 的设计。强烈推荐你看一下这个演讲的资料：<http://www.infoq.com/presentations/Are-We-There-Yet-Rich-Hickey>。



所有的引用类型总是包含某个值（即使这个被包含的值是 nil），访问这个值的语法是一样的：使用 deref 或者 @：

```
@(atom 12)
:= 12
@(agent {:c 42})
:= {:c 42}
(map deref [(agent {:c 42}) (atom 12) (ref "http://clojure.org")
(var +)])
:= ({:c 42} 12 "http://clojure.org" #<core$_PLUS_clojure.core$_PLUS_@65297549>)
```

解引用会返回 deref 被调用时这个引用的状态的一个快照。这并不意味着为了获得这个快照我们要做任何拷贝的工作，操作这个返回的状态跟操作 Clojure 集合类似，因为这个快照是不可变的值，但是引用的状态在我们获取这个快照之后是可以变化的。

关于 deref 有一个很重要的保证就是，对引用类型调用 deref 绝对不会阻塞——不管要解引用的引用类型的语义，也不管是不是在另外一个线程里面正对这个引用进行操作。类似的，对一个引用类型解引用绝不会跟其他的操作相互干扰。这个跟 delay、promise 以及 future 形成了鲜明的对比——你对它们解引用的时候，如果它们的值还没有实例化好的话，是有可能会阻塞的。而在其他大多数语言里面也是类似的：读线程会被写线程阻塞或者写线程会被读线程阻塞。

设置一个引用类型的值则是更加细致的事情了。每一种引用类型都有它自己对于状态变化的语义，它们各自有自己的一套函数来根据语义实施这些变化。本章余下的部分会讨论这些对应的语义以及相关函数。

这些引用类型除了都可以被解引用，还可以：

- 修饰一些元数据（参看 153 页的“元数据”一节）。跟普通的值不同的是，我们不能使用 `with-meta` 或者 `vary-meta`，要对引用类型的元数据进行修改必须使用 `alter-meta!`，它会直接修改引用类型的元数据——而不是返回一个新的引用类型。^{注 10}
- 我们可以指定一些函数，这样在它们的状态发生变化时会调用指定的函数以使我们得到通知；这些函数被称为“观察器”，会在 176 页的“观察器”一节详细讨论。
- 我们可以指定引用类型的状态必须符合的条件，甚至可以阻止对于状态的改变——使用函数（会在 178 页“校验器”一节详细介绍）。

并发操作的分类

在讨论 Clojure 的引用类型时，我们会经常碰到一些关键的概念，这些关键的概念可以用来对并行的操作进行分类。把这些概念放在一起可以帮助我们更清楚地理解每种类型最适合在什么样的场合下使用。

协调。一个“协调”的操作是指为了产生正确的结果，这个操作里面的多个角色必须互相合作（或者至少不要相互干扰）。一个典型的例子是银行转账的操作：从一个账户转钱到另外一个账户的过程必须保证这些转入的账户的金额必须在转出账户的金额划掉之后才能增加，而且如果转出账户没有足够的钱，那么整个过程必须失败。而且与此同时很多其他进程可能也会对这个账户进行操作，如果没有合适的方法对这些操作进协调的话，那么这个账户在某个时间内的金额可能就不对了，本来应该失败的转账可能会成功，而本来应该成功的转账可能会失败。

作为对比的是，一个“无须协调”的操作是指一个过程里面的多个角色不可能相互影响，因为它们操作完全独立的数据。比如两个不同的进程可以非常安全地对磁盘上两个不同的文件进行读写，而完全不用担心最后的结果会出错。

同步。“同步”操作是指调用线程会等待、阻塞或者睡眠，直到它获得对于指定上下文的独占访问，而“异步”操作是指调用线程不用阻塞在一个调用上面，它可以继续去做别的事情。

只需要这两个概念（或者四个，如果你把另外两个相对的概念也算上的话）就可以把你可能遇到的一些（如果不是大多数的话）并发操作进行分类。因此也很自然的，Clojure 中的引用类型被设计来实现这两个概念的各种组合，我们可以非常方便地根据它们各自适合的并发场景对这几种引用类型进行分类：^{注 11}

注 10：`atom`、`ref` 以及 `agent` 都接受一个额外的 `:meta` 关键字参数，允许你在创建引用类型的时候就提供一个初始的元数据 `map`。

注 11：`var` 并不能适配到这张图中来，因为它的主要用途是线程内对于状态的修改，因此也就跟我们这里说的“协调”或者“同步”扯不上关系了。

		coordinated	uncoordinated
synchronous	Refs	Atoms	
	Agents		

当为某个特定的问题来选择引用类型的时候，回忆一下这个分类；如果你可以把这个问题根据我们说的两个特征进行分类，那么到底使用哪种引用类型就很明显了。 ◀173



你可能已经注意到了，Clojure 的引用类型中没有实现“协调的”并且“异步的”语义。这种组合在分布式系统里面更常见，比如最终一致性的数据库通常只保证在一段时间之后所有对于状态的修改会合并到最终状态中去。而相对而言，Clojure 所感兴趣的是解决进程内的并发和并行问题。

一些帮助函数

为了加深对于并发的理解，在这一章的例子里面会使用一些帮助函数（宏）。`futures` 是一个宏，它接受表达式作为参数，返回 `n` 个 `future`，这里的 `n` 是传入的表达式的个数：

```
(defmacro futures
  [n & exprs]
  (vec (for [_ (range n)
             expr exprs]
         `(future ~expr))))
```

这给我们提供了一个简单的方法来在不同的线程里面对表达式求值。但是 `futures` 这个形式本身会被求值成它所创建的这些 `future` 的一个 `vector`。这在某些场景下可能是比较方便的，但是有时候我们就是想所有这些 `future` 都执行完毕，以使得我们可以肯定对所有的表达式都完全求值了。所以这里设计另外一个帮助函数：`wait-futures`，它的作用跟 `futures` 一样，但是它始终返回 `nil` 并且阻塞我们的 REPL 直到所有的 `future` 都实例化了：

```
(defmacro wait-futures
  [& args]
```

```
`(doseq [f# (futures ~@args)]
        @f#))
```

我们还没有太多地介绍宏，所以这里如果你不是很理解这些帮助函数的原理的话，没有关系，我们会在第 5 章详细介绍宏。

174 原子类型 (Atom)

原子类型是 Clojure 最基本的引用类型，它实现的是同步的、无须协调的、原子的“先比较再设值”的修改策略。因此，对于原子类型的修改操作都要阻塞直到这个修改完成，每一个修改操作都是完全隔离的——自动隔离的，没有办法协调对于两个 atom 的修改。

我们使用 atom 函数来创建 atom.Swap！则是最常见的对于原子类型的修改操作，它接受要修改的原子类型，一个函数以及一些额外的参数作为参数，它把当前这个原子类型的状态修改成对传入函数的返回值：

```
(def sarah (atom {:name "Sarah" :age 25 :wears-glasses? false}))
;= #'user/sarah
(swap! sarah update-in [:age] + 3)                                     ❶
;= {:age 28, :wears-glasses? false, :name "Sarah"}                      ❷
```

- ❶ 这里，当 swap! 调用返回的时候，sarah 这个原子类型的值会被更新成表达式 (update-in @sarah [:age] + 3) 的结果。
- ❷ swap! 总是返回原子类型的更新值。

原子类型是对于 Sarah 那个例子需要的最基本的引用类型：每一个对原子类型的修改都会原子地发生，所以我们可以非常安全地对原子类型的值调用任何函数，或者多个函数同时来对它进行修改。你可以放心，没有其他线程会看到这个原子类型在值被更改过程中的某个中间状态（值）：

```
(swap! sarah (comp #(update-in % [:age] inc)
                      #(assoc % :wears-glasses? true)))
;= {:age 29, :wears-glasses? true, :name "Sarah"}
```

有一件你必须要记住的事情是，swap! 的语义是先比较旧的值是否匹配，然后再设新值，所以如果原子类型的值在你的更新函数返回之前发生了变化（可能是被其他线程修改了），那么 swap! 会自动进行重试，再次以你的原子类型的新值来调用我们传入的函数。swap! 函数会一直重试，直到设值成功：

```

(def xs (atom #{1 2 3}))
:= #'user/xs
(wait-futures 1 (swap! xs (fn [v]
    (Thread/sleep 250)
    (println "trying 4")
    (conj v 4)))
(swap! xs (fn [v]
    (Thread/sleep 500)
    (println "trying 5")
    (conj v 5))))
:= nil
; trying 4
; trying 5
; trying 5
@xs
:= #{1 2 3 4 5}

```

- ❶ 这里想把 5 加入集合 xs 的线程需要重试，因为在它睡眠的时候，其他线程已经改变了集合（把 4 加入这个集合的线程），因此这个线程在第一次尝试修改的时候失败了。 175

我们可以这样图示 swap! 的重试语义。

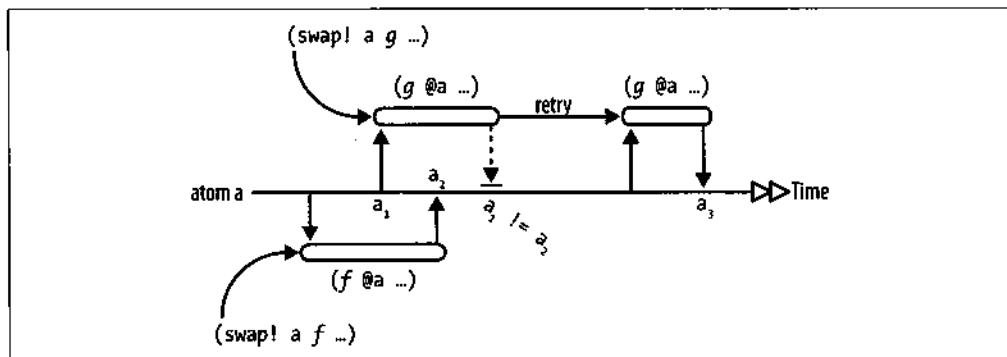


图4-1：在一个共享的原子类型上面相冲突的几个 swap! 之间的交互过程

如果原子类型 a 的值在我们调用函数 g 之后，在它返回之前发生了改变，那么 $swap!$ 会丢弃这个计算出来的新值，然后以这个原子类型的新值重新调用一遍这个函数，这个过程会一直持续，直到函数 g 在设置 a 的新值的时候发现没有别的线程对 a 的值做过修改。

我们没有办法对 $swap!$ 的重试语义进行任何控制，因此你提供给 $swap!$ 的函数必须是纯函数，否则程序执行的结果将很难预料。

作为一个同步的引用类型，修改 atom 值的函数会阻塞直到修改完成：

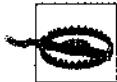
```
(def x (atom 2000))
:= #'user/x
(swap! x #(Thread/sleep %))      ①
:= nil
```

❶ 这个表达式至少需要 2 秒才会返回。

Clojure 还提供了一个简单版本的 compare-and-set! 操作，如果你已经知道要修改的原子类型当前的值是什么的话，可以使用这个函数；如果修改成功的话，这个函数会返回 true：

```
(compare-and-set! xs :wrong "new value")
:= false
(compare-and-set! xs @xs "new value")
:= true
@xs
:= "new value"
```

176



compare-and-set! 不使用值语义，它要求 atom 的值跟你传给它的第二个参数必须一样 (`identical?`)¹²：

```
(def xs (atom #{}))
:= #'user/xs
(compare-and-set! xs #{} "new value")
:= false
```

最后，我们有一个“终极武器”：如果你想对 atom 的状态进行设置而不管它现在的值究竟是什么，可以使用 `reset!`：

```
(reset! xs :y)
:= :y
@xs
:= :y
```

现在我们了解了 atom，再去看看另外两个所有引用类型都支持的特性，我们后面的例子会用到它们。

通知和约束

我们已经学习过 Clojure 的引用类型（参见 170 页）的一个共同的特性——解引用，它允许我们可以获取引用类型的当前值。引用类型还有另外两个共同的特性：分别让我们可以对引用的状态进行监控，对要赋给引用的新的状态的合法性进行校验。Clojure 的引用类型为这两件事情提供了回调，分别以“观察器”和“校验器”的形式提供。

¹² 这是由 `identical?` 定义的；详情请查看 433 页的“对象相同 (`identical?`)”一节。

观察器

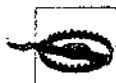
观察器是在引用的状态发生改变的时候会被调用的函数。如果你对于“观察者”设计模式很熟悉的话，那么一定能马上想到 Clojure 的观察器的使用场景。不过 Clojure 中的观察器要比设计模式中的“观察者”通用很多：Clojure 中的观察器只是一个函数而已，你不需要去实现某个特定的接口，而且这个通知的机制也是 Clojure 现成的，不用你自己实现。

所有的引用类型一开始都是没有观察器的，可以在任何时间给它注册一个观察器或者移除一个观察器。一个观察器函数必须接受 4 个参数：key、发生改变的引用（一个 atom、ref、agent 或者 var）、引用的旧状态以及现在的新状态：

```
(defn echo-watch
  [key identity old new]
  (println key old "=>" new))
;= #'user/echo-watch
(def sarah (atom {:name "Sarah" :age 25}))
;= #'user/sarah
(add-watch sarah :echo echo-watch)
;= #<Atom@418bbf55: {:name "Sarah", :age 25}>
(swap! sarah update-in [:age] inc)
; :echo {:name Sarah, :age 25} => {:name Sarah, :age 26}
;= {:name "Sarah", :age 26}
(add-watch sarah :echo2 echo-watch)
;= #<Atom@418bbf55: {:name "Sarah", :age 26}>
(swap! sarah update-in [:age] inc)
; :echo {:name Sarah, :age 26} => {:name Sarah, :age 27}
; :echo2 {:name Sarah, :age 26} => {:name Sarah, :age 27}
;= {:name "Sarah", :age 27}
```

177

- ① 我们的观察器函数在每次原子类型状态发生改变的时候把它的状态打印到控制台。
- ② 如果我们以一个新的 key 注册一个相同的观察器函数上去的话
- ③ 这样对于每个状态变化，这个函数会被调用两次。



因为观察器函数是在产生这个修改的线程上同步调用的，而此时另外一个线程可能又对这个引用进行了修改。因此观察器函数里面的代码应该使用这个传入的“旧值”和“新值”，而不应该手动去对引用进行解引用。

我们在调用 add-watch 的时候传入的 key 可以用来移除一个观察者：

```
(remove-watch sarah :echo2)
;= #<Atom@418bbf55: {:name "Sarah", :age 27}>
```

```
(swap! sarah update-in [:age] inc)
; :echo {:name Sarah, :age 27} => {:name Sarah, :age 28}
;= {:name "Sarah", :age 28}
```

注意，这些观察器函数在引用类型每次被修改的时候都会被调用，但是它不保证调用的时候引用的状态确实发生了变化：

```
(reset! sarah @sarah)
; :echo {:name Sarah, :age 28} => {:name Sarah, :age 28}
;= {:name "Sarah", :age 28}
```

因此我们经常在代码中对新值和旧值进行比较，如果确实不一样我们才进行一些处理。

一般来说，观察器函数对于使得本地改变可以即时通知到其他引用或者其他系统来说非常方便。比如，我们可以利用这个特性来很简单地实现一个记录引用所有历史状态的功能：

```
(def history (atom ()))

(defn log->list
  [dest-atom key source old new]
  (when (not= old new)
    (swap! dest-atom conj new)))
```

178 ➤

```
(def sarah (atom {:name "Sarah", :age 25}))
;= #'user/sarah
(add-watch sarah :record (partial log->list history))      ❶
;= #<Atom@5143f787: {:age 25, :name "Sarah"}>
(swap! sarah update-in [:age] inc)
;= {:age 26, :name "Sarah"}
(swap! sarah update-in [:age] inc)
;= {:age 27, :name "Sarah"}
(swap! sarah identity)                                     ❷
;= {:age 27, :name "Sarah"}
(swap! sarah assoc :wears-glasses? true)
;= {:age 27, :wears-glasses? true, :name "Sarah"}
(swap! sarah update-in [:age] inc)
;= {:age 28, :wears-glasses? true, :name "Sarah"}
(pprint @history)
;= ;;= nil
;= ;  {:age 28, :wears-glasses? true, :name "Sarah"}
;= ;  {:age 27, :wears-glasses? true, :name "Sarah"}
;= ;  {:age 27, :name "Sarah"}
;= ;  {:age 26, :name "Sarah"})
```

❶ 这里使用 `partial` 来把观察者函数要写日志历史的地方传递进来。

❷ 因为 `identity` 函数始终直接返回的是传入的参数，因此这里 `swap!` 函数会导致引用

状态发生变化，但是新值和旧值是一样的。`log->list` 只会在新值跟旧值不一样的时候才会记录这个状态，因此那些“重复”的状态是不会出现在历史状态列表里面的。

取决于对观察器函数的处理方式，你还可以根据注册的 `key` 的不同而做出不同的处理。一个简单的例子就是我们可以添加一个把历史版本记入数据库而不是内存的观察器函数：

```
(defn log->db
  [db-id identity old new]
  (when (not= old new)
    (let [db-connection (get-connection db-id)]
      ....))

(add-watch sarah "jdbc:postgresql://hostname/some_database" log->db)
```

我们会在 215 页“利用 `agent` 来记录引用状态的变更日志”一节把观察器函数、`ref` 以及 `agent` 结合起来使用，效果非常好。

校验器

校验器使你可以以任何想要的方式对引用的状态进行控制。校验器函数是一个只接受一个参数的函数，它在新状态被设置到引用之前被调用。如果校验器返回逻辑 `false` 或者抛出一个异常的话，那么这个状态修改会失败并且抛出一个异常。

179

这里说的对引用的修改可以是你对引用调用的任何修改函数，比如，我们对 `sarah` 这个 map 的 `:age` 字段进行递增，但是在 `swap!` 把这个新状态设置到引用里面之前校验器会被调用对新值进行校验。

```
(def n (atom 1 :validator pos?))
 ;;= #'user/n
 (swap! n + 500)
 ;;= 501
 (swap! n - 1000)
 ;;= #<IllegalStateException java.lang.IllegalStateException: Invalid reference state>
```

因为校验器函数只接受一个参数，因此可以使用 Clojure 中很多已经存在的谓词来作为校验器，比如 `pos?`。

虽然所有的引用类型都可以关联一个校验器，但是只有原子类型、`ref` 和 `agent` 可以在创建的时候通过 `:validator` 关键字参数直接指定一个校验器。如果要给一个 `var` 添加一个校验器，或者要改变 `atom`、`ref` 或者 `agent` 的校验器，可以使用 `set-validator!` 函数：

```
(def sarah (atom {:name "Sarah" :age 25}))
 ;;= #'user/sarah
```

```
(set-validator! sarah :age)
:= nil
(swap! sarah dissoc :age)
:= #<IllegalStateException java.lang.IllegalStateException:Invalid reference state>
```

可以抛出特定的异常来使我们知道为什么修改状态会失败，而不是简单地返回 `false` 或者 `nil`：^{注13}

```
(set-validator! sarah #(or (:age %)          ❶
                           (throw (IllegalStateException. "People must have `:age`s!"))))
:= nil
(swap! sarah dissoc :age)
:= #<IllegalStateException java.lang.IllegalStateException:People must have `:age`s!>
```

❶要记住，校验器必须返回逻辑 `true`，否则要修改的状态就会被否决的。在这个例子中，我们通过`#(when-not (:age %) (throw ...))`来实现校验器，当引用的状态确实有`:age`的时候，它返回的是 `nil` 而不是 `true`，因此最后的结果跟大家的期望是不一样的。

虽然校验器对几种引用类型都很有用，但是校验器对于 `ref` 来说有着更重要的作用，我们接下来会详细讨论，详见 189 的“通过校验器来保证本地的一致性”。

180

ref

`ref` 是 Clojure 提供的协调引用类型。使用它可以保证多个线程可以交互地对这个 `ref` 进行操作：

- `ref` 会一直保持在一个一致的状态，任何条件下都不会出现外界可见的不一致的状态。
- 在多个 `ref` 之间不可能会产生竞争条件。
- 不需要你手动地去使用锁、`monitor` 或者其他底层的同步原语。
- 不可能出现死锁。

这一切的保证都来自于 Clojure 实现的“软件事务内存”（STM），通过这个机制来协调各个线程对于 `ref` 的修改操作。

软件事务内存

广义地说，软件事务内存（STM）是任何对并发修改一系列内存地址的行为进行协调的方法。在任何其他语言中要实现 STM 的话，那就意味着你要手动地去使用锁，以及由锁所带来的一系列复杂性。而 STM 则提供了另外一种可能。

就像垃圾回收机制使我们不用再手动管理内存——从而避免了一堆微妙的 bug，STM 则

注 13：或者你可以选择一个 Slingshot 这样的库来抛出一个值，而不是把有用的信息编进一个字符串里：
<https://github.com/scgilardi/slingshot>。

经常被认为是对另一个非常容易出错的编程实践的系统性简化：手动锁管理。在这两种情况下，使用一个经过证明的、自动化的手段来把从那些跟我们领域不相关的、很底层的细节里面解放出来，而且通常最后的效果比那些在这些底层领域很专业的人手写出来的代码效果更好。^{注 14}

Clojure 实现 STM 所使用的技术已经被众多数据库管理系统使用几十年了。^{注 15} 就像它名字所暗示的那样，对于一系列 ref 的每个修改都是具有事务语义的（这个语义你在数据库领域应该已经很熟悉了），而每个 STM 事务保证对于 ref 的修改是：

1. 原子的，因此一个事务里面对于 ref 的所有修改要么都成功，要么都失败。
2. 一致的，因此如果对 ref 设置的新值不满足 ref 的约束条件，那么这个事务将会失败。
3. 隔离的，因此在一个线程内对于 ref 的修改不会影响到其他线程内对这些 ref 的读取。

<181

因此，Clojure 的 STM 满足 ACID 属性 (<https://en.wikipedia.org/wiki/ACID>) 里面的 A、C 和 I，A、C、I 的意思跟它们在数据库领域的意思是一样的。而关于 D 属性：持久性，不是 STM 所关心的，因为 STM 是纯内存的实现。^{注 16}

对 ref 进行修改的机制

有了前面介绍的那些背景知识，我们来看看可以用 ref 来做什么。在 172 页“并发操作的分类”一节，我们讨论了一个银行转账的例子，这是一个需要在多个线程之间进行协调的经典例子。不过要用这个例子来展示事务语义会让大家太紧张了（钱啊！）。可能通过实现一个多人游戏引擎来了解 Clojure STM 以及 ref 的特性会更加有趣。

现实中，有些问题被称为“高度并行”的，因为如果有了合适的工具，这些问题可以高度并行化，而多角色游戏就是高度并发的：涉及的数据集往往都很大，而且通常会有成百上千个独立的游戏玩家要对数据进行修改，这些修改必须以一种协调的、一致的风格发生，以保证游戏的规则不被破坏。

我们的“游戏”^{注 17} 将是一个 RPG 游戏，它会包括一些比如巫师、游侠、吟游诗人等角色。我们会用一个包含 map 的 ref 来表示每一个角色，这个 map 里面保存的就是关于这个角色的所有数据，无论角色的具体种类是什么，所有的角色都至少有以下属性：

注 14： 现代垃圾回收机制的实现使得程序在很多时候比手动进行内存管理的程序性能表现还要好；并且每当一个新的垃圾回收器的实现或者对于现有回收器的优化出现，所有程序的性能都可以得到提升，而不需要每个程序员去做任何事情。Clojure 的 STM 也有相同的效果。

注 15： 具体来说就是多版本并发控制（经常被简称为 MVCC）：https://en.wikipedia.org/wiki/Multiversion_concurrency_control。

注 16： 我们在 215 页的“利用 agent 来记录引用状态的变更日志”一节，展示了一种利用 agent 来实现 D（持久性）的方法。

注 17： 我们不是专业的游戏设计人员，在这里设计的显然只是简单的示意，但是这些简单代码背后的机制是可以用来构建一个真正的游戏引擎的。

- `:name`, 这个角色在游戏里面的名字。
- `:health`, 这是一个说明角色生命值的数字。当 `:health` 降到 0, 这个角色就死了。
- `:items`, 这个属性保存这个角色携带的所有武器装备。

当然, 每个角色还会都有自己的一些独特属性。`character` 是一个实现了所有这些的函数, 它所创建的角色会给 `:items` 和 `:health` 赋上一些默认值:

182 ➤

```
(defn character
  [name & {:as opts}]
  (ref (merge {:name name :items #{}} :health 500
              opts)))
```

有了这个函数之后, 我们可以定义一些具体的角色了:^{注18}

```
(def smaug (character "Smaug" :health 500 :strength 400 :items (set (range 50)))) ❶
(def bilbo (character "Bilbo" :health 100 :strength 100))
(def gandalf (character "Gandalf" :health 75 :mana 750))
```

- ❶ 我们创建了一个 `smaug` 角色, 它有一些武器, 这里这些武器只是一些数字, 在真实的游戏中, 这些数字可能对应到一个静态 map 或者外部数据库中的一条记录。

在类似这样的一个游戏里面, 如果 Bilbo 和 Gandalf 打败了 Smaug 的话, 它们可以抢掉 Smaug 的所有的武器。为了不过度陷入游戏的细节, 抢武器的意思其实就是把一个角色的 `:items` 里面保存的武器装备转移到另外一个人的 `:items` 里面。这个把武器从一个地方转到另外一个地方的过程从外界看必须要一致: 一个武器在同一时刻只能出现在一个人那里。

进入 Clojure 的 STM 和事务。`dosync` 划定了一个事务的边界^{注19}, 所有对于 `ref` 的修改必须要发生在一个事务里面, 对于这些修改的处理都是以同步的方式进行的。也就是说, 启动一个事务的线程在这个事务完成之前是不能继续执行其他代码的。

跟原子类型的 `swap!` 类似, 如果两个事务试图对一个或者多个 `ref` 同时进行有冲突的修改, 那么其中一个事务需要重试。两个对共享 `ref` 进行修改的事务冲突与否, 是由对 `ref` 进行修改所使用的函数来决定的。有三个修改函数: `alter`、`commute` 以及 `ref-set`——它们每一个都有不同的、是否产生(或者避免)冲突的语义。

说了这么多, 在游戏里面, 怎么实现把一个角色的武器转移到另外一个角色那里去呢?

^{注18}: 在一个真正的游戏引擎里面, 肯定不会使用 `var` 来表示角色; 更可能的办法是使用一个 `map` 来表示所有在线玩家的角色, 而这个 `map` 会被保持在一个 `ref` 里面。当游戏玩家上线下线的时候, 就利用 `assoc` 和 `dissoc` 加入这个 `map` 以及从这个 `map` 移除。

^{注19}: 这里要注意一下嵌套的事务范围——嵌套可以是词法上的嵌套的 `dosync` 形式, 也可以是一个事务参与到另外一个事务里面去了, 比如多个函数组成的一个逻辑事务, 这个逻辑事务作为一个整体, 在控制流要离开最外层的 `dosync` 的时候进行提交或者重试。

`loot` 函数负责把一个值从 (`:items @from`) 里面移除，并添加到 (`:items @to`) 里面去（假定这里 `items` 都是一个 `set`）^{注20}，并且返回 `from` 的新状态：

示例4-2：`loot`

```
(defn loot
  [from to]
  (dosync
    (when-let [item (first (:items @from))]
      ●
      (alter to update-in [:items] conj item)
      (alter from update-in [:items] disj item))))
```

183

- ① 如果 (`:items @from`) 是空的话，`first` 会返回 `nil`，`when-let` 中的代码不会被执行，因此整个事务也就什么都没做，而 `loot` 函数本身则会返回 `nil`。

那么假定 Smaug 被 Bilbo 和 Gandalf 打败了，现在它们要来瓜分他的武器了：

```
(wait-futures 1
  (while (loot smaug bilbo))
  (while (loot smaug gandalf)))
:= nil
@smaug
:= {:name "Smaug", :items #{}, :health 500}
@bilbo
:= {:name "Bilbo", :items #{0 44 36 13 ... 16}, :health 500}
@gandalf
:= {:name "Gandalf", :items #{32 4 26 ... 15}, :health 500}
```

这里比较重要的点是，`Bilbo` 和 `Gandalf` 会通过不同的 `future`（也就是不同的线程）来瓜分，并且对于武器的瓜分都是自动进行的：所有的武器必须都被瓜分，不能有一个武器同时被两个人持有。

示例4-3：验证瓜分武器的正确性

```
(map (comp count :items deref) [bilbo gandalf])
●
:= (21 29)
(filter (:items @bilbo) (:items @gandalf))
●
:= ()
```

- ① 如果最后它们武器的总数加起来不是 50（`Smaug` 的武器总数），或者……

- ② 最后 `Gandalf` 和 `Bilbo` 有相同的武器，那么说明 `loot` 事务执行的结果是不一致的。

这里对于武器的瓜分没有使用任何手工加锁，而且这个过程可以扩展到使用更多的 `ref`、`ref` 之间更多的交互。

注 20：回忆一下 105 页“`set`”一节所讨论的，`disj` 函数返回一个不包含指定值的新集合。

理解 alter

`loot` 函数里面使用了 `alter`, 到目前为止, 它看起来跟前面所使用的 `swap!` 是类似的, 它接收这些参数: 要被修改的 `ref`, 一个函数 `f` 以及这个函数所需要的其他参数。当 `alter` 函数返回的时候, `ref` 在这个“事务内的值”会被改变成函数 `f` 的返回结果。

这里有关“事务内的值”的说法是非常关键的。所有对 `ref` 的状态进行修改的函数是在一个独立的时间线上执行的, 这个时间线开始的时间是这个 `ref` 第一次被修改的时候。

184 接下来对于 `ref` 的所有修改、访问都是在这个独立的时间线上进行的, 而这个时间线只在这个事务内存在, 而且只能在这个事务中被访问。当控制流要离开这个事务的时候, STM 会尝试提交这个事务。在最乐观的情况下, 提交会成功, `ref` 的状态被修改成这个事务内 `ref` 的新值, 而这个新值会对所有的线程 / 事务可见——不只是在某个事务内可见了。但是如果外部的时间线已经对 `ref` 的状态进行了修改, 并且已经提交了, 那么事务提交就会跟它发生冲突, 这会导致整个事务重试, 利用 `ref` 的新值来重新执行一遍。

在这个过程当中, 任何只读线程(比如解引用)不会被阻塞住或者需要等待。而且那些对 `ref` 的值进行修改的线程直到成功提交之后, 它们对于 `ref` 的修改才会对其他线程可见, 也就不会影响其他线程对于 `ref` 的只读操作了。

`alter` 的独特语义是, 当一个事务要提交的时候, `ref` 的全局的值必须跟这个事务内第一次调用 `alter` 时候的值一样。否则整个事务会被重启, 从头再执行一遍。

我们可以把这个动态的过程可视化为两个事务之间的交互, t_1 和 t_2 , 它们都会使用 `alter` 对共享的 `ref a` 进行修改:

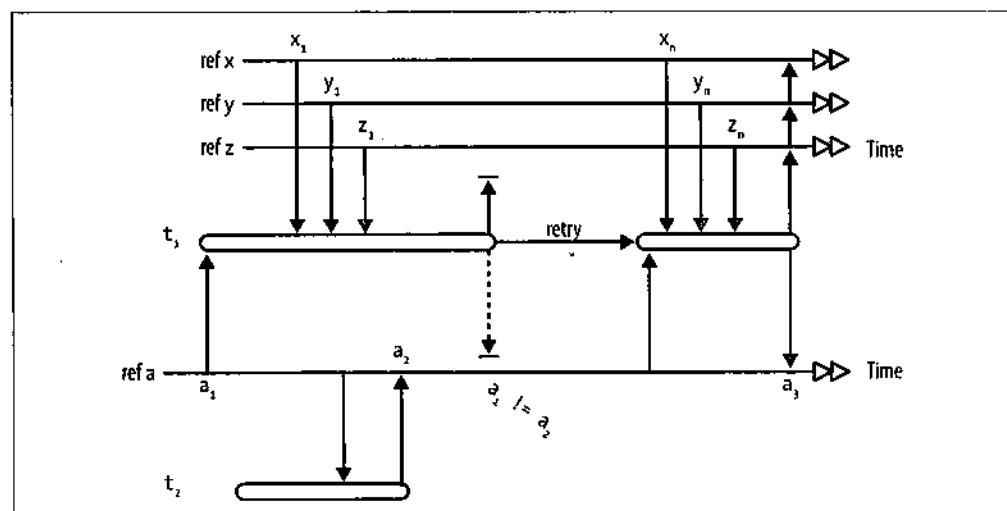


图4-2: 使用 `alter` 的事务之间的交互, 它们共享一个 `ref` 而且会冲突

虽然 t_1 在 t_2 之前开始，但是它要提交的对于 a 的修改会失败，因为 t_2 也对它进行了修改，并且已经提交了。 $a(a_2)$ 的当前状态已经跟 $a(a_1)$ 在 t_1 事务内第一次被修改的时候不一样了。这个冲突会终止事务内对于 ref 所有修改的提交，比如 x 、 y 、 $z \cdots t_1$ 然后会使用 ref 的新值重新执行。

<185

通过这样描述，你可以这样理解 Clojure 的 STM：它是一个乐观地尝试对并发的修改操作进行重新排序，以使得它们可以顺序地执行的一个过程。你肯定在数据库领域也看到过同样的语义，通常被称为“可串行化的快照隔离”(<https://en.wikipedia.org/wiki/Serializability>)。



在事务提交的时候，如果只有一个 ref 的状态冲突了，这个事务也不能提交，即使其他 100 个 ref 的状态都没有冲突。

利用 commute 来最小化事务冲突

因为 alter 没有对修改 ref 的可重排序性做任何假设，因此它是对 ref 状态进行修改的最安全的方式，不过，在一些情况下，可以安全地把对 ref 的修改的操作进行重新排序，这样就可以使用 commute 来代替 alter，可以减少潜在冲突几率以及事务重试次数，从而最大化总体的吞吐量。

从这个函数的名字就可以看出，commute 跟符合“交换原则”的函数有关 (https://en.wikipedia.org/wiki/Commutative_property) ——这些函数的参数的位置不重要，参数的位置相互调换之后，函数的返回结果仍然是一样的，比如，+、*、clojure.set/union 等——但是 commute 并不强制你传入的函数一定要符合交换原则。这里真正重要的是传给 commute 的函数可重新排序，而不会影响程序的语义。这里的前提是，符合交换原则的函数所产生的最终结果是我们所关心的，中间结果则不重要。

比如，虽然除法本身是不符合交换原则的，但是通常可以让它和 commute 一起合作——如果我们不关心中间结果的话：

```
(= (/ (/ 120 3) 4) (/ (/ 120 4) 3))
:= true
```

因此，当函数的组合是符合交换原则的，那我们就可以使用 commute：

```
(= ((comp #(/ % 3) #(/ % 4)) 120) ((comp #(/ % 4) #(/ % 3)) 120))
:= true
```

总的来说，commute 应该只用在可以对修改 ref 状态的操作进行重排的场景中。

commute 跟 alter 有两个方面的不同。首先，alter 的返回值是要更新到这个 ref 的全局

状态的值，这个事务内的值会是最终提交的值。而另一方面，由 `commute` 产生的事务内的值不一定会是最终提交的那个值，因为所有被 `commute` 的函数在最终提交的时候会利用 `ref` 的最新全局值重新计算一遍。

第二，利用 `commute` 对 `ref` 进行的修改从来不会导致冲突，因此也从来不会导致一个事务重试。这个显然潜在地提高了性能以及吞吐量：事务重试意味着要把所有的工作重做一遍，而且一个线程被阻塞着等待一个事务成功完成的时候才可以继续去做其他的事情。

可以非常形象地来演示这一点。假定有一个 `ref x`：

```
(def x (ref 0))
:= #'user/x
```

我们会用 10 000 个事务来操作这个 `ref`，每个事务做的事情很少（只是简单地去计算一些数字的总和），然后把这个结果用 `alter` 来更新 `x` 的值：

```
(time (wait-futures 5
  (dotimes [_ 1000]
    (dosync (alter x + (apply + (range 1000))))))
  (dotimes [_ 1000]
    (dosync (alter x - (apply + (range 1000))))))
; "Elapsed time: 1466.621 msecs"
```

这里面至少有一部分花在事务的重试上面，从而使得我们要重新计算这些数字的和。这里其实是可以把 `alter` 替换成 `commute` 的，因为这里的加、减都是符合替换原则的：

```
(time (wait-futures 5
  (dotimes [_ 1000]
    (dosync (commute x + (apply + (range 1000))))))
  (dotimes [_ 1000]
    (dosync (commute x - (apply + (range 1000))))))
; "Elapsed time: 818.41 msecs"
```

虽然这样要执行修改 `ref` 值的函数两次——一次用来设置事务内的值（使得在事务内如果要用这个 `x` 的更新值也是可以的），还有一次是在提交的时候来“真正”修改 `x` 的值——总的运行时间还是少了将近一半，因为 `commute` 从来不需要重试。

`commute` 其实也没有什么太神奇的东西在里面：要非常谨慎地使用它，否则会产生一些不正确的结果，来看看如果把示例 4-2 中 `loot` 函数里面使用的 `alter` 换成 `commute` 会怎么样：

示例 4-4：一个使用 `commute` 的有问题的 `loot` 函数

```
(defn flawed-loot
  [from to]
  (dosync
```

```
(when-let [item (first (:items @from))]
  (commute to update-in [:items] conj item)
  (commute from update-in [:items] disj item))))
```

把这些角色的状态恢复到初始状态，来看看新 `loot` 函数表现如何：

◀ 187

```
(def smaug (character "Smaug" :health 500 :strength 400 :items (set(range 50))))
(def bilbo (character "Bilbo" :health 100 :strength 100))
(def gandalf (character "Gandalf" :health 75 :mana 750))

(wait-futures 1
  (while (flawed-loot smaug bilbo))
  (while (flawed-loot smaug gandalf)))

;= nil
(map (comp count :items deref) [bilbo gandalf])
;= (5 48)
(filter (:items @bilbo) (:items @gandalf))
;= (18 32 1)
```

我们使用了和示例 4-3 相同的校验，可以看出，`flawed-loot` 的结果是有问题的：Bilbo 有 5 个武器，Gandalf 有 48 个武器（其中 18、32 和 1 这三个武器两个人都有），这种情况永远都不应发生，因为总的武器数目一共只有 50 个。

哪里出问题了？在三次的 `loot` 中，相同的值被从 Smaug 的 `:items` 集合里面取出来，并且同时加入到 Bilbo 和 Gandalf 的 `:items` 集合里面去了。如果使用 `alter` 是永远不会发生这种事情的，因为它可以保证 `ref` 在线程内的值与提交的值肯定是一样的。

在这个特定的例子里面，我们可以安全地使用 `commute` 来把抢过来的武器放入抢武器的人的武器库（因为我们把武器加入武器库的顺序对于最后的结果来说并不重要），这里的问题出在从被抢的武器库删除武器上，从被抢的武器库中删除武器一定要使用 `alter`，因为如果顺序不对，删除的武器就不对了：

示例 4-5：一个使用 `commute` 和 `alter` 的 `fixed-loot` 函数

```
(defn fixed-loot
  [from to]
  (dosync
    (when-let [item (first (:items @from))]
      (commute to update-in [:items] conj item)
      (alter from update-in [:items] disj item)))

    (def smaug (character "Smaug" :health 500 :strength 400 :items (set(range 50))))
    (def bilbo (character "Bilbo" :health 100 :strength 100))
    (def gandalf (character "Gandalf" :health 75 :mana 750))

    (wait-futures 1
      (while (fixed-loot smaug bilbo)))
```

```
(while (fixed-loot smaug gandalf))
:= nil
(map (comp count :items deref) [bilbo gandalf])
:= (24 26)
(filter (:items @bilbo) (:items @gandalf))
:= ()
```

188 而另一方面，commute 可以使用在游戏中其他很多函数上面。比如 attack 和 heal 函数，它们只是对角色的某个属性进行加减，这种修改可以安全地使用 commute：

```
(defn attack
  [aggressor target]
  (dosync
    (let [damage (* (rand 0.1) (:strength @aggressor))]
      (commute target update-in [:health] #(max 0 (- % damage)))))

(defn heal
  [healer target]
  (dosync
    (let [aid (* (rand 0.1) (:mana @healer))]
      (when (pos? aid)
        (commute healer update-in [:mana] - (max 5 (/ aid 5)))
        (commute target update-in [:health] + aid))))
```

再加上一些其他函数，我们可以模拟一个游戏玩家在游戏里面做一些动作：

示例4-6：游戏玩家模拟函数

```
(def alive? (comp pos? :health))

(defn play
  [character action other]
  (while (and (alive? @character)
              (alive? @other)
              (action character other))
    (Thread/sleep (rand-int 50))) ●
```

●当然，没有人可以在 1 秒之内做某个动作 20 次之多！

现在可以让角色之间决斗了：

```
(wait-futures 1
  (play bilbo attack smaug)
  (play smaug attack bilbo))
:= nil
(map (comp :health deref) [smaug bilbo]) ●
:= (488.80755445030337 -12.0394908759935)
```

● Bilbo 一个人打不过 Smaug。

或者说，“史诗般的”战斗。

示例4-7：三个角色之间的战斗

```
(dosync
  (alter smaug assoc :health 500)
  (alter bilbo assoc :health 100))

(wait-futures 1
  (play bilbo attack smaug)
  (play smaug attack bilbo)
  (play gandalf heal bilbo))
:= nil
(map (comp #(select-keys % [:name :health :mana]) deref) [smaug bilbo gandalf]) ②
:= {:health 0, :name "Smaug"}
:= {:health 853.6622368542827, :name "Bilbo"}
:= {:mana -2.575955687302212, :health 75, :name "Gandalf"})
```

189

● 这里重置一下各个角色的状态，给他们“充满血”。

● 只要 Gandalf 一直给 Bilbo 加血，Smaug 是可以被打败的！

用 ref-set 来设置 ref 的状态

ref-set 会把 ref 的事务内的状态设置到一个给定的值：

```
(dosync (ref-set bilbo {:name "Bilbo"}))
:= {:name "Bilbo"}
```

跟 alter 一样，如果在当前事务提交之前，ref 的状态在当前事务之外被改变了的话，这个事务会被重试。换个说法就是，ref-set 在语义上和 alter 是一样的，只不过你调用 alter 的时候传入的是一个函数以及一些参数，而调用 ref-set 传入的则是一个直接要设置的值：

```
(dosync (alter bilbo (constantly {:name "Bilbo"})))
; {:name "Bilbo"}
```

因为这种对于 ref 状态的改变跟 ref 的当前状态丝毫没有关系，很容易出现的情况就是这种改变符合 STM 的事务语义，但是会破坏应用级别的一些约定。因此，ref-set 通常用来重新初始化 ref 的状态到初始值。

通过校验器来保证本地的一致性

也许你已经注意到了，在示例 4-7 结束的时候，Bilbo 的生命值非常高。确实，我们没有对角色的生命值 :health 进行任何限制，因此由于角色可以被加血，使得最后 Bilbo 的生命值那么高。

这种游戏中通常都会限制角色的生命值不能超过某个阈值。但是不管从技术还是管理的角度来看——特别是当一个开发团队非常大，人非常多时，很难保证每个人写出来的代码对于角色的生命值都有一样的限制，但是必须要维护数据的完整性——也就是 ACID 里面的 C，这个工作就落在校验器的肩上了。

我们已经在 178 页“校验器”一节讨论过如何使用校验器了。对 ref 使用校验器跟对其他引用类型使用校验器的方法完全一样，它们跟 STM 的交互非常方便：如果一个校验器函数发现了一个非法的状态，就抛出一个异常（就像事务里面抛出的其他异常一样），然后就会导致当前这个事务失败。

190 了解了这个之后，可以对我们的实现进行一些微调。首先，character 应该修改成这样：

1. 对于每个角色应该有一些公共的校验器。
2. 对于每种特定的角色应该有一些额外的校验器，这些校验器会校验跟这些角色特定相关的一些状态：

```
(defn enforce-max-health
  [{:keys [name health]}]
  (fn [character-data]
    (or (<= (:health character-data) health)
        (throw (IllegalStateException. (str name " is already at max health!")))))

(defn character
  [name & {:as opts}]
  (let [cdata (merge {:name name :items #{}} :health 500)
        opts)
        cdata (assoc cdata :max-health (:health cdata))
        validators (list* (enforce-max-health name (:health cdata))
                           (:validators cdata))]
    (ref (dissoc cdata :validators)
         :validator #(every? (fn [v] (v %)) validators)))
```

- ❶ enforce-max-health 返回的是一个函数，这个返回的函数接受一个角色的“潜在的”新状态，如果这个“潜在的”新状态的生命值 :health 比这个角色的最高生命值还高的话，那么抛出一个异常。
- ❷ 把这个角色的最原始的生命值 :health 记录下来作为它的最高生命值 :max-health，这样对于接下来的处理会非常方便。
- ❸ 除了始终要保证角色的生命值不要超过它的最大生命值，我们也可以很容易地给每个角色添加它们自己的额外校验函数……
- ❹ 这些校验函数将用来校验它们各自特定的状态属性。

现在，任何角色的生命值都不可能超过它最开始初始化的生命值了：

```
(def bilbo {character "Bilbo" :health 100 :strength 100})
:= #'user/bilbo
(heal gandalf bilbo)
:= #<IllegalStateException java.lang.IllegalStateException:Bilbo is already at max
  health!>
```

校验器的一个局限是，它们是非常严格的，使得某些业务上可能认为正常的事情我们做不了：

```
(dosync (alter bilbo assoc-in [:health] 95))
:= {:max-health 100, :strength 100, :name "Bilbo", :items #{}, :health 95, :xp 0}
(heal gandalf bilbo)
:= #<IllegalStateException java.lang.IllegalStateException:Bilbo is already at max
  health!>
```

这里 Bilbo 的生命值 :health 是比它的最高生命值 :max-health 小的，因此，它应该是可以加血的。但是我们的 heal 实现只看角色的最大生命值 :max-health，而且也没有办法让相关的校验器做一些调整使得 Bilbo 的新生命值符合它的限制——在这个例子里面要么是 :health 和 :max-health 里面比较小的那个，要么是它当前的生命值 :health 和 Gandlf 给他加的血的总和。如果我们的校验允许做这样的调整，那么就很难避免在一个事务内引入不一致的数据了。记住，校验器只是为了保证数据的完整性。

<191

不过可以对 heal 函数做一些小调整，使得它可以进行“部分”加血——如果全部加会超过最大血量的话，那么只加到角色的最大血量：

```
(defn heal
  [healer target]
  (dosync
    (let [aid (min (* (rand 0.1) (:mana @healer))
                  (- (:max-health @target) (:health @target)))]
      (when (pos? aid)
        (commute healer update-in [:mana] - (max 5 (/ aid 5)))
        (alter target update-in [:health] + aid))))
```

现在 heal 函数可以保证给角色加血到它的最大血量了，如果角色不用加血，那么返回 nil：

```
(dosync (alter bilbo assoc-in [:health] 95))
:= {:max-health 100, :strength 100, :name "Bilbo", :items #{}, :health 95}
(heal gandalf bilbo)
:= {:max-health 100, :strength 100, :name "Bilbo", :items #{}, :health 100}
(heal gandalf bilbo)
:= nil
```

注意，这里对于 `target` 的修改潜在地依赖它之前的状态，所以这里用的是 `alter` 而不是 `commute`。这个其实不是严格需要的：你可以让校验器来拦截那些“过度的”加血，这种情况只会在一些别的并行发生的事务同时给这个角色加血的时候才会出现。不过这种做法对于现在对数据的建模方式来说有个缺点，现在一个角色的状态是一个 `map`，这个 `map` 被一个单个 `ref` 所持有：如果并发的几个事务修改这个状态不相关的几个属性，那么某些事务会不必要地重试。^{注21}

STM的一些缺点

就像我们在这章开始说的那样，Clojure 并没有任何解决并发问题的银弹。它提供的 STM 实现有时候看起来可能很魔幻——和那种传统的、需要手动进行锁管理的方式比起来确实很魔幻——但是即使是 STM 也有它自己的弱点，而你应该知道这些弱点。

事务内绝对不能执行有副作用的函数

适合在事务内执行的函数操作必须是可以安全地进行重试的，这就过滤掉了很多形式的 I/O。比如，如果在一个 `dosync` 块中去写文件或者数据库，那么结果很可能是文件 / 数据库中有重复的数据。

Clojure 没办法检测你在事务内执行的操作是不是安全的；当需要重试的时候，它简单地重试这些操作，而这种重试可能会带来灾难性的结果。因此，Clojure 提供了一个 `io!` 宏，它的作用是：当它被放在一个事务中执行，它会抛出一个错误。因此如果你有一个函数可能会被误用在一个事务里面，那么可以把其中有副作用的部分用 `io!` 包起来，这样可以防止万一被误用在一个事务里面，它会及时地报错：

```
(defn unsafe
  []
  (io! (println "writing to database...")))
:= #'user/unsafe
(dosync (unsafe))
:= #<IllegalStateException java.lang.IllegalStateException: I/O in transaction>
```



由此可以推论，对于原子类型的操作，一般来说可以认为是有副作用的，比如 `swap!`，因此它不能放在事务里面执行，如果一个事务重试三次的话，而这个事务里面包含有对 `swap!` 的调用，那么这个 `swap!` 会被调用三次，而这个被作用的原子类型则会被修改三次，而这很少是你想要的结果，除非你想用原子类型来记录事务重试的次数。

注21：要给模型确定一个理想的 ref 粒度是一个优化的步骤，需要通过性能测试、试验以及一定程度的远见才能做到。而编写代码的时候应该始终从最简单的办法开始——把所有值放在一个 ref 里面，大多数时候已经足够了，只有当真的需要时再去尝试更复杂的解决方案。<http://clj-me.cgrand.net/2011/10/06/a-world-in-a-ref> 这篇文章提供了一个潜在的方向。

同时要注意，被 ref 持有的值一定要是不可变的。^{注22} Clojure 并不会阻止你放一个可变的对象到 ref 里面去，但是重试将会导致可变对象最终处于你意料不到的状态：

```
(def x (ref (java.util.ArrayList.)))
:= #'user/x
(wait-futures 2 (dosync (dotimes [v 5]
    (Thread/sleep (rand-int 50))      ❶
    (alter x #(doto % (.add v))))))
:= nil
@x
:= #<ArrayList [0, 0, 1, 0, 2, 3, 4, 0, 1, 2, 3, 4]>      ❷
```

❶ 这个随机时间的 sleep 调用保证两个事务交替执行，至少一个会重试，从而……

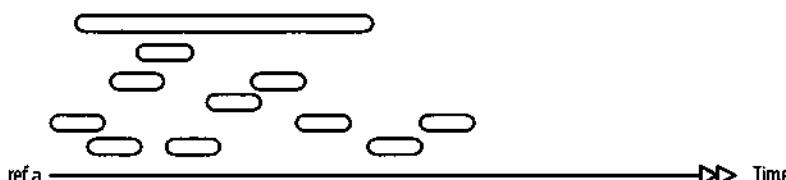
193

❷ 产生有问题的结果。

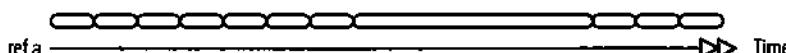
最小化每个事务的范围

回忆一下我们在图 4-2 中的讨论：STM 的职责是保证所有对 ref 进行修改的事务以串行的方式进行提交，而如果有必要的话，把这些对 ref 进行修改的操作进行重新排序。这意味着一个事务如果越短，那么 STM 调度起来就越简单，因此也就意味着更快的执行速度以及更高的总吞吐量。

如果你的程序里面有过长的事务，或者有很多大小不一的事务会怎么样？一般来说，最大的事务会被延迟提交（而那些依赖于这个事务的操作也会被延迟），看看下图里面的这些事务，假设它们都要去修改一个 ref a：



这些事务中很多会进行重试，直到它们可以串行地进行提交。而最长的事务会不断地重试，直到出现一个足够长的时间窗口，而在这个时间窗口中又没有别的事务跟它竞争，它才能成功提交：



注22： 或者，至少你对数据结构的用法是一种可变的方式。比如，完全可以将一个可变的 Java 列表放在 ref 里面，当产生新列表的时候，使用 Copy-On-Write 的策略，这样它也是可以达到事务特性的，但是这样做很奇怪，通常来说是完全没有必要这么做的。



我们知道 `commute` (在 185 页“利用 `commute` 来最小化事务冲突”一节介绍的) 不会引起事务冲突和重试。因此如果在你的场景中，你可以安全地对函数使用 `commute` 的话，就可以非常有效地避免长事务带来的问题。

做大量耗时的计算可以导致长事务，但是长事务也可能是跟别的事务竞争 ref 修改导致的。比如我们上面描述的长事务可能是在做一个很复杂的运算，而这个运算牵涉到一些别的事务也在不断更新的 ref，因此导致不断地进行重试。因此我们说，不要编写过长事务一方面是不要在事务内进行复杂耗时的计算，另一方面也要减少事务涉及的 ref 数量。

194 >

活锁。你可能会想，如果系统的负载非常高，一个很大的事务由于 ref 竞争永远都没有机会进行提交，那会怎么样？这种情况被称为“活锁”，相当于 STM 世界里面的死锁。在这种情况下，这个事务所在的线程被其他事务无限地阻塞了。如果没有合适的解决办法，那还不如自己手动地进行锁管理，因为那样最坏的结果也就是死锁而已，跟活锁在结果上没什么区别！

幸运的是，Clojure 的 STM 提供了一些解决办法。而第一个解决办法叫做 `barging`，在这个解决办法中，在某些情况下，当一个老事务跟一个新事务进行竞争的时候，系统会强迫较新的事务进行重试。而如果 `barging` 了一定次数之后这个老事务还是不能成功提交，那么系统会让这个事务失败：

```
(def x (ref 0))
:= #'user/x
(dosync
  @(future (dosync (ref-set x 0)))
  (ref-set x 1))
:= #<RuntimeException java.lang.RuntimeException:
:=   Transaction failed after reaching retry limit>
@x
:= 0
```

在上面的例子中，在 REPL 中运行的这个事务线程会启动一个新的 future，这个 future 中会启动另外一个事务对这个 ref 进行修改。通过对这个 future 进行解引用强制 REPL 线程等待这个 future 中的事务完成才继续执行，因此会导致 REPL 中的事务重试——然后又启动一个 future，无限循环。

Clojure 的 STM 是不会让一个事务无休止地重试下去的，重试一定次数之后会抛出一个异常。这样你可以去检查这个异常堆栈，至少知道到底发生了什么，比那种真正的死锁（或者是活锁）要好很多，因为发生真正死锁的时候，除了手动把应用线程杀掉之外没有什么好的办法，而杀掉之后你也不知道到底哪里出了问题。

读线程也可能重试

对引用类型来说，`deref` 是保证从来不会阻塞的。但是如果是在一个事务内的话，利用 `deref` 对一个 `ref` 解引用是可能触发事务重试的！

这是因为，如果另外一个线程在当前线程开始之后提交了 `ref` 的一个新值的话，我们就没有办法获取这个 `ref` 在事务开始时候的值了。^{注23} 幸运的是，STM 意识到了这个问题，并且会维护事务中涉及的 `ref` 的一定长度的历史版本值，这个历史版本的长度在每次事务重试的时候递增。这使得在某个时间点，我们的事务终于不用再重试了，因为虽然 `ref` 还在被其他事务并行地更新，但是我们需要的值还是在历史记录中。

这个历史版本的长度可以利用 `ref-history-count`、`ref-max-history` 以及 `ref-min-history` 来查询（以及调整）。而我们在创建 `ref` 的时候也可以通过指定 `:min-history` 和 `:max-history` 关键字参数来设置 `ref` 历史的最小长度和最大长度：

```
(ref-max-history (ref "abc" :min-history 3 :max-history 30))  
:= 30
```

这使得我们可以对历史长度进行调整来达到期望的负载要求。

`deref` 的重试通常出现在那些只读事务中，在这些事务中，我们想获取对于一些被活跃修改的 `ref` 的快照。可以通过一个单个 `ref` 以及一个很慢的只读事务来展示这种情况：

```
(def a (ref 0))  
(future (dotimes [_ 500] (dosync (Thread/sleep 200) (alter a inc))))  
:= #<core$future_call$reify__5684@10957096: :pending>  
@(future (dosync (Thread/sleep 1000) @a))  
:= 28         ❶  
(ref-history-count a)  
:= 5
```

❶ 这里读到的值 28 意味着读取事务在所有写事务完成之前成功提交了。

因此，这里的 `a` 通过一些历史记录使得读取事务可以读到它想要读的值了，那么如果写事务再快一点会怎么样？

```
(def a (ref 0))  
(future (dotimes [_ 500] (dosync (Thread/sleep 20) (alter a inc))))  
:= #<core$future_call$reify__5684@10957096: :pending>  
@(future (dosync (Thread/sleep 1000) @a))
```

注 23：在 196 页的“write skew”一节有关于 `deref` 在一个事务内的返回值的更细节的讨论。

```
;= 500
(ref-history-count a)
;= 10
```

这一次历史记录达到最高了，而读取事务是在所有写事务完成之后才完成的。这意味着这些写事务阻塞了读事务。如果把最大的历史长度调大一点，这个问题应该可以被修正：

```
(def a (ref 0 :max-history 100))
(future (dotimes [_ 500] (dosync (Thread/sleep 20) (alter a inc))))
;= #<core$future_call$reify__5684@10957096: :pending>
@(future (dosync (Thread/sleep 1000) @a))
;= 500
(ref-history-count a)
;= 10
```

没有成功，因为当有足够的历史的时候，这些写事务已经都执行完成了。因此这里的关键是设置最小历史长度到一个合适的值：

```
(def a (ref 0 :min-history 50 :max-history 100)) ❶
(future (dotimes [_ 500] (dosync (Thread/sleep 20) (alter a inc))))
@(future (dosync (Thread/sleep 1000) @a))
;= 33
```

❶ 这里选择 50 是因为读事务比写事务慢 50 倍。

196 > 这一次读事务很快成功完成了，而且没有重试！

write skew

Clojure 的 STM 保证了 ref 状态的事务性一致，但是到目前为止我们见识到的 ref 都是在当前事务被修改的。如果事务的一致性依赖一个 ref，但是对它只是读取，而不修改，那么 Clojure 的 STM 没有办法通过对 alter、commute 或者 set-ref 的调用知道这一点（因为不对 ref 进行修改，所以根本就不会调用这三个函数）。如果事务要读取的 ref 的值在事务进行一半的时候在别的事务中被修改了，而事务依赖的还是那个旧的值，那么当事务提交的时候，整个状态就不一致了，这种情况称为 write skew。

这种情况很少见，一般来说，在一个事务中的 ref 都会被某种形式进行修改。但是，当场景确实不会对一个 ref 进行修改，但是你又依赖它的值的时候，可以使用 ensure 来避免 write skew：它的作用也是对这个 ref 进行解引用，但是如果当前事务在提交之前，这个 ref 在别的事务中被修改了的话，会导致当前事务进行重试。

在这个游戏的例子中，我们来添加一个日光“daylight”因素，比如制订这么一条游戏规则：在中午的时候，攻击给对方造成的伤害要比傍晚高，我们来对 attack 做一些修改，把这一点体现出来：

```
(def daylight (ref 1))

(defn attack
[aggressor target]
(dosync
(let [damage (* (rand 0.1) (:strength @aggressor) @daylight)]
(commute target update-in [:health] #(max 0 (- % damage))))))
```

但是，如果 daylight 的状态在读取它之后、在提交事务之前发生了改变会怎么样？那么整个系统的状态就不一致了。比如一个单独的游戏进程可能会对 daylight 进行调整以表明现在已经傍晚了，如，(dosync (ref-set daylight 0.3))。假如 attack 调用的时候，这个进程对 daylight 进行了调整，那么这个 attack 的伤的生命值其实是多了。

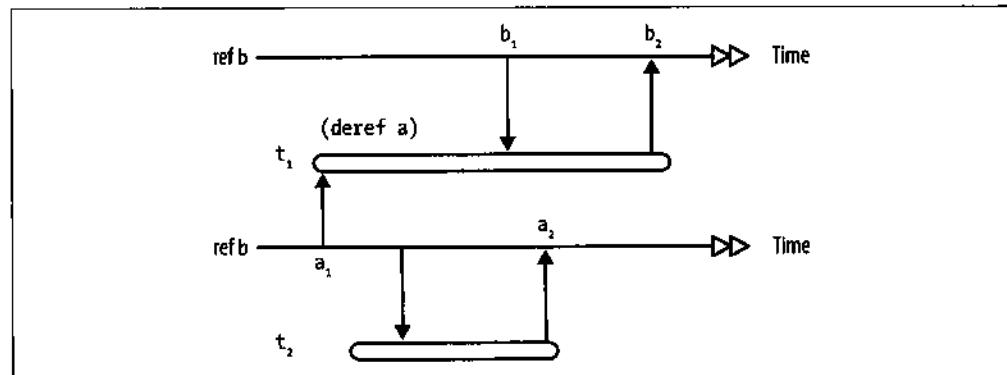


图4-3: write skew, b₂的状态依赖于在之前某个时间读取的a的状态

更通用地说，如果事务 t_1 要更新的 ref b 的状态为 b_2 ，而这个 b_2 的值依赖于 ref a 的 a_1 快照，而且事务 t_1 从来不会对 a 进行修改，而另一个事务 t_2 在事务 t_1 提交之前把 a 的状态修改成了 a_2 ，那么系统的状态就变得不一致了： b_2 依赖的是一个旧状态 a_1 ，而不是当前状态 a_2 。这就叫做 write skew。

197

我们简单地把 attack 里面的 @daylight 修改成 (ensure daylight) 就可以保证读取的 daylight 是最新的值。

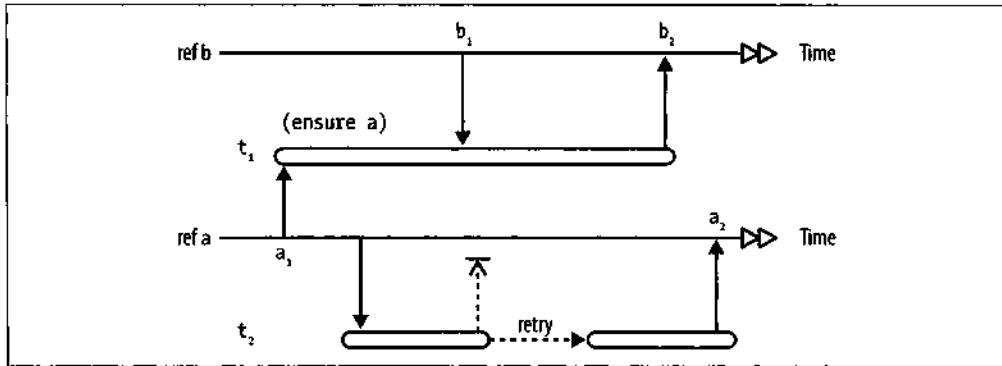


图 4-4：利用ensure来避免write skew

当一个事务 t_1 使用 `ensure` 而不是 `deref` 来读取一个 `ref a` 的时候，在事务 t_1 成功提交之前任何其他事务 t_2 对于 `ref a` 的修改都会导致事务 t_1 的重试，直到成功提交。这个可以避免 write skew：对于状态 b 的修改始终用最新状态的 a ，即使事务 t_1 从来没有修改过 a 。

198



从避免 write skew 的角度来说，`(ensure a)` 在语义上和 `(alter a identity)` 或者 `(ref-set a @a)` 是等效的——它们都是做了一个“空写”（dummy write）——作用都是保证读出的值在提交之前不发生改变。而跟空写相比，`ensure` 通常来说会最小化由于只读 `ref` 而引起的事务重试次数。

var

前面已经大量使用过 `var` 了，`var` 跟 Clojure 其他引用类型不同之处在于，它们的状态不是在时间尺度上进行维护的，它们提供的是一个命名空间内全局的实体，可以在每个线程上把这个实体绑定到不同的值。我们会从 201 页的“动态作用域”一节开始详细介绍这一点，不过首先还是先来看一下 `var` 的最基本的概念。因为 `var` 在 Clojure 中是最基本的一个类型，到处都在使用，不管你要不要处理并发问题。

在 Clojure 里面对一个符号进行求值，其实就是在当前命名空间寻找名字是这个符号的那个 `var`，并对它解引用以获取它的值。也可以直接引用 `var`，并且手动对它解引用：

```
map
:= #<core$map clojure.core$map@501d5ebc>
#'map
:= #'clojure.core/map
@#'map
:= #<core$map clojure.core$map@501d5ebc>
```

❶ 回忆一下 44 页“引用 :var”一节介绍的 `#'map` 是 `(var map)` 的语法糖。

定义 var

var 是组成 Clojure 的最基本的组件之一，我们在 26 页“定义 Var : def”一节提到过，顶层的函数和值都是保存在 var 中的，它们都是利用 def 或者它的引申物（deftype、defrecord 等）定义在当前的命名空间中的。

除了简单地把 var 以指定的名字“安装”到当前命名空间中，def 还会把这个符号上面的元数据赋到^{注 24}这个 var 上面。符号上面的一些特定的元数据可以改变 var 的语义，这里列举几个。

私有 var

私有 var 决定了哪部分代码是类库的一部分、跟实现相关的、不应该暴露给外部用户的；哪一部分是暴露给外部用户使用的方法。一个私有 var 具有如下特性：

1. 在其他命名空间里面只能通过全限定名称对它进行访问。
2. 要访问它的值只能通过解引用。

199

通过给 var 添加一个名为 :private、值为 true 的元数据，我们可以把这个 var 定义为私有的。下面是一个私有 var，保存一个其他代码需要使用的常量值：

```
(def ^{:private true} everything 42)
```

在 135 页“元数据”一节介绍过，这种表示方法跟下面的表示方法是等价的：

```
(def ^{:private true} everything 42)
```

从下面的例子可以看出，必须使用全限定名，才能在其他命名空间里面访问 everything：

```
(def ^{:private true} everything 42)
:= #'user/everything
(ns other-namespace)
:= nil
(refer 'user)
:= nil
everything
:= #<CompilerException java.lang.RuntimeException:
:=  Unable to resolve symbol: everything in this context, compiling:(NO_SOURCE_
PATH:0)>
@#'user/everything
:= 42
```

可以通过 defn- 来定义一个私有函数，它的作用跟 defn 完全一样，只是会自动给函数加上一个 ^{:private} 元数据。

注 24：135 页“元数据”一节有关于 Clojure 中的元数据的详细介绍。

文档字符串

Clojure 允许通过文档字符串来给顶级 var 添加文档，文档字符串通常是紧跟在命名这个 var 的符号后面：

```
(def a
  "A sample value."
  5)
:= #'user/a
(defn b
  "A simple calculation using `a`."
  [c]
  (+ a c))
:= #'user/b
(doc a)
; -----
; user/a
;   A sample value.
(doc b)
; -----
; user/b
; ([c])
;   A simple calculation using `a`.
```

正如你所看到的，文档字符串只是 var 上的另外一个元数据而已，`def` 在背后帮助我们自动把文档字符串加到 var 的元数据 map 里面去了：

```
(meta #'a)
{:ns #<Namespace user>, :name a, :doc "A sample value.",
 := :line 1, :file "NO_SOURCE_PATH"}
```

这也意味着，如果你想的话，可以显式地给 var 添加一个 :doc 元数据来添加文档，甚至可以在 var 定义之后通过对 var 的元数据进行修改以添加文档：

```
(def ^{:doc "A sample value."} a 5)
:= #'user/a
(doc a)
; -----
; user/a
;   A sample value.
(alter-meta! #'a assoc :doc "A dummy value.")
:= {:ns #<Namespace user>, :name a, :doc "A dummy value.",
 := :line 1, :file "NO_SOURCE_PATH"}
(doc a)
; -----
; user/a
;   A dummy value.
```

实际中很少这么做，不过在编写定义 var 的宏的时候，这个特性确实非常方便。

常量

定义常量是应用中很普遍的一个需求，通常是利用 `def` 来定义一个顶级 `var`。可以给你的 `var` 加上一个 `^:const` 元数据，这样可以告诉编译器它是一个常量：

```
(def ^:const everything 42)
```

这样也可以作为这个 `var` 自身的一个说明，一目了然它是一个常量，而且 `^:const` 不只是一个简单的标记而已，它在运行时会有一些实质影响的：任何对于常量 `var` 的引用都不是在运行时求值的（普通 `var` 是这个时候求值的）；这些引用在编译期会被全部替换成这个常量所对应的值。这个对于那种非常“热”的代码会带来些许的性能提升，但更重要的是，保证了你的常量确实保持了常量的特征，即使别人不小心改了这个 `var` 的值。

下面例子的结果肯定不是大家所期望的：

```
(def max-value 255)
:= #'user/max-value
(defn valid-value?
  [v]
  (<= v max-value))
:= #'user/valid-value?
(valid-value? 218)
:= true
(valid-value? 299)
:= false
(def max-value 500) ①
:= #'user/max-value
(valid-value? 299)
:= true
```

◀ 201

- ① `valid-value?` 在 `max-value` 被重新定义之后表现出了不同的语义，因为它依赖这个“常量”的运行时值。

可以通过给它加上一个 `^:const` 元数据来防止这种情况的发生：

```
(def ^:const max-value 255)
:= #'user/max-value
(defn valid-value?
  [v]
  (<= v max-value))
:= #'user/valid-value?
(def max-value 500)
:= #'user/max-value
(valid-value? 299)
:= false
```

这里给 `max-value` 加上了一个 `^:const` 元数据，它的值在编译期就被 `valid-value?` 函数

替换到代码里面。后续任何对于 max-value 的修改都不会影响 valid-value? 的语义——直到这个函数本身被重新定义。

动态作用域

大多数情况下，Clojure 中的作用域跟其他语言一样用的是词法意义上的作用域：也就是说，一个 var 的作用域在定义它的形式之内，比如：

```
(let [a 1
      b 2]
  (println (+ a b))          ①
  (let [b 3
        + -]
    (println (+ a b))))       ②
  ;;= 3
  ;;= -2
```

① a 和 b 是由 let 创建的两个本地绑定，+ 和 println 是在命名空间 clojure.core 中定义的两个函数的名字，在当前的命名空间可以访问到它们。

② 这里本地绑定 b 被绑定到一个不同的值，+ 也是，因为这些定义在词法上比外层的 b 绑定以及 + 比原始定义要近，因此它们在这个上下文里面“遮蔽”了它们原来的值。

这里有一个例外，var 提供的一种叫做“动态作用域”的特性是一个例外。每个 var 有一个根绑定，这是使用 def 及其变种定义 var 的时候赋上的值，同时也是一般情况下对它解引用所得到的值。但是如果把一个 var 定义成动态的（通过给它加上 ^:dynamic 元数据），^{注25} 然后就可以利用 binding 在每个线程上覆盖这个根绑定的值。

```
(def ^:dynamic *max-value* 255)
 ;;= #'user/*max-value*
 (defn valid-value?
   [v]
   (<= v *max-value*))
 ;;= #'user/valid-value?
 (binding [*max-value* 500]
   (valid-value? 299))
 ;;= true
```



对于那些你希望通过 binding 来对根绑定进行覆盖的动态 var，我们一般在命名的时候以星号开头、以星号结尾，比如 *this*——也被称为“护耳”。虽然这仅仅是一个命名约定，但是它可以很明确地提醒代码的阅读者，这是一个动态的 var，可以利用它来实现动态作用域。

注 25：如果对一个不是 :dynamic 的 var 使用 binding 会抛出异常。

这里虽然 `*max-value*` 是在 `valid-value?` 中被使用的，但是可以在外层利用 `binding` 来修改它的值。不过这仅是对线程本地的值的修改，我们可以看到 `*max-value*` 在其他线程中的值还是原来的：^{注 26}

```
(binding [*max-value* 500]
  (println (valid-value? 299))
  (doto (Thread. #(println "in other thread:" (valid-value? 299)))
    .start
    .join))
;= true
;= in other thread: false
```

动态作用域被广泛地用在各种类库以及 Clojure 本身^{注 27} 的实现里面，来提供一种修改 API 默认配置的方法，这种方法不需要你显式地把配置一层层地传到最终使用这个配置的函数里面去，使得代码非常简洁。我们会在第 15 章以及第 14 章介绍一些实际应用的例子，在那里动态作用域被利用来提供数据库的配置信息给类库。

形象化动态作用域。为了更形象化地解释动态作用域的原理，假设有这样一个 `var`：它有一个根绑定，而且对于每个线程，你可能有任意数量的线程本地绑定，它们通过 `binding` 形成一个套一个的嵌套动态作用域。◀ 203

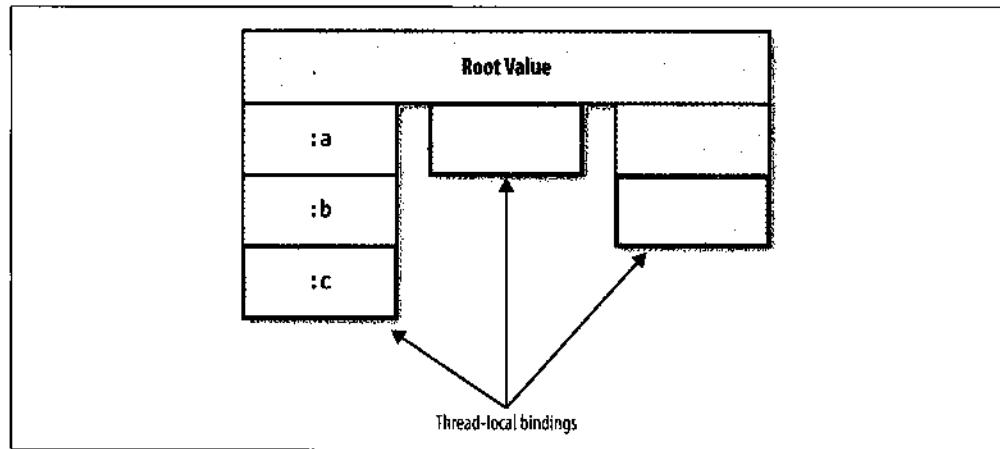


图4-5：一个var，它有一个根绑定，在每个线程上有一些线程本地绑定

注 26：请大家原谅这里使用了 Java 互操作的一些东西，但是这里要展示动态 `var` 的特性的话必须要使用 JVM 原生的线程。224 页的“使用 Java 的并发原语”以及第 9 章有关于这里为什么要使用原生线程的详细介绍。

注 27：比如 366 页“为了效率进行类型提示”以及 440 页“类型错误与警告”里面介绍的 `*warn-on-reflection*`；标准输出 `*out*`、标准输入 `*in*`、标准错误 `*err*`，还有一些对于 `binding` 的间接使用，比如 432 页“任意精度的小数操作和取整模式”介绍的 `with-precision`。

在这些绑定里面只有那些在栈顶的绑定值（图里面加粗的部分）会被访问到。一旦一个新的动态绑定被建立，那么在这个动态绑定的范围内，它外层的动态绑定就被覆盖掉了。所以这里 `*var*` 以及 `(get-*var*)` 将永远不会求值成 `:root`、`:a` 或者 `:b`：

```
(def ^:dynamic *var* :root)
:= #'user/*var*
(defn get-*var* [] *var*)
:= #'user/get-*var*
(binding [*var* :a]
  (binding [*var* :b]
    (binding [*var* :c]
      (get-*var*)))))
:= :c
```

每一层新的动态作用域都会向这个栈上面加一个新的“帧”：

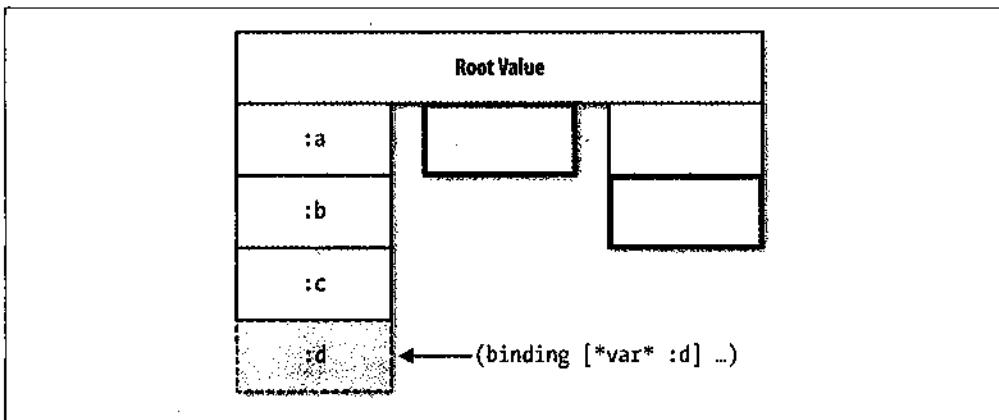


图4-6：使用binding给一个var建立一个新的本地绑定的效果

```
[204] (binding [*var* :a]
  (binding [*var* :b]
    (binding [*var* :c]
      (binding [*var* :d]
        (get-*var*)))))
:= :d
```

我们已经见识到了如何利用动态作用域来控制函数的行为，本质上其实是给函数提供了一个隐式的参数，而最后使用这个隐式参数的函数可能在函数调用树的最底层。关于动态作用域最后要介绍的一点是，它不仅可以把一个参数 / 配置从调用树的上层传到下层去，也能反过来让下层的函数通过这个动态作用域给上层的函数返回一些东西。

比如，虽然 Clojure 提供了一些方便得不可思议的 IO 函数来简单地获取一个 URL 的内容（比如，`slurp` 以及 `clojure.java.io` 命名空间里面的其他函数），但是如果需要这些

请求的返回码 (Response Code)，这些函数没有提供任何方法来让你获取（当你使用各种 HTTP API 的时候很可能需要处理返回码）。一个可能的办法是，始终把返回码跟请求的内容放在一个 vector 里面一起返回，[response-code url-content]：

```
(defn http-get
  [url-string]
  (let [conn (-> url-string java.net.URL. .openConnection)
        response-code (.getResponseCode conn)]
    (if (== 404 response-code)
        [response-code]
        [response-code (-> conn .getInputStream slurp)])))

(http-get "http://google.com/bad-url")
;= [404]
(http-get "http://google.com/")
;= [200 "<!doctype html><html><head>..."]
```

这个办法不算太糟糕，但是作为 http-get 的用户，这种方式迫使我们对于它的每次调用都要处理返回码，即使根本对它不感兴趣。◀ 205

而利用动态作用域的话，建立一个动态绑定，这样只有在我们对 HTTP 返回码感兴趣的时候 http-get 才会传回这个值：

```
(def ^:dynamic *response-code* nil) ❶

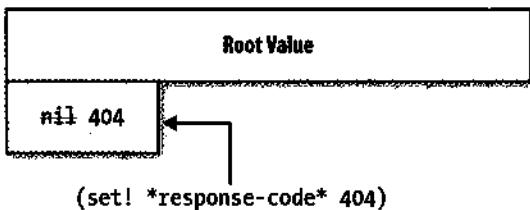
(defn http-get
  [url-string]
  (let [conn (-> url-string java.net.URL. .openConnection)
        response-code (.getResponseCode conn)]
    (when (thread-bound? #'*response-code*)
      (set! *response-code* response-code)) ❷
    (when (not= 404 response-code) (-> conn .getInputStream slurp)))) ❸

(http-get "http://google.com")
;= "<!doctype html><html><head>..."
*response-code*
;= nil
(binding [*response-code* nil]
  (let [content (http-get "http://google.com/bad-url")]
    (println "Response code was:" *response-code*)
    ; ... do something with `content` if it is not nil ...
  ))
;= Response code was: 404
;= nil
```

❶ 定义一个新的 var : *response-code*，http-get 的用户可以通过绑定这个 var 来返回这个返回码。

- ❷ 使用 `thread-bound?` 来检测 `http-get` 的调用者是否创建了一个 `*response-code*` 的线程本地绑定。如果没有绑定的话，那么我们什么都不做。
- ❸ 用 `set!` 来改变 `*response-code*` 的线程本地值，这样外层的调用者就可以访问到了。
- ❹ 现在，`http-get` 可以使用一个可选的 `*response-code*` 动态作用域来给调用者返回一些额外的信息，而它本身可以简单地返回 URL 的内容，而不用返回一个组合的 `[response-code url-content]` 了（假定这个 URL 不是 404）。

我们再以图形的方式展示一下：



206 因为 `set!` 会替换动态 `var` 在本地绑定的当前值，因此通过 `binding` 建立这个绑定的——不管这个调用者是在调用栈的上一层还是上 50 层的函数都可以拿到这个“返回值”。这个对于任意数量的 `var`，任意数量的绑定，也不管你通过 `set!` 设置的是什么类型的值都可以。这种灵活性使得我们能做很多事情，简单的如上面演示的返回一些附加的返回值，复杂的可以实现非本地的返回机制。

动态作用域会通过 Clojure 原生的并发形式进行传播。动态作用域的线程本地本质是非常有用的——它使得一个特定线程的运行跟其他线程完全隔离，但是如果有时候不想让它“隔离”却没有办法做到的话，比如，要把一个计算过程从一个线程转移到另外一个线程，那就有很大问题了。幸运的是，Clojure 的动态 `var` 绑定确实可以在线程之间进行传播，这种机制叫做“绑定传播”——当使用 `agent`（通过 `send` 和 `send-off`）、`future` 以及 `pmap` 和它的变种的时候：

```
(binding [*max-value* 500]
  (println (valid-value? 299))
  (@future (valid-value? 299)))
; true
;= true
```

虽然 `valid-value?` 是在另外一个独立的线程上被调用的，但是 `future` 把这个动态作用域传播到了这个线程上面。

不过要注意的是，虽然 pmap 是支持绑定传播的，但是一般惰性序列是不支持的：

```
(binding [*max-value* 500]
  (map valid-value? [299]))
:= (false)
```

这里一个可以绕过的办法是，手动将这个动态作用域“传播”到惰性序列中计算实际用到这个动态 var 的地方：

```
(map #(binding [*max-value* 500]
            (valid-value? %))
      [299])
:= (true)
```

var 不是变量

我们不应该把 var 和其他语言中的变量混淆在一起。如果用 Ruby 的话，你可能会写出类似这样的代码：

```
def foo
  x = 123
  y = 456
  x = x + y
end
```

◀ 207

对于 Clojure 的初学者来说，很容易会写出跟上面类似的代码：

```
(defn never-do-this []
  (def x 123)
  (def y 456)
  (def x (+ x y)
    x))
```

这样的代码在 Clojure 中来说是非常烂的。但是更糟糕的是，你知道下面的代码结果是什么吗？

```
(def x 80)
:= #'user/x
(defn never-do-this []
  (def x 123)
  (def y 456)
  (def x (+ x y))
  x)
:= #'user/never-do-this
(never-do-this)
:= 579
x
:= 579
```



❶ “喂！我在最开始的时候可是把 `x` 定义成 80 了的！”

`def` 定义的都是顶级 `var`——它并不是一个简单的赋值操作，它不只给一些本地绑定赋了值。这个例子里面的 `x` 和 `y` 在当前的命名空间中是全局可访问的，因此会重新定义你之前在其他地方对 `x` 和 `y` 已经赋过的值。

除了动态作用域这个例外，`var` 从本质上来说是被设计来保存一些值，然后直到程序或者 REPL 结束的时候都不再改变。如果你确实想要一种可以改变的东西，那么可以尝试 Clojure 提供的其他引用类型，定义一个 `var` 来保存它，然后利用合适的函数（`swap!`、`alter`、`send` 或者 `send-off` 等）来修改它的值。

修改一个 `var` 的根绑定。虽然我们已经提醒过好多次不要把 `var` 当做其他语言中的变量使用，但是在某些情况下确实需要对一个 `var` 的根绑定进行修改，这时候我们利用 `alter-var-root` 函数，它以这个 `var` 本身以及一个将要作用在 `var` 身上的函数作为参数：

```
(def x 0)
:= #'user/x
(alter-var-root #'x inc)
:= 1
```

如果这个 `var` 本身代表的是一个函数的话，那就可以用这种机制来实现面向切面的编程了，而且能够比其他面向切面编程框架提供的能力更强。我们会在 466 页“面向切面的编程”以及 351 页“构建多语言的项目”两节介绍一些具体的例子。

- 208 我们也可以通过 `with-redefs` 来暂时修改一些 `var` 的根绑定，它会在程序控制流离开它的作用域时把 `var` 的根绑定恢复到之前的状态，这个对于测试非常有用，可以让我们对一些函数或者函数所依赖的一些跟环境有关的变量进行 mock。472 页的“模拟”一节会介绍一些例子。

前置声明

在定义一个 `var` 的时候可以暂时不给它赋值，在这种情况下，称这个 `var` 是“未绑定”的，如果你对它进行解引用的话，它会返回一个“占位符”对象：

```
(def j)
:= #'user/j
j
:= #<Unbound Unbound: #'user/j>
```

这个特性在你需要先使用 `var`，后续代码再给它赋值的情况下很有用。比如要实现某种类型的算法，它的实现需要交替递归——或者你只是想把主要的、公共的 API 放在源文件的最前面，而这些公共 API 需要引用别的帮助函数。Clojure 编译和求值形式的时候

是按照源代码里面声明的顺序来进行求值的，因此要引用的任何 var 必须在引用之前定义好。假定这些 var 的值是只在运行时才需要的（比如，它们是某些函数的“占位符”），那么可以在后面再重新定义这些 var，给它们真正的值。这种编码方式就叫“前置声明”。

在这种情况下，更地道的做法是使用 declare 宏来声明。使用它而不是 def 可以更明确地告诉代码的读者你是在定义一个未绑定的 var（这样读者就不会猜测你是不是忘记给它赋值了），而且它允许你在一个表达式里面一次定义多个未绑定的 var：

```
(declare complex-helper-fn other-helper-fn)          ❶

(defn public-api-function
  [arg1 arg2]
  ...
  (other-helper-fn arg1 arg2 (complex-helper-fn arg1 arg2))    ❷

(defn- complex-helper-fn
  [arg1 arg2]
  ...)

(defn- other-helper-fn
  [arg1 arg2 arg3]
  ...)
```

- ❶ 在引用这些辅助函数 var 之前对它们进行前置声明。
- ❷ 尽量把主要的、公开的 API 放在源文件的最前面，而且可以自由地引用辅助函数。
- ❸ 这里再对辅助函数进行定义。

agent

209

agent 是一种不进行协调的、异步的引用类型。这意味着对于一个 agent 的状态的修改与对别的 agent 的状态的修改是完全独立的，而且发起对 agent 进行改变的线程跟真正改变 agent 值的线程不是同一个线程。agent 还有两个把它跟 atom 和 ref 区分开来的特点：

1. 可以很安全地利用 agent 进行 I/O 以及其他各种副作用操作。
2. agent 是 STM 感知的，因此它们可以很安全地用在事务重试的场景下。

agent 的状态可以通过两个函数：send 和 send-off 进行更新。使用风格上跟对其他引用类型进行更新是一样的：接受一个更新函数，这个更新函数以要更新的 agent 的当前状态以及额外的一些参数为参数，返回的结果则是这个 agent 的新状态。

这些元素加在一起：更新函数加上发送给 `send` 或者 `send-off` 额外的参数叫做一个 `agent action`，而每个 `agent` 维护一个 `action` 的队列。对于 `send` 和 `send-off` 的调用都会立马返回，它们只是简单地把这个 `action` 放到 `agent` 的 `action` 队列上去，然后这些 `action` 会以它们被发送过去的顺序在一些专门用来执行这些 `agent action` 的线程上串行执行。而每个 `action` 的结果都是 `agent` 的一个新状态。

`send` 和 `send-off` 的唯一区别是它们发送给 `agent` 的 `action` 类型。使用 `send` 发送的 `action` 是在一个固定大小的线程池中执行的，线程池的大小不会超过当前硬件的并行能力。²⁸ 因此，`send` 不适宜用来发送那种进行 I/O 操作的 `action`（或者其他类型的阻塞操作），因为它们会阻止其他 CPU 密集型 `action` 更好地利用计算机的计算资源。

而另外一方面，由 `send-off` 发送的 `action` 则会在一个不限制大小的线程池（跟 `future` 利用的是同一个线程池）中执行，因此适宜用来发送那些阻塞的、非 CPU 密集型的 `action`。

知道了这些之后，我们对 `agent` 到底是如何工作的有了一个大致的了解。

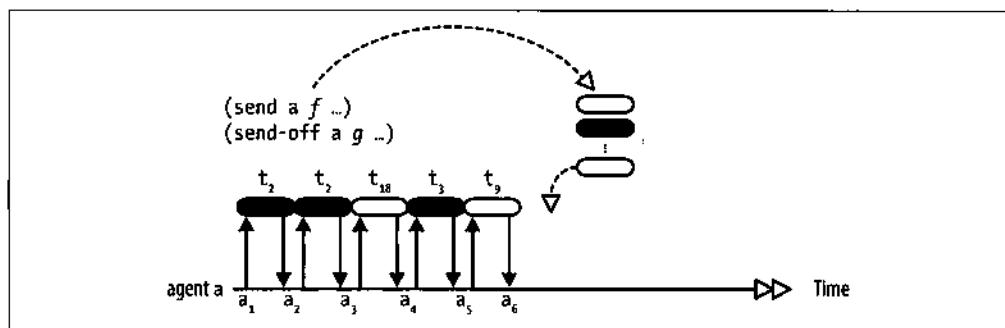


图4-7: `agent`的状态更新: `action`队列以及`action`的执行

action 是通过 `send` 或者 `send-off`（在图 4-7 中以不同的颜色来表示不同类型的工作）发送到 `agent` 的。`agent` 以 `action` 发送过来的顺序串行地执行这些 `action`，这个执行的操作会在一个跟发送这个 `action` 隔离的线程池里面的线程上执行。所以，如果黑色的 `action` 是 CPU 密集型的，那么线程 t_2 和 t_3 是从一个专门的、固定大小的 `send` 线程池中分配出来的，而 t_9 和 t_{18} 则是从一个对大小没有限制的 `send-off` 线程池里面分配出来的。每个 `action` 的返回值都会被设置为 `agent` 的新状态。

注 28：比如，对于一个双核 CPU，分配给 `send` 线程池最多 4 个线程，对于一个 4 核的 CPU 最多有 8 个线程等。

agent 并不跟线程一一对应

一个线程创建的 agent 的数目与为这些 agent 执行 action 的线程数目之间没有必然的联系。线程的数目更大程度上是由利用 send-off 发送的并发处理 action 数目决定的：因为它们是那种可以利用不受限制大小的线程池的 action，它们是那种可能会触发一个新的系统线程被创建的 action。记住一点：发送给 agent 的 action 都是被串行执行的，如果你想要强制创建 100 个线程来为你的 agent 服务，那么需要至少创建 100 个 agent，而每个都在并发处理由 send-off 发出的 action。

这意味着两件事情：

1. 只要内存足够，可以创建任意多个 agent。^{注 29}
2. 可以利用 send 来并发地发送任意多个 action，但是最终来执行这些 action 的线程数目是有限的，因此可以发送任意多个 action，唯一的限制是你的内存。

虽然 agent 的语义可能有点微妙，但是使用是非常简单的：

<211>

```
(def a (agent 500))          ①  
:= #'user/a  
(send a range 1000)          ②  
:= #<Agent@53d2f8be: 500>  
@a  
:= (500 501 502 503 504 ... 999)
```

① 创建了一个初始值是 500 的 agent。

② 利用 send 发送了一个 action 给 agent，这个 action 是由 range 以及一个额外的参数 1000 组成的；在另外一个线程上面，agent 的值会被设置成 (range @a 1000) 的结果。

send 和 send-off 都会返回相关联的 agent。当在 REPL 里面发送 action 的时候，有可能在解引用 agent 的时候马上看到它的更新值，取决于 action 逻辑的复杂度以及多快被调度，如果快的话，在刚调用完 send 或者 send-off 发送 action 之后去打印 agent 的值的时候，那个 action 就已经执行完成了：

```
(def a (agent 0))  
:= #'user/a  
(send a inc)  
:= #<Agent@65f7bb1f: 1>
```

注 29：对于一个默认配置的 heap，你可以创建数千个 agent，对 JVM 的 heap 做一些调整的话可以创建数百万个 agent。

而另一方面，你的代码可能需要等当前线程发送给 agent 的所有 action 都求值完成才能继续执行，你可以使用 `await` 函数：^{注 30}

```
(def a (agent 5000))
(def b (agent 10000))(send-off a #(Thread/sleep %))
;= #<Agent@da7d7b5: 5000>
(send-off b #(Thread/sleep %))
;= #<Agent@c0cd75b: 10000>
@a
;= 5000
(await a b)
;= nil
@a
;= nil
```

① 发送到 a 的函数需要 5 秒执行才能完成，因此现在它的值还没有被更新。

- 212 > ② 可以使用 `await` 来等待所有从这个线程上发送到 agent 的 action 都求值完。当前的这个调用会阻塞 10 秒钟，因为这就是发送到 b 的函数执行所需要的时间。
- ③ 当 `await` 返回之后，所有发送给 agent 的 action 都被执行了，agent 的状态也被更新了。需要注意的是，在对 a 进行解引用获取它的值之前可能有别的线程把它的值又给改变了！

还有一个 `await-for` 函数的作用是一样的，但是你可以提供一个超时时间。

处理 agent action 中的错误

因为 agent action 是异步执行的，从 action 里面抛出的一个异常跟发送这个 action 的不是一个线程。对这种情况的默认处理策略是这个 agent 默默地挂掉：你还是可以对它进行解引用以获取它最后的状态，但是进一步给它发送 action 会失败：

```
(def a (agent nil))
;= #'user/a
(send a (fn [_] (throw (Exception. "something is wrong"))))
;= #<Agent@3cf71b00: nil>
a
;= #<Agent@3cf71b00 FAILED: nil>
(send a identity)
;= #<Exception java.lang.Exception: something is wrong>
```

注 30：这里介绍一个实现细节——可能在将来的版本发生变化，不过目前你可以通过调用 (`.getQueueCount some-agent`) 来检查 `some-agent` 的 action 队列中有多少 action。

- ① 尝试给一个已经挂掉的 agent 发送 action 会返回导致这个 agent 挂掉的异常。如果想显式地去获取这个异常，可以使用 `agent-error` 函数，它会返回这个异常或者 nil——如果这个 agent 并没有挂掉的话。

可以通过 `restart-agent` 来重启一个挂掉的 agent，它会将 agent 的状态重置成我们提供的值，并且使它能够继续接收 action。它还接受一个可选的标志参数 `:clear-actions`，如果指定了这个参数的话，它会把这个 agent 挂掉的时候它上面阻塞着的所有 action 清除掉。否则，这些阻塞的 action 在 agent 重启之后会被立即执行，从而可能使得 agent 重启之后马上又挂掉。

```
(restart-agent a 42)
;= 42
(send a inc)                                ①
;= #<Agent@5f2308c9: 43>
(reduce send a (for [x (range 3)]           ②
                    (fn [_] (throw (Exception. (str "error #" x))))))
;= #<Agent@5f2308c9: 43>
(agent-error a)
;= #<Exception java.lang.Exception: error #0>
(restart-agent a 42)
;= 42
(agent-error a)                                ③
;= #<Exception java.lang.Exception: error #1>
(restart-agent a 42 :clear-actions true)       ④
;= 42
(agent-error a)
;= nil
```

213

- ① 重启一个 agent 会重置它的失败状态并且使得它可以继续接受 action。
- ② 但是，如果这个 agent 的队列里面阻塞的其他 action 会导致进一步的错误的话……
- ③ 那么要再次调用 `restart-agent` 函数。
- ④ 通过添加 `:clear-actions` 这个标志位，`restart-agent` 会把这个 agent 队列上面阻塞的 action 全部清除掉，保证这些阻塞的 action 不会在 agent 重启之后马上又使它挂掉。

默认的错误处理模式——agent 会被设置成失败状态，需要重启才能继续工作——这种方式通常是在人工可以干预程序执行的时候有用，比如通过 REPL。^{注31} 通过改变 agent 的默认值可以灵活、非手工地来处理异常。

注 31： 也就是说，通过一个 REPL 连接到应用环境。在 411 页“在 REPL 里调试、监测和打补丁”一节有介绍。

agent 的错误处理器以及模式

默认的错误处理行为：一个错误会导致 agent 进入失败状态是 agent 支持的两种失败模式之一。agent 接受一个 :error-mode 参数，它有两个可选值 :fail (默认值) 以及 :continue，^{注32} 如果一个 agent 的错误处理状态被设置成 :continue，那么当 action 在执行的时候会抛出异常，这个异常会被直接忽略掉继续执行队列中的其他 action，同时也可以继续接收新的 action：

```
(def a (agent nil :error-mode :continue))
;= #'user/a
(send a (fn [_] (throw (Exception. "something is wrong"))))
;= #<Agent@44a5b703: nil>
(send a identity)
;= #<Agent@44a5b703: nil>
```

这就使得我们没有必要去调用 restart-agent 了，但是把发生的错误默默地忽略掉而不做任何处理从来都不是好主意。因此在使用 :continue 错误处理模式的时候，我们始终会指定一个错误处理器，这是一个接受两个参数（发生错误的 agent 以及这个异常对象）的函数，它会在 agent action 抛出异常时调用，在创建 agent 的时候可以通过指定 :error-handler 来指定错误处理器：^{注33}

```
(def a (agent nil
  :error-mode :continue
  :error-handler (fn [the-agent exception]
    (.println System/out (.getMessage exception)))))

;= #'user/a
(send a (fn [_] (throw (Exception. "something is wrong"))))
;= #<Agent@bb07c59: nil>
; something is wrong
(send a identity)
;= #<Agent@bb07c59: nil>
```

214

这里只是简单地把异常信息打印到控制台，当然可以利用 :error-handler 函数做更加有用的处理：应用里面的某些数据可能需要被修改一下以避免这个错误，某些 action 或者操作可能需要重试一下，或者这个 agent 的错误模式可能需要设置回 :fail——如果知道关闭这个 agent 可能是唯一安全的办法：

```
(set-error-handler! a (fn [the-agent exception]
  (when (= "FATAL" (.getMessage exception))
    (set-error-mode! the-agent :fail)))
;= nil
(send a (fn [_] (throw (Exception. "FATAL"))))
;= #<Agent@6fe546fd: nil>
```

注 32：可以通过 set-error-mode! 来修改 agent 的错误模式。

注 33：可以通过 set-error-handler! 来修改一个 agent 的错误处理器。

```
(send a identity)
:= #<Exception java.lang.Exception: FATAL>
```

I/O、事务以及嵌套的 Send

跟 ref 和 atom 不同的是，可以非常安全地利用 agent 来协调 I/O 或者其他类型的阻塞操作。这使得 agent 成为任何使用 ref 以及 Clojure 的 STM 来维持状态的程序中不可或缺的一个组件。而且由于 agent 独特的语义使得它们成为简化涉及异步 I/O 操作的理想组件——即使你不使用 ref。

因为 agent 会串行执行所有发送给它的 action，因此它给有副作用的操作提供一个很自然的同步点。你可以建立一个 agent 来保持比如一个指向文件或者网络 socket 的 OutputStream、一个数据库的链接，或者消息队列的链接等。你可以确定发送到 agent 的每个 action 在执行期间都能独占这个连接。这使得它非常容易融合到 Clojure 的环境中来——包括 ref 和 atom，它们的目标就是最小化它们跟周围环境的副作用。

你可能在想，agent 怎么能用在 STM 的事务中呢？发送一个 agent action 本身就是一个有副作用的操作，那么看起来似乎不能把它放在一个事务里面去执行，因为会重试。进而可能会导致副作用发生多次，从而导致最终结果不对。不过实际上不是这样的。

agent 是跟 Clojure 的 STM 实现紧密融合的，在一个事务内通过 send 和 send-off 发送的 action 会被保持到事务成功提交。这意味着如果一个事务被重试 100 次，被发送给 agent 的 action 也在事务成功提交之后被执行一次。类似的，在一个 agent action 内部，对 send 和 send-off 的调用称为“嵌套发送”——也会被保持到直到这个 action 完成。在这两种情况下，被发送的 action 都可能因为一个校验器的失败而被全部抛弃掉。

215

为了展示这些语义并且看看它们到底能给我们带来什么，让我们通过几个例子看看如何使用 agent 跟 ref 以及 STM 合作，在一个高度并行化、高 IO 负载的情况下简化对 I/O 操作的协调。

利用 agent 来记录引用状态的变更日志

在 181 页“对 Ref 进行修改的细节”一节开发的游戏使用 ref 来保持角色的状态，证实了 Clojure 的 STM 在这种多人并发场景下的能力。但是任何跟这个类似的游戏，特别是那种多个玩家同时玩的游戏，都会对玩家的行为进行跟踪和保存，以及它们的角色所受的影响（加血、减血）。当然我们不会把这种类型日志、持久化以及其他 I/O 相关的东西写进核心游戏引擎：任何想做的持久化最终会由于事务重启而不一致。

解决这个问题最简单的方法是利用监视器以及 agent 来给游戏中的角色实现一个“后写”日志。首先来规划一下 agent 中要保存的东西，比如在这个例子中，会让所有的 agent 里

面包含一个 `java.io.Writer`——这是 Java 中定义输出流的 API：

```
(require '[clojure.java.io :as io])(def console (agent *out*))  
(def character-log (agent (io/writer "character-states.log" :append true)))
```

其中一个 agent 会包含 `*out*`（它也是一个 Writer），另外一个包含一个指向当前目录的文件 `character-states.log` 的 Writer，这些 Writer 实例会把通过 `action` 发送到它的内容写入这些输出介质，`write`：

```
(defn write  
  [^java.io.Writer w & content]  
  (doseq [x (interpose " " content)]  
    (.write w (str x)))  
  (doto w  
    (.write "\n")  
    .flush))
```

`write` 函数的第一个参数是 Writer（使用当前 agent 状态中的那个 writer），其他参数就是要写的内容。它会在每个要写的内容之间加一个空格，在最后加一个换行，然后调用 Writer 的 `flush` 函数，以把这些输出内容实实在在地写到磁盘或者命令行上，而不是滞留在 Writer 的内部缓存中。

最后，需要一个函数来给每个引用类型加一个监视器函数，这个监视器函数会把引用类型跟 agent 连接起来：

216>

```
(defn log-reference  
  [reference & writer-agents]  
  (add-watch reference :log  
    (fn [_ reference old new]  
      (doseq [writer-agent writer-agents]  
        (send-off writer-agent write new))))
```

然后每次引用类型的状态发生变化的时候，它的新的状态都会跟 `write` 函数一起被发送到由 `log-reference` 提供的那些 agent 那里。现在需要做的就是给每个想要记录状态变化的角色添加监视器，然后让这些角色重新再打一战：

```
(def smaug (character "Smaug" :health 500 :strength 400))  
(def bilbo (character "Bilbo" :health 100 :strength 100))  
(def gandalf (character "Gandalf" :health 75 :mana 1000))  
  
(log-reference bilbo console character-log)  
(log-reference smaug console character-log)  
  
(wait-futures 1  
  (play bilbo attack smaug)  
  (play smaug attack bilbo))
```

```

(play gandalf heal bilbo));

{:max-health 500, :strength 400, :name "Smaug", :items #{}, :health 490.052618}
; {:max-health 100, :strength 100, :name "Bilbo", :items #{}, :health 61.5012391}
; {:max-health 100, :strength 100, :name "Bilbo", :items #{}, :health 100.0}❶
; {:max-health 100, :strength 100, :name "Bilbo", :items #{}, :health 67.3425151}
; {:max-health 100, :strength 100, :name "Bilbo", :items #{}, :health 100.0}
; {:max-health 500, :strength 400, :name "Smaug", :items #{}, :health 480.990141}
; ...

```

- ❶ 在命令行中看到 Gandalf 每次给 Bilbo 加血的时候，:health 增长的记录。

会在 *character-states.log* 文件里面看到一样的内容。从根本上来说，把状态变化记录到命令行以及文件里面去了，因为它们是最常见的介质，可以用相同的方式把这些状态变化写入一个数据库、一个消息队列等。

以这样的方式来使用监视器使得我们可以在不修改业务函数的前提下把感兴趣的状态变化持久化（比如把它们写入磁盘或者数据库）。

为了监控和持久化一个事务内的信息——比如每次攻击的伤血量、每次的补血量、谁攻击了谁等，只需要在对应的函数里面把 write 函数分发到 writer：

```

(defn attack
  [aggressor target]
  (dosync
    (let [damage (* (rand 0.1) (:strength @aggressor) (ensure daylight)))
        (send-off console write
          (:name @aggressor) "hits" (:name @target) "for" damage)
        (commute target update-in [:health] #(max 0 (- % damage)))))

(defn heal
  [healer target]
  (dosync
    (let [aid (min (* (rand 0.1) (:mana @healer))
                  (- (:max-health @target) (:health @target)))]
      (when (pos? aid)
        (send-off console write
          (:name @healer) "heals" (:name @target) "for" aid)
        (commute healer update-in [:mana] - (max 5 (/ aid 5)))
        (alter target update-in [:health] + aid)))))

(dosync
  (alter smaug assoc :health 500)
  (alter bilbo assoc :health 100)
; {:max-health 100, :strength 100, :name "Bilbo", :items #{}, :health 100}
; {:max-health 500, :strength 400, :name "Smaug", :items #{}, :health 500}

(wait-futures 1

```

217

```
(play bilbo attack smaug)
(play smaug attack bilbo)
(play gandalf heal bilbo)
; {:max-health 500, :strength 400, :name "Smaug", :items #{}, :health 497.414581}
; Bilbo hits Smaug for 2.585418463393845
; {:max-health 100, :strength 100, :name "Bilbo", :items #{}, :health 66.6262521}
; Smaug hits Bilbo for 33.373747881474934
; {:max-health 500, :strength 400, :name "Smaug", :items #{}, :health 494.667477}
; Bilbo hits Smaug for 2.747103668676348
; {:max-health 100, :strength 100, :name "Bilbo", :items #{}, :health 100.0}
; Gandalf heals Bilbo for 33.37374788147494
; ...
```

把我们学到的 agent 技巧一起使用的结果是得到一个非常棒的引用类型持久化的机制，你可以安全地把它使用在可能会发生重试的事务里面。而如果只是使用 atom 以及 ref 事务是根本做不到的。这里为了例子的简单，把内容输出到了命令行以及日志文件里面了，但是对代码稍作修改就可以把它们写入数据库。不管怎样，这个例子展示了跟使用 atom 和 ref 类似，对于这种即使是在一个并发的环境下共享 I/O 资源也可以在不去使用锁的情况下完成。

利用 agent 来并行化工作量

一开始看起来可能很奇怪，为什么要把发送 action 的方式分成两种。但是如果不能区分阻塞性和非阻塞性的 action，我们就丧失了利用它们最有效地使用不同资源的能力——比如 CPU、磁盘 IO、网络等。

218 比如，我们的应用专门从一个消息队列中读取消息出来，读取消息是一个阻塞的动作，因为队列可能不是线程内的，而且根据队列的语义如果队列里面没有东西，线程要等在上面。但是对于取出来的每个消息的处理却是 CPU 密集型的。

这听起来很像是一个网络爬虫。利用 agent 可以很轻松地编写出一个高效可扩展的网络爬虫。我们这里要编写的将是一个非常“基本”的网络爬虫^{注34}，但是它可以给大家展示如何利用 agent 来并行地处理一些高负载的工作。

首先，需要一些基本的函数来对获取到的页面内容进行处理。links-from 以一个 URL 以及这个 URL 的 HTML 内容作为参数，返回从这个 HTML 内容里面获取到的链接的序列；words-from 则以网页的 HTML 内容为参数，抽取出其中的文本，返回这个网页里面发现的所有单词，并且转换成小写：

```
(require '[net.cgrand.enlive-html :as enlive])
(use '[clojure.string :only (lower-case)])
```

注 34： 并且表现得不是很好，因为爬虫不能给所爬网站过重的负载，这是一个好的爬虫的一个关键属性。我们这里要向 BBC 道个歉，它成了这个例子的受害者。

```

(import '(java.net URL MalformedURLException))

(defn- links-from
  [base-url html]
  (remove nil? (for [link (enlive/select html [:a])]
    (when-let [href (-> link :attrs :href)]
      (try
        (URL. base-url href)
        ; ignore bad URLs
        (catch MalformedURLException e))))))

(defn- words-from
  [html]
  (let [chunks (-> html
    (enlive/at [:script] nil)
    (enlive/select [:body enlive/text-node]))]
    (->> chunks
      (mapcat (partial re-seq #"\w+"))
      (remove (partial re-matches #"\d+"))
      (map lower-case))))

```

这段代码使用了 Enlive 库，这是一个对 HTML 内容进行抽取解析的库，我们会在 546 页“Enlive：基于选择器的 HTML 转换”一节详细介绍，但是关于 Enlive 的细节不是这里讨论的重点，我们要讨论的是如何利用 agent 来最大化爬虫的吞吐量。

跟爬虫相关的有三个“池”：

1. 我们会用一个 Java 中线程安全的队列来保存需要抓取的链接，并把它命名为 url-queue。然后对于每个要抓取的页面，我们会……
2. 找出页面中所有的链接，它们是接下来要抓取的，我们会把它们加入到 url-queue 里面去，而爬过的链接会将其加入到一个名叫 crawled-urls 的 atom 里面去。最后……
3. 要把每个页面里面的所有文本抽取出来，我们会从这个文本里面计算出每个单词出现的次数，然后把这个次数保存在一个叫做 word-freqs 的 atom 里面：

```

(def url-queue (LinkedBlockingQueue.))
(def crawled-urls (atom #{}))
(def word-freqs (atom {}))

```

为了完全充分地利用资源，我们会创建很多的 agents^{注35}，但是要考虑一下这些 agent 各自保存什么状态以及要给它们发送什么样的 action。在很多情况下可以把 agent 的状态以及这些导致状态变化的 action 联系起来画出一个有限状态机；我们前面已经大致了解

注 35：这个基本网络爬虫的另外一个缺点是：实际使用中的任何规模的爬虫都会用一个消息队列或者合适的数据库来保存状态——而不是内存。不过这一点丝毫不影响这个例子的主要目的，可以很简单地把例子代码修改成存入数据库。

了一下爬虫的工作流，这里用图表示出来。

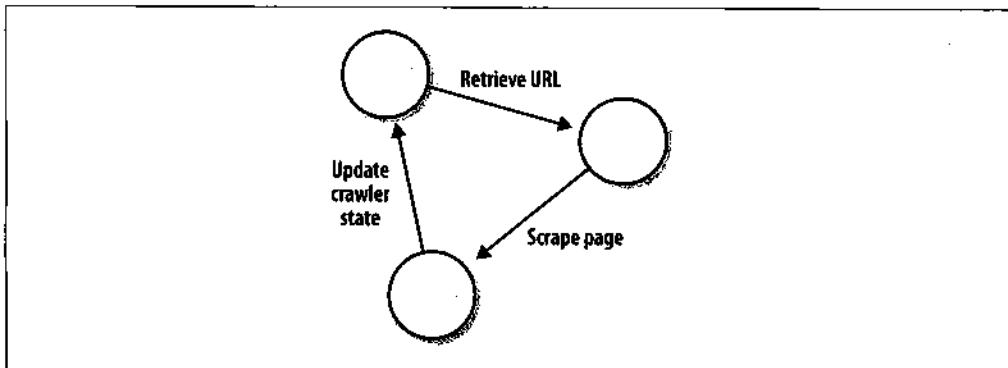


图4-8：一个网络爬虫的主要状态转变

agent 在这个流程上的每个点的状态应该是很明显的：在抓取一个 URL 之前，agent 需要知道要抓取哪个 URL；在抽取网页内容之前，agent 需要这个网页的内容；在更新爬虫状态之前，爬虫需要抽取网页的结果。因为不会有很多的状态，因此可以让每个 action 告知 agent 它下一步的 action 是什么，这样对编码来说会简单一点。

220> 让我们来定义所有的 agent，它们的初始状态，对应到图 4-8 中的“Retrieve URL”，是一个包含下一步要抓取的 URL 的队列的 map，以及 agent 要做的事情本身，我们把它定义为函数：get-url：

```
(declare get-url)

(def agents (set (repeatedly 25 #(agent {::t #'get-url:queue url-queue}))))*)
```

❶ agent 的状态始终会有一个 ::t 字段，^{注 36} 它里面保存这个 agent 下一步要做的事情的函数。^{注 37}

图 4-8 里面的 3 个转换是通过 3 个 agent action 来实现的：get-url、process 和 handle-results。

get-url 会等待在一个队列上面（记住，每一个 agent 都是以 url-queue 作为初始状态的）去获取要爬去的 URL。它会把 agent 的状态设置成它抓取出来的网页 URL 到网页内容的映射的一个 map：

注 36： 这里使用一个全限定的关键字以避免和用户加进状态的关键字名字冲突——如果这个爬虫会在当前这个命名空间之外使用的话。

注 37： 取决于要用 agent 保存的状态的数目，如果多的话，可以通过发送给 agent 一个多重方法或者协议函数来去掉这些状态转移，我们会分别在第 7 章和 264 页的“协议(Protocols)”介绍多重方法和协议。

```
(declare run process handle-results)

(defn ^::blocking get-url
  [{:keys [^BlockingQueue queue] :as state}]
  (let [url (as-url (.take queue))]
    (try
      (if (@crawled-urls url)
          state
          {:url url
           :content (slurp url)
           ::t #'process})
      (catch Exception e
        ;; skip any URL we failed to load
        state)
      (finally (run *agent*))))))
```

❶

❷

❶ 我们会在后面介绍 run 的实现。

❷ 如果从队列里面取出来的链接已经抓取过了（或者在抓取这个网页的内容时碰到了问题），就不去改变 agent 的状态。这个实现细节在前面的那个有限状态机上面加了一个圈——也就是说，get-url 有时候会在一个单个 agent 上面执行多次之后，agent 的状态才会发生流转。

process 会解析网页的内容，可使用 links-from 和 words-from 两个函数来获取这个网页里面的链接以及每个单词出现的次数。它会把 agent 的状态设置成这两个东西以及这个被解析的链接：

<221

```
(defn process
  [{:keys [url content]}]
  (try
    (let [html (enlive/html-resource (java.io.StringReader. content))]
      (::t #'handle-results
            {:url url
             :links (links-from url html)
             :words (reduce (fn [m word]
                             (update-in m [word] (fn[il inc 0])))
                           {}
                           (words-from html)))
            (finally (run *agent*))}))
```

❶

❶ :words 保存的是当前网页里面每个单词出现的次数，它是通过对所有单词进行归约 (reduce) 计算出来的。这里的 fnil 是一个高阶函数，它返回一个给参数加上默认值的函数（这里是 0），它使得我们不用显式地去检查参数是不是 nil，如果是 0 的话会直接返回 1。

handle-results 会更新三个关键状态数据：把刚刚爬过的链接加入到 crawled-urls 中，

把每个新发现的链接放到 url-queue 中，把刚刚统计的单词出现次数 map 合并到 word-freqs 里面。handle-results 返回一个包含 url-queue 以及 get-url 作为状态转换的 map，因此也就把 agent 设置回了最开始的状态。

```
(defn ^::blocking handle-results
  [{:keys [url links words]}]
  (try
    (swap! crawled-urls conj url)
    (doseq [url links]
      (.put url-queue url))
    (swap! word-freqs (partial merge-with +) words)

    {::t #'get-url :queue url-queue}
    (finally (run *agent*))))
```

你可能已经注意到了，每个作为 agent action 的函数都使用一个 try finally 结构，而 finally 里面包含了对 run 函数的调用，而 *agent* 则是它唯一的参数。^{注38} 我们并没有在任何地方定义过 *agent*，通常这个 var 是没有绑定的，这个 var 是 Clojure 提供的，当在一个 agent action 里面的时候，它被绑定到当前的 agent。因此，在每个 action 里面对于 (run *agent*) 的调用，其实都是传递给 run 当前正在处理这个 action 的 agent。

这种用法在使用 agent 的时候很常见，它使得 agent 可以连续运行。在爬虫的例子里面，
222 run 是一个根据 agent 状态里面的 ::t 属性来决定 agent 下一个要执行什么 action 的函数。如果每个 action 都知道自己后面要执行什么函数的话，那为什么还要加这一层 run 在这里呢？有两个原因：

1. 虽然技术上可以让每个作为 agent action 的函数知道它们 agent 下一步执行的 action 是什么，但是没有办法让它知道下一个 action 到底是不是阻塞的。这种事情最好还是留给转换本身来做，因此 run 会根据转换的元数据里面有没有 ::blocking 来判断每个 action 是应该用 send 还是应该用 send-off 来进行分发。^{注39}
2. run 会检查这个 agent 有没有被打上“暂停”的标记——看看它的状态的元数据里面有没有 ::paused。

示例4-8：run，网络爬虫的“主循环”

```
(defn paused? [agent] {::paused (meta agent)}))
```

```
(defn run
  ([] (doseq [a agents] (run a)))
```

注 38：已经在 3 个函数里面写过类似的代码了；如果再写这些重复的代码就是没脑子的表现了，我们可以写一个宏来把这个公共的逻辑抽出来。

注 39：这里对元数据的访问部分解释了为什么要用函数对应的 var 而不是函数本身来表示状态转移。除了这一点，使用 var 使得我们可以更容易地修改爬虫的行为，415 页的“重定义结构的限制”一节有解释原因。

```
([a]
  (when (agents a)
    (send a (fn [{transition ::t :as state}]
      (when-not (paused? *agent*)
        (let [dispatch-fn (if (-> transition meta::blocking)
          send-off
          send)]
          (dispatch-fn *agent* transition)))
      state)))))
```

`run` 可以用来启动一个 agent，当不给它传递参数的时候，它也可以用来启动所有的（没有被暂停的）agent。

可以暂停是非常重要的，因为你不可能希望你的爬虫一直运行而没有办法中断。可以通过使用元数据来告诉 `run` 暂停处理这个 agent 的下一个状态，这里 `pause` 和 `restart` 两个方法展示了通过元数据来暂停和重启 agent 的方法：

```
(defn pause
  ([] (doseq [a agents] (pause a)))
  ([a] (alter-meta! a assoc ::paused true)))

(defn restart
  ([] (doseq [a agents] (restart a)))
  ([a]
    (alter-meta! a dissoc ::paused)
    (run a)))
```

223

现在可以去爬行网页了！因为每次执行的时候要以一个全新的状态开始，所以最好开发一个函数来对那些状态进行重置。`test-crawler` 会做这个事情，而且它会把爬虫要爬的起点链接加入到 `url-queue` 里面去，而且为了方便，我们调整参数进行多次运行以进行对比，我们让程序每次运行 60 秒：

```
(defn test-crawler
  "Resets all state associated with the crawler, adds the given URL to the
  url-queue, and runs the crawler for 60 seconds, returning a vector
  containing the number of URLs crawled, and the number of URLs
  accumulated through crawling that have yet to be visited."
  [agent-count starting-url]
  (def agents (set (repeatedly agent-count
    #(agent {:::t #'get-url :queue url-queue})))) ❶
  (.clear url-queue)
  (swap! crawled-urls empty)
  (swap! word-freqs empty)
  (.add url-queue starting-url)
  (run)
  (Thread/sleep 60000))
```

```
(pause)
[(count @crawled-urls) (count url-queue)])
```

- ❶ 在前面提醒过大家很多次，不要在一个函数体内部重新定义一个 var（参见 206 页“var 不是变量”一节，但是还是有极少数的几种情况下是可以这么做的，比如在一个 REPL 里面定义函数，这个函数只是自己调试、试验用的，不可能会被外部调用。

为了建立一个基准线，先试试用一个 agent，并且把 BBC 的新闻页面作为抓取的起点：

```
(test-crawler 1 "http://www.bbc.co.uk/news/")
;= [86 14598]
```

一分钟内爬了 86 个页面。我们可以做得更好！使用 25 个 agent，这样计算机的网络和 CPU 都可以得到更充分的利用：

```
(test-crawler 25 "http://www.bbc.co.uk/news/")
;= [670 81775]
```

不错！60 秒内爬了 670 个页面，这么小的一个数字调整带来了效率上的指指数级提升。^{注 40}

来看看计算出来的单词频度。可以很简单地算出使用最多的词组以及使用最少的词组：

```
224 -> (sort-by val @word-freqs)
reverse
(take 10))
;= ([["the" 23083] ["to" 14308] ["of" 11243] ["bbc" 10969] ["in" 9473]
;= ["a" 9214] ["and" 8595] ["for" 5203] ["is" 4844] ["on" 4364])
(-> (sort-by val @word-freqs)
(take 10))
;= ([["relieved" 1] ["karim" 1] ["gnome" 1] ["brummell" 1] ["mccredie" 1]
;= ["ensinar" 1] ["estrictas" 1] ["arap" 1] ["forcibly" 1] ["kitchin" 1]])
```

现在看起来已经是一个可以正常工作的网络爬虫了。当然它不是很完美，就像之前说过的，它只实现了一些很基本的功能，如果要拿它到正常环境中去使用的话还需要做很多细致的调整，但是大致的样子就是这样了。

现在再回过头看看在这一节开始说过的，把这些 agent action 分成阻塞型（比如 I/O 或者其他需要等待的操作）和非阻塞型的操作（比如 CPU 密集型）可以最大化对于资源的利用。可以对于这一点进行测试，比如把 process 标示成阻塞的，这样它在发给 agent 的时候会使用 send-off 函数，从而总是会由不限制大小的线程池来处理：

```
(alter-meta! #'process assoc ::blocking true)
;= {:arglists ([{:keys [url content]}]), :ns #<Namespace user>,
;= :name process, :user/blocking true}
```

注 40：当然，实际测量出来的值跟你 CPU 的配置、网络链接速度以及延迟都有关系；不过从 1 到 25 个 agent 的这种相对提升应该是类似的。

这个小调整直接的后果是对于 HTML 的解析、抽取链接、抽取单词频率都会不受线程数目限制地进行。

```
(test-crawler 25 "http://www.bbc.co.uk/news/")
;= [573 80576]
```

这其实会给整体吞吐量带来负面影响——总体大概降了 15% 左右，因为现在有 25 个非常活跃（饥渴）的 agent 在抢 CPU 的资源，挤占了其他 CPU 密集型计算任务的时间。

使用 Java 的并发原语

到这里，我们已经深入介绍了 Clojure 的并发以及状态管理特性，值得指出的是，Java 的原生线程、锁原语以及它的一些非常有用的并发库——特别是 `java.util.concurrent.*` 这个包中的一些类在 Clojure 中都是非常有用的，而且 `java.util.concurrent` 这个包在 Clojure 自身的并发原语的实现过程中也有很广泛的使用，但是 Clojure 并没有对它们进行包装，因此我们也应该对它们好好掌握，以在合适的时候使用。

我们还没有去学习 Clojure 中所有与 Java 互操作相关的知识，这个会在第 9 章详细介绍，不过这里介绍的例子很简单，你应该能够看懂。

Java 里面定义了一些关键的接口——`java.lang.Runnable` 和 `java.util.concurrent.Callable`，Clojure 中所有无参函数都实现了这两个接口。这意味着你可以把无参函数传给任何需要这两个接口对象作为参数的 Java API，其中就包括 `Thread`：

```
(.start (Thread. #(println "Running...")))
;= Running...
;= nil
```

`java.util.concurrent.*` 包中提供了很多并发组件，这些组件在实现 Clojure 自身特性的时候也使用了，这些类在合适的时候也可以用在自己的应用里面。已经在 217 页“利用 Agent 来并行化工作量”一节使用过一种线程安全的队列实现：`LinkedBlockingQueue`，还有很多其他跟这个类似的但是在性能和语义上又有一些微妙但是有重要区别的类。然后还有一些线程池、线程安全的并发数据结构（如果你的程序需要跟一些 Java 代码共同对一些 Java 对象进行操作的话，那么它们将是比普通的数据结构如 `java.util.HashMap` 更好的选择），一些特别用途的对象比如 `CountDownLatch`，它使你可以阻塞一个线程（或者 `future`，或者由 `send-off` 发出的 Agent action）直到一些事情发生、一些条件满足为止。

如果想知道如何高效地使用这些并发类以及如何深入理解 JVM 底层的并发细节，推荐你阅读 Goetz 等人写的 *Java Concurrency in Practice* 一书。

Locking

即使 Clojure 提供了这么多（更安全的）并发原语，有时候还是需要使用底层的锁的，特别是当跟可变的 Java 类，比如数组打交道的时候。当然，一旦决定了要使用锁，那么你要知道：你不再能享受 Clojure 提供的并发原语的语义了。在任何情况下，你可以使用 `locking` 宏来获取一个给定对象上的锁，然后在程序控制流离开 `locking` 的时候，这个锁会自动释放。

因此，下面的 Clojure 代码：

```
(defn add
  [some-list value]
  (locking some-list
    (.add some-list value)))
```

跟下面对应的 Java、Ruby 以及 Python 代码效果是一样的：

```
// Java
public static void add (java.util.List someList, Object value) {
    synchronized (someList) {
        someList.add(value);
    }
}

# Ruby
require 'thread'
m = Mutex.new

def add (list, value)
    m.synchronize do
        list << value
    end
end

# Python
import threading
lock = threading.Lock()
def add (list, value):
    lock.acquire()
    list.append(value)
    lock.release()
```

总结

并发编程是很难的，而很多流行编程语言的设计使得这个问题更难。通过对标识和状态清楚地区分，提倡使用不可变的值，并且提供内置的、安全的并发编程的工具，Clojure 使得并发编程更加简单、可控。

构建抽象

宏

在计算机的历史上，Lisp 被称为“可被编程的语言”。这个描述同样适用于 Clojure，很大部分的原因是宏。宏是一种让程序员可以以其他语言里面很难、甚至不可能的方式来对语言进行扩展的机制。

编程语言是一种用来构建抽象的方式。它使得程序员可以避免无聊的手工重复工作，从而可以写一段代码块之后，把这个代码块作为可重用的单元。这个代码块可以被用在一个循环中反复执行；或者可以给这个代码块一个名字来定义成一个函数；或者如果这个代码块中使用了条件判断，那么相同的代码在不同的场景下还可以做不同的事情。

在那么多的编程语言里面，有的语言提供比其他语言更强大的抽象工具。想象一下，一个没有“循环”的语言。这样的一种语言也是可以用的，但是你需要手动去“循环”，这将是非常无聊的事情。类似的，一个没有函数的语言可以做任何“图灵完备的”语言能做的事情，但是同样的代码你要到处重复编写。

如果一门语言缺少某种进行抽象的方法，那么就意味着用它写代码的时候会出现很多相似的文件、重复的代码。这两个现象都是语言存在根本缺陷的信号。宏的威力在于，它让程序员可以在语言之上构建一层全新的抽象。宏是消灭模板文件，将语言打磨得符合你需要的终极武器。

宏到底是什么？

宏让我们可以控制 Clojure 的编译器。在它的作用域内，它可以被用来对语言的语法进行微调或者彻底改变语言的语法。我们通常把 Java 称为“没有枪、没有刀以及没有棍棒的 C++”，^{注1} 把 Ruby 和 Python 这样的语言称为“兵工厂”，Clojure 的宏使你可以构建任

注1：这个引用通常被认为是 Java 语言的作者 James Gosling 说的。

230 何武器，并且你自己构建的武器跟语言内置的武器没有任何区别。

要理解宏，首先要理解“运行期”和“编译期”的区别。

就像我们在 12 页“Clojure Reader”一节看到的那样，Clojure 的源代码被 Clojure reader 读入，reader 会从文本形式的 Clojure 代码产生 Clojure 的数据结构。比如从这个字符串 “(foo [bar] :baz 123)”，reader 会求值出一个列表，这个列表包含一个符号，一个包含符号的 vector，一个关键字以及一个数字。一门语言的代码可以用语言自身的数据结构来描述，这一特性称为“同像性”，正是这个特性使得宏成为可能。^{注2}

通常情况下，这些数据结构会被求值。每种数据有各自的求值规则：

- 很多字面量求值成它们自身（比如数字、字符串、关键字以及 vector）。
- 符号求值成某个命名空间内的一个 var 的值。
- 列表求值成调用，可以是函数调用、特殊形式调用或者宏调用。

编译发生在读入和求值之间，而且对于宏的求值跟对函数的求值是不一样的。函数调用会被直接转换成字节码，在运行时传给函数的参数会被求值成对应的值然后传给函数，而宏是被编译器调用的，调用的参数则是把传入的数据结构不做求值直接传给宏，而宏要返回一个数据结构，这个返回的数据结构本身必须是要能求值的。比如如果 foo 是一个函数，那么

(foo a b)

会被求值成运行时对 foo 函数的调用，调用的参数则是名为 a 和 b 的两个符号的值，而如果 bar 是一个宏的话，那么

(bar a b)

bar 会被 Clojure 的编译器调用，调用的参数是两个符号 a 和 b——而不是这两个符号的值。^{注3} bar 可以选择实现成跟函数一样的普通语义，或者也可以实现自己完全不同的语义，而它可以使用 Clojure 中所有的函数工具：宏并不限制使用语言的某一部分，它可以用语言提供的任何工具。在任何情况下，bar 都必须返回一个 Clojure 编译器可以求值的 Clojure 数据结构，求值出来的数据结构会代替宏原来的位置。这个过程是递归的，因为宏返回的数据结构中可能还包含了对于宏的调用，那么要继续对这个宏求值，这个求值的过程要一直持续到返回的数据结构中没有宏为止。

注 2： 在第一章的“同像性”一节详细讨论了同像性。

注 3： Clojure 编译器知道一个调用到底是调用函数，还是调用一个宏，因为宏在实现细节上其实也是函数，只是它有一些特别的元数据表明这是一个宏。你可以自己查看一下一个宏的元数据，比如 (meta #'or)。

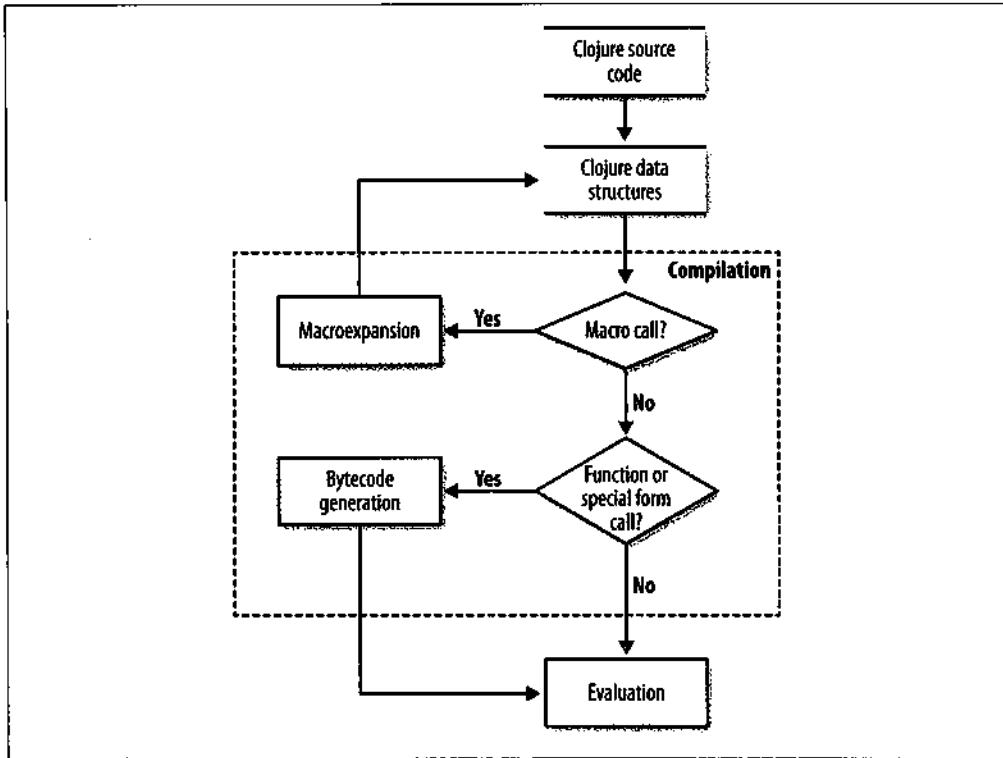


图 5-1: Clojure 的编译模型

作为抽象工具的宏，产生的代码通常比宏本身的代码要多。所以把宏求值成最终代码的过程被称为“宏展开”。就像我们最早在第 3 页“Clojure REPL”中提到的那样，所有的 Clojure 代码都会被编译，即使是在 REPL 里面，而宏展开是编译过程重要且不可分割的一部分。



编译的过程保证所有的宏调用在程序运行时之前被完全替换成它们的扩展形式，所以宏只在编译期被求值。

宏不是什么

写代码来对已有代码进行操作并不是 Clojure 所独有的，也不是 Lisp 语言所独有的。但是并不是所有语言的代码生成机制都是一样的。

比如，C 有预处理器，它会在编译期把一段源代码以文本替换的形式转换成另外一段源代码。这种文本宏系统是不能跟 Lisp 风格的宏相提并论的，因为文本宏系统对代码的操作是依赖字符串的操作，而 Lisp 则是把代码当做有结构的代码来处理的。这种系统的弱

<232>

点在类似 Ruby 的 eval 里面非常明显，我们将在 234 页“宏 vs. Ruby 的 eval”一节将它与 Clojure 的宏进行对比。

类似的，那些代码生成（Code Generation）机制跟宏相比也不是一个概念。代码生成通常是以一个高级别的表示，比如一个正式的语法或者是一个对象模型的描述作为输入，产生一段实现这个对象模型的一段代码。虽然这些对象生成系统还是很有用的，但是它们通常都需要编译器进行一个特殊的步骤来进行编译，而 Clojure 中的宏的编译跟普通 Clojure 代码是同一个步骤；对象生成系统所依赖的通常是一些专门的对象模型，而 Clojure 宏所使用的则是 Clojure 里面的普通数据结构；对象生成系统的代码通常不具有可组合性，而 Clojure 的宏则可以调用另外的宏。

最后，有很多语言提供编译器的 API，使得你可以对这种语言的代码进行修改。比如 Java 的注解（Annotation）处理器、Groovy 的 AST 生成器、模板 Haskell 以及 Scala 的编译器插件。这些都是非常有用的系统，它们使得你可以构造语法抽象并且进行优化——跟 Clojure 的宏是类似的。但是还是有很多不同之处，最大的不同就是这些系统根据它们自己的定义来说会暴露一套内部 API，而且这些系统使用的对象模型在用这种语言编写代码的时候是不会用到的。而 Clojure 作为一种“同像”的语言使我们编写宏的时候所面对的接口和数据模型就是普通的 Clojure 接口和数据结构。

有什么是宏能做而函数不能做的

一开始很难理解宏的威力，举一个简单的例子也许就清楚了。

Java 首次发布是在 1996 年，8 年之后，在 Java 5 里面，一个改进型的 for 循环被添加到语言里面——一个长久以来大家都很期盼的特性。这使得我们可以把诸如这种冗长的代码：

```
for (int i = 0; i < collection.size(); i++) {
    SomeType var = (SomeType)collection.get(i);
    ...
}
```

替换成比较简洁的形式：

```
for (SomeType var : collection) {
    ...
}
```

这种改变对于一门语言来说是非常重要的。改进型的 for 循环降低了代码的复杂性，同时也消除了使用 for 循环产生错误的可能性：使用改进型的 for，你不可能发生数组越界的问题，也不可能写错代码从下标 1 而不是 0 开始遍历。

作为程序员，对于一段使用改进型的 `for` 编写的代码，可以手动转换成等价的旧的 `for` 语法。在循环里面使用的变量名字以及要遍历的集合是唯一重要的事情。其余的，比如对下标进行跟踪、对边界进行检查等都是多余的。

<233>

那么为什么一个改进型的 `for` 循环在 8 年之后才添加到 Java 语言里面呢？根本原因是 Java 缺乏表达力，没有办法让你去写这种改进型的 `for`。它不能写成一个方法调用，但是方法调用是 Java 里面你唯一能利用的武器。方法没办法在方法本身之外设置本地变量，方法也没办法对参数进行选择性地求值——所有的参数都会被求值，而求值得到的值会被作为参数再传给方法。你可以写一个方法来完成类似改进型的 `for` 的功能，但是它看起来肯定跟 Java 里面提供的改进型 `for` 不一样。

添加改进型的 `for` 对 Java 来说需要在编译器层面进行一些修改，普通 Java 程序员不知道怎么改，也没能力去改。那么在没有改进型的 `for` 的这 8 年里，Java 程序员是怎么过的呢？他们只能忍着。

而形成鲜明对比的是，任何 Clojure 程序员能够在几分钟之内利用一些普通的 Clojure 代码写出一个宏来实现类似 Java 5 的改进型的 `for` 的功能：^{注4}

```
(defmacro foreach [[sym coll] & body]
  `(loop [coll# `coll]
    (when-let [[`sym & xs#] (seq coll#)]
      `@body
      (recur xs#)))
  := #'user/foreach
  (foreach [x [1 2 3]]
    (println x))
  ; 1
  ; 2
  ; 3
```

函数是一个进行抽象的优秀工具，但是某些事情它们确实做不了，因为它们只是运行期被调用，而且它们也访问不了编译器；在这种情况下，单靠函数的话，你没办法把一段没有求值的代码（`println` 调用）插入一个循环结构。简单来说，Clojure 程序员可以给语言添加新的语言结构。

内置操作符与宏。跟大多数语言相比，Clojure 的内置操作符——被称为特殊形式（见 23 页的“特殊形式”一节）的数量是非常少的。我们在第 1 章里面介绍过，一共只有 16 个。

你可能会想怎么有很多东西没有啊？比如进行数据结构遍历的 `while`、`for` 和 `doseq` 在哪里？进行数据定义的 `defn`、`defmacro` 和 `defrecord` 呢？进行条件判断的 `when`、`cond` 和

<234>

注 4：这里的 `foreach` 宏只是为了举例子方便；Clojure 本身已经提供了 Java 里面改进型 `for` 的对应操作符，包括命令式的 `doseq` 和函数式的 `for`。

`condp` 呢？所有的这些对于 Clojure 程序员来说都是非常基础的工具。

可能会让你非常惊讶的是，所有这些都是宏。如果它们不存在的话，你也可以利用普通的 Clojure 代码自己写出来，而不需要去求语言作者去实现。这个是跟很多语言相比的最大区别，在那些语言里面，你要写出一个自己的循环或者条件判断的结构是不可能的。当然这个问题也可以延伸到一些非常应用相关或者领域相关的需求，这些需求可能只是对于一个语法的微调，可以通过调整语言来适应你的特殊需求。所以宏消除了语言中“内置”的操作符与用户自定义的操作符之间的界限；因为有了宏，用户自定义的操作符跟内置的操作符看起来没有任何区别。

宏 vs.Ruby 的 eval

初看起来，宏可能跟 Ruby 里面的 eval 没有什么区别。^{注5} Ruby 里面的 eval 是一个内置的函数，它在运行时执行一些代码。Ruby 里面还有 `class_eval` 和 `instance_eval`，同样的，它们也是在运行时执行代码，只是在不同的上下文里面。

在 Ruby 里面，可以这样写：

```
x = 123
code = "puts VAR"
code.gsub!(/VAR/, 'x')
eval code
```

这段代码会打印 123。我们以字符串的形式创建了一段代码，对这段代码进行字符串操作，然后再执行它。

宏与 eval 之间一个最显著的不同是，eval 是在运行期执行代码。这意味着，比如，一些在编译期能够发现的问题对于 eval 代码是不可能的。比如：

```
code = <<END
def foo
    puts "foo!"      # oops, forgot a closing quote
end
END

if(rand(2) == 0)
    eval code
end
```

有一半的时间，这段代码会正确地编译并且正确运行，因为有一半的时间里这个字符串形式的代码根本就没有被求值。而另外一半时间里面，我们会在运行时碰到一个错误。

而作为对比的是，Clojure 的宏是在编译期被调用的。类似的错误如果发生在 Clojure 的

注5： 或者 Python、JavaScript、PHP、Perl 以及任何一种允许通过 eval 一个字符串来执行代码的语言。

宏里面，会被马上发现，因为宏只是普通 Clojure 代码，所以如果写出了一些不符合语法的代码是不会通过编译的。比如下面的代码是永远不可能通过编译的，更别提会被执行了：

```
(defmacro foo []
  `(if (= 0 (rand-int 2))
     (println "foo!))"      ;;= oops, forgot a closing quote
  ;;= #<Exception java.lang.Exception: EOF while reading string>
```

当你准备对代码进行操作的时候，会发现宏跟 eval 的另外一个显著区别。对于 Ruby 的 eval 来说，代码是一个字符串，这个字符串是没有结构的，要对它进行操作，对它进行操作的工具只有：正则表达式以及有关字符串的所有函数。

而作为对比，Clojure 的宏并不对无结构的字符串进行操作。因为 Clojure 是“同像”的，宏直接对 Clojure 的数据结构如列表、vector、符号等进行操作。

对字符串形式的代码进行操作是非常脆弱并且非常容易出错的，而这种错误直到代码在运行时被执行才会发现，使得这个过程非常危险。所以在 Ruby 的世界里，大家是反对使用 eval 的——这个建议也是对的。

所以，下面的代码在 Ruby 里面会被认为是很差的代码：

```
>> def print_sym(x)
>>   code = "p(" + x + ".to_sym)"
>> end
nil
>> eval print_sym "\"foo\""
:foo
nil
```

而 Clojure 里面的相应代码是没有问题的：

```
(defmacro print-keyword [x]
  `(println (keyword `x)))
 ;;= #'user/print-keyword
(print-keyword "foo")
; :foo
;= nil
```

编写你的第一个宏

让我们先写一个宏，这个宏做的事情是一个函数永远都做不到的事情。比如想捉弄一下同事，把 Clojure 代码里面所有的符号倒过来写。我们的目标是写这样的代码：

```
(reverse-it (nltnirp "foo"))
```

而 Clojure 会把它求值成：

```
(println "foo")
```

236 > 很明显的是，如果想用 Java 来实现这个功能，除非你对 Java 的解析器或者编译器进行修改，否则是实现不了的。但是在 Clojure 里面，因为有宏，能很容易地实现。

`reverse-it` 宏以 Clojure 数据结构的形式接受 Clojure 代码作为参数。我们需要做的是获取所有的符号，得到它们的字符串“名字”，把名字倒过来，再把这个倒过来的名字放回去，然后返回。

`clojure.walk` 命名空间里面提供了一个非常方便的函数：`postwalk`，它使我们递归地遍历一个嵌套的列表并且对于列表里面的某个元素做一些处理，这正好符合我们的需求：

示例 5-1：`reverse-it`，一个翻转符号名字的宏

```
(require '(clojure [string :as str]
                  [walk :as walk]))  
  
(defmacro reverse-it
  [form]                                     ①
  (walk/postwalk #(if (symbol? %)           ②
                  (symbol (str/reverse (name %))) ③
                  %)
    form))
```

① 宏接受一个名为 `form` 的参数。

② 它利用 `postwalk` 函数来递归地对 `form` 的每个元素执行给定的匿名函数。

③ 那个匿名函数把 `form` 里面的所有的符号替换成相反的名字，而对于非符号的元素则不进行处理。

现在我们就可以写出一些很奇怪的代码了：^{注 6}

```
(reverse-it
  (quesod [gra (egnar 5)]
         (nltnirp (cni gra)))
  ; 1
  ; 2
  ; 3
  ; 4
  ; 5
  := nil)
```

注 6： 虽然 `reverse-it` 提供了一个很醒目的展示宏的能力的例子，但是在实际生活当中还是不要这么写代码。

如果展开这个宏，会看到熟悉的代码：

```
(macroexpand-1 '(reverse-it
  (qesod [gra (egnar 5)]
         (nltnirp (cni gra))))
  := (doseq [arg (range 5)]
  :=  (println (inc arg))))
```

使用 macroexpand-1 函数来查看宏会产生怎样的代码。`macroexpand` 系列的函数对于测试和调试宏来说是很关键的工具，接下来会详细介绍。 <237>

调试宏

在没有工具的帮助下，宏是很难调试的。虽然 Clojure 编译器已经在编译期为我们提示很多错误，但是在使用宏的威力的时候还是要加倍小心。

想想如果引用一个没有定义的 var 会怎么样？如果是在函数里面的话，这会触发一个编译期的错误：

```
(defn oops [arg] (frobnciate arg))
;= #<CompilerException java.lang.Exception:
;=  Unable to resolve symbol: frobnciate in this context (NO_SOURCE_FILE:1)>
```

确实会报错，而如果定义一个类似的宏，那么编译器不会有任何出错提示：

```
(defmacro oops [arg] `(frobnciate ~arg))
;= #'user/oops
```

而当使用这个宏的时候会触发一个错误：

```
(oops 123)
;= #<CompilerException java.lang.IllegalStateException:
;=  Var user/frobnciate is unbound. (NO_SOURCE_FILE:0)>
```

怎么了？要记住，宏是在编译期执行的。而在编译期，Clojure 根本不知道，也不可能知道符号 `frobnciate` 到底有没有被定义。对于宏来说，它所看到的就是列表、符号以及其他数据结构，它返回的也是列表、符号以及其他数据结构，而这些数据结构里面的符号到底正确不正确不是宏所能确定的。这会使宏非常微妙，还好我们有一些工具可以帮助调试宏。

宏扩展

调试宏最基本的工具是 `macroexpand-1`。这个函数以一个数据结构（在调试的时候，通常就是一个被引号引住的宏形式）作为参数，记住，只是一个数据结构。这里能看到我

们以一个数字作为参数来调用 oops 宏会返回一个两个元素的列表，第一个元素是一个命名空间内的 frobnicate 符号（指向一个不存在的 var），第二个元素是一个数字：

```
(macroexpand-1 '(oops 123))
;= (user/frobnicate 123)
```

macroexpand-1 只会扩展宏一次。记住，如果一个宏被扩展之后还包含了对宏的调用的话，那么宏需要扩展多次；如果宏扩展之后产生的是对另外一个宏的调用，而你想继续扩展这个宏直到最顶级的形式不再是一个宏的话，那么使用 macroexpand。

由于 Clojure 里面很多核心工具（函数、宏）本身就是宏，因此使用 macroexpand-1 通常都能避免扩展出来的代码太冗长以至于没办法看出宏的逻辑。回忆在示例 5-1 介绍的 reverse-it，可以清楚地看出 macroexpand-1 和 macroexpand 之间的区别：

```
(macroexpand-1 '(reverse-it
                  (quesod [gra (egnar 5)]
                          (nltnirp (cni gra)))))

;= (doseq [arg (range 5)]
;=      (println (inc arg)))

(pprint (macroexpand '(reverse-it
                      (quesod [gra (egnar 5)]
                              (nltnirp (cni gra)))))

; (loop*
; ;  [seq_1647
; ;   (clojure.core/seq (range 5))
; ;   chunk_1648
; ;   nil
; ;   count_1649
; ;   (clojure.core/int 0)
; ;   i_1650
; ;   (clojure.core/int 0)])
; ; (if
; ;   (clojure.core/< i_1650 count_1649)
; ;   (clojure.core/let
; ;     [arg (.nth chunk_1648 i_1650)]
; ;     (do (println (inc arg)))
; ;       (recur
; ;         seq_1647
; ;         chunk_1648
; ;         count_1649
; ;         (clojure.core/unchecked-inc i_1650)))
; ;   (clojure.core/when-let
; ;     [seq_1647 (clojure.core/seq seq_1647)]
; ;     (if
; ;       (clojure.core/chunked-seq? seq_1647)
; ;       (clojure.core/let
```

```
; [c__3798__auto__(clojure.core/chunk-first seq_1647)]
; (recur
;   (clojure.core/chunk-rest seq_1647)
;   c__3798__auto__(clojure.core/int (clojure.core/count c__3798__auto__))
;   (clojure.core/int 0)))
; (clojure.core/let
;   [arg (clojure.core/first seq_1647)]
;   (do (println (inc arg)))
; (recur
;   (clojure.core/next seq_1647)
;   nil
;   (clojure.core/int 0)
;   (clojure.core/int 0))))))
;
```

reverse-it 返回一个 doseq 形式，doseq 本身又是一个基于 loop 特殊形式的宏，它扩展开来是非常长的。在大多数情况下，你会发现只需要看到宏的一级扩展就够了，从而也使得宏 macroexpand-1 成为最常用的宏扩展函数。◀239

完全扩展宏。无论 macroexpand 还是 macroexpand-1 都不会对嵌套的宏进行扩展。比如这里我们试着扩展一个 cond 形式，这会产生一个 if 调用，而这个调用的“else”分支还是由一个 cond 形式组成的，而这个“else”分支是没有扩展的：

```
(macroexpand '(cond a b c d))
;= (if a b (clojure.core/cond c d))
```

能够获得一个宏的彻底扩展有时候是非常有用的。在很多情况下，可以使用 clojure.walk/macroexpand-all 函数来达到这个效果：

```
(require '[clojure.walk :as w])
(w/macroexpand-all '(cond a b c d))
;= (if a b (if c d nil))
```

macroexpand-all 很有用，但是从根本上来说，它只是对于 Clojure 编译器完全宏扩展的一个模拟。比如它对于特殊形式的处理并不完全正确：

```
(w/macroexpand-all ''(when x a))
;= (quote (if x (do a)))
```

这个表达式应该扩展为 (quote (when x a))——宏扩展不应该去扩展 quote 这个特殊形式后面的代码。类似的，macroexpand-all 并不支持隐式参数 &env 和 &form，我们会在 251 页“隐式参数：&env 和 &form”一节详细介绍，同时也会介绍一个改进版的 macroexpand——可以支持隐式参数的版本。

语法

因为宏要返回 Clojure 数据结构，我们经常返回列表以表示进一步调用，这个调用可能是对函数、特殊形式或者宏的调用。所以需要一些语法工具来构建这些列表，完全可以使用熟悉的最简单的语法工具 `list` 函数：

```
(defmacro hello
  [name]
  (list 'println name))

(macroexpand '(hello "Brian"))
;= (println "Brian")
```

但是，对于更复杂的、不只是简单返回一个列表的宏来说，使用 `list` 就显得很笨拙了。下面是我们用 `list` 函数对标准库宏 `while` 进行的重写：

240 ➤

```
(defmacro while
  [test & body]
  (list 'loop []
    (concat (list 'when test) body)
    '(recur)))
```

这个宏实现的问题在于它已经淹没在对 `list` 和 `concat` 的调用之中了。

因此，Clojure 提供了一些语法糖来处理列表，把具名的值插入到列表。使用这些语法糖，`while` 宏⁷ 的实现会变得更具可读性：

```
(defmacro while
  [test & body]
  `(loop []
    (when ~test
      ~@body
      (recur))))
```

这个宏使用了三个语法工具：语法引述（syntax quote），⁸ 反引述（unquote）以及编接反引述（splicing-unquote）。这些语法不是专门为宏设计的。你也可以在函数或者其他任何代码中使用它们，但是因为宏里面会做很多列表处理，因此会发现大多数时候会在宏里面使用这三个语法工具。⁹

引述和语法引述

你应该对 `quote` 特殊形式很熟悉了，它会返回参数的不求值的形式。作为一个快捷方法，

注 7： 这里展示的跟 Clojure 标准库里面实现的一样。

注 8： 在其他 Lisp 里面也有类似的工具，称为“背引述”或者“准引述”。Clojure 的作者 Rich Hickey 选择使用语法引述以示区别。

注 9： 比如在 259 页“深入 -> 和 ->>”一节介绍了如何在普通函数里面使用语法引述和反引述。

我们可以把 (quote (a b)) 写成 '(a b)，这两种形式都会被求值成一个包含两个元素的列表，这两个元素 a 和 b 则会保持不求值的形式。

Clojure 包含另外一种形式的引述——语法引述，看起来跟引述很像，除了它使用的是反引号 (`)。

引述和语法引述之间有两个不同点。首先，语法引述把无命名空间限定的符号求值成当前命名空间的符号：

```
(def foo 123)
:= #'user/foo
[foo (quote foo) `foo `foo]
:= [123 foo foo user/foo]
```

这个被语法引述的 `foo (上面 vector 里面的最后一个 “foo”) 被求值成命名空间限定的符号：user/foo，因为它是在 user 命名空间被求值的。在另外一个命名空间里面，`foo 会被求值成那个命名空间的符号。

```
(in-ns 'bar)

`foo
:= bar/foo
```

241

但是如果这个符号本身就是命名空间限定的，那么语法引述也会把这个符号求值成对应命名空间里的符号。

```
(ns baz (:require [user :as u]))

`map
:= clojure.core/map
`u/foo
:= user/foo
`foo
:= baz/foo
```

符号的默认空间化对于产生正确的代码非常关键，它使我们不会因为疏忽而重定义一个已经定义过的值。这被称为宏卫生，我们会在 244 页“宏卫生”一节详细讨论。

普通引述与语法引述的第二个区别是语法引述允许反引述：一个列表里面的某些元素可以被选择性地反引述，从而使得它们在语法引述的形式内被求值。

反引述与编接反引述

在编写一个宏的骨架的时候，我们通常想保持列表里面的一些元素不进行求值，而另外一些元素则需要求值。如果只有最基本的方法可以用的话，那么将是很痛苦的：list 函数，

对于在那些不需要进行求值的元素上面加语法引述修饰，而那些需要求值的则不管。

```
(list `map `println [foo])
;= (closure.core/map closure.core/println [123])
```

一个更简短、更有可读性的办法是把整个列表进行语法引述，然后把其中那些需要求值的元素进行反引述。可以通过~来进行反引述。

```
`(map println `~[foo])
;= (closure.core/map closure.core/println [123])

`(~(map println `~[foo]))
;= (closure.core/~(map closure.core/println [123]))
```

注意，这个和显式使用 `list` 的版本相比较，使用语法引述的形式跟编写普通 Clojure 代码看起来没什么区别，只是在某些地方加上了` 和 ~，从而使得宏的代码具有高度的可写性与可读性。

注意一下第二个例子：反引述一个列表或者 `vector` 会把整个形式都反引述。这个特性可以用来在一个语法引述的形式里面对一个方法进行求值：

```
`(println `~(keyword (str foo)))
;= (closure.core/println :123)
```

242 另一个非常常见的场景是：有一个列表的形式，然后想把另外一个列表的内容解开加入到第一个列表里面去。同样道理，做这件事情最直接的办法是使用 `list` 和 `concat`，但是这么做同样也会降低代码的可写性与可读性：

```
(let [defs `((def x 123)
             (def y 456))]
  (concat (list 'do) defs))
;= (do (def x 123) (def y 456))
```

编接反引述操作符 '@ 是一个更好的办法，它自动帮你做列表的链接。

```
(let [defs `((def x 123)
             (def y 456))]
  `(~(do @defs))
;= (do (def x 123) (def y 456))
```

这里，`defs` 列表里的元素被分割到语法引述所在的列表，这在编写宏时是非常常见的模式。比如一个接受多个形式作为“代码体”的宏大概是这样的：

```
(defmacro foo
  [& body]
  `(~(do-something @body)))
```

```
(macroexpand-1 '(foo (doseq [x (range 5)]
                               (println x))
                           :done))
;= (user/do-something
;=   (doseq [x (range 5)]
;=     (println x))
;=   :done)
```

剩余参数 `body` 里面是传给宏的所有形式的列表，然后 `~@body` 把这些形式分割到宏所在的上下文里面。

语法引述结合反引述以及编接反引述允许我们把从宏返回的 Clojure 数据结构（也即 Clojure 代码）当做模板：可以通过语法引述来构建整个宏的结构，然后根据需要对这个宏返回的数据结构里的某些元素进行求值——通过反引述语法。

 语法引述、反引述以及编接反引述并没有什么特别神奇的地方：它们也是定义在 Clojure 标准库里的函数，然后在 Clojure reader 里面暴露出一个易于使用的语法：```、``~` 和 ``@`。通过对它们进行引述来看看它们到底是怎么实现的。这里看一下语法引述的实现：

```
`(map println ~[foo])
;= (clojure.core/seq
;=   (clojure.core(concat
;=     (clojure.core/list (quote clojure.core/map))
;=     (clojure.core/list (quote clojure.core/println)))
;=   (clojure.core/list [foo])))
```

什么时候使用宏

<243>

宏显然是很有用的工具，但是有着这么强大的能力也必然伴随着使用上的一些责任以及一些注意事项。

宏是在编译期被调用的。这意味着它们跟函数不一样，不是 Clojure 程序里面的头等公民。宏没办法访问运行时的信息，比如一个变量在运行时的值。在宏的眼里，它看到的就是一些从 Clojure 源代码读出的未求值的数据结构。

考虑一下怎么实现返回一个“欢迎语”的简单功能。可以把它实现成函数，也可以实现成宏：

```
(defn fn-hello [x]
  (str "Hello, " x "!"))

(defmacro macro-hello [x]
  `(str "Hello, " ~x "!"))
```

它们在一些用法下的表现是类似的：

```
(fn-hello "Brian")
:= "Hello, Brian!"
(macro-hello "Brian")
:= "Hello, Brian!"
```

而在另外一些上下文中表现就不一样了：

```
(map fn-hello ["Brian" "Not Brian"])
:= ("Hello, Brian!" "Hello, Not Brian!")
(map macro-hello ["Brian" "Not Brian"])
:= #<CompilerException java.lang.RuntimeException:
    := Can't take value of a macro: #'user/macro-hello,compiling:(NO_SOURCE_PATH:1)>
```

后一个对宏 `macro-hello` 的调用出错了，因为尝试去访问宏在运行时的值——宏根本就没有运行时值的概念，^{注10} 宏不能作为值来进行组合或者传递，因此要把宏作为一个值来传递给 `map` 函数进行计算是完全没有意义的。

要想在这种上下文里面使用宏，需要把宏包在一个 `fn` 或者匿名函数字面量里面。这使得宏的应用又返回到编译期——也就是在编译包外面的函数的时候。不过这个办法的问题是：如果直接传递一个函数就可以搞定的话，为什么还要用宏呢？

```
(map #(macro-hello %) ["Brian" "Not Brian"])
:= ("Hello, Brian!" "Hello, Not Brian!")
```

另一个避免这种难看的、用 `fn` 或者 `#(...)` 来包裹宏的办法是把整个 `map` 包裹在一个宏里面。但是这是个无底洞——你编写的宏越多，那么你就需要写更多的宏来解决宏所带来的问题——宏无法访问运行时信息所带来的问题，因此对于那些需要传递高阶函数的习惯用法，宏是不适合的。

244 >

这里可以学到两个东西，一个小的，一个稍大的。首先宏在某些上下文（编译期）中是很方便、很强大的，但是在另外一些场景下（运行期）也会使得你的代码很难写；可以考虑把主要逻辑从宏里面抽到函数里面，从而使得宏只是简单地做一些组织工作，真正的逻辑都通过调用函数来做。

第二个更重要的点是，应该只在需要自己的语言组件时才使用宏，换种说法就是，只在函数满足不了需要的时候才去使用宏。宏的使用场景主要有：

- 需要特殊的求值语义。
- 需要自定义的语法——特别是一些领域特定的表示法。
- 需要在编译期提前计算一些中间值。

注 10： 其实这种说法也是不完全对的，我们会在 253 页“测试和调试 &env 的用途”详细介绍。但是你也会看到，在运行时使用宏是非常麻烦的，通常这么做是不值得的。

另外一方面，应该始终问问自己：如果使用函数，能不能达到目的？我们需要特定的求值规则吗？

宏卫生

传统上来说，编写宏的最大的潜在问题是宏产生的代码可能会跟外部代码发生不正确的交互。虽然 Clojure 里面已经有一些其他 Lisp 方言所没有的防治措施，但是仍然可能会出错。

宏产生的代码通常会被嵌在外部代码中使用，而且我们通常也会把一段用户代码作为参数传入宏。在任何一种情况下，有一些符号已经被用户代码绑定到一个值了。那么就会有这样一种可能：在宏里面使用的某个符号名跟外部代码或者传入的用户自定义代码里面的某个符号名字可能会发生冲突，这样的问题是很难定位的。那些避免了这类问题的宏称为“卫生宏”。

考虑一个需要通过 let 来绑定一个本地绑定的宏，在宏里面到底用什么名字来表示这个本地绑定对宏的用户来说应该是透明的，但是我们还是要选择某个名字。比较幼稚的想法是使用一个可能不是那么常用的名字，比如 x：

```
(defmacro unhygienic
  [& body]
  `(let [x :oops]
    `@body))
 ;;= #'user/unhygienic
(unhygienic (println "x:" x))
 ;;= #<CompilerException java.lang.RuntimeException:
 ;;= Can't let qualified name: user/x,compiling:(NO_SOURCE_PATH:l)>
```

245

非常幸运的是，Clojure 很聪明地提示这个代码有问题。就如同在 240 页“引述和语法引述”一节所提到的那样，所有的符号都会被求值成语法引述里面的命名空间限定的符号。可以通过查看宏扩展来证实这一点：

```
(macroexpand-1 `(unhygienic (println "x:" x)))
 ;;= (clojure.core/let [user/x :oops]
 ;;=   (clojure.core/println "x:" user/x))
```

所有对 x 的引用都被扩展成 user/x，但是 let 需要的名字是没有命名空间限定的，所以这个宏始终会抛出编译期的错误。可以通过一些看似“聪明”的引述、反引述来避免这个错误：

```
(defmacro still-unhygienic
  [& body]
  `(let [~'x :oops]         ❶
```

```
  `@body))  
:= #'user/still-unhygienic  
(still-unhygienic (println "x:" x))  
; x: :oops  
:= nil  
(macroexpand-1 '(still-unhygienic  
                  (println "x:" x)))  
:= (clojure.core/let [x :oops]  
:=   (println "x:" x))
```

❶ ``x是使用反引述(`)来强制使用没有命名空间限定的符号x作为let里面绑定的名字。

这个虽然是可以运行的，但是给代码中引入了一个重大的bug。这里的问题是，这里使用的名字x，我们用来在本地范围内定义一个本地绑定，它可能会跟宏外部的符号以及传入宏的代码里的符号名字相冲突。看一个简单的例子：

```
{let [x :this-is-important]  
  (still-unhygienic  
    (println "x:" x)))  
; x: :oops
```

这里已经把x绑定到一个本地的值了，但是由宏产生的let会悄悄地把x绑定到另外一个值。宏的用户除非去看宏的源代码，否则永远也不知道到底发生了什么。

还好Clojure提供了一个轻松简单的机制使我们可以避免这种名字冲突问题。

246 > Gensym 来拯救

当要在宏里面建立一个本地绑定的时候，我们希望可以动态产生一个永远不会跟外部代码或者用户传入宏的代码冲突的名字。幸运的是，Clojure提供了一个简单的语法来解决这个问题：gensym。gensym函数返回一个保证唯一的符号。每次调用它都能返回一个新的符号。

```
(gensym)  
:= G_2386  
(gensym)  
:= G_2391
```

gensym也可以接受一个参数，这个参数作为产生的符号的前缀：

```
(gensym "sym")  
:= sym2396  
(gensym "sym")  
:= sym2402
```

虽然可以在任何想用的地方使用gensym来产生一个唯一的符号，但是它的主要使用场景还是用来编写卫生宏：

```
(defmacro hygienic
  [& body]
  (let [sym (gensym)]          ❶
    `(let [~sym :macro-value]     ❷
       `@body))
  := #'user/hygienic
  (let [x :important-value]
    (hygienic (println "x:" x)))
  ; x: :important-value
  := nil
```

❶ 这里我们不再手动指定一个符号名字，而是通过 `gensym` 来产生一个唯一的符号……

❷ 并且使用语法反引述把它放到宏要返回的代码里面去。

现在，代码是安全的了。可以随便选用什么 `x` 之类的名字来作为本地绑定的符号名字，都不会与 `gensym` 产生的名字相冲突。

因为 `gensym` 在宏里面是如此常用，Clojure 专门为它设计了一个简短的表示方式。在语法引述形式里面任何以`#`结尾的符号都会被自动扩展，并且对于前缀相同的符号，它们会被扩展成同一个符号的名字。这个被称为“自动 `gensym`”，下面这段代码跟上面的代码是等价的：

```
(defmacro hygienic
  [& body]
  `(let [x# :macro-value]
    `@body))
```

语法引述里面的 `x#` 会被转换成类似 `x__3507__auto__` 的名字。 247

而在同一个语法引述的形式里面，对于同一个前缀的所有自动 `gensym`，它们会被转换成同一个符号：

```
`(x# x#)
:= (x__1447__auto__ x__1447__auto__)
```

这使得我们在一个语法引述形式里面可以对一个 `gensym` 进行多次引用，而且读起来、写起来都是非常自然的。

```
(defmacro auto-gensyms
  [& numbers]
  `(let [x# (rand-int 10)]
    (+ x# `@numbers)))
  := #'user/auto-gensyms
  (auto-gensyms 1 2 3 4 5)          ❶
                                     ❷
```

```
;= 22
(macroexpand-1 '(auto-gensyms 1 2 3 4 5))         ❸
;= (clojure.core/let [x_570__auto__ (clojure.core/rand-int 10)]
;=   (clojure.core/+ x_570__auto__ 1 2 3 4 5))
```

- ❶ 这里使用一个自动 gensym `x#` 来在宏代码里面建立一个本地绑定。
- ❷ 以同样的名字 `x#` 来引用这个符号。
- ❸ 通过宏扩展来看看每个自动 gensym 是不是都被替换成同一个名字，而且不会跟其他名字冲突。

但是需要注意的是，对于自动 gensym，只能保证在同一个语法引述的形式里面所产生的符号的名字是一样的：

```
[`x# `x#]
;= [x_1450__auto__ x_1451__auto__]
```

这就意味着，像下面这种对 `doto` 进行重新实现的代码里面，想要在多个语法引述的形式里面使用同一个自动 gensym：

```
(defmacro our-doto [expr & forms]
  `(let [obj# `expr]          ❶
     `@(map (fn [[f & args]]
              `(~f obj# `@args)) forms)   ❷
     obj#))
```

- ❶ 在第一个语法引述形式里面对 `obj#` 的引用。
- ❷ 在第二个语法引述形式里面对 `obj#` 的引用。这样这段代码会产生两个 gensym，虽然我们的初衷是想使用同一个符号。

下面的代码将会出错：

```
248 ➤ (our-doto "It works"
  (println "I can't believe it"))
;= #<CompilerException java.lang.RuntimeException:
;= Unable to resolve symbol: obj_1456__auto__ in this context,
;= compiling:(NO_SOURCE_PATH:1)>
```

因为对两个 `obj#` 自动 gensym 的引用是在两个不同的语法引述形式里面。在这种情况下，还是需要手动调用 gensym：

```
(defmacro our-doto [expr & forms]
  (let [obj (gensym "obj")]
    `(let [~obj `expr]
       `@(map (fn [[f & args]]
```

```
`(~f `obj `@args)) forms) ❶  
`obj)))
```

- ❶ 在这里，这个语法引述其实是什么意义的，使用 (list* f obj args) 更好。

这样的话就没问题了：

```
(our-doto "It works"  
  (println "I can't believe it")  
  (println "I still can't believe it"))  
; It works I can't believe it  
; It works I still can't believe it  
:= "It works"
```

让宏的用户来选择名字

如果你的宏需要建立一个对用户可见的绑定，那么还是需要给绑定选择一个名字，并且还要去处理宏的“卫生性”的问题。

可以随便选择一个名字，然后写在宏的文档里面。但是这从来不是一个好主意，虽然有些人也确实这么干。如果你的宏产生的代码需要跟 Java 代码进行交互，那么很可能会让你的宏产生一个叫做 `this` 的绑定。^{注11} 一个宏如果故意向外部代码暴露出一个名字，那我们说这个宏是“不稳定的”。^{注12} 这是在编程当中要极力避免的一个现象，因为这使得宏的使用者要始终记住一个他实际上不应该太过关心的名字。

一个更简单也更干净的办法是让用户来选择这个绑定的名字。因为宏并不对传给它的参数进行求值，因此我们可以很简单地传一个符号给宏，然后在宏产生的代码里面使用这个符号：

```
(defmacro with  
  [name & body]  
  `(let [~name 5]  
    ~body)  
:= #'user/with  
(with bar (+ 10 bar))  
:= 15  
(with foo (+ 40 foo))  
:= 45
```

<249

重复求值

使用宏时一个普遍而又隐蔽的问题是重复求值。说它隐蔽是因为在很多情况下是碰不到这个问题的。重复求值发生在当传给宏的参数在宏的扩展形式里面出现多次的情况下。

注11：这是使用 proxy 的一个场景，也是 Clojure 里面“名字始终由用户来选择”的一个特例。另一个特例是定义宏 `defmacro`，在本章后面有关于“`&env` 和 `&form`”的讨论。

注12：在 507 页“去掉样板”一节有不稳定宏的一些例子。

比如看看下面这个 spy 宏：

```
(defmacro spy [x]
  `(do
    (println "spied" 'x `x)
    `x))
```

这个宏打印传入的表达式的值，然后返回这个值。只要传入的 x 求值出来的值是一个常量，那么这个宏就是没有问题的。但是如果对 x 的计算非常耗时或者对它的计算有副作用，那么就可能得到一些意想不到的结果：

```
(spy 2)
; spied 2 2
:= 2
(spy (rand-int 10))
; spied (rand-int 10) 9
:= 7
```

在第一种情况下，代码执行的结果是对的，因为 2 是一个常量值。在第二种情况下，(rand-int 10) 在宏的代码里面被求值两次，而且两次的值是不一样的。这个宏扩展之后可以很明显地看到这个问题：

```
(macroexpand-1 '(spy (rand-int 10)))
:= (do (println (rand-int 10))
      := (rand-int 10))
```

如果这里传入的表达式是要发射导弹，而不只是产生一个随机数，那么就会产生严重的问题了。要避免这种问题，可以引入一个本地绑定，把传入的表达式求值的值绑定到这个本地绑定就可以了：^{注13}

```
(defmacro spy [x]
  `(let [x# `x]
    (println "spied" 'x x#)
    x#))

(macroexpand-1 '(spy (rand-int 10)))
:= (let [x_725__auto_ (rand-int 10)]
    := (println x_725__auto_ '(rand-int 10))
    := x_725__auto_)
```

这会保证我们提供的表达式永远只会被求值一次：

```
(spy (rand-int 10))
; spied (rand-int 10) 9
:= 9
```

^{注13}：更精确的说法是，当一个宏的参数在代码的一条执行路径里出现多次。

重复求值的问题，即使通过某些手段绕过去了，也是代码“有问题”的一个信号，这意味着你把有些应该在函数里面实现的逻辑写到宏里面去了。

```
(defn spy-helper [expr value]
  (println expr value)
  value)

(defmacro spy [x]
  `(spy-helper ~x ~x))
```

这样的话，就不需要再引入一个自动 gensym 的本地绑定了。

宏的常见用法和模式

讨论宏的使用模式听起来可能有点讽刺，因为宏本身的使命就是要消除最后残存的“模式”。^{注14} 暂且先不讨论“模板代码”是不是一定应该被禁止，首先让我们把注意力集中在怎么写出“更 Clojure”的宏上面。

如果宏需要指定本地绑定，那么把绑定指定在一个 vector 里面。如果你的宏需要指定一些本地绑定，那么应该把这些绑定的名字和绑定的“初始化表达式”成对指定在一个 vector 里面，而且这个 vector 经常被指定为宏的第一个参数。这使得你的宏的风格跟 Clojure 内置的一些宏、函数的风格一致，比如 let、if-let、for 和 with-open 等：

```
(let [a 42
      b "abc"]
  ...)

(if-let [x (test)]
  then
  else)

(with-open [in (input-stream ...)
            out (output-stream ...)]
  ...)

(for [x (range 10)
      y (range x)]
  [x y])
```

for 是一个有趣的例子，因为它这里的“初始化表达式”提供的值并不是对应绑定的值：
x 并不会被初始化为 (range 10)，在 for 里面它的意思是 x 将作为遍历过程中 (range 10) 的每个元素的引用。所以你应该始终记住（初始化表达式的值不一定是你绑定的值。）

◀251

注 14： 另外的一些模式是被函数干掉的。

定义 var 的时候不要耍小聪明。在 Clojure 标准库里面有很多用来定义 var 的宏，它们都是基于 def 或者由 def 引申出来的宏。如果你也要编写一个这样的宏，那么把下面这些规矩记在心中会使你的宏的行为跟用户的期望吻合。

定义 var 的宏的名字应该以 def 开头。

这会让你的宏的风格和内置的定义 var 的宏的风格一致，比如 :defn、defn-、defmacro 等。这个 def 前缀对于用户来说就是一个提示：这个宏会定义一个 var，所以我应该把它用在顶级形式里面。

把 var 的名字作为第一个参数。

是不是第一个参数暂且不说，如果你的宏定义的 var 使用了一个特定的名字或者自动生成的名字，那么你的宏的用户需要去读你的宏的实现才能知道宏在干什么，你的宏定义的名字是什么，这跟我们编写代码时候强调的“面向抽象”的原则相违背。

每个宏形式只定义一个 var。

作为对于“最差实践”——“定义宏的时候使用特定的名字或者自动生成的名字”的补充，在一个宏形式里面不要定义多个 var。就像如果 def 或者 defn 会定义多个 var 会使你吃惊一样，如果你的宏会定义多个 var 也会使你的宏的用户吃惊。对于这个原则的一个特例是：如果宏生成的代码需要某个单独定义（通常是私有的）的 var，而这个 var 普通用户访问不到，或者根本就不知道，那这么做是没有问题的。

不要在宏里面实现复杂的行为。宏应该只是在函数（或者其他宏）的基础上薄薄地封装一层，我们应该很简单地实现宏里面的逻辑。这个原则其实回到了 243 页“什么时候使用宏”一节的讨论。

对于这个原则有一堆反例，包括著名的 for 宏，因为它的扩展形式非常复杂。但是 for 的实现可以很简单地用 map、filter、mapcat、fn 以及 let 组合起来实现。它的复杂性主要是为了优化性能以及语法：它并没有隐藏任何逻辑或者实现一些不该它实现的代码。

宏应该把大多数逻辑都代理给下层的函数来做，它应该专注于那些函数没办法实现的功能：对求值进行控制。

隐藏参数：&env 和 &form

在本章前面的“让宏的用户来选择名字”一节，我们提到了 defmacro 宏本身是 Clojure 里面几个不稳定的宏之一。defmacro 宏引入了两个隐藏的本地绑定：&env 和 &form。

&env 是一个 map，map 的 key 是当前上下文下所有本地绑定的名字¹⁵（而对应的值是未定义的）。这个对于调试非常有用：

```
(defmacro spy-env []
  (let [ks (keys &env)]
    `(prn (zipmap 'ks ['@ks])))

(let [x 1 y 2]
  (spy-env)
  (+ x y))
; {x 1, y 2}
:= 3
```

&env 的另外一个有用的用途是利用它可以在编译期安全地对表达式进行优化。下面就是一个例子，这个宏在编译期对于那些没有引用本地绑定的表达式提前进行求值，从而使得在运行期不用进行求值了：

```
(defmacro simplify
  [expr]
  (let [locals (set (keys &env))]
    (if (some locals (flatten expr))      ❶
        expr                         ❷
        (do
          (println "Precomputing: " expr)
          (list `quote (eval expr)))))) ❸
```

- ❶ 这里我们检查表达式中有没有引用当前代码上下文中的任何本地绑定。
- ❷ 如果表达式确实引用了本地绑定，那么原封不动地返回这个表达式。
- ❸ 否则，在编译期对这个表达式进行求值（并且打印一条信息）。

这个宏可以保持代码不进行改动，同时还能在编译期就对常量表达式进行优化（这个比在代码层进行手动优化——把优化后的某个“神奇”的数字写在代码里面要好很多）：

```
(defn f
  [a b c]
  (+ a b c (simplify (apply + (range 5e7)))))
; Precomputing: (apply + (range 5e7))
:= #'user/f
(f 1 2 3)           ;; returns instantly
:= 1249999975000006
(defn f'
  [a b c]
```

¹⁵ 注意，不能依赖 &env 这个 map 里面的 key 的元数据，特别是当这些 key 可能有本地别名的时候。

```
253 (simplify (apply + a b c (range 5e7)))  
:= #'user/f'  
(f' 1 2 3) ;; takes ~2.5s to calculate  
:= 1249999975000006
```

因为这里的 `apply` 表达式没有包含对于函数本地绑定的任何引用，因此 `simplify` 宏把它优化成一个常量值。这个优化使得函数 `f` 不需要在运行时花费时间（大约 2.5 秒）来计算 `range` 函数返回的数字的和。而另一方面，函数 `f'` 里面的 `simplify` 确实依赖本地绑定，`simplify` 没有做任何优化，所以这个表达式需要在运行时求值了。

测试和调试 `&env` 的用途。要测试一个使用了 `&env` 的宏是非常困难的，有两个办法。比较土的办法是在宏里面加很多日志，这样在使用这个宏的时候通过日志可以看出宏的执行是否正确（就像上面 `simplify` 函数里面用的 `println`）或者可以根据宏的实现原理来对宏进行测试。

现在的宏是被实现成函数的，这个函数在接受普通的参数之前，Clojure 编译器会传递两个额外的参数 `&form` 和 `&env`，但是如同在 243 页“什么时候使用宏”一节看到的那样，Clojure 为了避免很多潜在的问题，不让我们把宏当做函数使用。

可以通过直接使用实现宏的那个函数（其实就是模仿编译器的行为）来绕过这个限制：

```
(@#'simplify nil {} '(inc 1)) ●  
;  
; Precomputing: (inc 1)  
:= (quote 2)  
(@#'simplify nil {'x nil} '(inc x))  
:= (inc x)
```

❶ 这行代码 `simplify` 前面那么多前缀是解引用的一个语法糖，跟 `(deref (var simplify))` 是等价的，通常被用来获取私有 `var` 的值——使用它会触发一个警告信息。这种对于宏的滥用对于测试很有用，但是它依赖了 Clojure 的当前特定实现。

这里我们以两个额外参数调用宏的实现函数 `simplify`：一个 `nil` 的 `&form`（我们没有使用这个参数值）以及包含一个虚构的本地绑定 `&env` map。这使得我们可以看看在不同的本地绑定的情况下，`simplify` 会分别生成怎样的代码。注意，这里没办法使用 `macroexpand` 来测试 `simplify`，因为它没有提供任何方法来对 `&env` 这个 map 进行模拟，而直接调用宏的实现函数是可以的。^{注 16}

254 > **&form**

`&form` 里面的元素是当前被宏扩展的整个形式，也就是说，它是一个包含了宏的名字（用

注 16： 在 258 页“测试上下文相关的宏”一节我们提供了一个宏扩展函数的另一个实现，使得我们可以不用依赖这种对宏的实现函数进行解引用。

户代码里面引用的宏的名字——可能被重命名了) 以及传给宏的所有参数的一个列表。这个形式也就是 reader 在读入宏的时候所读入的形式。^{注 17} 这意味着, &form 保存了用户指定的所有元数据, 比如类型提示, 以及由 reader 添加的元数据, 比如调用宏的那行代码的行号。

我们来看两个使用 &form 的例子: 在宏里面打印有用的编译期错误信息以及在宏里面保留用户指定的类型提示信息。

在宏里面打印有用的错误信息

&form 的一个关键应用是使宏能够提供精确而有用的信息。比如, 假如要编写这样一个宏, 这个宏接受不定个数参数, ^{注 18} 但是它的每个参数必须都是包含三个元素的集合, 否则抛出异常:

```
(defmacro ontology
  [& triples]
  (every? #(or (= 3 (count %))
              (throw (IllegalArgumentException.
                      "All triples provided as arguments must have 3 elements")))
          triples)
  ;;= build and emit pre-processed ontology here...
)
```

只有当其中有一个参数不是包含三个元素的集合的时候, 这个异常才会抛出来。但是这个抛出来的异常信息并不是很理想:

```
(ontology ["Boston" :capital-of])           ❶
;= #<IllegalArgumentException java.lang.IllegalArgumentException:
;=   All triples provided as arguments must have 3 elements>
(pst)
;= IllegalArgumentException All triples provided as arguments must have 3 elements
;=   user/ontology (NO_SOURCE_FILE:3)           ❷
```

❶ 虽然这里确实抛出了一个异常……

❷ 但是这里显示的行号从用户的角度来说是不对的——这里的 3 是抛异常位置相对宏的源码定义开始的行号, 而不是调用这个宏的代码的行号。

对于这种简单的 REPL 交互的例子来说, 这个不精确的行号可能觉得没有什么, 但是对于那种稍微复杂一点的功能, 代码量稍微大一点, 那么这种不精确的行号就会浪费我们

<255>

注 17: 或者是由前一个宏扩展所返回的。

注 18: Triples 是一个“主体 - 谓词 - 对象”的表达式, 你会在比如语义网技术 RDF 当中看到这样的表达式。它的表达式方式和语义由于各自实现的不同会有所不同, 但是一个最简单的数组 triple 可能是这样的: ["Boston" :capital-of "Massachusetts"]。

很多时间来查找问题。还好我们可以利用 `&form` 来提供更准确的行号信息：

```
(defmacro ontology
  [& triples]
  (every? #(or (= 3 (count %))
              (throw (IllegalArgumentException.
                      (format "`%s` provided to `ontolo" %)
                      (first &form)
                      (-> &form meta :line))))) ❸
    triples)
  ;; ...
)
```

- ❶ 把整个有问题的数组参数都放在异常信息里面，虽然这个跟 `&form` 无关，但是写上更利于调试。
- ❷ `&form` 的第一个元素始终是宏的名字（虽然有可能是被重命名过的）。后面还会看到对这一点的应用。
- ❸ 这里我们从 `reader` 提供的元数据中读出了行号信息。这个行号就是用户代码使用宏的行号。

做了这些改动之后，这个宏对于编译期的错误提示就友好多了：

```
(ontology ["Boston" :capital-of])
:= #<IllegalArgumentException java.lang.IllegalArgumentException:
:=`["Boston" :capital-of]` provided to `ontology` on line 1 has < 3 elements>
```

很好，现在出错信息里面的行号已经比之前清楚很多了。这里为什么要把宏的名字打上呢？因为用户在一个命名空间里面引入这个宏的时候可能利用 `refer` 之类的函数对宏进行了重命名——这是很有用的特性，它 can 以用来避免从别的空间里面引入函数、宏的时候与当前空间里面的函数、宏的名字发生冲突。^{注19} 所以，如果切换到另外一个命名空间的时候，可以把 `ontology` 宏换成另外一个名字：

```
(ns com.clojurebook.macros)
:= nil
(refer 'user :rename '{ontology triples})
:= nil
```

现在，`ontology` 宏在命名空间 `com.clojurebook.macros` 里面的名字变成了 `triples`。可以看到在抛异常的时候，`ontology` 抛出了准确的宏的名字。当前使用引用这个宏的名字：

注 19： `refer` 我们会在“`refer`”一小节（参见 323 页）详细介绍，`use` 内部其实也用了这个函数，我们会在那一章的后面详细介绍。

```
(triples ["Boston" :capital-of])
;= #<IllegalArgumentException java.lang.IllegalArgumentException:
;= `["Boston" :capital-of]` provided to `triples` on line 1 has < 3 elements>
```

如果没有使用 (`first &form`) 来构造这个错误信息的话，那么报出来的错误信息就会有点让人混淆了，用户要始终记住它在当前命名空间把这个宏 `ontology` 重命名为 `triples` 了。

保持用户提供的类型提示

大多数宏把用户在形式上指定的元数据给丢弃掉了，包括类型提示信息。^{注20} 比如，这里的 `or` 这个宏就没有在生成的代码里面把 `^String` 类型提示信息带上，从而导致了一个反射警告：

```
(set! *warn-on-reflection* true)
;= true
(defn first-char-of-either
  [a b]
  (.substring ^String (or a b) 0 1))
; Reflection warning, NO_SOURCE_PATH:2 - call to substring can't be resolved.
;= #'user/first-char-of-either
```



这样的情况在实际代码中其实很少发生，因为通常不会这么指定类型提示信息，我们通常会把类型提示信息加在宏的参数上面，因此下面使用到参数的地方就可以通过类型推断来获取类型信息了：

```
(defn first-char-of-either
  [^String a ^String b]
  (.substring (or a b) 0 1))
;= #'user/first-char-of-either
```

可以通过检查它的元数据信息来看看用户在 `or` 表达式上面指定的类型提示信息：

```
(binding [*print-meta* true]
  (prn '^String (or a b)))
; ^{:tag String, :line 1} (or a b)
```

确实有，但是当检查 `macroexpand` 之后的代码的元数据信息的时候，这个元数据不见了：

```
(binding [*print-meta* true]
  (prn (macroexpand '^String (or a b))))
; {let* [or_3548__auto__ a]
;   (if or_3548__auto__ or_3548__auto__ (clojure.core/or b)))
```

但是要想在 `or` 上面保持用户指定的元数据信息是很简单的，只要在它的定义里面使用上 `&form` 就好了。首先来看看 `or` 在 `clojure.core` 中是怎么实现的：

^{注 20：} 我们在 366 页“为了效率进行类型提示”一节详细介绍“类型提示”。

```
(defmacro or
  ([] nil)
  ([x])
  ([x & next]
   `(let [or# `x]
      (if or# or# (or `@next))))
```

只需要把加在 `&form` 上面的元数据——包含了用户指定的类型提示信息（如果确实指定了的话）——加到宏产生的代码上去。在很多情况下，只需要先从 `&form` 上获取元数据信息，再在宏的代码体的最外层利用 `with-meta` 把元数据加上去。但是这里没办法这么做。^{注21} 这里需要把这个类型提示信息加在一个符号上面：

```
(defmacro OR
  ([] nil)
  ([x]
   `(let [result (with-meta (gensym "res") (meta &form)))
      `(let [~result `x]
         ~result)))
  ([x & next]
   `(let [result (with-meta (gensym "res") (meta &form))]
      `(let [or# `x
            ~result (if or# or# (OR `@next))]
         ~result))))
```

可以通过再次查看宏的扩展来确认元数据是不是真的被保留下来了（注意，这里用的是自定义的 `OR`）：

```
(binding [*print-meta* true]
  (prn (macroexpand '^String (OR a b))))
; (let* [or__1176__auto__ a
;        ^{:tag String, :line 2}
;        res1186 (if or__1176__auto__ or__1176__auto__ (user/or b))
;        ^{:tag String, :line 2} res1186)
```

用户指定的元数据被保留下来了。在这个例子里也就避免了反射调用警告：

```
(defn first-char-of-any
  [a b]
  (.substring ^String (OR a b) 0 1))
:= #'user/first-char-of-any
```

在上面 `OR` 使用的模式可以抽取出来作为一个可重用的函数，从而可以用在任何宏上面：

```
(defn preserve-metadata
  "Ensures that the body containing `expr` will carry the metadata"
```

^{注21}：这个限制是由于一个不幸的实现细节：特殊形式（比如 `let`，例子里面的 `or` 这个宏所在的最外层 `for`）是不能指定类型提示的。所以必须先引入一个本地绑定，再把这个类型提示加到这个本地绑定上去。

```

from `&form`."
[&form expr]
(let [res (with-meta (gensym "res") (meta &form))]
  `(let [^res ^expr]
    ^res))

(defmacro OR
  "Same as `clojure.core/or`, but preserves user-supplied metadata
  (e.g. type hints)."
  ([] nil)
  ([x] (preserve-metadata &form x))
  ([x & next]
   (preserve-metadata &form `(let [or# ^x]
                                (if or# or# (or ^@next)))))))

```

◀ 258

编写一个利用 `preserve-metadata` 来保留用户指定的元数据的版本的 `defmacro` 宏就留给读者作为练习了。

测试上下文相关的宏

我们已经知道使用了 `&form` 和 `&env` 的宏是不好测试的。但是基于对这两个参数的了解，以及宏的实现原理，可以编写出我们自己的 `macroexpand-1`，从而可以轻易地 mock 出 `&env` 以便于测试和调试：

```

(defn macroexpand1-env [env form]
  (if-let [[x & xs] (and (seq? form) (seq form))]
    (if-let [v (and (symbol? x) (resolve x))]
      (if (-> v meta :macro)
        (apply @v form env xs)
        form)
      form)
    form))

```

可以使用 `macroexpand1-env` 来测试上下文相关的 `simplify` 宏在不同的本地绑定的“环境”中会如何表现：

```

(macroexpand1-env '{} '(simplify (range 10)))
; Precomputing: (range 10)
;= (quote (0 1 2 3 4 5 6 7 8 9))
(macroexpand1-env '{range nil} '(simplify (range 10)))
;= (range 10)

```

并且可以验证我们的宏对于 `&form` 上面元数据处理的正确性，只要在传给 `macroexpand1-env` 的 `&form` 上面加上一些元数据就好了。比如可以修改一下前面例子中的那个 `spy` 宏，让它从 `&form` 里面获取行号然后打印出来：

```
(defmacro spy [expr]
```

```

`(let [value# -expr]
  (println (str "line #" "(-> &form meta :line) ",")
           `"-expr value#)
  value#))
:= #'user/spy
(let [a 1
      a (spy (inc a))
      a (spy (inc a))]
  a)
; line #2, (inc a) 2
; line #3, (inc a) 3
:= 3

```

如果想在不去执行这个宏的前提下，对这个宏发射出来的行号的正确性进行验证的话，可以使用 `macroexpand1-env` 宏：

```

(macroexpand1-env {} (with-meta '(spy (+ 1 1)) {:line 42})) ❶
:= (clojure.core/let [value__602__auto__ (+ 1 1)]
:=   (clojure.core/println
:=     (clojure.core/str "line #" 42 ","))
:=   (quote (+ 1 1)) value__602__auto__)
:=   value__602__auto__

```

- ❶ 把传给 `macroexpand1-env` 的 `:line` 元数据指定为一个特定的值，这样我们后面就可以确认这个 `:line` 值确实是我们传进来的。
- ❷ 然后可以看到宏确实用的是从 `&form` 里面拿出来的、我们传进来的那个行号。

看到这里也许你也注意到了，`macroexpand1-env` 的实现可以用宏来加以简化：那些嵌套的 `if-let` 表达式以及重复的 `form` 的表达方式并不十分优美，如果能够把这些 `if-let` 用一个单独的 `form` 代替会优美很多：

```

(defn macroexpand1-env [env form]
  (if-all-let [[x & xs] (and (seq? form) (seq form))
              v (and (symbol? x) (resolve x))
              _ (-> v meta :macro)]
    (apply @v form env xs)
    form))

```

我们希望 `if-all-let` 宏只有在所有的表达式都求值成逻辑 `true` 的时候才去执行 `then form`。可以把这个宏实现成这样：

```

(defmacro if-all-let [bindings then else]
  (reduce (fn [subform binding]
            `(~@binding ~subform ~else))
         then (reverse (partition 2 bindings))))

```

这个正确实现了 `macroexpand1-env` 宏，同时看起来也比那个嵌套的 `if-let` 优雅多了。

深入-> 和 ->>

为了探索怎么用宏来解决实际问题，我们来自己实现一下 Clojure 的“串行宏”：->，它跟它的“表兄弟”->>类似，经常被称为“串行宏”。它们被包含在 `clojure.core` 这个命名空间里面（在很多第三方的库里面也有从它引申出来的一些宏），这些宏对于清理那种多级函数调用以及多级 Java 方法调用的代码非常有用。

举个例子，可以把这样的代码：

```
(prn (conj (reverse [1 2 3]) 4))
```

写成更容易懂的形式：

<260

```
(thread [1 2 3] reverse (conj 4) prn)
```

这么写的话，读代码的时候不用从内向外一层层读了（这种读法对于那种嵌套调用很深的是很难读的），我们可以从左向右顺序地读：“首先有一个 `vector[1 2 3]`，想把它里面的元素反过来，然后 `conj` 一个 4 进去，然后把它 `prn` 出来。”

应该不难想象怎么去实现这么一个宏。对于传入的一堆形式，我们把第一个形式插入到第二个形式的第二个元素的位置上去，然后再把第二个形式插入到第三个元素的第二个元素位置上去，重复这样，直到最后一个形式。

而且如果第一个形式之后的任何一个形式不是一个列表的话，那么我们把它作为一个只有一个元素的列表，这使得我们对于单个参数的函数不用写成这样：

```
(-> foo (bar) (baz))
```

而是写成这种简洁的方式：

```
(-> foo bar baz)
```

首先，我们来编写一个简单的函数来验证一个形式是不是序列，如果不是序列，那么构建一个只包含这个形式的 `seq`，然后返回：

```
(defn ensure-seq [x]
  (if (seq? x) x (list x)))

(ensure-seq 'x)
;= (x)
(ensure-seq '(x))
;= (x)
```

现在，对于给定的两个形式 `x` 和 `ys`，我们希望把 `x` 插入到 `ys` 的第二个元素的位置上去，并且确保 `ys` 是一个序列。

<261

```
(defn insert-second
  "Insert x as the second item in seq y."
  [x ys]
  (let [ys (ensure-seq ys)]
    (concat (list (first ys) x)
            (rest ys))))
```

虽然这是一个普通的函数，但是要记住，这个函数会被宏调用，所以这里 x 和 ys 的值其实是一些没有求值的 Clojure 源代码。

可以利用语法引述以及反引述来更简洁地编写这段代码；再提醒一下，引述不只可以在宏里面使用：

```
(defn insert-second
  "Insert x as the second item in seq y."
  [x ys]
  (let [ys (ensure-seq ys)]
    `(~(first ys) ~x ~(rest ys))))
```

我们还可以利用 list* 来写得更简洁，这个函数在 92 页“创建序列”一节已经详细介绍过了，它在实现宏的时候经常用到：

```
(defn insert-second
  "Insert x as the second item in seq y."
  [x ys]
  (let [ys (ensure-seq ys)]
    (list* (first ys) x (rest ys))))
```

现在正式编写我们的宏，把这个宏命名为 thread。 (thread x) 应该直接返回 x。 (thread x (a b)) 应该利用前面的 insert-second 函数返回 (a x b)， (thread x (a b) (c d)) 先“串行”前两个参数，然后对“串行”的结果以及后面的参数循环调用 thread 宏。

```
(defmacro thread
  "Thread x through successive forms."
  ([x] x)
  ([x form] (insert-second x form))
  ([x form & more] `(thread (thread ~x ~form) ~@more)))
```

看起来它工作得很好，跟标准的 -> 宏表现一样：^{注 22}

```
(thread [1 2 3] (conj 4) reverse println)
;= (4 3 2 1)
(-> [1 2 3] (conj 4) reverse println)
;= (4 3 2 1)
```

注 22：要自己实现一个 ->> 宏也是很简单的，留给大家作为练习。提示：你将需要编写一个 insert-last 函数，而不是一个 insert-second 函数。

但是有一个问题，一直没有问我们自己：对于这个功能可以用一个函数而不是一个宏来实现吗？其实是可以用函数来实现的，只要稍作修改就可以了：

```
(defn thread-fns
  ([x] x)
  ([x form] (form x))
  ([x form & more] (apply thread-fns (form x) more)))

(thread-fns [1 2 3] reverse #(conj % 4) prn)
;= (4 3 2 1)
```

在这种情况下，需要把函数调用包在匿名函数字面量 #() 里面，代码看起来比较繁杂，也比较难读。而且这个用函数实现的版本不支持 Java 的方法调用；而我们的宏版本因为只是对于列表和符号的简单操作，因此是支持的。

```
(thread [1 2 3] .toString (.split " ") seq)
;= ("[1" "2" "3]")


(thread-fns [1 2 3] .toString #(.split % " ") seq)
;= #<CompilerException java.lang.RuntimeException:
;=   Unable to resolve symbol: .toString in this context,
compiling:(NO_SOURCE_PATH:1)> 262

;; This is starting to look a bit hairy...
(thread-fns [1 2 3] #(.toString %) #(.split % " ") seq)
;= ("[1" "2" "3]")
```

所以看起来，我们宏实现的版本确实比函数实现的版本在语法简洁性上有些优势。Clojure 标准库提供的“串行”宏也确实经常被用来对于单个值或者集合值的那种串行调用代码进行清理。

其实 Clojure 标准库里面的 -> 并不比我们实现的 thread 宏复杂多少。clojure.core 还提供了其他一些类似的“串行”宏，包括：

..
它的行为跟 -> 差不多，只是它只支持对于 Java 方法的调用（它还支持 -> 所不支持的 Java 静态方法）。.. 是在 -> 之前引入 Clojure 的，现在已经很少使用它了，你可能在社区的一些代码里面还能看到它。

->>

这个宏跟 -> 很类似，只是它是把前面一个 form 插入到后面一个 form 的最后一个元素的位置上，而不是第二个元素的位置上。这个宏经常被用来对一个序列或者其他数据结构进行转换，比如：

```
(->> (range 10) (map inc) (reduce +))  
:= 55
```

希望我们介绍的例子能让你感到宏的强大威力。

总结

跟其他 Lisp 方言一样，宏是 Clojure 的终极表达力的体现。正确合适地使用宏可以帮助我们消除代码里面的重复、把代码里面的一些通用的模式抽象出来。但是你不应该把它当做你写代码的首选：你可以用 Clojure 编程很久都不去用宏。

函数式编程和数据建模已经给了我们很强的表达力，并且使得我们可以把代码里面的重复模式抽象出来。宏应该被当做我们的终极武器，用它来简化控制流，添加一些语法糖以消除重复代码以及一些难看的代码。

数据类型和协议

在用其他语言编写代码的时候，我们可能都碰到过这样的情况：你设计了一些很漂亮的接口，它们都遵循那些最佳实践，突然发现你要处理的一个对象不是你所能控制的。通常来说，那个对象的维护者不会对这个对象进行修改以让它实现你的漂亮接口，不管是技术原因、政治原因还是法律原因。你可能马上想到的是用一个适配器模式或者是代理模式。

最幸运的情况是，如果我们用的语言是一种足够动态的语言，这种语言允许我们 *monkey-patching*，意思是说，这个语言里面的类是“开的”，我们可以在运行时向一个类里面添加一个方法，这样就可以在不改变那个对象的情况下，在运行时让它实现你的漂亮接口。但是在运用这个技巧之前，你肯定要想一想这个办法所带来的复杂性以及一些隐藏的陷阱。

这里我们问题的根源被命名为表达力问题：

表达力问题是对于一个旧问题的新名字。我们的目标是根据情况定义数据类型，也就是说，我们可以根据情况往数据类型里面添加新函数，而不用重新编译现有代码，而且保持了静态类型安全性（比如说不用进行类型转换）。

——Philip Wadler, <http://www.daimi.au.dk/~madst/tool/papers/expression.txt>

在业界有一些争论：动态语言里面的表达力问题还是不是 Philip Wadler 所指出的那个问题。在这里就不去讨论这个没有答案的问题了，我们把这个问题叫做动态表达力问题。用一个更面向对象的风格来描述就是：

动态表达力问题是一个旧问题的新名字。我们的目标是让一个新的类型来实现已有接口；让一个已有的类型实现一个新的接口，而不用重新编译现有代码。

这个目标的前半部分——让新的类型实现旧的接口，任何面向对象的语言都提供这个能力。而这个目标的后半部分——让旧的类型实现新的接口则是这个问题的有趣部分，这个部分也是很少有语言尝试解决的。我们在这一章来看看 Clojure 是怎么解决的。

协议

Clojure 里面接口的对应物我们称为协议（Protocol）。^{注1}Interface 在 Clojure 业界都是用来特指 Java 接口的。

一个协议由一个或者多个方法组成，而且每个方法可以有多个方法体。所有的方法至少有一个参数，这个参数对应到 Java 里面的 this 以及 Ruby 或者 Python 里面的 self。

协议里面的每个方法的第一个参数是特殊的，因为到底使用这个协议的哪个实现就是根据第一个参数来决定的。说得详细一点就是，根据第一个参数的类型来决定的。因此协议提供的是基于单个参数类型的函数分派，这是一种很受限的多态分派方式，实现的时候基于实用的原因选择了这种分派方式：

- 大多数宿主语言（比如 JVM、CLR 以及 JavaScript 的虚拟机）提供的就是这种分派方式，更重要的是，这种方式是被优化过的。
- 虽然这种分派方式是比较受限的，但是它确实是使用很广泛的分派方式。

如果确实需要不受限制的或者基于多个参数的分派，那么你可以使用 Clojure 的多重方法，我们将在第 7 章中详细介绍。

协议的定义大概是这样的：

```
(defprotocol ProtocolName
  "documentation"
  (a-method [this arg1 arg2] "method docstring")
  (another-method [x] [x arg] "docstring")) ❶
```

- ❶ 因为这个特殊的参数是显式指定的，所以可以使用任何你喜欢的名字 this、self 或者 _ 等。

跟 Clojure 中其他内容的命名方式都不一样的是，协议的命名用的是驼峰风格（CamelCase），因为它们最终会被编译成 JVM 的接口和类。这使得我们可以很容易把协议和类型跟 Clojure 中的其他东西区分开，也使我们可以以更地道的名字跟其他基于 JVM 的语言进行交互。

从用户的角度来看，协议的方法就是函数；在使用它的时候你不需要写什么特别的语法，

注1： 这个对于 Smalltalk 或者 Objective C 的开发者来说一定很熟悉。

可以通过 `doc` 来查看它的文档，可以把它们传给另一个高阶函数，等等。但是设计协议时应该是面向协议的实现者而不是面向最终使用者的。一个好的协议应该是由一些互相没有重复的方法组成的；一个好的协议应该是很容易实现的。一个协议完全可以只有一个方法。



虽然协议的方法是函数，但是在 `defprotocol` 形式里面不能对协议方法的参数使用解构或者剩余参数。比如基于在 38 页“解构函数参数”一节学到的，你可能想要这样定义一个协议的方法，让它接受两个固定参数，然后以一个单独的参数 `more` 来接收那些剩余参数：

```
(defprotocol AProtocol
  (methodName [this x y & more]))
```

但是因为协议最后会被编译成 JVM 的接口——JVM 的接口是不支持这种 Clojure 所支持的剩余参数的，所以如果这么写的话，那么在 JVM 看来你定义了一个接受 4 个参数的方法。

<265

那些附属函数或者工具函数不应该作为协议的一部分，它们应该是构建在协议上面的一层。

如果有必要，直接面向用户的函数与协议的方法可以放到不同的命名空间里面去，以更清楚地告诉用户哪些是公开的 API，哪些是协议实现者的 API。^{注2} 这种命名空间分离的例子在 Clojure 本身中就能找到：在 `clojure.core.protocols` 这个命名空间里面引入了一个名为 `InternalReduce` 的协议：

```
(defprotocol InternalReduce
  "Protocol for concrete seq types that can reduce themselves
   faster than first/next recursion. Called by clojure.core/reduce."
  (internal-reduce [seq f start]))
```

作为 Clojure 以及它的序列操作的使用者，我们可能从来都不需要去知道有这样一个协议。但是如果我们要实现自己的数据结构，并且想提供一个优化的 `reduce` 功能，那么就需要去实现这个协议了。

在讲述协议与数据类型的过程中，我们会使用下面这个 `Matrix` 协议作为例子，它定义了对二维顺序数据结构的访问和更新操作：

```
(defprotocol Matrix
  "Protocol for working with 2d datastructures."
  (lookup [matrix i j])
  (update [matrix i j value]))
```

注 2：这个跟很多 Java 类库里面的 API 与 SPI（服务提供者 API）的区别是一个道理。

```
(rows [matrix])
(cols [matrix])
(dims [matrix]))
```

266 扩展已有的类型

我们对这个协议的第一个实现是一个简单的包含 vector 的 vector。“包含 vector 的 vector”本身不是 Clojure 或者 JVM 里面的一个已有类型，^{注3}所以我们会将这个协议扩展到 Clojure 的 vector 上去：

```
(extend-protocol Matrix
  clojure.lang.IPersistentVector
  (lookup [vov i j]
    (get-in vov [i j]))
  (update [vov i j value]
    (assoc-in vov [i j] value))
  (rows [vov]
    (seq vov))
  (cols [vov]
    (apply map vector vov))
  (dims [vov]
    [(count vov) (count (first vov))])))
```

先介绍一下 `extend-protocol` 这个宏。这个宏的第一个参数是协议的名字（这里是 `Matrix`），紧接着是一些符号（比如指定类型名字的 `IPersistentVector`，它是 Clojure 里面 `vector` 所实现的接口）以及列表（前面指定的类型对于协议的方法的实现）。

方法的实现跟实现普通的函数没有什么区别。跟 Ruby 或者 Java 不同的是，这里没有什么隐式的 `self` 或者 `this`：第一个参数决定了使用协议的哪个实现。从这个角度来说，协议的实现跟 Python 的做法很像。

一个十分值得注意的点是，你不需要实现协议定义的所有方法：如果调用到那些没有实现的方法，Clojure 会简单地抛出一个异常。^{注4}

`extend-protocol` 并不是对一个协议进行扩展的唯一方法，扩展的办法还有：

- 内联实现
- extend
- extend-type

我们会在后面 285 页“重用实现”与 281 页的“内联实现”两节分别介绍 `extend` 以及

注3： 在字节码层面是没有范型信息的，它们在编译阶段就被去掉了——这被称为类型擦除，因为这一点，也使得我们在 JVM 的基础上实现动态语言比较简单。

注4： 到底抛出的是什么异常是跟实现有关的，因此不要在编写代码的时候依赖这个异常类型。

内联实现。

extend-type 跟 extend-protocol 是相对的：extend-protocol 让我们可以把一个协议扩展到多个类型，而 extend-type 则让我们可以把多个协议扩展到一个类型。编写它们的代码结构是类似的，只是指定协议与指定类型的地方互相调换一下就行了。◀ 267

示例 6-1：extend-type 与 extend-protocol 的使用比较

```
(extend-type AType
  AProtocol
  (method1-from-AProtocol [this x]
    (...implementation for AType
     )))
AnotherProtocol
(method1-from-AnotherProtocol [this x]
  (...implementation for AType
   )))
(method2-from-AnotherProtocol [this x y]
  (...implementation for AType
   )))

(extend-protocol AProtocol
  AType
  (method1-from-AProtocol [this x]
    (...implementation for AType
     )))
AnotherType
(method1-from-AProtocol [this x]
  (...implementation for AnotherType
   )))
(method2-from-AProtocol [this x y]
  (...implementation for AnotherType
   )))
```

还可以把一个协议扩展到 nil，这使得我们可以跟潜在的 NullPointerException 说“拜拜”了，因为在把协议扩展到 nil 的时候给它的方法提供了一个默认实现：

```
(extend-protocol Matrix
  nil
  (lookup [x i j])
  (update [x i j value])
  (rows [x] [])
  (cols [x] [])
  (dims [x] [0 0]))

(lookup nil 5 5)
:= nil
(dims nil)
:= [0 0]
```

268 要使用这个“包含 vector 的 vector”的实现，还需要一个工厂函数^{注5}，这个工厂函数用来创建一个指定长宽的空的“包含 vector 的 vector”。

```
(defn vov
  "Create a vector of h w-item vectors."
  [h w]
  (vec (repeat h (vec (repeat w nil)))))
```

现在，让我们实现：

```
(def matrix (vov 3 4))
;= #'user/matrix
matrix
;= [[nil nil nil]
;= [nil nil nil]
;= [nil nil nil]]
(update matrix 1 2 :x)
;= [[nil nil nil]
;= [nil nil :x nil]
;= [nil nil nil]]
(lookup *1 1 2)
;= :x
(rows (update matrix 1 2 :x))
;= ([nil nil nil]
;= [nil nil :x nil]
;= [nil nil nil])
(cols (update matrix 1 2 :x))
;= ([nil nil nil]
;= [nil nil nil]
;= [nil :x nil]
;= [nil nil nil]))
```

很好！我们已经高效地把一个协议给扩展（实现）到一个类型上面去了。而且这些实现函数是命名空间限定的：因此一个协议的方法不会跟另外一个协议的方法相冲突——即使有两个方法的名字是一样的。

为了证明可以把一个协议扩展到任何一个 Java 类，我们下面再提供这个协议的另外一个实现，这次使用一个 Java 的二维数组，数组里面保存的是 double 类型的浮点数，这和 IPersistentVector 不一样，不用怀疑 Java 的数组对协议会有任何潜在的支持了。

示例 6-2：把一个协议扩展到 Java 的数组

```
(extend-protocol Matrix
  (Class/forName "[[D")
```



注 5：Clojure 里面的工厂函数充当了其他语言里面的构造函数以及静态工厂方法的角色。Clojure 里面提供了很多工厂函数，我们已经见过的有 hash-map、sorted-set 等，读过 275 页“构造函数与工厂函数”一节之后，你会想定义自己的工厂方法的。

```
(lookup [matrix i j]
  (aget matrix i j))
(update [matrix i j value]           ②
  (let [clone (aclone matrix)]
    (aset clone i
      (doto (aclone (aget clone i))
        (aset j value))).
    clone))
(rows [matrix]
  (map vec matrix))
(cols [matrix]
  (apply map vector matrix))
(dims [matrix]
  (let [rs (count matrix)]
    (if (zero? rs)
      [0 0]
      [rs (count (aget matrix 0))))]))
```

- ❶ 这里的 `Class.forName` 调用允许获得一个指向两维 `double` 数组的 `Class` 引用，Java 里面表示成 `double[][] .class`，在 444 页“数组类”一节有关于这种表示法的更详细介绍。
- ❷ 数组是可变的，但我们通过 `Matrix` 实现提供了某种程度的不可变语义，使得它跟前一个例子“包含 `vector` 的 `vector`”的表现一致：对于每个 `update` 操作先克隆最外层的数组，克隆受影响的行，然后对要更新的位置进行更新，再把克隆的行放回最外层数组，然后返回。当然，这里为了维护这种不可变性对性能有所损耗，可以定义另外一个可变版本用于那种对性能要求很高的场景。

当调用 `Matrix` 方法的时候，第一个参数传入数组的话，那么 Clojure 会自动帮我们选择刚刚实现的这个数组版本，而且这些实现的方法跟我们的期望也是一致的：

```
(def matrix (make-array Double/TYPE 2 3))
:= #'user/matrix
(rows matrix)
:= ([0.0 0.0 0.0]
:= [0.0 0.0 0.0])
(rows (update matrix 1 1 3.4))
:= ([[0.0 0.0 0.0]
:= [0.0 3.4 0.0]])
(lookup (update matrix 1 1 3.4) 1 1)
:= 3.4
(cols (update matrix 1 1 3.4))
:= ([0.0 0.0]
:= [0.0 3.4]
:= [0.0 0.0])
(dims matrix)
:= [2 3]
```

现在应该很清楚了，虽然 Clojure 里面的协议跟 Java 里面接口是一样的角色，但是协议比接口更好，它没有动态版本的表达力问题：可以让任何已有的类实现我们的一个协议，而且不需要对这个类进行任何修改或者重新编译。

但是，到现在为止，我们都是把协议扩展到用 Java 实现的类。还好，Clojure 提供了定义新类型的办法。

定义你自己的类型

一个 Clojure 类型是一个 Java 类，不过定义 Clojure 的类型要简单很多：

```
(defrecord Point [x y])
```

或

```
(deftype Point [x y])
```

我们在讨论 `deftype` 和 `defrecord` 的不同点之前先讨论一下它们的共同点。

这两种定义方法都会定义出一个新的 `Point` Java 类，Java 类里面有两个以 `public` 和 `final` 修饰的名为 `x` 和 `y` 字段。跟协议一样（跟 Clojure 里面其他东西不一样），类型的名称是采用驼峰风格的，而不是小写字母加横线风格的，因为它们被编译成了 Java 的类。要创建一个新的 `Point` 实例，直接调用它的构造函数 (`Point. 3 4`) 就好了——跟定义的时候一样传递对应的参数，不管你定义的是普通类型还是记录类型，因为它们都是 Java 对象上的普通字段：

- 访问与更新这些字段的值比对应的对 Clojure 的 map 的操作要快很多。
- 比如，你可以通过 Java 互操作的语法来访问 `deftype` 或者 `defrecord` 定义的类的字段：^{注6}

```
(.x (Point. 3 4))
  := 3
```

定义的每个字段都是 `java.lang.Object` 类型，这对于很多建模需求来说都已经足够了，而且如果你的模型里面的某些字段会变的话，这一点也是必须的。但是如果你需要把字段定义成基本类型，那么可以用类型提示来指定。可以通过类型提示指定某个字段为非基本类型，但是这个不会改变这个字段的具体类型。

所以下面的例子定义了一个具有 `long` 类型的、名字分别为 `x` 和 `y` 的两个字段，以及一个类型为 `Object` 的名字为 `name` 的字段（虽然类型提示它是 `String` 类型的）：

^{注6}： 记录类型提供了更灵活的字段访问方法，在 273 页“记录是一种关系型数据结构”一节有详细描述。

```
(defrecord NamedPoint [^String name ^long x ^long y])
```

◀ 271

我们会在 366 页“为了效率进行类型提示”以及 438 页“声明函数接受和返回原始类型”两节分别详细讨论类型提示与类型声明，但是现在并不影响我们理解 `deftype` 和 `defrecord` 的实现原理。

有时候知道一个类型到底有哪些字段是很有用的，特别是对于记录类型，记录类型支持在运行时添加字段。这个类里面所有字段的集合被称为 `basis`，如果你的类型是通过 `deftype` 或者 `defrecord` 定义的，那么可以通过下面的方法获取：

```
(NamedPoint/getBasis)
:= [name x y]
```

在这个 `basis` 里面的每个符号都保持了定义时指定的所有元数据，包括类型信息：

```
(map meta (NamedPoint/getBasis))
:= ({:tag String} {:tag long} {:tag long})
```

我们将首先讨论记录类型：因为这种类型就是设计来表示应用级别的数据的，^{注7} 而 `deftype` 则是设计来表示比较底层的一些类型的，比如如果你要实现一种新的数据结构，那么应该使用 `deftype`。

这两种方式的主要区别在于，`defrecord` 对于所定义的类型提供了与 Clojure 以及 Java 进行互操作的一些默认行为，而 `deftype` 则提供了一些对底层操作进行优化的能力。所以最后你会发现大多时候用的是 `defrecord`，而很少用 `deftype`。

类型并不是命名空间限定的

当你用 `defrecord` 或者 `deftype` 定义了一个新类型，这个定义好的类型是被定义在跟所在命名空间对应的一个 Java package 里面的，而且它会被默认引入所定义的命名空间里，因此你可以以无限定名称去直接引用它。但是如果你切换到另外一个命名空间，那么即使已经 `use` 或者 `require` 了定义它的命名空间，还需要显式地 `import` 这个类，因为它是宿主语言的类，不是 `var`。

```
(def x "hello")
:= #'user/hello
(defrecord Point [x y])
:= user.Point
```

❶

注7： 你可以这么认为：记录类型和 map 是 Clojure 世界的“POJO”。

```
(Point. 5 5)
:= #user.Point{:x 5, :y 5}
(ns user2)
(refer 'user)          ❸
x                      ❹
:= "hello"
Point                  ❺
:= CompilerException java.lang.Exception:
:=   Unable to resolve symbol: Point
(import 'user.Point)    ❻
Point
:= user.Point
```

- ❶ 我们在命名空间 `user` 里面定义了一个新类型 `Point`，它会定义出一个全限定名为 `user.Point` 的类。可以直接以无限定名引用它，因为它被隐式地引入了定义它的命名空间。
- ❷ `refer` 跟 `use` 类似，但是它没有加载的语义，它会假定这个命名空间已经存在了。
- ❸ `x` 现在在当前命名空间 (`user2`) 被 `referred` 了。
- ❹ 但是现在是找不到 `Point` 这个类的。
- ❺ 因为 `Point` 是一个类，要想直接引用它的话，需要把它 `import` 到当前命名空间里面来。

关于 Clojure 命名空间的更多细节可以查看 322 页“[定义和使用命名空间](#)”一节。

记录

通常被称为记录类型，由 `defrecord` 定义的记录类型其实是由 `deftype` 定义的类型的一种特例。添加了如下这些额外的特性：

- 值语义。
- 实现了关系型数据结构的接口。
- 元数据的支持。
- 对于 Clojure reader 的支持，比如我们可以通过 Clojure reader 直接读入一个记录类型。
- 一个额外的、方便的构造函数，使得我们可以在创建实例的时候添加一些元数据以及一些额外的字段。



Clojure 里面还提供了一种比较弱的“struct” map 实现——可通过下面这些方法来构造：`defstruct`、`create-struct`、`struct-map` 以及 `struct`，这些函数都应该被认为已经过时了，应该避免使用。如果你想要一个很灵活的 struct，那么可以直接使用 `map`，在 118 页“`map` 作为临时的 struct”已经介绍过了；而如果你想要一个比较正式的模型表示的话，那么记录类型在任何方面都要比 `struct-map` 好。我们会在 277 页“什么时候使用普通 `map`，什么时候使用记录”一节介绍如何选择。

值语义。值语义意味着两件事情：记录类型是不可变的；如果两个类型的所有对应字段都相等，那么这两个记录本身也是相等的：

```
(defrecord Point [x y])
:= user.Point
(= (Point. 3 4) (Point. 3 4))
:= true
(= 3 3N)
:= true
(= (Point. 3N 4N) (Point. 3 4))
:= true
```

273

你在使用 Clojure 的数据结构的时候应该已经习惯了这种语义，这里记录类型提供一样的保证，记录通过自动提供正确而一致的 `Object.equals` 与 `Object.hashCode` 实现来提供这个保证。

记录是一种关系型数据结构。记录类型实现了关系型数据结构的接口，^{注8} 所以那些对 `map` 进行操作的函数你都可以用在记录类型的实例上。

比如，虽然对于普通类型以及记录类型的字段你可以通过 Java 互操作的操作符 (`.x instance`) 来访问，对于记录类型的字段，可以通过关键字来访问，就像我们在 111 页“访问集合元素的简洁方式”一节所介绍的那样，关键字本身是函数，可以通过它来找到集合里面 `key` 为这个关键字所对应的值。这样的调用在记录类型上与在普通 `map` 上效果一模一样：

```
(:x (Point. 3 4))          ●
:= 3
(:z (Point. 3 4) 0)          ●
:= 0
(map :x [(Point. 3 4)
           (Point. 5 6)
           (Point. 7 8)])
:= (3 5 7)
```

注 8：由 `clojure.lang.Associative` 接口定义，在 99 页“Associative”一节有详细描述。

- ① 注意，当关键字被显式地放在函数位置上时，编译器会对这个调用进行优化，使得这种访问的速度跟直接字段访问 (.x instance) 差不多。
- ② 还可以像调用 map 一样提供一个“默认值”。

而要对一个字段进行更新也只要简单调用 assoc 就好了。所以其他关系型集合相关的函数 keys、get、seq、conj、into 等都可以用在记录类型上。并且跟 Clojure map 一样，记录类型也实现了 Java 的 java.util.Map 接口，所以可以把记录类型传给任何需要 Java Map 的函数 / 方法来使用。

虽然在定义的时候指定了记录类型有哪些字段，但是你还是可以在运行时给它添加新的字段：^{注9}

```
274  (assoc (Point. 3 4) :z 5)          ❶
      := #user.Point{:x 3, :y 4, :z 5}
      (let [p (assoc (Point. 3 4) :z 5)]
        (dissoc p :x))
      := {:y 4, :z 5}
      (let [p (assoc (Point. 3 4) :z 5)]
        (dissoc p :z))
      := #user.Point{:x 3, :y 4}
```

- ❶ 可以给记录类型添加新的字段。
- ❷ 从记录类型实例去掉一个预定义的字段的话，返回的就不是记录类型了，而是被降级成一个普通的 map。
- ❸ 但是如果你去除的是运行时额外添加的字段，那么返回的仍然是记录类型，而不会发生降级。

注意，这些运行时额外添加的字段是被保存在一个单独的 Clojure hashmap 里面的，因此它们的语义也是普通 map 的语义——它们并没有真的被添加到底层那个 Java 类上去：

```
(:z (assoc (Point. 3 4) :z 5))
:= 5
(.z (assoc (Point. 3 4) :z 5))
:= #<java.lang.IllegalArgumentException:
:= No matching field found: z for class user.Point>
```

元数据支持。跟 Clojure 里面的其他集合一样，你可以通过 meta 来获取记录的元数据信息，通过 with-meta (以及 vary-meta) 来设置记录的元数据信息——而不会影响记录的值语义：

^{注9：} 这种可以往一个有着固定字段的类型对象里面添加新的字段的特性，在其他语言或者框架里面有时候被称为 expando 特性。

```
(-> (Point. 3 4)
     (with-meta {:foo :bar})
     meta)
:= {:foo :bar}
```

135页“元数据”一节有关于元数据以及它的使用场景的详细介绍。

可读入的表示法。你可能已经意识到了，REPL用一种比较特殊的表示法来打印记录的实例，跟普通的Clojure map不一样，跟其他任何Java对象也不一样：

```
#user.Point{:x 3, :y 4, :z 5}
```

这是一个记录字面量，跟表示vector的中括号、表示关键字的冒号的作用是一样的。这意味着你可以从记录的一个文本表示读入一个记录实例——跟Clojure的其他字面量一样：

```
(pr-str (assoc (Point. 3 4) :z [:a :b]))
:= "#user.Point{:x 3, :y 4, :z [:a :b]}"
(= (read-string *)
    (assoc (Point. 3 4) :z [:a :b]))
:= true
```

这使得使用记录来存储/获取数据非常方便（保存在一个文件或者数据库里面），因为它
是被Clojure的reader直接支持的。<275>

附加构造函数。除了那个接受预定义字段的标准构造函数，记录类型还提供了一个附加的构造函数，这个构造函数也展示了它的一些独特的能力：对于扩展字段以及元数据的支持。这个附加的构造函数接受两个额外的参数，一个包含额外字段的map以及一个包含元数据的map：

```
(Point. 3 4 {:foo :bar} {:z 5})
:= #user.Point{:x 3, :y 4, :z 5}
(meta *)
:= {:foo :bar}
```

这个在语义上跟下面的代码是等价的（但是更高效）：

```
(-> (Point. 3 4)
     (with-meta {:foo :bar})
     (assoc :z 5))
```

构造函数与工厂函数

构造函数通常不应该作为你的公开API的一部分。相反，应该对于你的类型提供一些工厂函数，因为：

1. 工厂函数可能更适合调用者使用，因为由底层 `deftype` 或 `defrecord` 生成的构造函数通常太底层，包含了一些调用者可能根本不关心的细节在里面。
2. 可以把工厂函数作为普通函数一样传给其他高阶函数，以对生成的记录进行处理。
3. 可以最大化你的 API 的稳定性——即使在底层模型发生变化的时候。

工厂函数不会像构造函数那么脆弱：只要改变了记录的字段列表，就改变了构造函数的签名，但是工厂函数使你可以在工厂函数内部添加一些逻辑给新加的字段一些默认值。`Clojure` 的类型不允许自定义构造函数，因此在其他语言里面你想写进构造函数的逻辑，在 `Clojure` 里面你都应该写到工厂函数里面去。

`deftype` 和 `defrecord` 都会隐式创建一个形如 `->MyType` 的工厂函数，它接受的参数跟定义类型时候的字段列表一样：

```
(->Point 3 4)
:= #user.Point{:x 3, :y 4}
```

记录类型还隐式生成另外一个工厂函数 `map->MyType`，它接受一个 `map` 作为参数，这个 `map` 包含了要填充给新 `MyType` 实例的信息：

```
[276] > (map->Point {:x 3, :y 4, :z 5})
:= #user.Point{:x 3, :y 4, :z 5}
```

这些对于创建普通类型以及记录类型都是很有用的，特别是在跟高阶函数一起使用的时候：

```
(apply ->Point [5 6])
:= #user.Point{:x 5, :y 6}

(map (partial apply ->Point) [[5 6] [7 8] [9 10]])
:= (#user.Point{:x 5, :y 6}
:= #user.Point{:x 7, :y 8}
:= #user.Point{:x 9, :y 10})

(map map->Point [{:x 1 :y 2} {:x 5 :y 6 :z 44}])
:= (#user.Point{:x 1, :y 2}
:= #user.Point{:x 5, :y 6, :z 44})
```

对于记录类型，这个 `map->MyType` 函数还可以通过静态方法 `create` 来访问，这对于 Java 调用者来说很方便：

```
(Point/create {:x 3, :y 4, :z 5})
:= #user.Point{:x 3, :y 4, :z 5}
```

虽然提供的这些工厂函数很有用，但是还是会需要编写自己的工厂函数，比如你可以在你的工厂函数里面添加一些校验逻辑等：

```

(defn log-point
  [x]
  {:pre [(pos? x)]}
  (Point. x (Math/log x)))

(log-point -42)
;= #<AssertionError java.lang.AssertionError: Assert failed:
(pos? x)
(log-point Math/E)
;= #user.Point{:x 2.718281828459045, :y 1.0}

```

很多情况下，工厂函数自然出现在从使用普通 map 转换到使用记录类型的时候，我们可能开始对代码进行重构，引入工厂函数，让工厂函数来创建普通 map，以让代码看起来更清晰。比如：

```

(defn point [x y]
  {:x x, :y y})

```

后面如果想要把某个 map 转换成记录类型的话，我们只需要重写对应的工厂函数让它创建新的记录类型就可以了。而其他使用 map 的代码经常是不需要改变的，而且也不知道所操作的对象类型发生了变化。

什么时候使用普通 map，什么时候使用记录类型

277

虽然有很多场景都适合使用记录类型，^{注10}但是通常鼓励先使用 map，然后在实在需要的时候再换成记录类型。

map 是进行编程以及数据建模的最简单的方法，因为它不强制你去预先定义任何类型，从而给了我们很大的灵活性。但是，当需要基于类型的多态的时候（由协议以及记录类型、普通类型提供的），或者需要对字段的访问具有很好的性能，那么就可以切换到记录类型上了：大多数代码不需要做修改（可能有极少代码需要修改），因为 map 和记录类型实现了同样的关系型数据结构的接口。

从 map 切换到记录类型的时候，有一个陷阱是：记录类型不是函数。所以，((Point. 3 4) :x) 这样的代码是不能工作的，而 ({:x 5 :y 6} :x) 是可以的。但是如果你遵守了 111 页“访问集合元素的简洁方式”一节提到的什么时候适合把 map 当做函数来使用的建议的话，那么作为函数使用的 map 以及能被记录类型替代的那种 map 的使用场景应该是没有交集的。

另外一个陷阱是普通 map 和记录类型永远不可能相等，所以如果你的代码里面 map 和记录类型混合使用的话，那么要特别注意了。

注 10： 第 18 章提供了一个图帮助你思考这个领域的问题。

```
(defrecord Point [x y])
:= user.Point
(= (Point. 3 4) (Point. 3 4))
:= true
(= {:x 3 :y 4} (Point. 3 4))
:= false
(= (Point. 3 4) {:x 3 :y 4}) ●
:= false
```

- 还好，Clojure 没有打破相等的对称性规则。

类型

`deftype` 是 Clojure 里面最底层的定义形式，`defrecord` 其实只是包装了 `deftype` 功能的一个宏。你可能已经预料到了，很多由记录类型所提供的方便的特性在 `deftype` 所定义的类型里面是没有的。它本身就是被设计来定义那种最底层的框架类型的，比如定义一种新的数据结构或者引用类型。而相对应来说，普通 `map` 以及记录类型则是应该用来表示你的应用级别的数据的。

而这种低级别的类型确实提供了一种在编写底层应用或者库时“有时候”所不能避免的特性：可修改的字段！

在深入了解可修改字段之前，先澄清一下，对于由 `deftype` 定义的、普通（不可修改）字段的访问只能通过 Java 互操作的语法：

278 ◆

```
(deftype Point [x y])
:= user.Point
(.x (Point. 3 4))
:= 3
(:x (Point. 3 4))
:= nil
```

`deftype` 定义的类型没有实现关系型数据结构的接口，¹¹ 所以那种使用关键字来访问字段内容的函数在这种类型里面是没办法用的。所以我们必须依赖这个事实：这个定义的类型最终会被编译成 Java 类，而所有的不可修改字段都被定义成这个 Java 类里面的 `public final` 字段。

可修改字段则有两种类型：`volatile` 或者非 `synchronized` 的。要把一个字段定义成可修改的，可以给它加上元数据 `^:volatile-mutable` 或者 `^:unsynchronized-mutable`，比如：

```
(deftype MyType [^:volatile-mutable fld])
```

注 11：除非你是要实现 Clojure 的 `clojure.lang.Associative` 接口；我们会在 292 页“自己实现一个 set”一节做类似的事情。

这个`*-mutable`元数据可以跟其他要指定的元数据一起作用在要定义的字段上。

这里的“Volatile”跟Java里面字段修饰符的那个`volatile`意义是一样的：它保证对于该字段的读写是原子的^{注12}并且必须以代码中指定的顺序发生。也就是说，它们不能被JIT编译器或者CPU进行重新排序。所以`Volatile`修饰的字段是多线程安全的，但是多线程之间是没有协调的，因此还是可能会产生竞争条件。

而另一方面，非`synchronized`字段则是一个“普通”的Java可修改字段，只有在它上面显式地加一个锁^{注13}它才会线程安全。

我们定义的不可修改字段始终是`public`的，但是可修改字段则始终是`private`的，并且只能在定义类型的那个形式的那些内联方法里面使用。关于类型的内联实现我们会在280页“实现协议”一节详细介绍；而现在提供一个简单的例子，这个例子类型包含一个可修改的字段，并且它实现了`IDeref`接口的`deref`方法：^{注14}

薛定谔的猫

◀ 279

```
(deftype SchrödingerCat [^:unsynchronized-mutable state]
  clojure.lang.IDeref
  (deref [sc]
    (locking sc
      (or state
          (set! state (if (zero? (rand-int 2))
                          :dead
                          :alive))))))

(defn schrödinger-cat
  "Creates a new Schrödinger's cat. Beware, the REPL may kill it!"
  []
  (SchrödingerCat. nil))

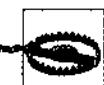
(def felix (schrödinger-cat))
;= #'user/felix
@felix
;= :dead
(schrödinger-cat)
;= #<SchrödingerCat@3248bc64: :dead>
(schrödinger-cat)
;= #<SchrödingerCat@3248bc64: :alive>
```

注12：根据Java内存模型，写一个非`volatile`的`long`或者`double`字段是不能保证原子性的，详情查看JSR-133，第11节。

注13：如果你足够勇敢，而且确实很需要可修改字段，那么你可以使用`locking`宏，这个宏在225页“Locking”一节有介绍。

注14：任何实现了`IDeref`接口的类型可以利用`deref`来解引用，因此也就可以使用`@`这个Clojure reader语法，这一点在160页的注意栏中有描述。

- ❶ Felix 现在是活的也是死的，直到……
- ❷ 直到我们（或者这个 REPL）杀掉它的时候才能知道——`deref` 的一个副作用。



使用可修改字段还是引用 / 数据流类型？

因为我们的 `SchrödingerCat` 的状态是不确定的，直到我们“去盒子里看看它”，它和下面的代码是等价的：

```
(delay (if (zero? (rand-int 2))
      :dead
      :alive))
```

大多数情况下，一个应用的可修改性需求可以由 Clojure 的引用类型 (`agents`、`atoms` 以及 `refs`)、数据流类型 (`futures`、`promises` 以及 `delays`) 来满足，或者可以使用 `java.util.concurrent` 里面的那些类。使用这些手段——如何使用在第 4 章有详细介绍——比使用可修改的字段简单很多：它们消除了很大的复杂性，它们专注于一个限定轨迹的可修改性，所以通常大多数情况下它们帮你解决了所有的问题。所以简单来说，在有更好的工具的情况下，不要去碰可修改的字段！

[280] 从上面例子的代码可以看出，要对一个可修改字段进行设值可以简单地使用 `set! : (set! field value)`。^{注 15} 在类型的实现体里面我们可以通过直接引用这个可变字段来访问它。^{注 16}

现在理解了 Clojure 里面各种不同的类型，可以继续我们的协议探索之旅了。

实现协议

要让一个给定的类型实现一个协议有两种方法：

1. 在用 `deftype` 或者 `defrecord` 定义类型的时候直接把这些方法给实现了，这种方法叫做内联实现。
2. 使用 `extend*` 系列函数来把一个类型的实现注册到协议上去。

我们通过例子 6-3 和 6-4 来比较一下这两种方法把协议 `Matrix` 扩展到 `Point` 记录类型时候的区别，在这些例子里面我们把一个 `Point` 实例当做一个 2×1 的 `Matrix`：

^{注 15}：这个跟 Java 里面的 `this.field = value`、Python 里面的 `self.field = value` 以及 Ruby 里面的 `@field = value` 相对应。

^{注 16}：可修改字段在方法体内跟本地绑定看起来差不多，因此你可以直接引用它，也就是说，你不需要用什么特殊的语法，比如 `(.x this)`，而且如果你使用“`let`”定义一个同名的本地绑定的话，那么这个可修改字段会被“遮住”，从而你使用的则是那个本地绑定，而不是这个可修改字段。

示例6-3：内联实现一个协议

```
(defrecord Point [x y]
  Matrix
  (lookup [pt i j]
    (when (zero? j)
      (case i
        0 x
        1 y)))
  (update [pt i j value]
    (if (zero? j)
      (condp = i
        0 (Point. value y)
        1 (Point. x value))
      pt))
  (rows [pt] [[x] [y]])
  (cols [pt] [[x y]])
  (dims [pt] [2 1]))
```

示例6-4：把一个协议扩展到已存在的类型

```
(defrecord Point [x y])

(extend-protocol Matrix
  Point
  (lookup [pt i j]
    (when (zero? j)
      (case i
        0 (:x pt)
        1 (:y pt))))
  (update [pt i j value]
    (if (zero? j)
      (condp = i
        0 (Point. value (:y pt))
        1 (Point. (:x pt) value)))
      pt))
  (rows [pt]
    [[[(:x pt)] [(:y pt)]]])
  (cols [pt]
    [[[(:x pt) (:y pt)]]])
  (dims [pt] [2 1])))
```

281

这两种方法的一个细微区别是你如何访问字段的值：当你在类型之外访问类型的字段时你需要以关键字（或者 Java 互操作方法）来访问，比如 (.x pt)，而当你内联实现的时候可以直接引用字段名字来获取字段的值，因为内联的时候这些字段名在词法范围内。

除了这个之外，内联实现跟用 extend-* 实现之间还有很多区别。

内联实现

一般来说，内联实现的性能会好一些。有以下两个原因：内联实现的时候能直接访问类型的字段；内联实现的时候调用协议的方法跟 Java 里面调用一个接口的方法一样快。

因为每个协议最终是被编译成一个 Java 的接口，以内联方式实现一个协议的方法会产生一个实现了协议方法的 Java 类，而那个 Java 类的实现就是对应的协议的方法的实现。更进一步说，每个对于协议函数的调用首先会做一个针对对应接口的测试，而这个测试可以使用我们所能有的最快的办法：一个普通的方法调用，这是 JVM 所能识别的，同时也进行了高度优化的。

但是使用内联实现的话也会有一些潜在的问题，比如如果两个协议有同样签名的方法，那么这两个方法就会冲突。你没办法以内联的方式为冲突的两个方法提供实现，甚至不能为任何跟 `java.lang.Object` 方法冲突的协议方法提供实现；或者更让人迷惑的是，如果你尝试提供那些 `defrecord` 会自动提供的方法实现，包括 `java.util.Map`、`java.io.Serializable` 以及 `clojure.lang.IPersistentMap` 里面的方法，那么会抛出一个错误：

```
282 > (defprotocol ClashWhenInlined
          (size [x]))
      := ClashWhenInlined
  (defrecord R [])
    ClashWhenInlined
    (size [x]))           ①
    := #<CompilerException java.lang.ClassFormatError:
    :=  Duplicate method name&signature in class file user/R,
        compiling:(NO_SOURCE_PATH:1)>

  (defrecord R [])
  := user.R
  (extend-type R           ②
    ClashWhenInlined
    (size [x]))
  := nil
```

① 这个方法会跟 `java.util.Map` 的 `size` 方法冲突。

② 而用 `extend-type` 来把一个类型的实现注册到协议就没有这个问题了，因为这个注册的行为不会影响到 `R` 类型本身的创建——为了实现这个协议，这个类本身并没有发生什么变化。

因为用内联实现的时候，这些实现代码是直接写入 `deftype` 或 `defrecord` 所定义的类文件里面的，所以除非你重新定义整个类型，否则无法在运行时改变这些实现。而且如果

你真的这么做的话，那就意味着所有依赖这个类的代码都需要重新编译。^{注17} 而内联实现最致命的问题是，`deftype` 和 `defrecord` 都会创建一个新的类型，因此在它之前已经存在的对象都无法利用任何更新的内联实现。

所以，内联实现因为看起来很熟悉（像 Java 类里面的方法），大家都想这么用，但是它对协议进行实现的方法都要“静态”，因此应该只在进行性能优化的时候才去考虑这种实现方法。

作为这个通用规则的一个特例，要实现 Java 的接口的话，只能用这种方法。

Java 接口的内联实现

因为以内联的方式实现一个接口只是简单地实现这个协议所对应的 Java 接口，那么我们可以用同样的办法来实现任何 Java 的接口。作为一个特例，我们可以把 `java.lang.Object` 的方法也作为接口来实现。^{注18} 跟实现协议一样，你也不需要实现接口的所有方法，对于没有实现的方法在调用的时候会自动抛出一个异常。

```
(deftype MyType [a b c]
  java.lang.Runnable
  (run [this] ...)
  Object
  (equals [this that] ...)
  (hashCode [this] ...)
  Protocol1
  (method1 [this ...] ...)
  Protocol2
  (method2 [this ...] ...)
  (method3 [this ...] ...))
```

◀ 283

可以对 `Object` 里面的方法进行重写使得我们可以通过 `deftype` 定义的 `Point` 类型获得值语义：

```
(deftype Point [x y]
  Matrix
  (lookup [pt i j]
    (when (zero? j)
      (case i
        0 x
        1 y)))
  (update [pt i j value]
    (if (zero? j)
```

注 17：这么多的冗长代码会大大缩短，如果你遵从 275 页“构造函数和工厂函数”一节的建议：为类型实现一个工厂函数。

注 18：任何其他类，不管它是不是抽象的，都不能内联继承。你必须使用 `gen-class` 或者 `proxy` 来处理，374 页“定义具名的类”以及 372 页“匿名类的实例：proxy”分别对这两种方法进行了详细介绍。

```

(case i
  0 (Point. value y)
  1 (Point. x value))
pt))
(rows [pt]
  [[x] [y]])
(cols [pt]
  [[x y]])
(dims [pt]
  [2 1])
Object
>equals [this other]
  (and (instance? (class this) other)    ❶
    (= x (.x other)) (= y (.y other))))
(hashCode [this]
  (-> x hash (hash-combine y)))

```

- ❶ 内联实现无法获得所定义类型的引用。



在 Clojure 里面实现 equals 和 hashCode 跟在 Java 里面实现一样无聊而且有风险——让 defrecord 来帮我们实现最好了。要为由 deftype 定义的类型实现这些方法就更微妙了：你不能把 (class this) 替换成 Point，替换之后代码能够编译通过，但是 (instance? Point other) 会始终返回 false。这是当前编译器的一个已知缺陷。其他可能的办法是在这些方法内调用一个能够引用到所定义类的函数；或者干脆不要测试是否是类型的一个实例，而测试是不是接口（或者协议）的一个实例。

284 用 reify 来定义匿名类型

除了 defrecord 和 deftype，还有一种结构可以接受内联实现：reify。

跟 deftype 和 defrecord 不同的是，reify 不是一个顶级形式，它不会定义出一个有名类型，它直接求值出一个无名类型的实例。从根本上来说，这是一种创建实现任何协议（或者实现任何接口或者 Object 的方法）的类实例的方法。它其实对应到 Java 里面的匿名内部类。

reify 的整体结构跟 defrecord 或者 deftype 类似，但是不定义任何字段：

```

(reify
  Protocol-or-Interface-or-Object
  (method1 [this x]
    (implementation))
  Another-Protocol-or-Interface
  (method2 [this x y]
    (implementation)))

```

```
(method3 [this x]
  (implementation)))
```

跟实现普通类和记录类一样，不需要对协议或者接口的所有方法提供实现。

由 `reify` 所创建的类实例创建了一个闭包，使得方法的实现体可以直接访问当前词法范围内的任何本地绑定。这个对于创建适配器（比如示例 6-5）或者创建一次性的对象（比如示例 6-6）是非常方便的。

示例6-5：把一个函数包装成一个ActionListener

```
(defn listener
  "Creates an AWT/Swing `ActionListener` that delegates to the given function."
  [f]
  (reify
    java.awt.event.ActionListener
    (actionPerformed [this e]
      (f e))))
```

示例6-6：使用一个reified的FileFilter实现来获取目录文件

```
(.listFiles (java.io.File. ".")
  (reify
    java.io.FileFilter
    (accept [this f]
      (.isDirectory f))))
```

这样的场景跟 Clojure 里面 `proxy` 的场景有重合，^{注19} 但是：

- `reify` 更简单。它的方法实现是“嵌入”到所定义的匿名类里面去了——跟 `deftype` <285> 和 `defrecord` 一样，因此对于方法实现的动态更新是不支持的。
- `reify` 更受限。它只能实现协议、实现 Java 接口的方法以及 `Object` 的方法。它不能继承一个具体的类，不管这个类是不是抽象类。
- 因为所有 `reify` 的方法被嵌入到宿主语言的类文件里面去了，因此调用这些方法没有什么额外的开销。

重用实现

在 Clojure 的协议与类型的体系里面，没有“层级”的概念。就像我们将要看到的那样，这并不是一个限制，因为到目前为止，大家都认为基于类型的继承层级本身是复杂而又受限的，而 Clojure 提供了一种更有效同时也更灵活的方式。

类型只能实现协议或者实现接口——没有办法像在其他语言里面一样让一个类型去继承另一个具体类型，从而继承它的实现。Clojure 对于这个问题的解决办法是利用 `extend-`

^{注 19：} 详情请查看 372 页“匿名类的实例”一节。

`type` 和 `extend-protocol` 宏的基础：`extend` 来重用方法实现。

`extend` 接受的第一个参数是要进行扩展的类型，然后是要实现的协议的名字以及具体的方法实现的 map，这个 map 里面以方法名作为 key（方法名所对应的关键字），map 的 value 是这个方法的实现。

让我们用 `extend` 来让 `Point` 记录类型实现 `Matrix` 协议：

```
(defrecord Point [x y])

(extend Point
  Matrix
  {:lookup (fn [pt i j]
             (when (zero? j)
               (case i
                 0 (:x pt)
                 1 (:y pt))))
  :update (fn [pt i j value]
            (if (zero? j)
                (condp = i
                  0 (Point. value (:y pt))
                  1 (Point. (:x pt) value))
                pt))
  :rows (fn [pt]
          [[[(:x pt)] [(:y pt)]]])
  :cols (fn [pt]
          [[[(:x pt) (:y pt)]]])
  :dims (fn [pt] [2 1]))
```

因为 `extend` 是一个函数，而不像 `extend-type` 和 `extend-protocol` 是宏，因此这个实现 map 被当做一个值来到处传递，对它进行操作、转换或者跟其他的实现 map 合并等。有了这种灵活性，我们可以进行任何形式的实现重用，可以利用它来实现“继承”，或者更复杂的“traits”和“mixin”。

一个简单的应用是为一些方法提供默认实现，比如我们这里的 `:rows` 和 `cols`，它们跟具体类型的 `dims` 和 `lookup` 实现有关：

```
(def abstract-matrix-impl
  {:cols (fn [pt]
           (let [[h w] (dims pt)]
             (map
               (fn [x] (map #(lookup pt x y) (range 0 w)))
               (range 0 h))))
  :rows (fn [pt]
          (apply map vector (cols pt)))))
```

现在我们把 `Matrix` 扩展到 `Point` 类型，只需把默认的实现 map `assoc` 到具体类型的实现

map 上去就可以了：

```
(extend Point
  Matrix
  (assoc abstract-matrix-impl
    :lookup (fn [pt i j]
      (when (zero? j)
        (case i
          0 (:x pt)
          1 (:y pt))))
    :update (fn [pt i j value]
      (if (zero? j)
        (condp = i
          0 (Point. value (:y pt))
          1 (Point. (:x pt) value)))
        pt))
    :dims (fn [pt] [2 1])))
```

虽然这个例子非常简单，而且对于主流的面向对象的设计思想也没有多少创新，但是它确实演示了 extend 提供的这种灵活性，可以把方法实现简单地看做命名函数，然后你可以利用这些函数在类型之间构建任何继承关系，而不用管具体的类型原本到底有什么关系。

更有趣的例子可能是利用这个特性来实现 mixins 特性——也就是说，一个方法的实现是把其他几个具体的实现通过某种有意义的方式组合起来。

为了这个例子，我们定义一个新的协议——Measurable：^{注 20}

```
(defprotocol Measurable
  "A protocol for retrieving the dimensions of widgets."
  (width [measurable] "Returns the width in px.")
  (height [measurable] "Returns the height in px."))
```

< 287

然后定义一个新的记录类型 Button，让它来实现 Measurable 协议。同时还定义一个名为 bordered 的方法：

```
(defrecord Button [text])

(extend-type Button
  Measurable
  (width [btn]
    (* 8 (-> btn :text count)))
  (height [btn] 8))

(def bordered
```

注 20：这个例子很大一部分灵感来自于维基百科上面关于 mixin 的例子：<https://en.wikipedia.org/wiki/Mixin>。

```
{:width #(* 2 (:border-width %))
  :height #(* 2 (:border-height %)))}
```

现在可以定义一个 `BorderedButton` 类了，它的实现就是把 `Button` 的实现和 `bordered` 的实现组合起来。但是这里有一个问题：怎么获得 `Button` 类的实现 map 呢？协议不只在作为静态名字跟 `extend` 一起使用时很有用，它内部定义了一些 var，这些 var 可以给我们一些提示：

示例6-7：一个协议的var map的内容

```
Measurable
:= {:impls
:=  {:user.Button
:=    {:height #<user$eval2056$fn_2057 user$eval2056$fn_2057@112f8578>,
:=      :width #<user$eval2056$fn_2059 user$eval2056$fn_2059@74b90ff7>},
:=  :on-interface user.Measurable,
:=  :on user.Measurable,
:=  :doc "A protocol for retrieving the 2D dimensions of widgets.",
:=  :sigs
:=    {:height
:=      {:doc "Returns the height in px.",
:=        :arglists ([measurable]),
:=        :name height},
:=    :width
:=      {:doc "Returns the width in px.",
:=        :arglists ([measurable]),
:=        :name width}},
:=  :var #'user/Measurable,
:=  :method-map {:width :width, :height :height},
:=  :method-builders
:=    #'user/height #<user$eval2012$fn_2013 user$eval2012$fn_2013@27aa7aac>,
:=    #'user/width #<user$eval2012$fn_2024 user$eval2012$fn_2024@4848268a>}}
```

288 这里有很多有趣的东西——如果不是全部都有趣的话——是协议的实现细节。^{注21} 跟我们的 mixin 例子有关的是，我们可以看出来，`(get-in Measurable [:impls Button])` 可以获得 `Button` 类对于 `Measurable` 协议的实现 map：

```
(get-in Measurable [:impls Button])
:= {:height #<user$eval1251$fn_1252 user$eval1251$fn_1252@744589eb>,
:=  :width #<user$eval1251$fn_1254 user$eval1251$fn_1254@40735f45>}
```

现在还缺的就是把这个实现 map 跟 `bordered` 函数组合起来的函数。这个函数接受两个函数作为参数，然后返回一个以某种规则把两个函数合并起来的新函数：

```
(defn combine
  "Takes two functions f and g and returns a fn that takes a variable number
```

注21：289页“协议自省”一节介绍了一些函数可以对协议进行自省。

```
of args, applies them to f and g and then returns the result of
(op rf rg) where rf and rg are the results of the calls to f and g."
[op f g]
(fn [& args]
  (op (apply f args) (apply g args))))
```

最后，可以定义新 `BorderedButton` 类了，并且让它实现 `Measurable`，使用 `+` 来把 `bordered` 函数的实现以及来自 `Measurable` 中 `Button` 类的实现合并起来：

```
(defrecord BorderedButton [text border-width border-height])

(extend BorderedButton
  Measurable
  (merge-with (partial combine +)
    (get-in Measurable [:impls Button])
    bordered))
```

现在来测试一下 `BorderedButton` 示例计算的长宽是否正确：

```
(let [btn (Button. "Hello World")]
  [(width btn) (height btn)])
:= [88 8]

(let [bbtn (BorderedButton. "Hello World" 6 4)]
  [(width bbtn) (height bbtn)])
:= [100 16]
```



另一个关于不要对协议过早进行内联实现的原因是，上面提到的这种对方法实现进行重用的方法是不支持内联实现的。所以如果你内联实现了一个协议，你还想对实现进行重用的话，那么只能利用代理或者宏了。

协议自省

◀ 289

既然在示例 6-7 中看过协议的底层细节，这里介绍一些方便的函数用来对协议进行自省：`extenders`、`extends?` 以及 `satisfies?`，这些 API 的作用就是回答协议与类型之间的关系。

extenders

返回实现了某个协议的所有类。比如在看过那个 `Measurable` 协议之后，可以看看有哪些类型实现了它：

```
(extenders Measurable)
:= (user.BorderedButton user.Button)
```

这个就相当于问“哪些类实现了 Java 接口 `X`？”注意，由于一个类型可以在运行期的任何时候对一个协议进行扩展，所以这个函数调用返回的结果是调用 `extenders` 时

所有实现了这个协议的类型。

extends?

如果一个类型扩展了一个协议，那么返回 true：

```
(extends? Measurable Button)
:= true
```

satisfies?

它是 Clojure 世界的 instance?，它测试某个特定的类型实例是否扩展了某个协议，要么是这个实例的类通过 extend 扩展了这个协议，或者通过 extend 引申出来的其他宏扩展了这个协议：

```
(satisfies? Measurable (Button. "hello"))
:= true
(satisfies? Measurable :other-value)
:= false
```

或者这个实例是内联实现了这个协议：

```
(deftype Foo [x y]
  Measurable
  (width [_] x)
  (height [_] y))
:= user.Foo
(satisfies? Measurable (Foo. 5 5))
:= true
```

在后一种场景下，因为提供了内联实现的类型实际上就是实现了协议所对应的 Java 接口，因此这时候用 satisfies? 和 instance? 效果是一样的：

```
(instance? user.Measurable (Foo. 5 5)) ①
:= true
```

290 ➤ ① user.Measurable 是由 Measurable 协议所生成的接口。

协议函数分派的边界场景

因为协议的方法产生的是命名空间限定的函数，因此两个协议的函数永远都不会相互冲突。但是也有一些边界场景——在典型的面向对象的语言里面没有的场景——会产生一些让人吃惊的结果。

互相竞争的实现。协议这种可以在运行期的任何时候进行扩展的特性对于交互式开发十分有用，但是也产生了一些奇怪的建模或者优化问题。^{注22} 但是，如果一个类型对于一

注 22： 这里 <http://dosync.posterous.com/51626638> 有关于后一种场景的一个例子。

个协议有两个不同的实现，那么后一个实现会被加载进内存覆盖掉前一个实现，这个可能会与你的期望不符。

哎，这其实应该是开发策略的问题，而不是技术本身的问题。避免这种冲突的原则是：如果你不是这个协议的所有者，也不是这个类型的所有者，那么准备好删除你的实现吧。对于这种潜在冲突的解决办法是：如果协议是先定义的，那么就应该由类型的所有者来对协议进行扩展，反之亦然。

不要扩展协议到类继承链上的两个类。而另一个相关联的场景是对于一个协议存在两个实现，而且这两个实现关联到两个相关联的类。比如我们扩展一个协议到两个相关联的接口 `java.util.List` 和 `java.util.Collection`，然后来调用这个协议方法的时候使用一个对象，而这个对象同时又实现了这两个接口：

```
(defprotocol P
  (a [x]))
:= P
(extend-protocol P
  java.util.Collection
  (a [x] :collection!)
  java.util.List
  (a [x] :list!))
:= nil
(a [])
:= :list!
```

在这种情况下，协议的分派机制使用类继承层次来做决定，它始终选择继承层级比较低的那个实现，在这里就是 `java.util.List`，因为它的层级比 `Collection` 要低，因为前者是后者的子接口。

无法分派的情况就随机分派。但是，扩展一个协议的两种类型本身没有任何继承关系，◀ 291 那怎么分派？答案是协议分派机制会随机选择一个实现，并缓存在内存中。最可能出现这种场景的例子是把一个协议扩展到两个层级很高的 Java 接口。

比如把一个协议扩展到两个完全没有关系的高层级的接口：`java.util.Map` 和 `java.io.Serializable`：

```
(defprotocol P
  (a [x]))

(extend-protocol P
  java.util.Map
  (a [x] :map!)
  java.io.Serializable
  (a [x] :serializable!))
```

现在我们以一个 Clojure map 的对象来调用协议的方法，那么到底使用的是哪个实现？

```
(a {})
:= :serializable!
```

好吧，这个结果看起来还算有道理，但是为什么选择的是 `Serializable` 而不是 `Map` 实现呢？这里的陷阱是，如果重启你的程序或者 RPEL 再来调用相同的函数，结果可能是不同的！对于上面的例子意味着在运行代码之前不知道 `(a {})` 返回的到底是 `:serializable!` 还是 `:map!`

有很多办法来解决这个问题：

1. 把协议扩展到你要支持的具体的类，而不是上面例子里的那种层级很高的接口。对于具体的类的分派从来都是明确的。^{注23}
2. 而要把一个协议扩展到两个完全没有关系的高层接口说明设计是有问题的——你的协议的范围太广。重新设计你的协议。
3. 使用多重方法。跟协议相比，对于这种没有办法分派的场景，多重方法会直接报错。而且多重方法还提供了一种“偏好”机制使我们可以指定在自动分派不能分派的情况下，优先分配给谁。^{注24}

自己实现一个 set

在 279 页“薛定谔的猫”一节，我们小窥了一下如何定义一个自己的类型来实现 Clojure 的核心接口——那个例子里面我们实现的是 `derefereceable` 接口。现在让我们更进一步来看看如何完整实现一个数据结构，使得它可以完全融入 Clojure 的其他抽象：一个用数组实现的 `set`，在保存的数据数量比较少的时候，它将比标准的 `set` 有更好的性能，使用更少的内存。^{注25}

要参与到 Clojure 的这种核心抽象里面去意味着我们要去实现某个 Java 接口，而那个 Java 接口则定义了这种抽象。所以，我们所有的实现功能都可以以内联的方式完成。^{注26}

一般来说，参与到 Clojure 的抽象里面最难的部分是搞清楚到底要实现哪个接口的哪个方法，因为这些信息在官方的文档里面是没有描述的。^{注27} 一个工具函数可以帮助我们：

注23： 所以对于普通类或者记录类的协议的实现，从来不会有无法分派的场景。

注24： 多重方法将在第 7 章详细介绍，而它所提供的“偏好”机制将在 313 页“多重继承”一节进行介绍。

注25： 对于大多数 Clojure 集合所采用的基于树的实现，我们在 125 页“持久性图示：maps（以及 vectors 和 sets）”有详细介绍，它们对于普通用途来非常合适，但是对于那种特殊场景，我们可以实现自己的数据结构。

注26： 从长远来看，以接口来表示抽象会被协议所代替；在 ClojureScript 里面已经是这么做的了，在 584 页“ClojureScript”一节有介绍。

注27： Clojure Atlas 是一个用来查看 Clojure 抽象背后实现了哪些接口的一个工具：<http://www.clojureatlas.com>。

```
(defn scaffold
  "Given an interface, returns a 'hollow' body suitable for use with `deftype`."
  [interface]
  (doseq [[iface methods] (-> interface
                                .getMethods
                                (map #(vector (.getName (.getDeclaringClass %))
                                              (symbol (.getName %))
                                              (count (.getParameterTypes %))))
                                (group-by first))]
         (println (str "  " iface)))
    (doseq [[_ name argcount] methods]
      (println
        (str "    "
              (list name (into '[this] (take argcount (repeatedly gensym))))))))))
```

通过查看 (ancestors (class #{})) 的输出，我们知道 clojure.lang.IPersistentSet 是标准 set 实现的主要的接口，把这个接口传给 scaffold 函数，它输出了一个需要实现的接口及方法的列表，这个可以当做我们工作的起点：

```
(scaffold clojure.lang.IPersistentSet)
; clojure.lang.IPersistentSet
;  (get [this G_5617])
;  (contains [this G_5618])
;  (disjoin [this G_5619])
; clojure.lang.IPersistentCollection
;  (count [this])
;  (cons [this G_5620])
;  (empty [this])
;  (equiv [this G_5621])
; clojure.lang.Seqable          ❶
;  (seq [this])
; clojure.lang.Counted
;  (count [this])               ❷
```

293

❶ 如果只是想让你的数据结构 seqable，那么你只要实现 clojure.lang.Seqable 接口的 seq 方法就可以了。

❷ 注意，这里有两个签名一样的 count 方法，它们中的一个会被“遮住”。

要实现一个 Clojure 的 set，我们要实现上面那些方法以及 Object 类的 hashCode 方法和 equals 方法，以保证 set 的正确的值语义。除了 cons 和 equiv，其他方法如何实现应该都是显而易见的。cons，不管它的名字，它是 conj 函数调用的方法；equiv 跟 equals 类似，但是它确保数字之间的值语义（435 页“等值让你不致发疯”一节有介绍）。这里不需要考虑这个，因为我们的 set 不是数字类型。

示例6-8：用vector实现的set

```
(declare empty-array-set)
(def ^:private ^:const max-size 4)

(deftype ArraySet [^objects items
                    ^int size
                    ^:unsynchronized-mutable ^int hashcode]
  clojure.lang.IPersistentSet
  (get [this x]
    (loop [i 0]
      (when (< i size)
        (if (= x (aget items i))
            (aget items i)
            (recur (inc i))))))
  (contains [this x]
    (boolean
      (loop [i 0]
        (when (< i size)
          (or (= x (aget items i)) (recur (inc i)))))))
  [294] (disjoin [this x]
    (loop [i 0]
      (if (== i size)
          this
          (if (not= x (aget items i))
              (recur (inc i))
              (ArraySet. (doto (aclone items)
                            (aset i (aget items (dec size)))
                            (aset (dec size) nil))
                          (dec size)
                          -1))))))
  clojure.lang.IPersistentCollection
  (count [this] size)
  (cons [this x]
    (cond
      (.contains this x) this
      (== size max-size) (into #{x} this)
      :else (ArraySet. (doto (aclone items)
                           (aset size x))
                         (inc size)
                         -1)))
  (empty [this] empty-array-set)
  (equiv [this that] (.equals this that))
  clojure.lang.Seqable
  (seq [this] (take size items))
  Object
  (hashCode [this]
    (when (== -1 hashCode)
      (set! hashCode (int (areduce items idx ret 0
                                    (unchecked-add-int ret (hash (aget items idx))))))))
```

```

(hashcode)
>equals [this that]
(or
  (=identical? this that)
  (and (or (instance? java.util.Set that)
            (instance? clojure.lang.IPersistentSet that))
       (= (count this) (count that))
       (every? #(contains? this %) that)))))

(def ^:private empty-array-set (ArraySet. (object-array max-size) 0 -1))

(defn array-set
  "Creates an array-backed set containing the given values."
  [& vals]
  (into empty-array-set vals))

```

- ①② set 实现的一个关键点是，当 set 里面所保持的元素个数已经不在这种实现的“最佳范围”，那么这个实现应该自动升级成另外一种适合这种数据规模的实现——而不用显式地改变上层抽象。而在这里对于我们这个专为小数据量而优化的 set 实现，会在当数据量多于 4 个的时候自动升级为一个普通的 Clojure HashSet。对于一个具体的使用模式可以设置不同的阈值。

<295>

- ③ equiv 的实现直接代理给 equals 方法，以使得 ArraySet 的行为跟 clojure.lang.IAPersistentSet 的行为一致。
- ④ 因为我们是用一个数组来实现的，因此需要用这些操作——areduce、aget 以及 aset 等，这些我们前面都没有介绍过，但是它们跟现在介绍的话题关系不是太大。442 页“合理使用原始类型的数组”一节详细介绍了数组操作。
- ⑤ ArraySet 的构造函数是不适合最终用户直接使用的，它的参数只跟它的实现相关，最终用户不应该关心这个。基于此，就像我们在 275 页“构造函数和工厂函数”一节介绍的那样，我们提供了对用户友好的 array-set 工厂函数。

我们的这些实现对不对呢？

```

(array-set)
:= {}
(conj (array-set) 1)
:= {1}
(apply array-set "hello")
:= {"h" "e" "l" "o"}
(get (apply array-set "hello") \w)
:= nil
(get (apply array-set "hello") \h)
:= \h

```

```
(contains? (apply array-set "hello") \h)
:= true
(= (array-set) #{})
:= true
```

到目前为止还挺好的，但是……

```
((apply array-set "hello") \h)
; #<ClassCastException java.lang.ClassCastException:
;   user.ArraySet cannot be cast to clojure.lang.IFn>
```

抛出这个异常我们不会太惊讶，因为到目前为止我们并没有实现任何东西以使得我们的 `ArraySet` 可以当做一个函数来调用。为了实现这个，要实现 `clojure.lang.IFn` 接口的一个合适的方法，这个接口是所有 Clojure 函数都实现的。我们完全可以不实现这个接口——一个 set 不是必须是可以调用的才有用，但是如果它是可以调用的，写代码的时候会比较方便，因此我们来把这个实现加上吧。^{注 28}

296 > 更严重的问题是：

```
(= #{}) (array-set))
:= false
```

我们打破了 = 的对称性，这是程序设计里面必须遵守的一个原则。这个问题的根源在于 Clojure 的 set 同时也是 Java 的 `java.util.Set`。^{注 29}

让我们再次使用 `scaffold` 函数来看看我们要实现 `java.util.Set` 的那些方法：

```
(scaffold java.util.Set)
; java.util.Set
;   (add [this G_6140])
;   (equals [this G_6141])
;   (hashCode [this])
;   (clear [this])
;   (isEmpty [this])
;   (contains [this G_6142])
;   (addAll [this G_6143])
;   (size [this])
;   (toArray [this G_6144])
;   (toArray [this])
;   (iterator [this])
;   (remove [this G_6145])
;   (removeAll [this G_6146])
;   (containsAll [this G_6147])
```

注 28：类似的，`ArraySet` 还不支持元数据，它是由 `clojure.lang.IObj` 所定义的。它定义了两个很容易实现的方法。这个就留给读者来实现了。

注 29：这个对于实现 Clojure 的 map 来说也是一样的，我们在实现的时候也要同时实现 Java 的 `java.util.Map` 接口。

```
;     (retainAll [this G_6148])
```

不要慌：我们只需要那些读操作，因为 `ArraySet` 是只读的并且是持久化的。`equals` 和 `hashCode` 以及 `contains` 已经实现了，那么我们只需要实现：

```
java.util.Set
  (isEmpty [this])
  (size [this])
  (toArray [this G_6144])
  (toArray [this])
  (iterator [this])
  (containsAll [this G_6147])
```

这些方法都不难实现。唯一可能的陷阱是，从 `toArray` 返回的 `items` 数组会把我们内部存储的数据结构暴露出来，从而破坏了不可变性的保证。而如果以幼稚的方法实现 `iterator` 方法，那么实现可能会是冗长而无聊的，而如果你记得序列也是 Java 集合的话，我们可以重用序列的 `Iterator`：

示例6-9：改进版的使用数组实现的set

297

```
(deftype ArraySet [^objects items
                     ^int size
                     ^:unsynchronized-mutable ^int hashCode]
  clojure.lang.IPersistentSet
  (get [this x]
    (loop [i 0]
      (when (< i size)
        (if (= x (aget items i))
          (aget items i)
          (recur (inc i))))))
  (contains [this x]
    (boolean
      (loop [i 0]
        (when (< i size)
          (or (= x (aget items i)) (recur (inc i)))))))
  (disjoin [this x]
    (loop [i 0]
      (if (= i size)
        this
        (if (not= x (aget items i))
          (recur (inc i))
          (ArraySet. (doto (aclone items)
                      (aset i (aget items (dec size)))
                      (aset (dec size) nil))
                     (dec size)
                     -1))))))
  clojure.lang.IPersistentCollection
  (count [this] size)
  (cons [this x]
```

```

(cond
  (.contains this x) this
  (= size max-size) (into #{x} this)
  :else (ArraySet. (doto (aclone items)
                           (aset size x))
                     (inc size)
                     -1)))
(empty [this] empty-array-set)
(equiv [this that] (.equals this that))
clojure.lang.Seqable
(seq [this] (take size items))
Object
(hashCode [this]
  (when (== -1 hashCode)
    (set! hashCode (int (areduce items idx ret 0
                                  (unchecked-add-int ret (hash (aget items idx)))))))
  hashCode)
>equals [this that]
  (or
    (identical? this that)
    (and (instance? java.util.Set that)
         (= (count this) (count that))
         (every? #(contains? this %) that))))
clojure.lang.IFn
(invocation [this key] (.get this key))
[298] applyTo [this args]
  (when (not= 1 (count args))
    (throw (clojure.lang.ArityException. (count args) "ArraySet")))
  (this (first args)))
java.util.Set
(isEmpty [this] (zero? size))
(size [this] size)
(toArray [this array]
  (.toArray ^java.util.Collection (sequence items) array))
(toArray [this] (into-array (seq this)))
(iterator [this] (.iterator ^java.util.Collection (sequence this)))
containsAll [this coll]
  (every? #(contains? this %) coll))

(def ^:private empty-array-set (ArraySet. (object-array max-size) 0 -1))

```

① equals 现在可以简化为测试 that 是否实现 java.util.Set 接口，因为现在这个接口是由 ArraySet 实现的了。

② 对于直接调用 clojure.lang.IFn 包含了 21 个参数列表的 invoke 方法组成。因为 set 只接受一个参数（也就是我们用来查找的 key，比如通过 get 来查找），所以我们只实现那个参数列表；如果以多个参数或者 0 个参数来调用 ArraySet 的话，那么会抛出一个错误。

- ① 对于通过 `apply` 来产生的调用，`IFn` 定义了 `applyTo` 函数，它的参数是 `apply` 参数的序列。
- ② 这里最好使用 `sequence` 而不是 `seq`，因为它从来不会返回 `nil`；而 `seq` 会返回 `nil`，从而对于空 `set` 会导致 `NullPointerException` 异常。
- ③ 因为序列本身是 Java 的集合，可以直接返回序列上的那个 `Iterator`。

现在我们有了一个可以完全工作的 `set` 实现，它可以跟其他 `set` 交互合作，同时也可以像普通 `set` 一样当做函数来调用：

```
(= #{3 1 2 0} (array-set 0 1 2 3))
;= true
((apply array-set "hello") \h)
;= \h
```

但是，`ArraySet` 对于我们真的有实在的意义吗？让我们把它跟普通的 Clojure `hash-set` 做个对比，使用各种查找函数 `disj` 以及 `conj` 操作：

```
(defn microbenchmark
  [f & {:keys [size trials] :or {size 4 trials 1e6}}]
  (let [items (repeatedly size gensym)]
    (time (loop [s (apply f items)
                n trials]
            (when (pos? n)
              (doseq [x items] (contains? s x))
              (let [x (rand-nth items)]
                (recur (-> s (disj x) (conj x)) (dec n)))))))

(doseq [n (range 1 5)
        f #'array-set #'hash-set])
  (print n (-> f meta :name) ": ")
  (microbenchmark @f :size n))
; size 1 array-set : "Elapsed time: 839.336 msecs"
; size 1 hash-set : "Elapsed time: 1105.059 msecs"
; size 2 array-set : "Elapsed time: 1201.81 msecs"
; size 2 hash-set : "Elapsed time: 1369.192 msecs"
; size 3 array-set : "Elapsed time: 1658.36 msecs"
; size 3 hash-set : "Elapsed time: 1740.955 msecs"
; size 4 array-set : "Elapsed time: 2197.424 msecs"
; size 4 hash-set : "Elapsed time: 2154.637 msecs"
```

299

从结果来看，当保存的元素个数很少的时候，`array-set` 往往会比 `hash-set` 有更好的性能，而且因为它使用一个简单的 Java 数组来做存储，它使用的内存也比普通 `set` 使用的那种基于“树”的实现使用得要少。

总结

普通类型、记录类型以及协议一起构建起了一个强有力的框架，这个框架主要关注数据并且避免了不必要的复杂性。这个以数据为中心进行建模的办法使得我们可以更加专注建模，使用整个语言提供的工具来处理模型，比如可以把作用于 map 的函数用在记录类型上，使我们不需要选择复杂类层次这种不必要的复杂性。

多重方法

我们前面已经谈到协议：协议引入了一种常见却有限的形式进行多态转发——即基于类型的单转发。在本章里，我们将探索“多重方法”，扩大转发灵活度，不仅提供多重转发，而且可以基于参数类型之外的其他内容进行转发。即对于一个多重方法，可以根据参数的任何属性选择调用的具体实现，没有哪一个属性有特权。此外，多重方法支持任意层级，支持解歧多重继承。

 在 Java 里，一个方法名可以有参数数量相同的多个签名，差别只是参数的类型不同，这种情况称为“重载”。不过这不算是多重转发，正确的签名是在编译时基于方法的参数类型选择的。唯一的“动态”转发是基于特权参数 `this` 的类型进行的转发。

多重方法基础

一个多重方法是用 `defmulti` 形式创建的，而多重方法的实现是由 `defmethod` 形式提供的。可帮助记忆的是这两个形式的顺序正是 `multimethod` 这个单词里的顺序：首先定义“多重”转发然后定义“方法”（多重方法调用就是转发给这些方法的）。

我们来看一个例子：一个填充 XML/HTML 节点、行为依赖于标签名的函数，使用 `closure.xml` 命名空间定义的 XML 表示。一个 XML 元素是包含三个键的映射：`:tag` 是元素的名称（作为关键字），`:attrs` 是属性名称（作为关键字）到值（作为字符串）的映射，而 `:content` 是子节点和内容的一个集合类。

```
(defmulti fill
  "Fill a xml/html node (as per closure.xml)
  with the provided value."
  (fn [node value] (:tag node)))
```



```
(defmethod fill :div
  [node value]
  (assoc node :content [(str value)]))

(defmethod fill :input
  [node value]
  (assoc-in node [:attrs :value] (str value)))
```

❶ 这是转发函数。传递给多重方法的参数被传递给这个函数，产生一个转发值，这个值用于为这些参数选择调用哪种方法。

❷ 这里的 :div 是一个转发值。当转发函数的返回值与这个转发值匹配时，所提供的方法实现（只不过是另一个函数而已）被选中并被调用。

一个多重方法的操作一点儿也不复杂：

1. 接受参数。
2. 用所给的参数调用转发函数，计算转发值。
3. 选择定义为支持这个转发值的实现方法。
4. 用最初的参数调用这个实现方法。

就这么多了。利用原子和宏，^{注1} 你可以用几行 Clojure 代码或者一定量的 Ruby 或 Python 代码^{注2} 轻易实现这样一个系统。

多重方法看起来确实与其他函数定义有点不一样，差别在于：

- 一个多重方法——尽管是定义一个函数——并不显式指定它的参数个数。它支持它的转发函数支持的所有参数个数。
- defmulti 形式真正定义了一个新 var，在上例中是 fill。每个 defmethod 形式只是在“根”多重方法上注册一个新的实现方法；与直觉相反的是，defmethod 并不定义或重新定义任何 var。

目前还没有使用过我们的代码，先来看看一切是否符合预期：

```
(fill {:tag :div} "hello")
:= {:content ["hello"], :tag :div}
(fill {:tag :input} "hello")
:= {:attrs {:value "hello"}, :tag :input}
(fill {:span :input} "hello")
```

303 ➤

注 1：这个作为练习留给读者。

注 2：与 Clojure 的多重方法相似的多重转发系统在其他语言中也存在。Philip J. Eby 的用于 Python 的 PEAK-Rules (<http://pypi.python.org/pypi/PEAK-Rules>) 是特别成熟的一个。探索用 Python 实现 Clojure 风格的多重方法的描述可以在 <http://codeblog.dhananjaynene.com/2010/08/clojure-style-multi-methods-in-python/> 找到。

```
;= #<IllegalArgumentException java.lang.IllegalArgumentException:  
;=   No method in multimethod 'fill' for dispatch value: null>
```

这是目前的方法的一个缺点：因为我们没有一个基础情形，`fill` 只能工作于它所知道的元素，即那些通过 `defmethod` 提供了实现的情形。

幸亏有一个特别的转发值：`:default`。

```
(defmethod fill :default  
  [node value]  
  (assoc node :content [(str value)]))  
  
(fill {:span :input} "hello")  
;= {:content ["hello"], :span :input}  
(fill {:span :input} "hello")  
;= {:content ["hello"], :span :input}
```

这行得通，而且我们甚至可以去掉 `:div` 的实现，因为它可以由基础情形覆盖！

不过，我们的正常转发值已经是关键字了，这意味着如果想要用特别的方式扩展 `fill` 来覆盖 `<default>` 标签^{注3}，我们可能会遇到麻烦。

令人高兴的是，`defmulti` 接受选项，有一种方式可以指定默认转发值应是什么样的：

```
(defmulti fill  
  "Fill a xml/html node (as per clojure.xml)  
  with the provided value."  
  (fn [node value] (:tag node))  
  :default nil) ●  
  
(defmethod fill nil ●  
  [node value]  
  (assoc node :content [(str value)]))  
  
(defmethod fill :input  
  [node value]  
  (assoc-in node [:attrs :value] (str value)))  
  
(defmethod fill :default ●  
  [node value]  
  (assoc-in node [:attrs :name] (str value)))
```

● `defmulti` 的选项是关键字 / 值对，这里常把这个默认方法值设置为 `nil`。

● 这是对应的默认实现。

● 这不再是默认实现，而是对 `<default>` 元素的实现。

注3： 让我们假设处理的是某种特别的 XML 格式或有人在 HTML 5 里引进了这个标签。

我们来完善一下 `fill`, 对于不同的 `<input>` 标签施用不同的行为。例如当 `value` 属性与传给 `fill` 的 `value` 参数匹配时我们想选中单选按钮或多选框。

当前的转发函数没有提供有关属性的信息，因而要根据 `type` 属性来转发，我们需要修改转发函数：

```
(ns-unmap *ns* 'fill)

(defn- fill-dispatch [node value]
  (if (= :input (:tag node))
      [(:tag node) (-> node :attrs :type)]
      (:tag node)))

(defmulti fill
  "Fill a xml/html node (as per clojure.xml)
   with the provided value."
  #!fill-dispatch
  :default nil)

(defmethod fill nil
  [node value]
  (assoc node :content [(str value)]))

(defmethod fill [:input nil]
  [node value]
  (assoc-in node [:attrs :value] (str value)))

(defmethod fill [:input "hidden"]
  [node value]
  (assoc-in node [:attrs :value] (str value)))

(defmethod fill [:input "text"]
  [node value]
  (assoc-in node [:attrs :value] (str value)))

(defmethod fill [:input "radio"]
  [node value]
  (if (= value (-> node :attrs :value))
      (assoc-in node [:attrs :checked] "checked")
      (update-in node [:attrs] dissoc :checked)))

(defmethod fill [:input "checkbox"]
  [node value]
  (if (= value (-> node :attrs :value))
      (assoc-in node [:attrs :checked] "checked")
      (update-in node [:attrs] dissoc :checked)))
```

```
(defmethod fill :default
  [node value]
  (assoc-in node [:attrs :name] (str value)))
```

- ❶ 使用 #'fill-dispatch 而不是简单的 fill-dispatch 增加了一层重定向，这让我们可以修改转发函数而不需碰 ns-unmap 或损失已经定义的任何方法。如果使用 fill-dispatch，则在 defmulti 被求值时捕获了转发函数的值，转发函数随后就不会被更新。这个技巧在 REPL 里演化代码时非常有用。

305

重定义一个多重方法不会更新这个多重方法的转发函数

注意在上面的例子里，对命名空间用了 ns-unmap fill，因而我们可以重新定义它。在重新定义函数时一般是不需要这样做的，defmulti 有 defonce 的语义，所以如果不先撤销对多重方法根 var 的映射，转发函数就不能改变。这意味着你不得不从当前的命名空间撤销对它的映射再重新定义它，否则你的修改会被悄悄地忽略掉！

这次迭代有意思的一点是，转发值现在可能是 nil、关键字或关键字和字符串对。转发值不限于关键字。尽管如此，很快我们就会看到，在多重方法中关键字仍然发挥着特别的作用。

让我们先来测试一下新代码：

```
(fill {:tag :input
        :attrs {:value "first choice"
                :type "checkbox"}}
      "first choice")          ❶
:= {:tag :input,
:=  :attrs {:checked "checked",
:=      :type "checkbox",
:=      :value "first choice"}}
(fill *1 "off")           ❷
:= {:tag :input
:=  :attrs {:type "checkbox",
:=      :value "first choice"}}
```

- ❶ 用多选框元素的映射和多选框的 :value 调用 fill，将会正确地产生一个选中的多选框。
❷ 而用那个多选框和任何其他值调用 fill，将产生一个没被选中的多选框。^{注4}

这些结果看来不错，不过最后的迭代里有相当多的重复代码：多选框和单选按钮应该共享同一个实现，显式和隐式 (nil) 文本字段也是如此。而且 text 行为应该是未知类型的默认情形。

注 4： *1 持有在 REPL 里求值的最后一个表达式的值。详情参见 399 页“REPL 绑定的 var”一小节。

- 306> 简而言之，我们想指定多选框和单选按钮是可勾选的输入而未知输入类型应该看做文本输入。这可以通过定义一个层级来实现，层级让我们可以表达转发值间的关系，多重方法可以用这个转发值来细化如何选择具体的方法实现。

层级

Clojure 的多重方法让我们定义层级来支持应用领域所需的任何关系，包括多重继承。这些层级是用具名对象^{注5}（关键字或符号^{注6}）和类之间的关系定义的。

“层级”用复数，这不是巧合：你可以有多个层级。有一个全局（默认的）层级，还有在需要时通过 `make-hierarchy` 创建的层级。此外，层级和多重方法不受限于单个命名空间：可以从任何命名空间——不是非得在它们被定义时的命名空间里——扩展一个层级（通过 `derive`）或多重方法（通过 `defmethod`）。

全局层级是共享的，访问它更受保护。即不带命名空间的关键字（或符号）不能用于全局层级。这有助于防止两个函数库由于无意中选择使用了同一个关键字来表示不同的语义而互相干扰。

用 `derive` 来定义一个层级关系：

```
(derive ::checkbox ::checkable)      ❶
:= nil
(derive ::radio ::checkable)
:= nil
(derive ::checkable ::input)
:= nil
(derive ::text ::input)
:= nil
```

❶ 回忆一下，`::keyword` 是 `:current.namespace/keyword` 的简写；因而这里的 `::checkbox` 等值于 `:user/checkbox`。还要记得，`::keyword` 对 `:keyword` 来说就像 `'symbol` 对 `'symbol` 一样。详情参见 14 页“关键字”一节。

我们刚描述了不同“类”之间存在的关系：多选框和单选按钮是“可勾选的”，所有“可勾选的”和“文本”元素都是输入元素。你可以用 `isa?` 来测试层级里的这些关系：

```
(isa? ::radio ::input)      ❶
:= true
(isa? ::radio ::text)
:= false
```

注 5： 具名对象是你可以对它调用 `name` 和 `namespace` 的对象，实现了 `clojure.lang.Named` 接口。
注 6： 不过关键字一般更受偏爱。

- ❶ 派生是传递性的：`::radio` 是一个 `::checkable`，后者是一个 `::input`。



isa? 很少在 REPL 之外使用。如果发现自己使用它的次数较多，这意味着你应该提取一个多重方法出来。在这方面，它与 instance? 非常类似：一般来说，它的存在提示你应该提取一个转发设施（Java 接口、协议或多重方法）。

307

还有一些其他内省的函数：`underive`、`ancestors`、`parents` 和 `descendants`，在 REPL 下工作或利用它们来完成某些元编程的特技时这些是非常有用的。

类和接口也可以在层级里使用，不过只能作为派生的子节点，永远不能作为父节点，^{注7} 换言之，在 Clojure 环境的 classpath 隐式定义的类层级之外，^{注8} 类和接口只能成为一个层级的叶子节点。

```
(isa? java.util.ArrayList Object)
;= true
(isa? java.util.ArrayList java.util.List)
;= true
(isa? java.util.ArrayList java.util.Map)      ❶
;= false
(derive java.util.Map ::collection)           ❷
;= nil
(derive java.util.Collection ::collection)
;= nil
(isa? java.util.ArrayList ::collection)       ❸
;= true
(isa? java.util.HashMap ::collection)
;= true
```

- ❶ 在 Java 集合类框架里，Map 和 Collection 完全是分离的。只依赖这些静态类型的转发机制没有能用单个方法来同时处理 Map 和 Collection 的（转而依靠 Object 之类的不算）。
- ❷ 可以声明 Map 和 Collection 是从全局层级里的一个新标识符派生的，称为 `::collection`。
- ❸ 现在可以用 `::collection` 作为 defmethod 里的目标转发值，因为它现在匹配实现 Map 或 Collection 的任何类。

不过层级有关类和接口的方面与我们的例子无关。我们将在 313 页“多重继承”一节再回过头来谈在层级中使用类和接口的话题。

注7：从一个类或接口派生的唯一办法是用互操作或 deftype、defrecord、reify 创建一个类型。

注8：关于 classpath 的信息请参见 331 页“classpath 入门”一节。



整个 Java 类层级总是每个层级的一部分，即使这个层级是刚用 `make-hierarchy` 创建的。

```
(def h (make-hierarchy))
:= #'user/h
(isa? h java.util.ArrayList java.util.Collection)
:= true
```

因而 `isa?` 是 `instance?` 的超集，它可用于测试一个类是否是从另一个类派生的，或一个类是否实现一个接口。

独立层级

目前，`fill-dispatch` 返回 `nil`、关键字或向量。其中有两种不能参与层级（`nil` 和向量），而第三种不能参与全局层级，因为这些关键字没有带命名空间。

所以我们的选择只剩下让 `fill-dispatch` 返回带命名空间的关键字或使用一个私有层级。

`derive` 隐式地修改了全局层级，但如果使用一个自定义的层级，就得自己管理修改了。这其实很简单，就是把层级放在一个引用类型里，如 `ref`、原子或 `var`。使用 `var` 是一个安全的选择：^{注9} 全局层级用的就是 `var`。

例7-1：用自定义层级实现fill

```
(ns-unmap *ns* 'fill)

(def fill-hierarchy (-> (make-hierarchy) ❶
                           (derive :input.radio ::checkable)
                           (derive :input.checkbox ::checkable)
                           (derive ::checkable :input)
                           (derive :input.text :input)
                           (derive :input.hidden :input)))

(defn- fill-dispatch [node value]
  (if-let [type (and (= :input (:tag node))
                     (-> node :attrs :type))]
    (keyword (str "input." type))
    (:tag node)))

(defmulti fill
  "Fill a xml/html node (as per clojure.xml)
   with the provided value."
  #fill-dispatch
  :default nil
  :hierarchy #'fill-hierarchy) ❷
```

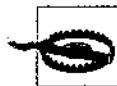
注9：这是安全的选择，因为层级不会经常变化，而你希望层级的变化对所有线程都是可见的。如果你有其他需求（例如事务式变化或动态作用域），请选择相关的引用类型。参见第4章。

```
(defmethod fill nil [node value]
  (assoc node :content [(str value)]))

(defmethod fill :input [node value]
  (assoc-in node [:attrs :value] (str value)))

(defmethod fill ::checkable [node value]
  (if (= value (-> node :attrs :value))
    (assoc-in node [:attrs :checked] "checked")
    (update-in node [:attrs] dissoc :checked)))
```

- ❶ `input.type` 和 `::checkable` 都是为了避免与合法的标签名称发生潜在冲突。关键字里的`.`没有特别的意义——它只是名称的一部分。
- ❷ 多重方法预期某种类型的引用类型作为 `:hierarchy` 选项的值，而不是一个层级值；这让你在必要时可以在运行时更新层级。在这种情形下，我们提供 `fill-hierarchy-var`，而不是它在求值 `defmulti` 形式时的值；可以用 `alter-var-root` 来动态修改层级，不需要重新定义多重方法的根。



所有层级相关的函数（如 `derive`、`isa?`、`parents` 等）要求层级值作为附加的第一个参数。`derive` 是这组函数里最特别的，因为如果没有显式的层级，这是一个产生副作用的函数——修改全局层级，而如果有显式的层级，它却是一个纯函数！

我们对 `fill` 代码的最新的迭代没有重复代码，因而这个目标是实现了。不过第二个目标却不太成功：未知的输入类型，如像文本输入，没有被处理。

```
(fill {:tag :input
        :attrs {:type "date"}}
      "20110820")
:= {:content ["20110820"], :attrs {:type "date"}, :tag :input}
```

这可不是我们想要的！我们还没有表达出未知输入类型应该视作文本。问题是未知类型的输入是一个开放的集合，所以无法预先创建所需的所有派生。而且，因为转发值的集合比只含输入标签的那些转发值的集合大，也不能用默认值来覆盖意料之外的输入类型。

我们能吗？如果动态地思考，而不是假设必须静态地提前定义这个层级，我们就会意识到层级不是静态的。这意味着我们的默认情形可以作为一个安全网，动态定义需要的派生以便对于给定的一个新的输入类型，默认情形将只会碰到一次。

例7-2：动态更新fill用到的层级

310

```
(defmethod fill nil [node value]
  (if (= :input (:tag node))
```

```

(do
  (alter-var-root #'fill-hierarchy
    derive (fill-dispatch node value) :input) ❶
    (fill node value)) ❷
  (assoc node :content [(str value)])))

```

- ❶ 我们动态改变 var 的值，指定未知 :input 节点的转发值应该从我们层级的 :input 派生出来。
- ❷ 做了这一修改之后，我们递归地调用 fill，它将转发到 :input 方法实现，因为我们已经用适合这一情形的新派生更新了这个层级。

这个技巧工作得很好：

```

(fill {:tag :input
        :attrs {:type "date"}}
      "20110820")
:= {:value "20110820", :type "date"}, :tag :input}

```

实现这一结果的一个不那么聪明的办法是引入一个 fill-input 多重方法，从 fill 的 :input 这种情形来调用它。

```

(ns-unmap *ns* 'fill)

(def input-hierarchy (-> (make-hierarchy)
                           (derive :input.radio ::checkbox)
                           (derive :input.checkbox ::checkbox))) ❶

(defn- fill-dispatch [node value]
  (:tag node))

(defmulti fill
  "Fill a xml/html node (as per clojure.xml)
   with the provided value."
  #'fill-dispatch
  :default nil) ❷

(defmulti fill-input
  "Fill an input field."
  (fn [node value] (-> node :attrs :type))
  :default nil
  :hierarchy #'input-hierarchy)

(defmethod fill nil [node value]
  (assoc node :content [(str value)]))

(defmethod fill :input [node value]
  (fill-input node value))

```

```

(defmethod fill-input nil [node value]
  (assoc-in node [:attrs :value] (str value)))

(defmethod fill-input ::checkable [node value]
  (if (= value (-> node :attrs :value))
    (assoc-in node [:attrs :checked] "checked")
    (update-in node [:attrs] dissoc :checked)))

```

◀ 311

- ❶ 从默认情形显式派生 :text 和 :hidden 不再是必要的了。而且那也完全是不可能的了，因为默认转发值是 nil，不能参与到层级中。
- ❷ fill 不再依赖一个自定义的层级——只依赖 fill-input。

真正实现多重！

目前我们的例子运用了基于非类型的转发，但还不是多重转发：第二个参数（value）在前面所有的转发函数里被有意忽略了。

这不是什么大不了的事情，因为多数多重转发以相同方式工作：转发函数计算出一个值，根据适用的层级与一个实现匹配。多重方法系统不知道我们的转发函数只是检查第一个参数。

不过转发值是向量的时候，isa? 按特殊情形对待转发值，一个元素一个元素地处理：^{注10}

```

(isa? fill-hierarchy [:input.checkbox :text] [::checkable :input])
:= true

```

正如我们已经指出，Java 的类层级是包含在所有的层级里的，这意味着如果需要的话，我们可以把某些类加入进来：

```

(isa? fill-hierarchy [:input.checkbox String] [::checkable CharSequence])
:= true

```

我们将利用这一特性使得 fill 更聪明，让它根据 value 参数的类型做出适当的反应。

第一步是修改 fill-dispatch，返回一个关键字和一个类的向量。^{注11}

```

(defn- fill-dispatch [node value]
  (if-let [type (and (= :input (:tag node))
                     (-> node :attrs :type))]
    [(keyword (str "input." type)) (class value)]
    [(:tag node) (class value)]))

```

注 10： 它递归地工作：你可以用向量的向量作为转发值！

注 11： 这个例子是基于例 7-1 的，可以像在例 7-2 里演示的那样使用动态层级修改。

312 现在我们想说，基础情形是把 value 转变成一个 String，不管它的类型是什么。不过，多选框接受集合为值：一个多选框被勾选当且仅当它的值属于这个集合。

```
(ns-unmap *ns* 'fill)

(def fill-hierarchy (-> (make-hierarchy)
                           (derive :input.radio ::checkbox)
                           (derive :input.checkbox ::checkbox)))

(defn- fill-dispatch [node value]
  (if-let [type (and (= :input (:tag node))
                     (-> node :attrs :type))]
    [(keyword (str "input." type)) (class value)]
    [(:tag node) (class value)]))

(defmulti fill
  "Fill a xml/html node (as per clojure.xml)
   with the provided value."
  #'fill-dispatch
  :default nil
  :hierarchy #'fill-hierarchy)

(defmethod fill nil
  [node value]
  (if (= :input (:tag node))
      (do
        (alter-var-root #'fill-hierarchy
          derive (first (fill-dispatch node value)) :input) ●
        (fill node value))
      (assoc node :content [(str value)])))

(defmethod fill
  [:input Object] [node value]
  (assoc-in node [:attrs :value] (str value)))

(defmethod fill [::checkboxable clojure.lang.IPersistentSet]
  [node value]
  (if (contains? value (-> node :attrs :value))
      (assoc-in node [:attrs :checked] "checked")
      (update-in node [:attrs] dissoc :checked)))
```

❶ 这里新出现的 first 是为了只取转发值的关键字部分。记住，在层级里你只能使用关键字、符号或类。

现在我们可以为可勾选值使用这个更灵活的集合记法勾选或取消勾选多选框：

```
(fill {:tag :input
        :attrs {:value "yes"
                :type "checkbox"}})
```

```

#{"yes" "y"})
:= {:attrs {:checked "checked", :type "checkbox", :value "yes"}, :tag :input}
(fill *1 #{"no" "n"})
:= {:attrs {:type "checkbox", :value "yes"}, :tag :input}

```

而其他输入元素和非输入元素还是按我们的预期填充的：

```

(fill {:tag :input :attrs {:type "text"}} "some text")
:= {:value "some text", :type "text"}, :tag :input}
(fill {:tag :h1} "Big Title!")
:= {:content ["Big Title!"], :tag :h1}

```

还有几件事

多重继承

在这个 `fill` 例子中，层级还没复杂到需要引入多重“继承”。在你有多重方法处理接口时这样的复杂关系经常出现。

假设我们想要一个 `run` 函数，能够执行任何隐约可以运行的东西（像 `java.lang.Runnable` 和 `java.util.concurrent.Callable`）：

```

(defmulti run "Executes the computation." class)
(defmethod run Runnable
  [x]
  (.run x))

(defmethod run java.util.concurrent.Callable
  [x]
  (.call x))

```

让我们用一个函数来测试它：

```

(run #(println "hello!"))
;= #<IllegalArgumentException java.lang.IllegalArgumentException:
;=   Multiple methods in multimethod 'run' match dispatch value:
;=   class user$fn_1422 -> interface java.util.concurrent.Callable and
;=                           interface java.lang.Runnable, and neither is
;= preferred>

```

这个异常相当显而易见：由于 Clojure 函数实现了 `Runnable` 和 `Callable`，多重方法不知道应该挑选哪一个实现、应该偏爱哪一个提示。

偏好是通过 `prefer-method` 函数表达的。这个函数预期三个参数：多重方法和两个转发值，第一个是应该比第二个更受偏爱的转发值：

```
(prefer-method run java.util.concurrent.Callable Runnable)
:= #<MultiFn clojure.lang.MultiFn@6dc98c1b>
(run #(println "hello!"))
:= hello!
:= nil
```

现在这个多重方法知道应该选择、偏爱哪个实现，运行毫无问题。

- 314 > 这个偏好机制让我们可以声明式地解决菱形继承问题，^{注 12} 即一个类通过两个或多个其他中间超类从同一个超类派生出来的情形。因而你不需要因担心多重继承而防御性地设计你的层级。偏好使得多重继承关系明确，因而易于推理。

内省多重方法

有一些罕用的函数让你在多重方法上元编程：`remove-method`、`remove-all-methods`、`prefers`、`methods` 和 `get-method`。这些函数让你查询、更新多重方法。

不过你会注意到没有 `add-method`。好消息是，`defmethod` 宏尽管是一个 `def-` 类型的形式，但不要求它是一个顶层表达式。不过你可能偏爱把已有的函数注册为一个方法实现；通过瞧瞧它是如何实现的，我们发现可以这样做：

```
(macroexpand-1 '(defmethod mmethod-name dispatch-value [args] body))
:= (. mmethod-name clojure.core/addMethod dispatch-value (clojure.core/fn [args]
body)) ●
```

- ❶ 这里用 `clojure.core/addMethod` 而不是简单的 `addMethod`，是由 `syntax-quote` 造成的。防止它是有点别扭，但 Clojure 相当聪明，知道忽略 Java 方法名称上的命名空间。

这提示可以有一种简单的方法实现 `add-method`：

```
(defn add-method [multifn dispatch-val f]
  (.addMethod multifn dispatch-val f))
```

一个旁注：碰巧这个函数可以在 `clojure pprint` 命名空间里找到，是一个私有函数，名为 `use-method`。不过不管是用 `use-method` 或是上面所示的 `add-method`，你都是在依赖一个实现细节，因而得准备在以后的 Clojure 发布版里继续维护它。

type 或 class；或者说，映射的报复

`class` 有一个叫 `type` 的近亲。`(type x)` 通常返回与 `(class x)` 同样的值，除非 `x` 有 `:type` 元数据：

```
(class {})
:= clojure.lang.PersistentArrayMap
```

注 12： 不总是菱形的，因为 Clojure 的层级没有一个共同的根。在 Clojure 里，这个问题最好称为“V 形问题”。

```
(type {})
:= clojure.lang.PersistentHashMap
(class ^{:type :a-tag} {})
:= clojure.lang.PersistentHashMap
(type ^{:type :a-tag} {})
:= :a-tag
```

:type 元数据是一种把数据分成不同的类型并让这些类型用于多重方法的简洁办法。注意，把这个元数据加到其他类型的对象也是可行的：向量、集合、函数，等等。

例如，让我们扩展 313 页“多重继承”一节里的例子，使得我们的 run 多重方法可以接受 Runnable、Callable、常规的 Clojure 函数或者包含这些东西之一，并在 :run 槽里“类型标记”为可运行的映射：

```
(ns-unmap *ns* 'run)

(defmulti run "Executes the computation." type) ❶

(defmethod run Runnable
  [x]
  (.run x))

(defmethod run java.util.concurrent.Callable
  [x]
  (.call x))

(prefer-method run java.util.concurrent.Callable Runnable)

(defmethod run :runnable-map
  [m]
  (run (:run m)))

(run #(println "hello!"))
;= hello!
;= nil
(run (reify Runnable
        (run [this] (println "hello!"))))
;= hello!
;= nil
(run ^{:type :runnable-map}
      {:run #(println "hello!") :other :data})
;= hello!
;= nil
```

- ❶ 现在我们有 type 用做 run 的转发函数，对任何传入的映射它将返回这个 :type 元数据，如果没有这个元数据才转而返回参数的类。

当然，你可以修改转发函数，显式地检查映射的 :run 槽以实现同样目标。不过，如果你

想要 run 某个函数，而又知道那个函数是你的应用程序里其他地方产生的向量的最后一个元素，那会怎么样？你将需要修改转发函数允许这样的情况，但那样就推翻了使用多重方法（实际上也包括 Clojure 许多其他设施）动因的一大部分：让你从被操作的数据中解放出来。

316 转发函数的范围是无限的

最后这个例子将有助于显示多重方法给你带来的灵活性。目前我们所用的多重方法的转发函数只依赖参数返回值。不过，根本就没有要求是这样子的。

考虑一个信号系统，对每个信号的处理根据它的优先级而显著不同：

```
(def priorities (atom {:911-call :high
                        :evacuation :high
                        :pothole-report :low
                        :tree-down :low}))

(defmulti route-message
  (fn [message] (@priorities (:type message)))))

(defmethod route-message :low
  [{:keys [type]}]
  (println (format "Oh, there's another %s. Put it in the log." (name type))))

(defmethod route-message :high
  [{:keys [type]}]
  (println (format "Alert the authorities, there's a %s!" (name type))))
```

这看起来相当简单明了^{注13}：

```
(route-message {:type :911-call})
;= Alert the authorities, there's a 911-call!
;= nil
(route-message {:type :tree-down})
;= Oh, there's another tree-down. Put it in the log.
;= nil
```

不过，信号的优先级如果能动态改变将会如何呢？没问题：只要调整驱动转发函数的数据，route-message 的行为可以显著地改变，不需要对代码或数据进行任何修改：

```
(swap! priorities assoc :tree-down :high)
;= {:911-call :high, :pothole-report :low, :tree-down :high, :evacuation :high}
(route-message {:type :tree-down})
;= Alert the authorities, there's a tree-down!
;= nil
```

注 13：我们的 route-message 实现可能会做更多的事，而不仅是向标准输出打印东西。

这给了你许多自由和威力，也许正好足以解决你的这个问题。同时——抱歉这里用了个双关语——它也可能正好足以让你自缚。一个多重方法的行为不是严格地依赖于参数所提供的值，则根据定义，它不是幂等的。^{注14} 那不会使得这样的函数不可靠或无用，只是与其他幂等的同类相比更难以理解、测试和与其他函数组合。

最后的思考

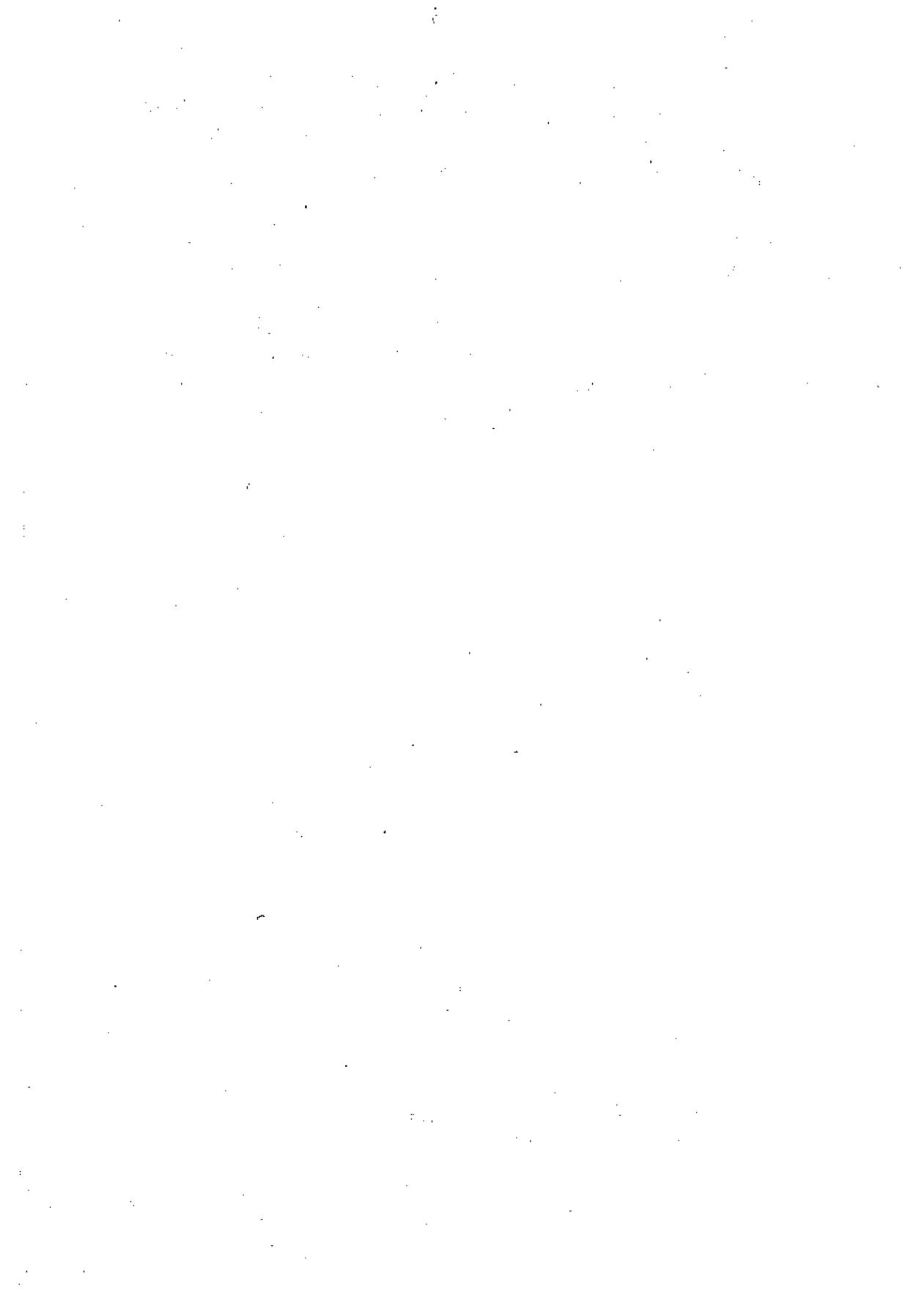
<317

通过 `fill` 这个例子，我们已经覆盖了 Clojure 多重方法的所有基本特性。^{注15}

需要一些时间来习惯它的威力，跳出基于类型的单转发来思考。每次在想写一堆嵌套的条件或很大的 `cond` 语句或定义许多类型以把数据与具体的功能连接起来时，你应该问自己：一个多重方法是否能更好地满足你的需要。

注 14：更多关于幂等性、纯函数及其好处，请参见 76 页“纯函数”一节。

注 15：在第 15 章你会找到更多使用多重方法的例子。



工具、平台以及项目

Clojure项目的组织与构建

讽刺的是，采用一门有前途的新编程语言面临的最大挑战之一经常与语言本身没多大关系：用新语言写成的代码库需要好好组织起来，构建成一个个构件，便于分发、便于其他程序员作为函数库来使用或者最终用户直接使用，便于如 Web 应用那样在服务器环境下安装。这一挑战的具体情况依赖于你是把新语言作为已有项目的一部分来用还是在一个全新项目里使用它，以及部署的具体需求。

我们不可能涉及 Clojure 项目组织和结果分发的所有方式，在这些领域的观点分歧经常超越其重要性，^{注1} 但我们有责任让你走对路，符合 Clojure 社区里的典型作法。本章将给出一些通用的提示，思考如何组织 Clojure 代码库，用 Clojure 社区里两个最受欢迎的构建工具 Leiningen 和 Maven 来展示解决 Clojure 项目构建问题的最佳办法。

项目布局

在开始介绍构建机制前，我们需要先确定如何组织 Clojure 项目，像文件的物理存放和代码库的功能组织。这意味着谈论命名空间。

定义与使用命名空间

如我们在 20 页“命名空间”一节所说，^{注2} Clojure 的命名空间：

- 是从符号到 Java 类名和 var 的动态映射，后者包含你指定的任何值（最常见的是函数、常量数据和引用类型）。
- 与 Java 中的包和 Python 与 Ruby 中的模块大致相当。

注 1： 整个“构建”问题也许是人们构建的最大的口水战，在争议与好恶度上也许只逊于制表符与空白符之争和 emacs 与 vi 之争：<http://bikeshed.org>。

注 2： 如果你还没能消化那一节的内容，现在再试试。我们在那一节介绍了命名空间的最基本知识，讲到了符号、var 以及符号如何解析到相应的 var。

Clojure的所有代码都是在命名空间里定义的。如果你没有定义自己的命名空间，你定义的所有 var 都会映射到默认的 user 命名空间。虽然这对于在 REPL 下交互使用很好，但是一旦想要编写一个长期使用和给别人使用的东西时，这几乎从来不是什么好主意。我们需要知道如何以符合 Clojure 语言习惯的方式定义命名空间，命名空间是如何映射到每个源文件的，以及如何更好地使用命名空间来为你的 Clojure 代码库提供高级结构和组织。Clojure 为操作命名空间的方方面面提供了不同函数（这在 REPL 中非常有用），并提供了一个宏把这些函数整合起来，让我们可以在一个地方声明命名空间的名称、顶级文档以及对其他命名空间和 Java 类的依赖。

in-ns、def 及其所有变体（如 defn）在当前命名空间里定义 var，即总是绑定在 *ns* 里：

```
*ns*
:= #<Namespace user>
(defn a [] 42)
:= #'user/a
```

使用 in-ns 可以转到其他命名空间（如果不存在就创建新的命名空间），从而让我们在其他命名空间里定义 var：

```
(in-ns 'physics.constants)
:= #<Namespace physics.constants>
(def ^:const planck 6.62606957e-34)
:= #'physics.constants/planck
```

不过，我们很快就会发现在新的命名空间中出了点岔子：

```
(+ 1 1)
:= #<CompilerException java.lang.RuntimeException:
:=  Unable to resolve symbol: + in this context, compiling:(NO_SOURCE_PATH:1)>
```

函数 +（以及 clojure.core 命名空间里的所有其他函数）不可用，这与我们一直默认在 user 命名空间里工作的情形不同了，得使用带命名空间的符号，这些函数才可用：

```
(clojure.core/range -20 20 4)
:= (-20 -16 -12 -8 -4 0 4 8 12 16)
```

323 ➤ 别忘了命名空间是符号到 var 的映射；in-ns 把我们转到了我们指定的命名空间，这就是它做的全部工作。特殊形式还是可用的（包括 def、var 和 . 等），但需要从其他命名空间载入代码，把其他命名空间命名的 var 映射到新的命名空间，以便我们比较简洁地使用那些代码。

refer。假设某个命名空间已经载入，我们可以用 refer 把这个命名空间里的映射加入到我们的命名空间。前面在 user 命名空间里已经定义了一个没做什么事的函数 a，可以在还是空的命名空间里为 user 的所有公用 var 建立映射，从而让我们访问 a 更便利：

```
user/a
:= #<user$a user$a@6080669d>
(clojure.core/refer 'user)
:= nil
(a)
:= 42
```

a在当前的命名空间里映射到 var user/a，我们可以像在本地定义的一样来使用。与不得不在其他命名空间里处处用带命名空间的符号相比，这肯定更方便。

refer能做的可比简单的“import”要多：把别的命名空间里的 var 映射到当前命名空间时，你可以使用可选的关键字参数 :exclude、:only 和 :rename 来指定包含、排除或重命名某些 var。例如，下面用 refer 引用 clojure.core，但排除某些函数，并把一些算术操作符映射到不一样的本地名称：

```
(clojure.core/refer 'clojure.core
  :exclude '(range)
  :rename '{+ add
            - sub
            / div
            * mul})
:= nil
(-> 5 (add 18) (mul 2) (sub 6))
:= 40
(range -20 20 4)
:= #<CompilerException java.lang.RuntimeException:
:=   Unable to resolve symbol: range in this context, compiling:(NO_SOURCE_PATH:1)>
```

现在可以使用 clojure.core 里的所有公用函数^{注3}（但 range 除外，这是我们明确排除了的），而且对一些算术操作符我们要使用不同的名称。

虽然 clojure.core 总是预载入（而且在 user 命名空间里总是 refer 引入），我们经常需要能做更多的事，为了更合理地组织自己的代码库，我们还想定义多个命名空间，需要有办法载入命名空间。



refer 很少直接用，而是通过广泛使用的 use 来间接使用它。

324

require 和 **use**。当某些代码需要使用别的命名空间里公共 var 定义的函数或数据时，require 和 use 可用：

注3： refer 不会引进源命名空间里的私有 var。关于私有 var 的详情，见 198 页“var”一节。

1. 确保所说的命名空间被载入。
2. 为命名空间里的那些名称建立别名，这是可选的。
3. 触发隐性使用 refer，让代码不需要限定就可以指称其他命名空间的 var。

require 提供了第 1 项和第 2 项；use 基于 require 和 refer 以简洁的方式提供第 3 项。

新开一个 REPL，我们来调用 clojure.set 命名空间里的 union 函数：

```
(clojure.set/union #{1 2 3} #{4 5 6})  
;= #<ClassNotFoundException java.lang.ClassNotFoundException: clojure.set>
```

等等，那个命名空间还没载入呢，只有 clojure.core 是预载入的。可以用 require 来从 classpath 载入 clojure.set 命名空间；注⁴ 这之后，就可以用那个命名空间里的任何函数了：

```
(require 'clojure.set)  
;= nil  
(clojure.set/union #{1 2 3} #{4 5 6})  
;= #{1 2 3 4 5 6}
```

不过必须用全限定的符号来指称 var 会很痛苦，如果你所用的函数库提供的命名空间较长或有不少节段就尤其痛苦。好在 require 提供了为命名空间指定别名的途径：

```
(require '[clojure.set :as set]) ●  
;= nil  
(set/union #{1 2 3} #{4 5 6})  
;= #{1 2 3 4 5 6}
```

- 提供给 require 和 use 的向量参数有时被称为 *libsspecs*：用于指定一个函数库是怎么载入的及在当前命名空间里如何引用。

如果需要请求有共同前缀的多个命名空间，可以给 require 提供一个顺序性集合类，第一个元素是命名空间前缀，其余元素是想要载入的命名空间的其余节段。因而，如果你想要同时 require clojure.set 和 clojure.string，就可以不用重复前缀 clojure：

```
(require '(clojure string [set :as set]))
```

use 提供了 require 的所有功能，只是在指定的命名空间载入后还会默认 refer 指定的命名空间。因而 (use 'clojure.xml) 等价于：

```
(require 'clojure.xml)  
(refer 'clojure.xml)
```

注 4：关于 Clojure 命名空间如何与磁盘文件对应，见 328 页“命名空间与文件”一节，关于什么是 classpath 及为什么要在意它，见 331 页“classpath 入门”一节。

此外，use 会把它的所有参数传给 refer，因而你可以充分利用后者的 :exclude、:only、和 :rename 选项。举例说明一下，假设需要同时使用 clojure.string 和 clojure.set 的情景：

1. 我们愿意 refer 第二个命名空间里的所有 var 到当前的命名空间，但……
2. 本地有一些函数的名称与 clojure.string 里的有冲突，简单地命名空间别名（在 require 里用 :as）也是可以的，但……
3. 我们需要多次使用 clojure.string/join，而这并不与当前的命名空间里的任何函数发生冲突，所以在这种情形下我们不想用命名空间别名。
4. clojure.string 和 clojure.set 都定义了 join 函数；试图把两个都引入会导致出错，所以我们更想要 clojure.string/join。

use 能轻易地满足这些标准：

```
(use '(clojure [string :only (join) :as str]
              [set :exclude (join)]))

;= nil
join
;= #<string$join clojure.string$join@2259a735>
intersection
;= #<set$intersection clojure.set$intersection@2f7fc44f>
str/trim
;= #<string$trim clojure.string$trim@283aa791>
```

现在可以直接访问 clojure.string 的 join 函数，不需要任何命名空间的限定，而且 clojure.set 的其余部分也被 refer 引入我们的命名空间（包括 intersection），而整个 clojure.string 命名空间通过 str 别名全部可用。

有效使用 require、refer 和 use

这些函数合在一起提供了许多精细的选项，与 Java 里的 import 和 Ruby 里的 require 等笨拙工具相比尤其如此。有效而符合 Clojure 习惯地使用这些函数可能成为一些 Clojure 初学者的难点。

总是用 require，一般给每个命名空间提供别名，应是不错的默认作法：

```
(require '(clojure [string :as str]
                  [set :as set]))
```

这与 Python 里 `import sys, os` 大致相当。因为命名空间通常有多个节段（而 Python 里常见单个词的模块名），Clojure 并不为 `require` 请求的命名空间提供默认别名，但让你控制所使用的别名名称。当然，如果所涉及的命名空间很短，或只需少数几次用到这个命名空间的 `var`，不带别名的光杆 `require` 是完全合适的。

另一个经常推荐的模式是偏爱 `use`，同时用命名空间别名和明确的包含列表列出要 `refer` 引入到当前命名空间的所有 `var`：

```
(use '[clojure.set :as set :only (intersection)])
```

只要 `use` 这种形式能提供 `require` 和 `refer` 所提供的功能的超集，你可以在单一的 `use` 形式里整合进所有的命名空间引用。即使在你可能使用带别名的 `require` 形式的地方，等价的 `use` 形式几乎不会更长，还让你很容易把 `refer` 引入的函数加到 `:only` 参数里。

无论如何，总的说来，好的作法是要避免无节制地使用 `use`，即那些不包含 `:only` 选项（以明确指定应该 `refer` 引入当前命名空间的函数）的形式。这样做可以表明你的代码使用了其他命名空间的什么部分，避免因上游函数库修改，增加了你已经在本地声明的函数而产生名称冲突警告。

`import`。Clojure 的命名空间主要是把符号映射到 `var`，而 `var` 按照约定经常在多个其他命名空间里定义，但也把符号映射到 Java 类和接口。可以用 `import` 把这样的映射加入当前的命名空间。

`import` 的参数是要引入的类的全名，或一个描述要引入的包名和类名的顺序性集合类。引入一个类使得类的“简短名称”在当前命名空间可用：

```
(Date.)
;= #<CompilerException java.lang.IllegalArgumentException:
;=   Unable to resolve classname: Date, compiling:(NO_SOURCE_PATH:1)>
(java.util.Date.)
;= #<Date Mon Jul 18 12:31:38 EDT 2011>
(import 'java.util.Date 'java.text.SimpleDateFormat)
;= java.text.SimpleDateFormat
(.format (SimpleDateFormat. "MM/dd/yyyy") (Date.))
;= "07/18/2011"
```

- `Date` 类在 `java.util` 包里，所以在把它引进当前的命名空间前直接用简名会导致错误。
- 无须显式引进就可以用 Java 的类和接口，但这种用法要求完整限定的类名，这可能累赘得令人不快。

❸ 可以给 `import` 提供指称这些类的符号来引进类。

❹ 引进之后，这些类可以用简名来引用了。

`java.lang` 包里的所有类默认总是被引进每个命名空间，例如 `java.lang.String` 类以 `String` 符号提供，不需要另外用 `import` 引进。 327

想从同一个包里引进多个类时，可以给 `import` 提供以包名为前缀的集合类，这与 `require` 接受有共同前缀的多个命名空间一样：

```
(import '(java.util.Arrays Collections))
;= java.util.Collections
(-> (iterate inc 0)
  (take 5)
  into-array
  Arrays/asList
  Collections/max)
:= 4
```

这种情况很少见，但要知道你不能用相同简名把两个类引进同一个命名空间：

```
(import 'java.awt.List 'java.util.List)
;= #<IllegalStateException java.lang.IllegalStateException:
;= List already refers to: class java.awt.List in namespace: user>
```

这里的解决办法和在 Java 里一样，把用得最多的类引进你的命名空间，另一个类就用完整类名。



Clojure 的 `import` 与 Java 的 `import` 命令概念上是相似的，但有两点重要区别。

首先，它没有提供 Java 里常用的通配引进类似的机制，例如 `import java.util.*;`。如果需要从同一个包里引进多个类，需要枚举这些类，肯定如上所示，作为以包名为前缀的列表的一部分。

其次，如果需要引用一个内部类（如 `java.lang.Thread.State`、`java.util.Map.Entry`），需要用 Java 的内部记法（如 `java.lang.Thread$State`、`java.util.Map$Entry`）。这适用于对 Java 内部类的任何引用，不限于 `import`。

`ns`：本节目前为止所看到的这些命名空间工具函数一般保留在 REPL 下使用。只要还想在 REPL 外重用代码，就应该在 `ns` 宏里定义自己的命名空间。^{注5}

注5：把在 REPL 下得到的能工作的代码复制、粘贴到一个 `.clj` 文件（加上必要的 `in-ns` 和 `refer` 等形式）就算交差可能很有诱惑力。我们督促你要战胜任何这样的诱惑。下一节我们将要讨论到，在如何组织 Clojure 代码上有一些保持健康的规则，不用 `ns` 来指定完整的命名空间将与那些指南相悖而没有什么好处。

328 ➤ ns 让你可以声明式地指定一个命名空间的名称和它的顶级文档以及成功载入、正常工作所需 require、refer、use 和 import 而做的一切，是对这些函数的非常轻量级的包装。因而下面这些工具函数调用：

```
(in-ns 'examples.ns)
(clojure.core/refer 'clojure.core :exclude '[next replace remove])
(require '(clojure [string :as string]
                   [set :as set])
         '[clojure.java.shell :as sh])
(use '(clojure zip xml))
(import 'java.util.Date
        'java.text.SimpleDateFormat
        '(java.util.concurrent Executors
          LinkedBlockingQueue))
```

与这个 ns 声明等价：

```
(ns examples.ns
  (:refer-clojure :exclude [next replace remove])
  (:require (clojure [string :as string]
                    [set :as set])
            [clojure.java.shell :as sh])
  (:use (clojure zip xml))
  (:import java.util.Date
          java.text.SimpleDateFormat
          (java.util.concurrent Executors
            LinkedBlockingQueue)))
```

require 和 refer 等的所有语义保持不变，但 ns 是一个宏（注意，这里用的是关键字，如 :use 而不是 use），所以不需要大量地用引号。



在前面的例子中，我们从 clojure.core 排除了一些 var，因为这些名称（next、replace 和 remove）与在 clojure.zip 中定义的同样名称的 var 重名了，后者在几行后面没用排除直接使用 use。对 clojure.zip 直接用 use 将会撤销从 clojure.core 引用的那些 var 的映射关系（同时报个警告），但明确排除这些 var 可以让后来的维护者清楚我们知道有这个冲突。

定义命名空间之后，就可以在运行中查看、修改，通常通过一个 REPL。我们在 399 页“最简的 REPL”里谈到了可用于命名空间的不同工具。

命名空间与文件

关于 Clojure 源文件该如何组织有一些确定的规则：^{注6}

注 6：像所有规则一样，如果你有合理的理由，这些规则大多数都是可以违反的，但那样的理由很少。

一个命名空间一个文件。每个命名空间应该在单独的文件中定义，文件在项目的 Clojure 源文件根目录下的位置必须与命名空间的节段对应。例如，`com.mycompany.foo` 命名空间对应的代码应该在 `com/mycompany/foo.clj` 文件里。^{注7}当 `require` 或 `use` 这个命名空间时，如 `(require 'com.mycompany.foo)`，将会载入文件 `com/mycompany/foo.clj`，之后必须定义这个命名空间，否则会出错。

命名空间含连字线时文件名用下画线。简单地说，如果命名空间是 `com.my-project.foo`，命名空间对应的源代码应该在 `com/my_project/foo.clj` 文件里。只有命名空间节段对应的文件名和目录受影响——在 Clojure 代码中应该继续用声明的名称来引用命名空间，例如，`(require 'com.my-project.foo)`，而不是 `(require 'com.my_project.foo)`。这是因为 JVM 不允许类名或包名里有连字线，而在命名包括命名空间、`var`、局部量在内的 Clojure 实体时符合 Clojure 语言习惯的通常是用连字线而不是下画线。

每个命名空间都以完整的 ns 形式开始。每个命名空间的根文件（通常也是唯一的）的第一个 Clojure 形式应该是精心设计的 ns 形式，像 `require` 和 `refer` 这样的命名空间操作函数在 REPL 环境外完全没有必要。除了是不错的形式外：

1. ns 的使用鼓励整合，否则可能得分别使用 `require` 等。
2. ns 的使用让代码的读者和后来的维护者立刻了解这个命名空间与它的依赖是如何关联的，因为它总是放在每个文件的顶部。
3. ns 的使用为代码重构和其他代码操作工具打开了方便之门，这些工具需要修改所请求的命名空间、函数和引进的类的集合，因为 ns 是一个宏，只能接受这些东西的未求值的名称。^{注8}无限制求值与可能修改命名空间的底层形式会使那样的工具不可行。

避免命名空间循环依赖。在任何应用程序里，Clojure 命名空间的依赖关系应该形成一个有向无环图，即命名空间 X 不能要求命名空间 Y，而 Y 又（直接或通过其依赖关系间接）要求命名空间 X。这样做会产生如下错误：

```
#<Exception java.lang.Exception:  
  Cyclic load dependency:  
  [ /some/namespace/X ]->/some/namespace/Y->[ /some/namespace/X ]>
```

用 `declare` 启用前向引用。Clojure 按顺序载入每个命名空间的文件中的每个形式，同时把引用解析为前面已经定义的 `var`。这意味着引用未定义的 `var` 会导致出错：

注7：这些路径是相对于你所使用的源文件的根目录。在 332 页“位置、位置、位置”一节里将讨论 Clojure 项目的物理布局。

注8：例如，`slamhound` 会基于给定文件的代码调整 ns 形式里什么命名空间被 `require` 和 `use` 以及什么被 `import`：<https://github.com/technomancy/slamhound>。

```
(defn a [x] (+ constant (b x)))
;= #<CompilerException java.lang.RuntimeException:
;=   Unable to resolve symbol: constant in this context, compiling:(NO_SOURCE_
PATH:1)>
```

许多编程语言定义编译单元，这允许它们找到程序里所有还未被定义的标识符，不用先解析指向它们的引用；而 Clojure 不这样做。不过这一切并没有都失去，不管是出于清晰或风格的考虑，如果你想先定义高层函数后定义高层函数所引用的低层函数，可用 `declare` 在当前命名空间内化一个 var，然后定义自己的高层函数（自由引用由 `declare` 声明的 var），后来才定义前面已经用 `declare` 声明的 var：

```
(declare constant b)
;= #'user/b
(defn a [x] (+ constant (b x)))
;= #'user/a
(def constant 42)
;= #'user/constant
(defn b [y] (max y constant))
;= #'user/b
(a 100)
;= 142
```

需要注意的一个小问题是，如果忘了定义前面 `declare` 声明的 var，在运行时解引用那个 var 会产生一个不可用的占位值，你的高层代码试图使用它时几乎肯定会导致异常。

避免单节段命名空间。命名空间应该有多个节段，例如 `com.my-project.foo` 命名空间有 3 个节段。原因有二：

1. 如果提前编译一个单节段命名空间，这个过程会在默认包里产生至少一个 class 文件（即一个光杆 class 文件，不在 Java 包里），在某些环境里这会阻止命名空间被载入，并且总会阻止这个命名空间对应的 class 文件被 Java 使用，因为 Java 语言限制默认包里 class 文件的使用。
2. 即使你肯定不会为单节段命名空间分发提前编译的 class 文件，无论你命名时多么聪明，仍然冒着更大风险出现命名空间冲突，这并不明智。

关于命名空间节段的深度，不要以为我们推荐你到达荒唐的高度。没人喜欢像 `com.foo.bar.baz.factory.Factory` 的名称。不过，在那与单节段、容易出现冲突的命名空间如 `app` 或 `util` 之间有令人满意的折中方案。

- 331> 不管如何组织你的命名空间，这些命名空间（以及你的函数库或应用程序依赖的所有其他代码和资源）最终将通过 classpath 载入。

classpath 入门

对于不熟悉 Java 的程序员，classpath 经常成为人们困惑的根源。classpath 是 JVM 在查看用户自定义的函数库和资源时所搜索的路径，包括目录和.zip 归档文件，也包括.jar 文件。Clojure 寄生在 JVM 上，也继承了 Java 的 classpath 系统。

classpath 有它的特点，但这并非独一无二，与其他搜索路径机制（肯定有你熟悉的）有很多相似之处。例如，UNIX 和 Windows 环境下的 shell 定义了一个 PATH 环境变量，这个变量储存了一组串连的路径，可执行程序可以在这些路径里找到。Ruby 和 Python 也有搜索路径：Ruby 把它保存在运行时变量 \$LOAD_PATH 里，^{注9} 而 Python 依赖 PYTHONPATH 环境变量。在所有这些情况下，搜索路径倾向于由系统范围的设置和依赖管理工具（如 Ruby Gems 或 Python 的 easy_install 和 pip）两者的组合自动处理。

classpath 自动配置可以通过最常用于管理 Clojure 项目依赖关系的工具 Leiningen 和 Maven 以及最受欢迎的 Java 集成开发环境和 Emacs 得到。例如，一旦在 *project.clj* 或 *pom.xml* 文件中定义了依赖关系，通过任何一种这样的工具启动 REPL 都将会导致所依赖的被自动加到 REPL 的 classpath 里。如果是用 Leiningen 或 Maven 插件来引导完整的应用程序，如通过 lein-ring 或 jetty:run 运行本地 Web 应用程序，情况是同样的。^{注10}

不过，如果需要在 shell 里直接启动一个 Java 进程，就得手工构造 classpath 了。即使你从不从命令行使用 Clojure，了解 classpath 是如何以最基础的方式定义的将有助于你理解更高级的工具到底做了什么。

定义 classpath。 classpath 默认情况下是空的，与我们提到的其他路径机制相比，这个差异是不方便的，其他的默认都把当前工作目录(.) 包含在内，因而根在这里的函数库在运行时会被找到。

要设置一个 Java 进程的 classpath，在命令行用 -cp 标志来指定。例如要包含当前工作目录、src 目录，clojure.jar 归档文件和在 lib 目录里的所有 jar 文件，在类 UNIX 系统上得这样做：

```
java -cp '.:src:clojure.jar:lib/*' clojure.main
```



如同所有其他搜索路径机制一样，由于不同系统上文件名约定的差异，classpath 是以平台依赖的方式定义的。在类 UNIX 系统中，classpath 是以 : 分隔、以 / 定义的路径列表，在 Windows 系统中是以 ; 分隔的、\ 定义的路径列表。因而上述类 UNIX 系统的 classpath 例子在 Windows 系统中将翻译为：

```
'.;src;clojure.jar;lib\*'
```

332

注9： 又名为 \$: 变量。

注10： 详情见 565 页“在本地运行 Web 应用”一节。

classpath 与 REPL。classpath 可以在运行时在 Clojure 里查看：

```
$ java -cp clojure.jar clojure.main  
Clojure 1.3.0  
(System/getProperty "java.class.path")  
:= "clojure.jar"
```

主要的 classpath（保存在 `java.class.path` 系统属性里）是在 JVM 进程启动时通过命令行参数或环境变量定义的，遗憾的是不能在运行时修改。这与 Clojure 正常的开发周期不符，后者倾向于打开一个持续运行的 REPL 会话并让它一直开着。改变 classpath 需要 JVM 重启，所以 REPL 也得重启。^{注 11}

位置、位置、位置

有两个占主导地位的项目布局惯例用于 Clojure 项目，默认值是由 Clojure 项目所用的主要构建工具定义的。^{注 12}

第一个是“Maven 风格”，把所有源文件放在顶级的 `src` 目录里，但根据语言和在项目中的作用把源文件放在不同的子目录里。定义公用 API 或要发布的特性的主要源代码在 `src/main` 目录里，定义单元和功能测试的代码通常不会分发，在 `src/test` 里，等等：

333 例 8-1：“Maven 风格”的项目布局

```
<project dir>  
|  
|- src  
  |- main  
    |- clojure  
    |- java  
    |- resources  
    |- ...  
  |- test  
    |- clojure  
    |- java  
    |- resources  
    |- ...
```

注 11：有一些绕过这个问题的办法。Clojure 本身就提供了一个 `add-classpath` 函数，不过已经弃用，一般不推荐使用。另一个是 `pomegranate` (<https://github.com/cemerick/pomegranate>)，为 `add-classpath` 提供继续维护的替代方案，提供了把 `.jar` 文件和 Leiningen/Maven 传递性依赖加到运行中的 Clojure 的办法。最后，所有类型的 JVM 模块系统，包括 OSGi、NetBeans 模块系统和 JVM 应用服务器都提供了便利的办法扩展或重定义应用程序或单个模块内的 classpath。所有这些机制都用到了 JVM 内建的设施（如受管理的 ClassLoader 层级）以启用这样的功能。

注 12：所有（不错的）构建工具（包括 Leiningen 和 Maven）都让你把源文件放在任意地方。这些布局只是默认情形，不过难以想象值得费劲不按默认方式来做的情况。

用这种项目布局时，Clojure 源文件的根在 `src/main/clojure`，Java 源文件^{注 13} 的根在 `src/main/java`，等等。文件的作用和类型在目录结构上反映出来，这可以让某些活动变得更容易。例如，不需用文件过滤器就可以从源文件根目录选择出某一类型的一组文件，可以用每种类型的文件根目录来“无选择地”指向一组文件。这样就可以大大地简化资源的打包：如果你有一组资源需要包含在 Web 应用（图像、JavaScript 文件等）中，可以在 `src/main/webapp` 下分组，不需分发的资源可以安全地放在一个不同的源文件根目录下，可以肯定在构建和打包过程不会被引用。

Maven 风格的布局是最标准化的选择——它鼓励源文件的位置遵循惯例，意味着使用这种风格的项目绝少违反。Maven 风格的项目布局最主要的缺点是文件路径变长了，因为要使用 `src/main`、`src/test` 等前缀。

项目布局的另一个主流风格则难于刻画，因为项目与项目之间可以有很大的变化：

例8-2：“自由”风格的项目布局举例

```
<project dir>
|
|- src
|- test

<project dir>
|
|- src
  |- java
  |- clojure
|- test
|- resources
|- web
```

334

与 Maven 风格相比，自由风格的项目布局文件路径长度更短（部分原因是为了使之在命令行容易引用文件），除了 `src` 和 `test`，项目和项目间一般重用的惯例更少。不同类型的源文件经常混在同一个源文件根目录下（如 Java 和 Clojure 源文件可能都在根目录 `src` 里），虽然有时候并不是这样，这取决于特定项目的构建配置。你会发现用这种布局的项目可使用 Maven 之外的其他构建工具，其中包括 Leiningen。

Clojure 代码库的功能组织

目前还只讨论了机械的基本规则——文件放哪儿、命名规则、命名空间与文件的对应关系，等等。更微妙的是，从功能角度看该如何组织 Clojure 代码：

注 13：假设没有 Java 源文件；如果项目是一个 Java/Clojure 混合项目，看一下 351 页“构建混合源文件项目”一节里的提示。

- 实现特定算法应该用多少函数？
- 一个命名空间应该有多少函数？
- 一个项目应该有多少命名空间？

与其他编程语言相关的直接问题通常更容易回答，部分原因是经常有具体要求，最终明确定义了什么是“好的风格”。多种语言里许多常用框架对如何定义插件、组件、模型、扩展有特定的预期（如“每个数据库表一个类”或“每个用户接口组件一个模块”），因而代码库的形状很大程度上取决于所用的函数库附带的或机械的特点以及将要部署的大环境。

相比之下，你很少会由于使用某个函数库或框架而被迫改变 Clojure 应用程序的组织方式。^{注14} 广泛应用函数式编程的技术和少量、审慎地使用宏，这尤其让你能把 Clojure 函数库和应用程序组织得远比其他语言更能反映应用域的面貌。可以这么说，Clojure 鼓励你更清晰地思考应用域，多年来你都没能这么清晰地思考过，最后结果是你的数据或模型会自然地支配程序的结构，并且超出你过去认为可能的限度。

这也就是说，除了一些非常一般的原则，Clojure 程序也许根本就没有所谓的“典型结构”。

[335] > 这个理念也许令人困惑，也许非常动人，视你的背景和预期而定。对我们来说，我们一直觉得这是让人欣喜的，可以关注一个特性或算法或应用域的基本要素，不受很久之前做出的决定所包含的某些要求干扰，那些要求与我们在代码中要解决的问题无关。

项目组织的基本原则

如果只是模糊地说说一般原则，不提我们心里想的几条，那是我们所不齿的：

- 不同的事物要分开，也许是放在不同的命名空间：与自定义的 record 相关的代码也许应该在同一个命名空间，与载入 Web 内容的模板的命名空间分开。
- 相关的事物放在一起，也许按自然类别分组，体现为命名空间。例如，用命名空间的名称所暗含的层级来表示如高层 API（如 foo.ui）和低层 API（如 foo.ui.linux 和 foo.ui.windows）之间的关系。
- 尽量把包含实现私有的数据或函数的 var 定义为 `^:private`（即私有，或者用 defn- 便利形式来定义私有函数）。这让客户程序不至于无意中依赖很可能改变的东西，同时如果真有必要，通过 var 特殊形式（或其 reader sugar #'）仍然提供了访问“后台”函数和数据的后门。
- 不要重复自己：只在指定的命名空间定义常量一次，如有必要，分离出共同的功能为工具函数和工具命名空间。

注 14： 即使是用 Clojure 扩展或集成已有的 Java 函数库或框架，很少会无法隔离 Clojure 函数和数据接入框架相关的互操作，让你自由组织 Clojure 代码库以最适合应用域或所选择的架构。

- 如有可能，用通用的抽象如引用类型、集合类、序列，不要把这些抽象的特定的具体实现结合进去。
- 不纯的函数应该视为有害，只有绝对必要才那样实现。^{注15}

总的说来，Clojure 项目会从注重模块性、关注分离上获得与其他语言写的项目一样的益处。此外，要记住命名空间是一个组织工具，完全是为了方便你而提供的：一个用 500 个命名空间写成的大型应用程序与把所有东西堆在一个巨大的命名空间里的功能和效率是一样的。所以，你应该自由地组织应用程序，以与应用域匹配，适应团队的工作方式。

构建

336

“构建”是一个统称，包括我们写完代码之后到代码发布之前越来越多的东西；代码发布是又一个含义丰富的词，软件作为服务、云计算等使情况更复杂。

对我们的意图来说，构建表示：

- 编译。
- 依赖管理，让你可以系统地使用外部函数库。
- 把编译结果和项目其他资源打包成构件。
- 在依赖管理的环境下分发这些构件。

这样过分正式的描述听起来比所描述的活动更复杂。很可能你已经在做这些事了：

表8-1：比较不同编程语言的“构建”方案

	编译	依赖管理	打包	分发
Ruby	rake	gem、rvm	Gems	rubygems.org
Python	distutils、SCons	pip、virtualenv	Eggs	PyPI ^a
Java	javac、Ant、Maven、Gradle 等	Maven 模型、Ivy	jar 文件及其变体	Maven 构件库

^a <http://pypi.python.org/pypi>

因为 Clojure 是一种 JVM 语言，自然大量地重用了这个生态系统的构建、打包和分发的基础设施和机制：

- Leiningen 重用了 Maven 许多的基础设施，但提供了令人愉悦得多的“用户界面”和 Clojure 原生开发体验。
- Maven、Gradle 和 Ant 都有插件帮助从这些工具驱动 Clojure 的构建。

注 15： 纯函数对成功而有效地应用函数式编程至关重要，是设计地道的 Clojure 函数库和应用的基石。关于函数式编程，见第 2 章，关于纯函数，见 76 页“纯函数”一节。

- Clojure 函数库打包成 `.jar` 文件, Clojure Web 应用(通常)打包成 `.war` 文件,^{注¹⁶}等等。
- Clojure 函数库通过 Maven 库分发, 每个 Java(因而也包括 Clojure)构建工具都可用。

Clojure 和 Java 的构建工具与实践的对应关系让 Clojure 应用程序与函数库可以依赖、使用 Java 库, 让你分发 Clojure 写的函数库而其他 JVM 语言的程序员(如 Java、Groovy、Scala、JRuby 和 Jython 等)可以依赖和使用它们。

- [337] >** 如果已经在使用 Java 或其他某种 JVM 语言, 你会发现增加一些 Clojure 代码到代码库里对已有的构建过程产生的影响极小。另一方面, 如果你是来自 Ruby、Python 或其他某种非 JVM 语言, 令人安慰的是, Clojure 的构建过程和配置几乎总是比为 Java 定制的工具要简单。

提前编译

在第 3 页“Clojure REPL”一节里已经提到, Clojure 代码“总是”编译的——没有 Clojure 解释器。编译涉及为给定片段的 Clojure 代码生成字节码并把字节码载入主 JVM, 可以有两种方式:

- 在运行时。这正是在使用 REPL 或把 Clojure 源文件从磁盘载入时发生的情况。源文件的内容被编译成字节码, 然后载入 Java 虚拟机。字节码及其所定义的类在所运行的 JVM 终止后不会保留下。
- “提前”(AOT) 编译与运行时编译一样, 但生成的字节码会作为 Java 虚拟机 class 文件^{注¹⁷}存储在磁盘上。这些 class 文件在随后的 Java 虚拟机实例中可以代替原始的 Clojure 源文件被重用。

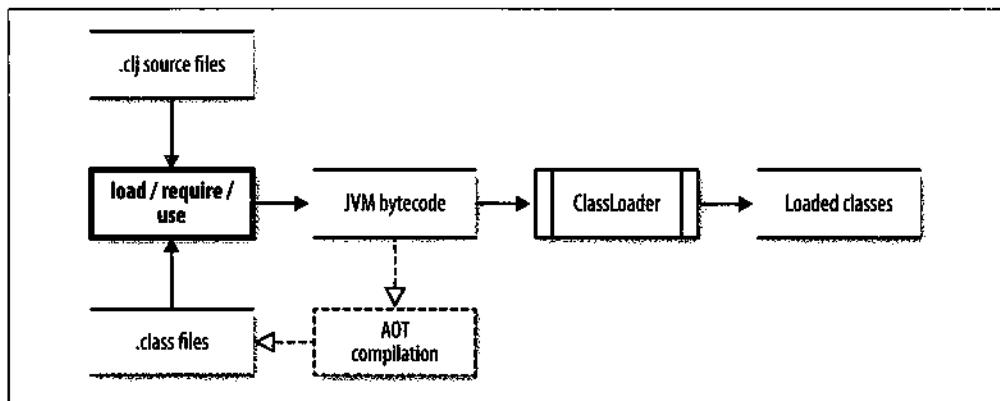


图8-1: Clojure的编译过程

注 16: Heroku 在这方面是一个例外, 请参见 587 页“Heroku 上的 Clojure”一节。

注 17: 这与 javac 类似, 它存储基于 Java 源文件内容生成的 class 文件。

Clojure 代码可以从源文件载入或从提前编译的 class 文件载入，两者可以互换而没有功能上的差别。请求一个命名空间——例如 (require 'clojure.set)——将在 classpath 里搜索定义这个命名空间的 *closure/set.clj* 源文件，或从对应的提前编译的 class 文件里搜索。^{注 18}

因而，除了几个特别的情况，提前编译完全是可选的。既然 Clojure 函数库和应用程序一般没有技术原因要求提前编译，如果可能的话优选分发 Clojure 源程序是合理的。事实上，提前编译可能有一些重要的消极面：

1. Java 虚拟机中的 class 文件从定义上讲与生成它的 Clojure 源文件相比在磁盘上要大得多。
2. 提前编译给原本轻量级的 Clojure 开发周期加上了单独编译这一步。
3. 提前编译一个项目使得 Clojure 函数库或应用程序锁定在提前编译源文件时所用的 Clojure 版本。不能假设用一个版本的 Clojure 提前编译的代码可以部署在另一个版本的 Clojure 上。
4. 提前编译具有传递性。如果提前编译了命名空间 `foo`，而它请求了命名空间 `bar`，那么 `bar` 也会被提前编译，依此类推。这可能导致大范围的提前编译，远比依据构建配置预期的范围要广泛得多，具体情况取决于你的函数库或应用程序里命名空间的具体依赖关系。



如果只需生成单个（或几个）class 文件作为 Java 框架的入口，有一个简单办法可以切断这个传递关系：使用 `gen-class`（或 `ns` 的 `:gen-class` 选项）并用 `:impl-ns` 指定不同的命名空间。编译不会跟随那些依赖，因而请求编译 `gen-class` 的命名空间不会触发整个代码库的编译。

提前编译应该限于以下几种情况：

1. 不能或不想分发源代码。
2. 打包过程中想用 class 文件的各种加工工具，如 obfuscator。
3. 应用程序的启动时间绝对至关重要，因而宁愿承受提前编译的代价。载入提前编译的 class 文件远比从 Clojure 源文件开始要快得多。
4. 预计 Clojure 代码会通过 `gen-class`、`defrecord`、`deftype` 或 `defprotocol` 生成具名的 Java 类或接口被 Java 或别的 JVM 语言调用。^{注 19}

^{注 18}：如果某个命名空间既有源文件，又有 class 文件，Clojure 将优先用 class 文件，除非源文件的修改日期更新。这让你在 REPL 会话的 classpath 里既可以有源文件根目录又有提前编译的目标目录，需要时仍然可以从更新的源文件重新载入命名空间。

^{注 19}：通常确保只提前编译那些包含使用这些形式的命名空间是值得的，即使只是尽量减少打包的构件文件大小。

我们在 349 页“提前编译的配置”一节里将讨论在 Leiningen 和 Maven 里进行提前编译的一些选择。

依赖管理

每个软件开发团队都应该在自己的项目里使用依赖管理。Clojure 开发团队也不例外。

不久以前的黑暗时期，把所有依赖用 `.zip` 和 `.tar.gz` 文件在互联网上发来发去是常态，手工地把这些文件小心地放在项目特别的地方（经常是 `lib` 目录），还把结果加入版本控制系统。构建进程会用简单的文件路径来引用这些依赖，而且在分发的二进制或其他构件里捆绑这些依赖，或者毫不担忧地假定项目的下游用户已经有同样的版本，或者知道从哪儿得到文件。

考虑到这样做依赖手工修改、过程完全不可重复，现代需求在很大程度上已经把这样的作法淘汰了。

Perl 有 CPAN（这也许是所有依赖管理作法的前辈），Python 有 `pip` 和 `virtualenv`，Ruby 有 `gem` 和 `rvm`，JVM 上的依赖管理是 Maven 主导的，Maven 的模型是在 2000 年定义的。Clojure 完全采纳了这个模型，不管使用什么构建工具。

Maven 的依赖管理模型

Maven 的依赖管理模型提供了：

- 用坐标来标识构件和版本。
- 声明构件的依赖和产生这些构件的项目。
- 在 Maven 库里存储、检索构件及其依赖的详细规范。
- 计算构件的传递性依赖。

下面来详解这些概念和涉及的机制所产生的广泛影响。

340 构件与坐标

一个构件是项目构建进程产生的任何文件。Clojure 函数库和应用程序打包就像用 Java 和其他 JVM 语言写的一样。具体来说，这意味着你会主要用到和产生两种构件：

- `.jar` 文件，这是 `.zip` 文件但包含：^{注 20}
 - 组织成层次结构的 Clojure 源文件、JVM class 文件和其他文件（如静态配置文件、图形文件等），与在源文件根目录或者在编译的目标目录里的位置一致。
 - 元数据（可选），在 `META-INF` 目录里。

注 20：关于 `.jar` 文件，更多内容请参见 <http://java.sun.com/developer/Books/javaprogramming/JAR/basics>。

- `.war` 文件，这也是 `.zip` 文件，是 JVM 打包 Web 应用的标准。关于 `.war` 文件的更多信息及其用法，参见 560 页“Web 应用程序打包”一节。

罕见的打包类型

常规的、没有装饰的 `.zip` 或 `.tar.gz` 文件偶尔也会在 Maven 库里碰到，不过这些多半是用于传输非代码的文件。例如，你可能想把一个 JVM 安装程序包括进一个在构建时自动创建的客户端应用的安装程序里，把 JVM 安装程序压缩并部署到机构自己的 Maven 库可以确保自己的项目能声明对 JVM 安装程序的依赖并用脚本把它包括进安装程序里。

此外，像 Eclipse 和 NetBeans 等胖客户端平台有自己的打包类型和需求，不过这些文件格式大致上只是 `.jar` 文件格式的变体，如果有也很少用于这些特别框架以外的格式。

所有类型的构件都是用坐标标识的，这是由一系列的属性组成、可唯一标记构件的特定版本。

- `groupId`, 一般是组织或项目标识名，如 `org.apache.lucene` 或 `com.google.collections`。
- `artifactId`, 构件在组织或项目内的标识名，如 `lucene-core` 或 `lucene-queryparser`。在同一个 `groupId` 下，项目通常产生多个相关联的构件，不过对于规模较小的 Clojure 开源函数库来说，如果机构和项目没有区别，`groupId` 和 `artifactId` 常用同一个名称。
- `packaging`, 标识构件的类型，与构件自身的文件扩展名对应。默认是 `jar`，在默认情况时通常不指定。
- `version`, 版本字符串，理想情况下遵循按语义标记版本的惯例。^{注21}

341

在文本环境下，Maven 的坐标经常以这样的格式指定：`groupId:artifactId:packaging:version`，因而 Clojure 的 1.3.0 版的 jar 包是用 `org.clojure:clojure:1.3.0` 来指代的（别忘了默认是 `jar` 方式打包的）。每个项目都定义自己的坐标——有时是在 Maven 的 `pom.xml` 文件里，有时是在 `project.clj` 文件里（如果用 Leiningen 的话）。无论如何，每次项目构建后且作者希望分发生成的构件时，对应的 `pom.xml` 文件会与构件一起上传到一个 Maven 库。

注 21：语义版本编号是标明软件版本间变化规模的众所周知的习惯。更多内容请参见 <http://semver.org>。

库

当一组构件上传到一个 Maven 库后，通常 Maven 库会对与构件一起提供的 *pom.xml* 里的版本信息和依赖信息建立索引，这称之为“部署”。从这一刻起，Maven 库就可以把这些构件分发给任何想要获得它的客户（几乎总是其他开发者，他们有项目依赖于这些构件）。

全世界也许有成百上千个公共的 Maven 库在运行，但只有少数几个大型的库包含主要的 Maven 构件：

- Maven 中央库，这是最大的库，也是基于 Maven 的构建工具搜索构件的默认位置。Clojure 的官方发布版和核心函数库都是发布到 Maven 中央库的。
- Clojars.org，这是 Clojure 社区的一个公共库，Leiningen 为它简化了集成过程。许多非常受欢迎的开源 Clojure 函数库是部署到 Clojars 上的，包括 Ring、Clutch 和 Enlive。
- 大型开源组织像 Apache、JBoss 和 RedHat 也维护着自己的 Maven 库。

很可能在某个时候你也会用到另外两种 Maven 库：

- 私有库 / 内部库，常由公司或其他组织维护，用于容纳自己项目产生的构件，有时也代理像 Maven 中央库或 Clojars 这样的公共库。
- 本地库，这是由 Maven 和其他基于 Maven 的构建工具在 `~/.m2/repository` 里创建的。你的项目所有的依赖包会先下载到这里，缓存起来供未来使用。也可以选择从你构建的项目里把构件“安装”到这里（在概念上与部署是同样的过程，这样指称是为了区别安装动作是指向本地库）。

342

依赖

每个项目都定义了自己的坐标，同样，每个项目都要定义它的“依赖”。依赖是用坐标表示的对其他项目构件的引用。利用依赖的这些规格说明，Maven 和其他基于 Maven 的构建工具可以：

- 确定项目的传递性依赖的集合——即项目的依赖的依赖的依赖，依此类推。更确切地说，一个项目的依赖构成一个有向无环图，根是要构建、运行、查看的项目。项目的依赖形成引用循环是严重的错误。
- 给定项目传递性依赖的完整集合，这些依赖的构件可以加到 JVM 的 classpath 里，新的 REPL 和编译与应用进程就可以启动了；这让在这些进程中运行的代码可以引用这些依赖里的类、资源和 Clojure 源文件。

要解决项目的依赖，直接遍历直方图就可以，一般你不需要知道详情。^{注22} 唯一需要提醒的是，要注意指定依赖的版本的灵活方式，主要形式有快照和版本范围两种。

Clojure “只是” 另一个依赖

Python、Ruby和其他许多语言带有自己专门的运行时，有这些语言背景的人经常预期“安装”Clojure，就像你得安装某个版本的Ruby那样。事实并非如此。

按说，Clojure只是另一个Java/JVM函数库，所以在你的项目里它只是另一个依赖，必须安装的是JVM。然后必须把Clojure加到应用程序的classpath里，在开发和测试时通常有Leiningen或Maven辅助。

Clojure是一个JVM函数库的事实在部署上^{注23}和团队与客户接受上会有一些实在的好处。^{注24}

快照与发布版本。在Maven的依赖模型上，版本分为快照和发布两种。大多数版本字符串，包括像1.0.0、3.6、0.2.1-beta5这样的例子——被视为发布版，暗示着这样的版本在时间上是固定的、不会再变化了。这是所有健康的Maven库强制保证的，从而阻止之前部署的带发布版本字符串的构件被更新。这样就支持了可重复构建的目标：因为发布版的构件在部署到一个Maven库后不能再改变，一旦针对特定版本的依赖构建、测试成功，这些结果可以永远有效。

<343

快照版本则完全不同。版本字符串用-SNAPSHOT后缀表示，例如1.0.0-SNAPSHOT、3.6-SNAPSHOT和0.2.1-beta5-SNAPSHOT，快照版是用来标识从开发最前沿产生的构件。因而，随着开发版不断创建、分发到Maven库，同样的版本号可以指不同时间的具体构件。

例如，假设想跟踪某个函数库到2.0.0版前正在进行的开发，因为2.0.0版会提供你的项目所需的一些关键特性。这个函数库可能会分发多个版本为2.0.0-SNAPSHOT的构建，只要在项目依赖里指定了这个版本号，就总能使用最新的发布前版本来开发和测试。^{注25}

注22：像Leiningen和Maven这样的工具可处理麻烦的细节。如果要了解Maven模型里依赖解决的细节，<http://docs.codehaus.org/display/MAVEN/Dependency+Mediation+and+Conflict+Resolution>是不错的参考文献。

注23：至于应用程序的部署，有各种打包的选项，包括Web应用的.war文件（在560页“Web应用程序打包”一节里讨论）、一般应用程序部署的uberjar文件（这是Leiningen直接支持的，这在347页“Leiningen”一节里提到了）和其他一些方法，目标都是生成一个总文件，包括项目所有的代码和传递性依赖，也包括Clojure。

注24：请参见579页“Clojure真的‘只是又一个jar文件’”一节。

注25：Maven和依赖它的其他工具典型地会每24小时检查一次是否有新快照。如果用的是Maven，传一个-U选项可以强制检查。

一旦作者工作完成、把最后的发布版部署到项目的 Maven 库，你就可以转而使用库的 2.0.0 发布版了。

版本范围。假设需要依赖某个函数库的 1.6.0 版，而你可能知道如果下一个主要发布版前没有 API 兼容性问题依赖这个库是安全的。目前几乎所有构件都使用语义标记版本，就像这个案例，用“版本范围”来定义所依赖的版本经常很有用。Maven 支持好几种版本范围格式。

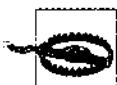
表8-2：Maven版本范围格式^a

范围格式	语义
(,1.0]	$x \leq 1.0$
1.0	“软性”要求 1.0
[1.0]	硬性要求 1.0
[1.2,1.3]	$1.2 \leq x \leq 1.3$
[1.0,2.0)	$1.0 \leq x < 2.0$
[1.5,)	$x \geq 1.5$

^a 取自 <http://docs.codehaus.org/display/MAVEN/Dependency+Mediation+and+Conflict+Resolution>。

所以，对这个库的依赖可以指定为 [1.6.0,2.0.0)，这会让我们的项目构件可以与从 1.6.0（包括在内）到 2.0.0（不包括在内）的任何版本的库一起使用。一旦 2.0.0 版发布

344 （既然主要版本改变了，也许打破了 1.x.x 发布版兼容性），将需要在 2.0.0 版下测试项目（而且做些小修改），再分发一个新的发布版——也许指定对这个库依赖的版本范围 [2.0.0,3.0.0)。



注意，在表 8-2 里，“光杆”版本号（例如 1.0）标为“软性”版本要求。这意味着如果依赖某个构件两个不同的光杆版本，例如依赖某个库的版本一个是 1.2，另一个是 1.6，则后面的版本会被选中并用在所有构建、REPL 等。这几乎从来不会导致什么问题，但如果出问题了，指定一个版本范围可以消除歧义。这个例子里，如果把对这个函数库的直接依赖改为 [1.2,1.5]，那么将会选择 1.5 版。

这样的情景和版本范围的用法取决于对产生你所依赖的函数库的评判。如果稳定地使用（某种程度的）语义版本编号，版本范围可以是一个不错的办法，确保你生成的构件只用于与它兼容的版本的依赖。另一方面，如果你的依赖是用某种其他方案进行版本编号，^{注26} 版本范围对你可能就不那么有用。

注 26：一个不幸的方法是用发布日期作为版本编号，如 20110705。更糟糕的是，构件的发行版本用 Git 或 Mercurial 提交的 SHA 如 8c7be13792 之类的东西，这不仅不适合用于版本范围，甚至不是单调递增的——因而不可能从构件的坐标看出来哪一个版本更新一点。

构建工具与配置模式

既然已经谈及了项目组织和构建概念的关键背景，下面来看看 Clojure 社区里最流行的两个构建工具——Leiningen 和 Maven 的实际使用例子。不过这只能看做快捷的概述，因而考虑如下几点。



这些工具每种自身都相当深奥，而这里也不是全面讨论这些工具的功能与不足的地方。不论选择哪种构建工具，一定要参考它的文档和社区资源，以尽量充分利用它所提供的功能。最后，我们在第 17 章里为 Web 应用额外提供了几个构建的例子。

能用的继续用。如本章开头所说，我们无法提供什么绝对真理——尤其是关于构建工具。尽管如此，有一些启发式的知识可以帮助你决定自己的 Clojure 项目该用什么工具，下面关于 Leiningen 和 Maven 的小节将详细讨论。不过有一个启发式知识也许可以推翻其他的：

345

如果你的组织已经统一使用一种主要的基于 JVM 的构建工具，坚持用它。Ant^{注 27}、Maven、Gradle^{注 28} 和 Buildr 都有 Clojure 插件，可能还有其他我们不熟悉的系统，因而把 Clojure 引入使用这些工具的项目里很简单，不需要搅乱一个在用的、工作得很好的构建和项目管理过程。

Maven

Maven 也许是 Java 世界里最常用的构建工具。作为一个开源的 Apache 项目，^{注 29} 它的使用范围远比大多数构建工具广阔：通过广泛第三方插件社区，它的目标是为管理软件项目“全生命周期”提供办法，除了代码编译和打包等典型的构建过程，还包括像集成与功能测试、代码覆盖、测试报告和发布管理等事情。

Maven 的特性之一是你将发现其他基于 JVM 的工具提供了与 Maven 很好的集成，如集成开发环境 Eclipse、NetBeans 和 IntelliJ，团队设施如 Hudson/Jenkins、Clover 和 Cobertura，构建辅助工具如 NSIS、IzPack、javacc、ANTLR 和 Selenium 等。如果想要在这样的工具与你的构建工具间实现最大程度的集成，或那些“全生命周期”活动对你或你的组织很重要，就应该为你的 Clojure 项目认真考虑用 Maven。

下面这个基本的 *pom.xml* 会让你在简单的 Clojure 项目里使用 Maven：^{注 30}

注 27：可参见 <https://github.com/jmcconnell/clojure-ant-tasks>。

注 28：可参见 <https://bitbucket.org/kotarak/clojuresque/wiki/Home>。

注 29：推荐用 Maven 3 的最新版本，可从 <http://maven.apache.org> 获得。

注 30：关于 Maven 有大量的文档可以在线获得，包括 Maven 作者写的两本免费电子书。这两本书可以从 <http://www.sonatype.com/books.html> 获得：*Maven by Example* 是一本教程式的 Maven 入门指南，*Maven: the Complete Reference* 则值得留在身边备查以理解这个工具的大小小的细节。

例8-3：适合简单的Clojure项目的基本pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.clojurebook</groupId>
  <artifactId>sample-maven-project</artifactId>
  <version>1.0.0</version>
  <packaging>clojure</packaging>

  <!-- 346 --> <dependencies>
    <dependency>
      <groupId>org.clojure</groupId>
      <artifactId>clojure</artifactId>
      <version>1.3.0</version>
    </dependency>
  </dependencies>

  <build>
    <resources>
      <resource>
        <directory>src/main/clojure</directory>
      </resource>
    </resources>
    <plugins>
      <plugin>
        <groupId>com.theoryinpractise</groupId>
        <artifactId>clojure-maven-plugin</artifactId>
        <version>1.3.8</version>
        <extensions>true</extensions>
        <configuration>
          <warnOnReflection>true</warnOnReflection>
          <temporaryOutputDirectory>true</temporaryOutputDirectory>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

这个 *pom.xml* 文件：

- 定义了 Maven 坐标 `com.clojurebook:sample-maven-project:1.0.0`。
- 定义了 `clojure` 打包，这将提示 `clojure-maven-plugin` 把它的提前编译和单元测试目标加到对应的 Maven 生命周期阶段。
- 设置了对 Clojure v1.3.0 的单一依赖 (`org.clojure:clojure:1.3.0`)。

- 把 Clojure 的标准源文件目录加为 Maven 源文件目录，这会把那个目录里的 Clojure 文件加到项目构建生成的 *jar* 文件里。
- 配置 `closure-maven-plugin` 插件，按 349 页“提前编译的配置”一节所描述的“健康检查”模式运行提前编译器，遇到反射性互操作调用就报警，将生成的 class 文件保存到一个临时目录里（这些文件不会包括在打包的构件里）。

使用这个 *pom.xml* 的一些典型工作流程将是：

- `mvn closure:repl` 将启动一个新的 Clojure REPL，classpath 设置为包括项目所有的传递性依赖。
- `mvn package` 将提前编译 Clojure 源文件作为健康检查，构建包含这些文件的 *jar* 包。
- `mvn test` 将运行项目里所有的 Java 和 Clojure 测试（根目录分别在 `test/src/java` 和 `test/src/closure` 里）。347
- `mvn install`——除了做 `mvn package` 所做的一切，将在本地 Maven 库安装 *jar* 文件。
- `mvn deploy`——除了做 `mvn install` 所做的一切，将把 *jar* 文件部署到一个远程的 Maven 库，这需要你在 *pom.xml* 文件中指定。

Maven 里与 Clojure 相关的所有功能都是由 `closure-maven-plugin` 提供的，^{注 31} 它提供了大量的选项用于控制如何及何时调用 Clojure 的提前编译、运行项目里特别的 Clojure 脚本、运行 Clojure 单元和函数测试，等等。此外，它还提供了除 `closure:repl` 外的许多目标，这些在开发过程中非常有用。

除了 `closure-maven-plugin`，还有数百个 Maven 插件可以帮助自动化各种构建、测试和项目管理活动，以及启用 Maven 项目与外部工具和环境的强化集成。总之，在 Web 上简单搜索 `foo maven plugin` 你就可能找到你感兴趣的 `foo` 的 Maven 插件。

Leiningen

Leiningen^{注 32} 宣传为“一个用于 Clojure 的构建工具，使用时不会让你着急上火”。^{注 33} Maven 和 Ant 对 Clojure 项目最常见的构建需求来说还是相当复杂的，Leiningen 的这一使命很大程度上就是在受挫中诞生的。它的目标是提供一个对常见任务比 Ant 或 Maven 更简单的总工作流。

如果来自 JVM 以外的世界——如 Ruby、Python 或相似的语言——你很可能会觉得 Leiningen 比 Maven 更有吸引力。Leiningen 在底层重用了 Maven 部分代码，但它对

注 31：参见 <http://github.com/talios/closure-maven-plugin>。

注 32：Leiningen，口头上经常称为“lein”，发音却像 LINE-ing-en，尽管另一发音 LINE-in-gen 也不算罕见。

注 33：推荐使用 Leiningen 的最新版本，至少是 1.7.0 版：<http://leinigen.org>。书中所有的 Leiningen 例子已经在 1.7.0 版上测试过，也在最近发布的 2.0 版的预览版上测试过。

Maven 依赖模型提供了更符合 Clojure 习惯的处理办法和更轻量的开发过程。特别是改变或扩展 Leiningen 的构建过程可以完全在 Clojure 里完成，^{注34} 这与构建 Maven 插件所要求的相比是非常令人欣慰的体验。代价是 Leiningen 不提供对 Maven 生态里大量第三方插件和集成选项的支持，不过可用的 Leiningen 插件在不断增长。^{注35}

348 > 下面这个基本的 *project.clj* 文件会让你在简单的 Clojure 项目里开始用 Leiningen：

例8-4：适合简单的Clojure项目的*basic-project.clj*

```
(defproject com.clojurebook/sample-lein-project "1.0.0"
  :dependencies [[org.clojure/clojure "1.3.0"]])
```

`defproject` 是一个宏，给你的项目定义了一个适合 Leiningen 的模型。除了键 - 值对组成了项目配置的主体外，`defproject` 要求它的前两个参数是项目的坐标（用一个 Clojure 符号指定，这里是 `com.clojurebook/sample-lein-project "1.0.0"`，对应于 Maven 写法的 `com.clojurebook:sample-lein-project:1.0.0`）。

`defproject` 提供了用不带命名空间的符号把 group ID 和 artifact ID 设置成同一个值。这常用于项目名就是组织名的开源项目，例如 Ring（一种 Clojure Web 框架）就是给 `ring "1.0.1"` 作为 `defproject` 的前两个参数，产生的 Maven 坐标是 `ring:ring:1.0.1`。



Leiningen 提供了生成新项目骨架的命令，用 `lein new my-project` 将会创建一个新目录 `my-project`，里面包含一个 *project.clj* 文件（坐标是 `my-project "1.0.0-SNAPSHOT"`），以及占位的源文件和测试源文件。

这个基本的 *project.clj* 文件提供了 345 页“Maven”一节里讨论的 Maven 样例 *pom.xml* 大致相同的配置选项。区别在于，Clojure 源文件的根目录是 `src` 而不是 `src/main/clojure`，Clojure 测试源文件目录是 `test` 而不是 `src/test/clojure`。对这样一个简单的项目，两者真正差异是 Leiningen 构建的行为取决于发出的是哪个 `lein` 命令及其顺序，而 Maven 构建对每一构建阶段强制执行一致的顺序和语义。

除了这些差异，Leiningen 的基本工作流与 Maven 的非常相似：

- `lein repl` 将开启一个新的 Clojure REPL，classpath 设为包括项目的所有传递性依赖。
- `lein test` 将运行项目里所有的 Clojure 测试（通常在 `test` 目录里）。
- `lein jar` 执行与 `mvn package` 相同的打包动作，只是不会自动对 Clojure 源文件进行提前编译。
- `lein uberjar` 会生成一个超级 `jar` 包、一个 `.jar` 文件，就像 `lein jar` 所生成的一样。

注 34：对 Leiningen 构建过程的局部小修改的例子见 351 页“构建混合源文件项目”一节。

注 35：Leiningen 插件一个不完整的列表可以在 <https://github.com/technomancy/leiningen/wiki/plugins> 找到。

但项目的所有传递性依赖都“解包”到里面了。超级 jar 经常用于简单的部署，全部应用程序作为单个文件发送，可以用单个 java 调用来执行。^{注 36}

- `lein compile` 将基于 `project.clj` 里的 :aot 配置提前编译项目的所有 Clojure 源文件。我们将在下一节 349 页的“提前编译的配置”里为提前编译配置 Leiningen。
- `lein pom` 将生成一个与 Maven 兼容的 `pom.xml` 文件，包含项目的 `project.clj` 文件里的项目和依赖信息。部署到远程 Maven 库或安装到本地的 Maven 库用的就是这个 `pom.xml` 文件。
- `lein deps` 确保项目指定的所有依赖都可用，必要时下载下来。这通常是自动完成的（即只要 :dependencies 向量发生变化就会进行下载）。



如你所见，Leiningen 和 Maven 描述依赖的写法在语法上差别很大，不过传达的信息是相同的，这个 Leiningen 依赖：

```
[org.clojure/clojure "1.3.0"]
```

与下面这个 Maven 依赖是完全等价的：

```
<dependency>
  <groupId>org.clojure</groupId>
  <artifactId>clojure</artifactId>
  <version>1.3.0</version>
</dependency>
```

因为它更简略，也因为它在 Clojure 社区里用得更广泛（例如项目 `README` 文件更可能提供 Leiningen 风格的依赖向量而不是 Maven 里的 `<dependency>` XML 片段），从现在起，在例子中引用 Clojure 函数库时，本书都用 `Leiningen` 的写法提供相应的依赖。

提前编译的配置

除了定义依赖和打包构件，在 Clojure 构建过程中需要做的最常见事情是提前编译 Clojure 源文件。不过在进行前应该考虑提前编译的动机并权衡利弊。请先消化 337 页“提前编译”一节再决定你的项目是否需要提前编译，很多时候不需要提前编译。

无论如何，Leiningen 和 Maven 都提供了直接的方法启用、禁用和配置提前编译。

Leiningen。 默认情况下，Leiningen 不会提前编译项目里的 Clojure 源文件。要配置成那样，需要在项目的配置里加上 :aot 槽，槽的值可以是：

^{注 36} 例如，`java -cp <path-to-uberjar> com.foo.MainClassName` 或 `java -cp <path-to-uberjar> clojure.main -m com.foo.namespace-with-main-fn`。后者常见于 Clojure 项目，不需要提前编译，详情参见 `clojure.main/main` 文档。

- :all，这会提示 Leiningen 提前编译项目里所有的命名空间。
- 命名空间的向量，指定项目里哪些命名空间应该编译。

一旦加入了 :aot 配置，运行 `lein compile` 将使项目的 Clojure 命名空间被提前编译。

Maven。在 `clojure-maven-plugin` 里提前编译默认是启用的，至少在使用 `clojure` 打包方式时是这样的，如例 8-3 所示。`clojure` 打包把 `clojure-maven-plugin` 的编译目标绑定到 Maven 的 `compile` 阶段，因而提前编译在 Maven 的默认的 Java 编译完成后运行。

如果不用 `clojure` 打包，则需要明确地配置这个阶段的绑定：

```
<plugin>
  <groupId>com.theoryinpractise</groupId>
  <artifactId>clojure-maven-plugin</artifactId>
  <version>1.3.8</version>
  <configuration>
    <warnOnReflection>true</warnOnReflection>
    <temporaryOutputDirectory>false</temporaryOutputDirectory>
  </configuration>

  <executions>
    <execution>
      <id>compile-clojure</id>
      <phase>compile</phase>
      <goals>
        <goal>compile</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

这里我们也看到与提前编译最相关的两个配置参数：`warnOnReflection` 和 `temporaryOutputDirectory`。这两个参数分别控制着 *warn-on-reflection* 是否启用、提前编译的 `class` 文件是保存到默认的 `classes` 输出目录还是一个临时目录。

提前编译用于健康检查。即使不想分发提前编译的 `class` 文件，在构建过程中包括提前编译（同时忽略生成的 `class` 文件）可以对 Clojure 项目的代码进行健康检查，这也是有用的。因为提前编译要求载入代码，所有引用到的函数库、命名空间、`var` 和 Java 类都必须被解析、载入，如果这些引用有任何问题，提前编译这一步就会发现它。^{注37}

`clojure-maven-plugin` 为这种情景做了准备，只用在插件的 `<configuration>` 元素上加上 `<temporaryOutputDirectory>true</temporaryOutputDirectory>`，提前编译结果就

注 37： 把代码载入 REPL 可以完成同样的检查，但总的说来，与在交互式、很少重复的 REPL 会话里运行相比，确保这样的质量检查在可重复的构建中定期执行是更好的作法。

会指向一个临时目录（随后被删除）。这保证了执行提前的健康检查，而结果又不会进入打包的发布版里。

Leiningen 目前还没有提供简便的方法既执行提前编译，在打包项目的构件时又忽略产生的 class 文件。目前最好的选择是调用 `lein compile`（以执行提前编译），在用 Leiningen 为项目分发打包时先调用 `lein clean`。

同样，可以在提前编译的过程中启用 `*warn-on-reflection*`，这会让 Clojure 对每个互操作反射或参数类型不匹配的情形发出警告。^{注38} 要启用这些警告，在 `clojure-maven-plugin` 的配置中加入 `<warnOnReflection>true</warnOnReflection>` 元素，或者把 `:warn-on-reflection true` 加到 Leiningen 的 `project.clj` 文件中。

构建混合源文件项目

如果想把 Clojure 加到已有的项目里，就需要特别注意这样的混合项目不同部分的编译顺序。^{注39} 这非常简单，如果 Java 源代码需要同一个项目里 Clojure 类型定义形式生成的具名类，就需要先编译 Clojure 源代码再编译 Java 源代码。同样，如果 Clojure 源代码需要引用 Java 源代码定义的类，Java 源代码就需要先编译。



如果 Java 代码并不使用同一个项目里的 Clojure 代码，或者虽然 Java 代码与 Clojure 代码有互操作但并没有引用任何 Clojure 定义的类型，那么与混合源代码项目相关的这些构建上的考虑就与你无关。^{注40}

无论如何，都需要在构建过程中确定编译步骤的顺序，以反映项目内语言间的依赖关系。³⁵² Leiningen 和 Maven 都直接提供了这样做的办法。

Maven。`clojure-maven-plugin` 默认让 Maven 的 Java 编译先于 Clojure 代码库的提前编译。把 `clojure-maven-plugin` 编译目标的执行绑定到 `compile` 前的某个 Maven 阶段，如 `process-resources`，是可以改变默认设置的：

```
<plugin>
  <groupId>com.theoryinpractise</groupId>
  <artifactId>clojure-maven-plugin</artifactId>
  <version>1.3.8</version>
  <executions>
```

注 38：关于互操作类型提示的详情参见 366 页“为了效率进行类型提示”一节，关于 Clojure 能够抛出参数类型不匹配的警告的进一步情况，参见 440 页“类型错误与警告”一节。

注 39：为了清晰起见，这里假设混合项目只包含 Java 为非 Clojure 语言。如果是 Clojure/JRuby 项目或者 Scala/Clojure 项目，类似的建议仍然适用。

注 40：关于如何在 Java 里使用 Clojure 定义的功能而不需在 Clojure 里定义新的 Java 引用类型的详情，参见 385 页“在 Java 里使用 Clojure”一节。

```
<execution>
  <id>clojure-compile</id>
  <phase>process-resources</phase>
  <goals>
    <goal>compile</goal>
  </goals>
</execution>
</executions>
</plugin>
```

Leiningen。Leiningen 默认也是在配置的提前编译前用 `javac` 编译 Java 代码，只要用 `defproject` 的 `:java-source-path` 槽定义了在什么位置能找到 Java 代码。

Leiningen 让你逆转这个顺序，但是通过编程手段而不是像在 Maven 里那样修改配置。首先在 `defproject` 的配置里，`:java-source-path` 应该保持未定义，否则，Leiningen 的默认编译顺序会继续生效。我们想做的是改变 `compile` 任务的行为，以便 `javac` 在任务的默认活动结束后运行。Leiningen 捆绑了 `robert-hooke` 函数库，^{注41} 后者在 `project.clj` 文件里提供了简洁的方法做到这点：

```
(defproject com.clojurebook/lein-mixed-source "1.0.0"
  :dependencies [[org.clojure/clojure "1.3.0"]]
  :aot :all)

(require '(leinigen compile javac)) ❶

(add-hook #'leinigen.compile/compile
  (fn [compile project & args]
    (apply compile project args)
    (leinigen.javac/javac (assoc project :java-source-path "srcj")))) ❷
```

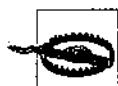
❶ 首先请求关键的 Leiningen 命名空间。

- 353 ❷ 想给 Leiningen 的 `compile` 任务的驱动函数加一个钩子，为此目的，`robert-hooke` 的 `add-hook` 函数需要访问这个驱动函数的 var，`#'leinigen.compile/compile`。
- ❸ 给钩子函数提供了原来占据 `#'leinigen.compile/compile` var 的 `compile` 函数，目前的 `project` 配置，以及原本要传递给原 `compile` 函数的所有其他参数。`add-hook` 返回后，钩子函数将会完全控制 `compile` 任务的实现。
- ❹ 首先，控制交给原来的 `compile` 函数，这样通常的 Clojure 提前编译可以运行。

注 41： `robert-hooke` 是一个 Clojure 函数库，为 Clojure 提供了面向方面编程（AOP）的超集；关于它是如何解决在 Java 里通常要用 AOP 框架解决的问题，请参见 466 页“面向方面的编程”一节里的描述。

- ⑤ 需要给 `:java-source-path assoc` 进一个值，以便 `javac` 知道在哪里可以找到 Java 源文件。这通常包含在权威的项目模型里（由 `defproject` 在顶层 `project.clj` 文件里定义），不列在那里可以确保 `compile` 不会在编译项目的 Clojure 源代码前试图运行 `javac` 命令。

这些就位后，运行 `lein compile` 将调用钩子函数，从而运行通常的 Clojure 提前编译，随后通过使用 `javac` 编译根在 `src/j` 目录的 Java 代码。



不论要如何组织项目里的代码，都应该绝对避免交织的源代码依赖。例如，Java 代码用了 Clojure 里定义的类型，而那是实现 Java 里的接口，都在同一个项目里，这就形成了交织依赖。这样的拓扑结构是可以解决的（也许在 Maven 里比其他地方更容易，因为它有能力使用任何数量的不同源文件根目录和任何数目的编译目标执行），不过可以安全地认定那是设计糟糕的症状。

最后的思考

有效地组织代码和软件项目本身就是一个应用领域。关于如何以 Clojure 的方式组织和构建项目，从而让你专注于学习 Clojure 并用它来完成真正的工作，希望我们为你提供的内容足以让你的项目运转起来。

Java 及 JVM 互操作

许多语言提供了专为自己建立的运行时，常见的例子包括 Python (CPython)、Ruby (MRI) 和 Java (JVM)。^{注1} 与此相反，Clojure 基本上是一门“寄生”语言，意即它的目标是已有的运行时，即 JVM。^{注2} 这意味着不需要重新实现许多基础设施（如垃圾收集、即时编译、线程、图形上下文，等等）和各种各样的函数库（从基本的字符串处理到像密码学函数这样深奥的东西），Clojure 简单地重用在 JVM 上已有的所有工作。

除了单纯节约了实现的时间，以成熟平台为目标对 Clojure 程序员来说也是有利的：

- JVM 核心设施和函数库生态系统是由显要的工程组织支持的。这意味着那些一般是经过充分测试、已经被广泛使用、大力优化过的，因而性能表现符合多数实际用户的典型需求。
- 以 JVM 为基础设施意味着有标准的互操作途径。用一种语言写的程序可以调用其他语言写的函数库所提供的功能，这些语言都在同一个运行时里运行，而 Java 接口和 Java 对象模型就是大家的通用语。
- Java 及更广大的 JVM 社区非常巨大，这保证了有大量的函数库针对各个领域、有大量的支持材料、有一支对平台有深刻理解的庞大开发者队伍。
- 有各种工具可用于支持软件开发的所有阶段，包括开发环境、构建工具、剖析器、调试器和运营支持。

要从 Clojure 取得最大收获，你可能需要对 Clojure 与寄主的关系有透彻的理解，以便利用 JVM 最好的部分、针对 JVM 的函数库以及前期在上面的投资。

注 1： 不过从 Java 7 开始，JVM 的变化完全是为像 Clojure 这样的 Java 之外的语言而设的——明确地使它成为了多语言运行时。

注 2： 还有其他的 Clojure 实现，是针对其他平台的，请参见 583 页“(dissoc Clojure ‘JVM’)”一节。

JVM 是 Clojure 的基础

你可能会以为 Clojure 作为一个寄生语言意味着它在某些方面会与 JVM 显著不同——就像某些寄生的脚本环境建立起的沙箱那样，你轻易不能摆脱。那绝对是假话。Clojure 在很多方面重用了 JVM 的基础设施，仅举几个最明显的例子：

- Clojure 的字符串是 Java `String` 类型的。
- Clojure 的 `nil` 是 Java 的 `null`。
- Clojure 的数字是 Java 的数字。^{注3}
- Clojure 的正则表达式是 `java.util.regex.Pattern` 的实例。
- Clojure 的数据结构是用 `java.util.*` 集合类接口的只读部分实现的，因而 Clojure 的映射实现了 `java.util.Map`，向量、序列和列表实现了 `java.util.List`，而集合实现了 `java.util.Set`。
- Clojure 函数实现了 `java.lang.Runnable` 和 `java.util.concurrent.Callable`，使得 Clojure 函数可以轻松集成到已有的函数库和框架里，这些库和框架要求实现 Java 这些核心接口。
- 在语法和抽象的背后，Clojure 的函数调用就是 Java 的方法调用，因而 Clojure 的函数和函数调用没有什么特别的运行时开销。
- Clojure 从来不是解释执行的，总是在运行前编译成高效的 JVM 字节码，即使在 REPL 这样的交互环境下也一样。
- 在 Clojure 里调用 Java 的 API 在语义和机制上与在 Java 里调用这些 API 是同样的操作。
 - Clojure 的函数编译成 Java 类。
 - Clojure 的 `defrecord` 和 `deftype` 形式编译成含有通常 Java 字段的 Java 类。
 - 用 `defprotocols` 定义的协议生成对应的 Java 接口。

这样深层次的整合意味着在 Clojure 里使用 Java 函数库^{注4} 通常不需要特别的包装代码、转换或其他花招，与等值的 Java 代码相比也没有性能上的损失，在 Java 里使用 Clojure 也一样。

而且，由于 Java 标准函数库的深度和 JVM 函数库周围聚集的社区的规模，对于每种潜在需求很少会没有 JVM 函数库（有时还会有多个相互竞争的库）。对于来自那些常常需要用首选语言重新实现功能的人来说，这通常是一个可喜的变化。

Java 类、方法和字段的使用

Clojure 提供了一些简单的形式用于与寄主的类、方法和字段互操作，这使得在 Clojure

注 3： 这里有一点微妙的小问题，详情见表 11-1。

注 4： 别忘了，Clojure 自身也是 Java 库。请参见 342 页的“Clojure ‘只是’ 另一个依赖”。

里使用 Java 函数库非常自然，肯定比等价的 Java 代码要简练得多。

表9-1：Clojure的互操作形式及其对应的Java形式^{*}

操作	Clojure 形式	对应的 Java 形式
创建类 <code>ClassName</code> 的新实例	<code>(ClassName.) (ClassName. arg1 arg2 ...)</code>	<code>new ClassName() new ClassName(arg1, arg2, ...)</code>
调用 <code>object</code> 的一个实例方法	<code>(.methodName object)</code>	<code>object.methodName()</code>
	<code>(.methodName object arg1 arg2 ...)</code>	<code>object.methodName(arg1, arg2, ...)</code>
调用类 <code>ClassName</code> 的静态方法	<code>(ClassName/staticMethod)</code>	<code>ClassName.staticMethod()</code>
<code>staticMethod</code>	<code>(ClassName/staticMethod arg1 arg2 ...)</code>	<code>ClassName.staticMethod(arg1, arg2, ...)</code>
访问类 <code>ClassName</code> 的 <code>FIELD</code>	<code>ClassName/FIELD</code>	<code>ClassName.FIELD</code>
静态字段 <code>FIELD</code>		
引用 <code>Class</code>	<code>ClassName</code>	<code>ClassName.class</code>
访问对象 <code>object</code> 实例	<code>(.field object)</code>	<code>object.field</code>
字段 <code>field</code> 的值		
给对象 <code>object</code> 的实例	<code>(set! (.fieldName object) 5)</code>	<code>object.fieldName = 5</code>
字段 <code>field</code> 赋值为 5		

* 我们在这里撇了个小说：其实与寄主只有两个基本的互操作形式：`.`（英文句点）和 `new`，前者提供方法调用和字段访问，后者调用构造器。除了 `set!`，以上所有形式都扩展为使用 `.` 和 `new` 的形式。这种语法糖在 44 页“和 Java 的互操作：`.` 和 `new`”一节里有解释。

例 9-1 和例 9-2 展示了 REPL 下几个回合的交互结果，每种互操作形式都利用了 Java 标准函数库里的核心类。

例9-1：用Java函数库提取Web页面

```
(import 'java.net.URL)           ❶
:= java.net.URL
(def cnn (URL. "http://cnn.com")) ❷
;#'user/cnn
(.getHost cnn)                  ❸
;= "cnn.com"
(slurp cnn)                     ❹
;= "<html lang=\"en\"><head><title>CNN.com..."
```

358

- ❶ `import` 宏把指定类引进到当前命名空间，这里是平台标准的 `URL` 类。
- ❷ 这里用一个字符串类型的参数创建了一个 `URL` 实例，结果保存在一个 `var` 里。构造器的使用与 Java 里的 `new URL("http://cnn.com")` 等值。
- ❸ 调用 `getHost` 方法返回我们预期的结果，这与 Java 里的 `url.getHost()` 等值。

- ❶ Clojure 有一个函数 `slurp`, 它会返回好几种类型的对象的字符串内容, 包括 `java.io.File`、`java.net.Socket` 和字节数组, 等等^{注5}。这里我们把 URL 实例传给 `slurp`, 返回这个 URL 页面的字符串内容。

使用静态方法和字段同样容易：

例9-2：Java的静态方法和字段的用法

```
Double/MAX_VALUE
:= 1.7976931348623157E308
(Double/parseDouble "3.141592653589793")
:= 3.141592653589793
```

设计这些互操作形式要考虑的关键因素是要完全与函数位置一致：^{注6}执行的“操作”总是每个形式里的第一个符号。因而，互操作形式不仅与 Java 自身的方法和构造器调用在功能上是平行的，而且完全与 Clojure 的语法一致。这使得读和写互操作的代码与使用常规 Clojure 函数的非互操作的代码一样自然，也使得互操作形式可与 Clojure 自身的东西（如 `->` 系列的宏）一起用。例如，下面是一个地道的 Clojure 函数，给定十进制数的字符串，返回全大写的十六进制的字符串：

```
(defn decimal-to-hex
  [x]
  (-> x
    Integer/parseInt
    (Integer/toString 16)
    .toUpperCase))
:= #'user/decimal-to-hex
(decimal-to-hex "255")
:= "FF"
```

- 359> 串行宏 `->` 把每个形式的结果作为第一个参数传给后面的形式（顺带把单个符号转换成单元素的列表），返回值是最后那个形式的结果。因而 `decimal-to-hex` 函数的主体与 `(.toUpperCase (Integer/toString (Integer/parseInt x) 16))` 等价，也与下面的 Java 代码等价：

```
public String stringToHex (String x) {
    return Integer.toString(Integer.parseInt(x), 16).toUpperCase();
}
```

注 5： `slurp` 其实依赖于 `clojure.java.io/reader` 函数来为所有这些不同类型的对象返回一个 `java.io.Reader`——在这个例子里，用了不同的 Java 互操作形式来打开指向 URL 的一个 `java.net.Socket`，从 `socket` 获得一个 `java.io.InputStream`，后者又被一个 `Reader` 包装起来。然后 `slurp` 取得控制，简单地从 `Reader` 读出字符串数据。

注 6： 关于函数位置请参看第 7 页的“表达式、操作符、语法和优先级”一节。

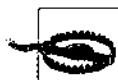
至少在我们看来，使用 `->`（以及类似的宏如 `->>注7`）让代码容易读得多（可以看成一个线性处理的渠道，消除了表达式忽前忽后而求值顺序也不是简单的从左到右的问题）而且仍然与互操作形式完全兼容，这多亏了函数位置使用一致。

访问对象字段。在 Java API 里，公共字段很稀罕，可改变的公共字段更是这样。不过 Clojure 提供了易用的方式访问公共字段，还有一个 `set!` 特别形式来赋值给它。^{注8}

例9-3：访问、设置Java对象的字段

```
(import 'java.awt.Point)
;; java.awt.Point
(def pt (Point. 5 10))
;; #'user/pt
(.x pt)
;; 5
(set! (.x pt) -42)
;; -42
(.x pt)
;; -42
```

注意，`set!` 用了与获取字段的值相同的语法——这里是 `(.x pt)`——来引用要赋值的字段。



Java 的可变性（陷阱门 | 逃生门）

360

Clojure 鼓励使用不可变的值作为好的默认情况（这在第 2 章里已经讨论过），但与 Clojure 共享 JVM 的主要语言则不是这样的。Java 的集合类以及一般情况下，Java 的默认情况是鼓励可变的对象状态。既然可以在 Clojure 里相当自然地使用 Java 的函数库、类和对象，不需要太多麻烦就可以选择可变状态。这可以成为一个有用的逃生门（特别是在你需要的功能只作为 Java 函数库提供时）或者成为一个讨厌的陷阱门（如果在处理可变 Java 对象时不够小心）。无论如何，在 Clojure 里使用 Java 对象时头脑要清醒，尤其是在你开始欣赏并依赖 Clojure 里合情合理的默认值以后。

便利的互操作工具

使用 Java 函数库的绝大部分情况涉及表 9-1 里列出的核心互操作形式。不过有两三个常用的工具你应该知道：

`class`

返回参数的 `java.lang.Class`，例如 `(class "foo") ⇒ java.lang.String`。

注 7：关于 `->` 和 `->>` 的详细解释见 259 页“深入 `->` 和 `->>`”一节。

注 8：`set!` 也用于一些其他需要直接改变一个可变位置的情形，见 45 页“状态修改：`set!`”一节。

`instance?`

一个谓词函数，如果第二个参数是第一个参数命名的 Class 的实例，就返回真，如 `(instance? String "foo")` \Rightarrow true。

`doto`

一个宏，调用一系列的嵌套形式，把 `doto` 的第一个参数作为第一个参数传给后面所有的形式。

`class` 和 `instance?` 的用途和用法相当明显，不过对于 `doto` 有必要做进一步解释。

Java 世界里的很多类是可改变的，并且经常要求在构造函数所提供的处理之外进行初始化。一个很好的例子是 `java.util.ArrayList`，经常需要用一些初始数据去填充：

```
ArrayList list = new ArrayList();
list.add(1);
list.add(2);
list.add(3);
```

当然，Clojure 给自己的向量提供了字面量，^{注9}一些不幸的情形下已有的 Java API 可能要求用 `ArrayList`（而不是接受任何 `java.util.List`，否则就可以直接传递一个 Clojure 向量作为参数）。这时，可以用 `let` 来创建 `ArrayList`，合适地填充数据，然后返回 `ArrayList` 引用：

例9-4：“手工”填充 `ArrayList`

```
(let [alist (ArrayList.)]
  (.add alist 1)
  (.add alist 2)
  (.add alist 3)
  alist)
```

这是可以工作的，不过和 Java 同样累赘。`doto` 用在这里要好多了：^{注10}

例9-5：用 `doto` 来简练地给 `ArrayList` 填充数据

```
(doto (ArrayList.)
  (.add 1)
  (.add 2)
  (.add 3))
```

这里完成与例 9-4 完全同样的事情，不过显然简练得多。`doto` 的用处在需要对单个对象执行更大量的操作时就更明显了。例如：

注9：注意，Clojure 的向量与 `java.util.Vector` 没任何关系。关于 Clojure 数据结构的完整讨论，参见第 3 章。

注10：`ArrayList` 是大家熟悉的，可以作为一个简单的 `doto` 例子的合理基础，不过可以直接用一个 Clojure 集合类来填充它，如 `(ArrayList. [1 2 3])`。

例9-6：用doto来操作java.awt.Graphics2D上下文

```
(doto graphics
  (.setBackground Color/white)
  (.setColor Color/black)
  (.scale 2 2)
  (.clearRect 0 0 500 500)
  (.drawRect 100 100 300 300))
```

上例对 Graphics2D 对象应用一系列操作，结果在图形上下文里画了一个黑方框在一个白色方框里，最后 doto 形式返回这个对象。

Clojure 的主要惯用法是对值进行操作，而不是一步一步地改变引用（引用自身固定不变）。doto 通过提供简练的方式从语法上绕过任何过程式的初始化或其他有副作用的操作，并且隐式返回这些操作的对象，以便后面的 Clojure 代码可以把它当做值来看待，因而 doto 有助于在互操作寄主的可变、含状态的对象时也采用这一惯用法。

异常与错误处理

362

Clojure 完全重用了 JVM 的异常处理机制，^{注11} 采纳了人们熟悉的 try/catch/finally/throw 惯用语。因而对使用过 Java、Ruby 和 Python 等同样采用这种惯用法的语言的人来说，Clojure 这种错误处理模式并不陌生。

因此，例 9-7 里的代码片段在错误处理上是等价的：

例9-7：把字符串用Java、Ruby和Clojure语言解析为整数，包括错误处理

```
// Java
public static Integer asInt (String s) {
    try {
        return Integer.parseInt(s);
    } catch (NumberFormatException e) {
        e.printStackTrace();
        return null;
    } finally {
        System.out.println("Attempted to parse as integer: " + s);
    }
}

# Ruby
def as_int (s)
begin
    return Integer(s)
rescue Exception => e
    puts e.backtrace
```

注 11：有很多 Clojure 函数库提供了错误处理扩展，比这里描述得要广，特别是 Slingshot 提供了更为复杂的模型和错误状态处理：<https://github.com/scgilardi/slingshot>。

```
ensure
    puts "Attempted to parse as integer: " + s
end
end

; Clojure
(defn as-int
[s]
(try
(Integer/parseInt s)
(catch NumberFormatException e
(.printStackTrace e))
	finally
  (println "Attempted to parse as integer: " s))))
```

解析整数是相当普通的操作，不过它可以清楚表明 Clojure 的 try、catch 和 finally 形式在语义和功能上与 Java 的以及如像 Ruby 里对应的形式（begin、rescue 和 ensure）相似。

363 ➤ try

限定异常处理形式的范围，可以包括任何数量的 catch 形式和一个可选的 finally 形式，前面都可以有任何数量的表达式代表代码的“正常路径”。^{注12} 如果是走正常路径，在 try 里面但在 catch 或 finally 形式前面的最后一个表达式决定 try 形式的结果。

catch

指定如果在 try 形式的主体执行过程中抛出指定类型的异常（即 `java.lang.Throwable` 或它的任何子类）将要转往的代码。抛出的异常将与异常类型后面提供的名称绑定。激活的 catch 形式里的最后的表达式决定包含它的 try 形式的返回结果。可以指定任意数量的 catch 形式。

finally

指定控制流从 try 形式退出前要执行的代码，不论退出的性质是什么（即使 try 形式主体里的某些代码抛出了未被捕获的异常，仍然会执行 finally 形式里的表达式）。finally 形式不会影响 try 形式返回的结果，因而只是在 try 形式完成后需要进行某些副作用的动作时它才有用处。唯一的小问题是，从 finally 主体抛出的异常将会取代从对应的 try 形式的主体抛出的任何异常（在 Java 里也是这样）。

这些就是 JVM（因而也是 Clojure 的）通用错误处理功能的基础。

抛出异常。Clojure 代码可以用 throw 形式表明异常，这与 Java 的 throw、Ruby 和

注 12： 值得指出的是，Clojure 的 catch 和 finally 形式必须包括在一个 try 形式里，这样使得 try（或者如 Ruby 的 begin）与 catch 和 finally 的语义关系明确，不过几乎每个其他语言在语法上都把 finally 当成与 try 同级别的“顶层”。

Python 的 `raise` 完全类似：

```
(throw (IllegalStateException. "I don't know what to do!"))
;= #<IllegalStateException java.lang.IllegalStateException: I don't know what to do!>
```

在 Python 里可以合乎语法地 `raise` 任何类或类的实例，在 Ruby 里甚至可以 `raise` 字符串，但 Clojure 的 `throw` 必须接受一个扩展 `java.lang.Throwable` 的类或它的子类：

```
(throw "foo")
;= #<ClassCastException java.lang.ClassCastException:
;=     java.lang.String cannot be cast to java.lang.Throwable>
```

重用已有的异常类型。Java 代码里常见的一个惯用法，但在 Clojure 代码里却非常罕见的是定义新的异常类型。在需要抛出异常时，地道的 Clojure 代码抛出一个标准类型的异常。Java 标准类库里有 350 多种异常类型。这么多的异常类型在语义上还不足以描述手头的错误情况？虽然有可能发生，但很少会碰到这样的情景。公道地说，这些“核心”异常类型可以合理地满足 90% 的使用情况：

- `java.lang.IllegalArgumentException`
- `java.lang.UnsupportedOperationException`
- `java.lang.IllegalStateException`
- `java.io.IOException`

364

这是总的规则，不过有时候也需要定制的异常类型。如果身处这种情景，378 页“[定义自定义的异常类型](#)”一节里定义定制的异常类型的例子对你可能有帮助。

摆脱检查型异常

Clojure 虽然运行在 JVM 上，但它并没有继承 Java 的检查型异常。这些是由 Java 方法声明为可能抛出的异常类型，Java 编译器要求调用这种方法的一方要么捕获并处理抛出的检查型异常，要么自己也声明可能抛出这些异常。检查型异常长期以来一直是 Java 生态系统里的争议话题，^{注 13} 大致的共识是检查型异常一般使得 Java 代码更冗长、更难于维护。

幸亏在 Clojure 代码里调用 Java 方法时 Clojure 并不受制于所声明要抛出的检查型异常的限制，因为检查型异常是 Java 编译器制造的，根本就不存在于运行时的 JVM 里。^{注 14} 因而可以用一个指定了可能抛出检查型异常 `java.io.IOException` 的方法来创建临时文件：

```
(File/createTempFile "closureTempFile" ".txt")
```

既不需要包含在 `try/catch` 表达式里，也不需要像 Java 那样用 `throws` 声明来表明代码

注 13： <http://www.mindview.net/Etc/Discussions/CheckedExceptions> 是探索这个话题不错的的地方。

注 14： 碰巧的是，与 Java 的泛型非常像，这多亏类型擦除。

可以抛出一个 `IOException`。当调用带检查型异常声明的方法时，这显然让 Clojure 比等价的 Java 代码更简练。

with-open : finally 的挽歌

典型的、最常见的 `finally` 用法用于确保资源合理管理。例如，下面是 Java 里的一个静态方法，用于向文件附加一些文本：

例9-8：在Java里通过finally进行手工资源管理来向文件附加内容

```
public static void appendTo (File f, String text) throws IOException {
    Writer w = null;
    try {
        w = new OutputStreamWriter(new FileOutputStream(f, true), "UTF-8");
        w.write(text);
        w.flush();
    } finally {
        if (w != null) w.close();
    }
}
```

注意 `finally` 代码块，我们有条件地关闭在前面的 `try` 代码块里打开的 `Writer`。这一模式（或者说它太容易被滥用）一直是与文件、socket、数据库以及其他需要明确管理的资源打交道的很多程序里错误的根源。由于这个原因，Java 7 引入了 `try-with-resources` 语句，^{注15} 它会在控制退出其范围时自动关闭如文件句柄等资源，与 Python 中的 `with` 非常相似：

例9-9：在Java 7里用try-with-resources向文件附加内容

```
public static void appendTo (File f, String text) throws IOException {
    try (Writer w = new OutputStreamWriter(new FileOutputStream(f, true), "UTF-8")) {
        w.write(text);
        w.flush();
    }
}
```

Clojure 提供了等价的 `with-open` 形式来确保控制退出范围前关闭资源。^{注16} 下面是 `appendTo` 方法翻译成地道的 Clojure，用 `with-open` 形式来确保 `Writer` 被正确关闭：

例9-10：在Clojure里通过with-open进行自动资源管理向文件附加内容

```
(require '[clojure.java.io :as io])
```

注 15：如果你还不熟悉，可以看 <http://download.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html>。

注 16：我们不会放过提一下这个的机会：`with-open` 是用 9 行 Clojure 代码实现的宏，你也可以轻易地自己写出来——与之形成鲜明对比的是，缺乏宏的语言等待戈多式体验，一直等待语言维护者发布改进的版本。在第 5 章里你可以读到有关宏及宏能做什么的内容。

```
(defn append-to
  [f text]
  (with-open [w (io/writer f :append true)] []
    (doto w (.write text) .flush)))
```

- 虽然可以自己手工创建 `OutputStreamWriter` 和 `FileInputStream`, 不过 `writer` 辅助函数太方便而没这个必要了。

也可以在一个 `with-open` 形式里绑定多个资源, ^{注17} 再控制从 `with-open` 形式退出时每个都会关闭。下面的例子简单实现了复制函数, 显示了这一点:

<366

例9-11: 在 Clojure 里通过 `with-open` 的自动资源管理复制文件

```
(defn copy-files
  [from to]
  (with-open [in (FileInputStream. from)
             out (FileOutputStream. to)]
    (loop [buf (make-array Byte/TYPE 1024)]
      (let [len (.read in buf)]
        (when (pos? len)
          (.write out buf 0 len)
          (recur buf))))))
```

这是使用 `with-open` 的好例子, 不过最好的选择是用 `clojure.java.io` 命名空间^{注18} 里的 `copy` 函数 (以及其他 I/O 工具) 而不是选择像这样的简单实现。

为了效率进行类型提示

你可能已经注意到有些代码例子使用一种引用 Java 类名的语法, 如这里的 `^String`:

```
(defn length-of
  [^String text]
  (.length text))
```

这里的 `^ClassName` 语法定义的是类型提示, 给 Clojure 明确指出一个表达式、`var` 的值或具名绑定的对象类型。

典型的类型提示由 Clojure `reader` 展开, 纳入提示后面的值的元信息。因而, `^String text` 这个有提示的绑定形式被 `reader` 扩展成等值的表达式 `^{:tag String} text`, 这会对带元信息 `{:tag String}` 的符号 `text` 求值。Clojure 内部使用术语 “tag” 来表示类型。

这些提示只是被编译器用来避免发出反射性的互操作调用, 除此之外提示是不必要的。

注 17: 任何对象的类如果提供了空的 `close` 方法都可在这儿用。这会与在 `java.lang.Closeable` 接口里定义的 `close` 方法匹配, 但因为 Clojure 是一门动态语言, `with-open` 使用的资源不需要实际实现这个接口。这在许多其他语言中称为 “duck typing”。

注 18: 详情请参见 <http://clojure.github.com/clojure/clojure.java.io-api.html>。

因而，如果代码并不包括互操作调用，就没有必要包括类型提示；

367 (defn silly-function
[v]
(nil? v))

为上面的 v 提供的任何提示都用不上。



函数参数或返回值的类型提示都不是类型签名声明：不会影响函数能接受或返回的类型。类型提示唯一的作用是让 Clojure 用编译时生成的代码调用 Java 方法、访问 Java 字段，而不是在运行时使用反射来搜索能匹配所涉及的互操作形式的方法或字段，后者要慢得多。因而，如果一个提示没有关于互操作的信息，它就什么也不做。例如，这个函数提示它的参数是 `java.util.List`，但它也能接受任何类型的参数：

```
(defn accepts-anything  
[^java.util.List x]  
x)  
:= #'user/accepts-anything  
(accepts-anything (java.util.ArrayList.))  
:= #<ArrayList []>  
(accepts-anything 5)  
:= 5  
(accepts-anything false)  
:= false
```

这与签名声明形成对比，但只适用于原始类型的参数和返回类型。我们在 438 页“声明函数接受和返回原始类型”一节里讨论原始类型声明。

避免反射调用是保证计算密集型代码最大性能的关键。在实践中，很少需要类型提示来完全避免反射，因为 Clojure 编译器基于已知的字面量类型、构造函数的调用和方法的返回类型提供了类型推导。

作为示例，下面给某些代码加上类型提示以优化它。下面这个函数返回一个所提供的 `String` 的大写：

例9-12：没有提示的大写函数

```
(defn capitalize  
[s]  
(-> s  
(.charAt 0)  
Character/toUpperCase  
(str (.substring s 1))))
```

这个实现工作得不错，不过我们也许想提高点速度。

例9-13：计时测量大写“foo”10万次

```
(time (doseq [s (repeat 100000 "foo")]
  (capitalize s)))
; "Elapsed time: 5040.218 msecs"
```

在这种情况下，如果你猜测类型提示可能会产生一些好处，打开 Clojure 编译器的反射警告是有益的，这样做会显示哪儿发出了反射调用：^{注19}

例9-14：给例9-12的大写函数加上反射警告

```
(set! *warn-on-reflection* true)
:= true
(defn capitalize
  [s]
  (-> s
    (.charAt 0)
    Character/toUpperCase
    (str (.substring s 1))))
; Reflection warning, NO_SOURCE_PATH:27 - call to charAt can't be resolved.
; Reflection warning, NO_SOURCE_PATH:29 - call to toUpperCase can't be resolved.
; Reflection warning, NO_SOURCE_PATH:29 - call to substring can't be resolved.
:= #'user/capitalize
```

可以看到，这里反射地发出了3个互操作调用。可以加上一个类型提示来处理（注意添加的^{^String}）：

```
(defn fast-capitalize
  [^String s]
  (-> s
    (.charAt 0)
    Character/toUpperCase
    (str (.substring s 1))))
```

这会去除所有3个反射调用。仅一个类型提示是如何影响到3个反射调用的呢？这就是 Clojure 编译器的类型推理起作用的地方：

1. 由 let 绑定的名称 s 显式地类型提示为 String，因而……
2. .charAt 调用可以编译成对 String.charAt 的直接调用。编译器知道这个方法返回一个 char 类型的值，因而……
3. 它可以正确地选择 char 变体的 Character.toUpperCase 方法（而不是 int 版）。
4. 最后，在编译 .substring 互操作方法调用时，编译器再次引用对 s 的显式 String 类型提示来决定选择 String.substring 方法。

这对修改后的 fast-capitalize 函数效率有什么影响呢？来看一看：

^{注19} 这里显示的技巧在 REPL 里也是有用的，但如果想要，你也可以从构建过程得到同样的警告，并有实际的行号。关于详情，请参考 350 页“提前编译用于健康检查”一节。

```
(time (doseq [s (repeat 100000 "foo")]
              (fast-capitalize s)))
      ; "Elapsed time: 154.889 msecs"
```

从 5 秒降到了 0.155 秒。不错。

任何表达式都可以加类型提示。看下面这个函数，它有一个对 String.split 的反射调用：

```
(defn split-name
  [user]
  (zipmap [:first :last]
          (.split (:name user) " ")))
; #'user/split-name
; Reflection warning, NO_SOURCE_PATH:3 - call to split can't be resolved.
(split-name {:name "Chas Emerick"})
; {:last "Emerick", :first "Chas"}
```

如果只能对 let 绑定的名称加提示，就不得不加一个 let 形式，只是为了能够给中间值 (:name user) 加提示：

```
(defn split-name
  [user]
  (let [^String full-name (:name user)]
    (zipmap [:first :last]
            (.split full-name " ")))
; #'user/split-name
```

虽然没有反射，但得多不少事儿。幸亏可以直接提示 (:name user) 表达式：

```
(defn split-name
  [user]
  (zipmap [:first :last]
          (.split ^String (:name user) " ")))
; #'user/split-name
```

没有反射，也不冗长。同样，可以提示函数的返回值，从而调用它的函数就不需要对结果提示互操作调用了：

```
(defn file-extension
  [^java.io.File f]
  (-> (re-seq #"\.(.+)" (.getName f))
       first
       second))

(.toUpperCase (file-extension (java.io.File. "image.png")))
; #'user/file-extension
; Reflection warning, NO_SOURCE_PATH:1 - reference to field toUpperCase can't be
; resolved.
; "PNG"
```

返回值的提示是加在函数的参数向量上的：

```
(defn file-extension
  ^String [^java.io.File f]
  (-> (re-seq #"\.(.+)" (.getName f))
       first
       second))

(.toUpperCase (file-extension (java.io.File. "image.png")))
:= "PNG"
```

最后，可以给 var 名提供类型元数据，以指明 var 所含的值的类型：

```
(def a "image.png")
:= #'user/a
(java.io.File. a)
; Reflection warning, NO_SOURCE_PATH:1 - call to java.io.File ctor can't be resolved.
:= #<File image.png>
(def ^String a "image.png")
:= #'user/a
(java.io.File. a)
:= #<File image.png>
```

数组

在 `java.util` 集合类 API 被引入 Java 1.2 之前，数组^{注20}是少数几种能容纳大量对象的办法之一。现在很少用数组来容纳对象，但在涉及数值和其他原始数据集时，数组仍是重要的工具。无论如何，Clojure 可以应付自如地处理 Java 数组，不过一般你会觉得数组处理是 Clojure 比 Java 冗长的少数几种情况之一，因为后者直接为数组提供了专门设计的语法。

表9-2：数组操作比较

操作	Clojure 表达式	对等的 Java 代码
从集合类创建数组	(into-array ["a" "b" "c"])	(String[])coll.toArray(new String[list.size()]);
创建空的数组	(make-array Integer 10 100)	new Integer[10][100]
创建原始 long 类	(long-array 10) (make-array	new long[10]
型的空数组	Long/TYPE 10)	
访问数组的值	(aget some-array 4)	some_array[4]
设置数组的值 [*]	(aset some-array 4 "foo") (aset ^ints int-array 4 5)	some_array[4] = 5.6

* 在给原始类型的数组设置值时，Clojure 的表达式提示机制会起作用，详情见 442 页“合理使用原始类型的数组”一节。

注 20：不要混淆 Clojure/Java 的数组与 Ruby 的 Array。Ruby 的 Array 与 Java 的向量相似。

371 如果你的 Clojure 代码只是使用对象数组，可能是已有的 Java API 返回的，那么没必要把它显式地转成列表或其他集合类。数组已经获得 Clojure 的序列抽象支持，^{注21} 因而可以像其他可序列化的集合类一样使用：

```
(map #(Character/toUpperCase %) (.toCharArray "Clojure"))
;= (\C \L \O \J \U \R \E)
```

不过 Clojure 的确为原始类型的数组提供了一些特别支持。这一话题在 442 页“合理使用原始类型的数组”一节里讨论。

定义类、实现接口

能够调用 Java 方法和实例化类是好的开始，但经常还需要定义类、实现接口，有时还要扩展一个已有的类。Clojure 提供了很多类的定义手段，每个分别提供不同功能的组合以满足不同的用例。

表9-3：定义Java类的Clojure形式^a的关键特性比较

	proxy	gen-class	reify	deftype	defrecord
返回一个匿名类的实例？	✓		✓		
定义具名的类？		✓		✓	✓
能够扩展一个基础类？	✓	✓			
能够定义新的字段？				✓	✓
提供 Object.equals、Object.hashCode 和各种 Clojure 接口的默认实现？					✓

^a 如果需要在 Clojure 里定义新的类型，但不确定使用哪种类型定义形式，请参见第 18 章特别为这一目的设计的流程图。

所有这些形式都可用来定义类、实现接口。其中一些形式——`deftype`、`defrecord` 和 `reify`——也服务于 Clojure 内一个独特的目的，与 Java 类和接口互操作无关，因而单独在第 6 章里讨论。

另一方面，其他两个——`proxy` 和 `gen-class`——存在的唯一目的就是支持在互操作情景下的使用，我们将在这里讨论。

匿名类的实例：proxy

`proxy` 产生一个匿名类的实例，实现任意数目的 Java 接口和 / 或单个具体类。^{注22} 匿名类

注 21：特别的，构成 Clojure 各种各样序列操作基础的 `seq` 函数在给出一个数组时会正确地返回一个 `seq` 序列。

注 22：如果不需要从已有的具体类派生子类，应该首选 `reify` 而不是 `proxy`；前者在 284 页“用 `reify` 来定义匿名类型”一节里有描述。

只生成一次，在编译时，基于指定的类和接口。之后，每次运行时，proxy 调用的代价就只是调用所生成的类的构造函数一次的代价。这让 proxy 与 Java 里定义和实例化一个匿名的内部类等价。

下面来演示 proxy 的用法，看看如何用它实现一个基本的、最近最少使用（LRU）缓存^{注23}的，有些元件使用 Java 标准函数库里现成的东西。

JDK 提供了一个哈希映射的实现，java.util.LinkedHashMap 提供了实现一个简单的 LRU 缓存的基本钩子：它能基于最后访问维护条目迭代顺序，并且定义了一个方法，removeEldestEntry(Map.Entry<K,V>)，子类可以推翻它以告诉 LinkedHashMap 最旧的条目（基于插入或访问顺序）是否应该删除。

用 proxy 来做可以展示它所有的特点。

例9-15：用 LinkedHashMap 和 proxy 实现一个简单的 LRU 缓存

```
(defn lru-cache
  [max-size]
  (proxy [java.util.LinkedHashMap] [16 0.75 true]
    (removeEldestEntry [entry]
      (> (count this) max-size))))
```

① 首先，设立一个工厂函数以便在需要时获得这个缓存的实例。

373

② proxy 的方法实现形成闭包，因而在使用 proxy 的范围内绑定的任何值在这些实现里都可用。这里工厂函数接受一个 max-size 参数，removeEldestEntry 方法的实现闭包了它。我们将比较这个值和映射的大小来作为条目是否过期的标准。

③ proxy 前两个参数要求是两个向量：

1. 超类和 / 或所实现的接口的名称；注意超类必须在前。
2. 提供给它的超类的构造函数的参数。这个例子里，我们提供的是 LinkedHashMap 的初始映射大小的默认值（16）和负荷因子（0.75），以及布尔值 true，这表示底层的映射应该维护访问顺序而不是它默认的插入顺序。这就是把 LinkedHashMap 变成一个可用的 LRU 缓存的东西。注意，我们可以把任何表达式放在构造函数向量里，不仅是字面量。

④ proxy 不要求在方法实现里提供 this 作为第一个参数（这与 Clojure 的其他类定义形式 reify、defrecord 和 deftype 不同）。this 是隐含地与 proxy 的实例绑定的……

注 23：如果对 LRU 缓存不熟悉，请参见 http://en.wikipedia.org/wiki/Cache_algorithms#Least_Recently_Used。

- ❸ 我们用它来检查映射的大小。如果它变得比闭包的 `max-size` 值还要大，就返回 `true`，表示所提供的最近最少用到的条目应该驱逐出缓存。当然，任何标准都可用来实现缓存驱逐政策，但映射的大小是一个合理而常见的基准。

提供给 `proxy` 的方法实现可以是任何顺序。

让我们在 REPL 里使用一下这个简单的 LRU 缓存，确保它像我们预期的那样工作：

```
(def cache (doto (lru-cache 5) ❶
  (.put :a :b)))
;= #'user/cache
cache
;= #<LinkedHashMap$0 {:a=:b}>
(doseq [[k v] (partition 2 (range 500))] ❷
  (get cache :a) ❸
  (.put cache k v)) ❹
;= nil
cache
;= #<LinkedHashMap$0 {492=493, 494=495, 496=497, :a=:b, 498=499}> ❺
```

- ❶ 首先，用在例 9-15 里定义的 `lru-cache` 函数创建了新的缓存，加入一个条目，从 `:a` 到 `:b` 的映射关系。²⁴ 我们将频繁访问这一条，保证它永远不会作为最近最少使用的而在缓存里过期了。

- 374 ❷ 我们用 `partition` 组合出 250 个双元素列表的序列，每个包含由 `range` 提供的单调递增的整数。²⁵ 每个双元素列表用 `[k v]` 解构形式解构²⁶ 成“键”和“值”。
- ❸ 在添加新条目到缓存前，先访问与 `:a` 键对应的条目。这将保持这一条目的“热度”，维持它在 `LinkedHashMap` 里的地位，让它免于成为 `removeEldestEntry` 方法的潜在驱逐对象。
- ❹ 我们把每个键 - 值对从在步骤 ❷ 建立的包含 250 对的序列放入缓存。一旦 `max-size` 的值 5 超过了，`.put` 将会导致最近最少用到的条目在缓存里过期。
- ❺ 在结束批量 `.put` 操作后，检查缓存的状态……

而结果的确是对的，缓存只包含 5 个条目，而且我们一直保持热度的条目 `[:a :b]` 还在，它从没被送给基于 `proxy` 实现的 `removeEldestEntry` 方法，因而从来没被驱逐。只有 `[498 499]` 一条比 `[:a :b]` 在更近的时间使用过，因为最后一次操作就是添加前者。

注 24：`:a` 和 `:b` 是 Clojure 的关键字变量。Clojure 关键字在 14 页“关键字”一节里有描述。

注 25：使用可打印数量的数据的一个例子是 `(partition 2 (range 10)) ((0 1) (2 3) (4 5) (6 7) (8 9))`。

注 26：关于 Clojure 的解构形式可以阅读 28 页“解构 (let, 第 2 部分)”一节。

定义具名的类

虽然 proxy 让你在 Clojure 中在运行时定义匿名类，但经常有情况需要你提供静态、独立、具名的 Java 类给 Java 为中心的客户使用。^{注27} Clojure 提供了三种形式生成具名的类，每种都体现了不同的优缺点。

`deftype` 和 `defrecord` 与协议一起代表了 Clojure 对数据和领域建模的原则方法。结果是，它们回避了 Java 对象模型的一些较复杂的方面，为 Clojure 的惯用法让路。另一方面，为了获得最高效率，`deftype` 和 `defrecord` 结构上与“常规”Java 类极为相似：让你定义新的（可选原始类型的）字段，而函数体被内联进它们产生的 class 文件，这使得效率可以与用 Java 自身定义的一样高。可以说，`deftype` 和 `defrecord` 重用了 Java 对象模型“好的部分”。它们可以并且在可能的情况下确实用于互操作目的，^{注28} 但是在 Clojure 里它们有更大的用途，因而我们在第 6 章单独而广泛地讨论了它们。

< 375

相反，`gen-class` 对 Java 对象模型提供了更完整的支持——包括定义静态方法、从具体基类派生子类、定义多重构造函数和新的实例方法，不过结构上和典型的 Java 类差别很大，所生成的类的方法在运行时把实现委托给通常的 Clojure 函数。

gen-class

`gen-class` 让你定义 Java 类，方法实现是由通常的 Clojure 函数支持的。这完全是为了互操作情景准备的，支持 Java 对象模型很大的子集，从而可能满足框架和函数库 API 的要求而少有例外。它允许你：

- 在任何包、用任何名称生成 Java 类。
- 扩展已有的基类，访问基类的保护字段。
- 实现任何数量的 Java 接口。
- 定义任何数量的构造函数。
- 定义静态方法和更多实例方法，超出父类和所实现的接口所定义的那些。
- 方便地生成静态工厂函数。
- 方便地生成一个静态的 `main` 方法，以便类在命令行操作。

^{注 27}：这些客户可以是 Java 程序员，如果你把 Clojure 代码作为函数库来分发，或者已有的 JVM 函数库、框架和服务器要求某些实现类在配置文件里静态命名，等等。后者一个常见的例子是 servlet 容器；它的 `web.xml` 文件，将在例 17-1 里描述，要求指定具名的 servlet 类。

^{注 28}：本章后面我们将看到一些例子，见 383 页“实现 JAX-RS Web 服务端点”和 388 页“使用 `deftype` 和 `defrecord` 类”两小节。



`gen-class` 是 Clojure 里唯一必须提前（AOT）编译的形式。否则，`gen-class` 形式就是空操作，因为 `gen-class` 不像其他 Clojure 的类定义形式那样在运行时定义一个类。经过提前编译，`gen-class` 形式生成 Java class 文件，可以在 `.jar` 文件里发布、被其他 Java 函数库使用、在 Java 程序里引用，等等。

在 337 页“提前编译”一节可以学到更多关于提前编译的内容。（本章使用 `gen-class` 的所有例子都假设已经相应地提前编译过了。）

全面地描述 `gen-class` 的所有选项很可能需要一整章的篇幅。不过，以 `gen-class` 用法一两个有代表性的例子可以作为不错的起点，帮你理解它是如何工作的。这里是一个 Clojure 命名空间，实现了非常简单的图形缩放，我们可以把它打包成一个自足的、可执行的 `.jar` 文件，多亏有了 `gen-class` 定义。

376 例9-16：通过`gen-class`提供静态方法和命令行工具

```
(ns com.clojurebook.imaging
  (:use [clojure.java.io :only (file)])
  (:import (java.awt Image Graphics2D)
           javax.imageio.ImageIO
           java.awt.image BufferedImage
           java.awt.geom.AffineTransform))

(defn load-image
  [file-or-path]
  (-> file-or-path file ImageIO/read))

(defn resize-image
  ^BufferedImage [^Image original factor]
  (let [scaled (BufferedImage. (* factor (.getWidth original))
                               (* factor (.getHeight original))
                               (.getType original))]
    (.drawImage ^Graphics2D (.getGraphics scaled)
               original
               (AffineTransform/getScaleInstance factor factor)
               nil)
    scaled))

(gen-class
  :name ResizeImage
  :main true
  :methods [^:static [resizeFile [String String double] void]
            ^:static [resize [java.awt.Image double] java.awt.image BufferedImage]])
```

```

(def ^:private -resize resize-image)

(defn- -resizeFile
  [path outpath factor]
  (ImageIO/write (-> path load-image (resize-image factor))
    "png"
    (file outpath)))

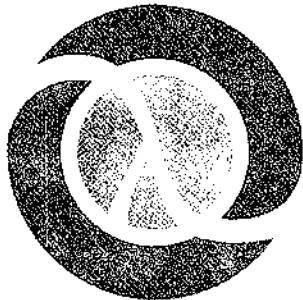
(defn -main
  [& [path outpath factor]]
  (when-not (and path outpath factor)
    (println "Usage: java -jar example-uberjar.jar ResizeImage [INFILE] [OUTFILE]
      [SCALE]")
    (System/exit 1)))
  (-resizeFile path outpath (Double/parseDouble factor)))

```

- ❶ gen-class 默认生成所在命名空间同名的类。在这个例子里，命名空间 (`com.clojurebook.imaging`) 是符合 Clojure 习惯的，不过与 Java 的实践不太吻合，对命令行工具来说也太长了。因而我们指定在默认包里生成一个类名 `ResizeImage`。377
- ❷ 我们想能够作为命令行工具来使用这个类，因而启用 `gen-class` 的主方法选项。这会产生一个 `public static void main String(args[])` 方法，并委托给这个命名空间的 `-main` 函数。
- ❸ 这里在生成的类上定义了两个方法，以 `[methodName [parameter types] returnType]` 的形式提供。这个例子中这些方法都是静态的，因为每个方法的签名向量里有 `^:static` 的元信息。
- ❹ `gen-class` 默认会在与 `gen-class` 定义的名字相同的命名空间里找实现方法的函数，名称和定义的方法相同但带一个前缀 `-`。²⁹ 这里定义的静态方法 `resize` 方便地与 `resize-image` 函数有相同的签名和语义。因此，我们直接给 `resize-image` 函数用一个新 var 取别名，名称与 `resize` 方法将委托的 `(-resize)` 一样。我们为 `resizeFile` 方法提供了专门的实现。
- ❺ 我们的主方法提供了一些简单的用法信息，把命令行参数传递给 `-resizeFile`。

一旦提前编译这个命名空间，将会有一个 `ResizeImage class` 文件，可以从命令行运行它或通过静态方法在 Java 应用程序里调用它。例如，假设在当前目录里有一个图形文件叫 `closure.png`：

注 29：给 `gen-class` 的 `:prefix` 指提供一个字符串的值可以指定不同的前缀。



378 > 可以从命令行运行这个 `ResizeImage` 工具：

```
java -cp gen-class-1.0.0-standalone.jar ResizeImage clojure.png resized.png  
0.5
```

生成缩放的图像：



`ResizeImage` 的这些静态方法对任何 Java 代码都是可以访问的：

```
ResizeImage.resizeFile("clojure.png", "resized.png", 0.5);
```

关于 `gen-class` 值得注意的一件事是，它不要求你对 Clojure 代码做任何改动。这个 `com.clojurebook.imaging` 命名空间将是一个很好的（虽然有点小）Clojure API，在各方面都是符合习惯的，我们只是用 `gen-class` 来提供对 Java 友好的桥梁来利用这个命名空间的功能。

定义自定义的异常类型。正如在 363 页“重用已有的异常类型”一节所说，通常的情况是 Clojure 代码重用已经由 Java 标准函数库或某个应用程序可能使用的第三方函数库提供的异常类型。不过在有些情况下需要特别的异常类型，尤其是如果在与以 Java 为中心的同事广泛合作时，他们预期看到每种错误情况有大量的异常类型。

看看一个自定义的异常类型，除了通常 Java 异常类型典型的带有 `String` 和根源 `Throwable` 外，它让我们提供一个数据的映射：

例9-17：用gen-class定义一个自定义的异常类型

```
(ns com.clojurebook.CustomException
  (:gen-class :extends RuntimeException
              :implements [clojure.lang.IDeref]
              :constructors {[java.util.Map String] [String]
                            [java.util.Map String Throwable] [String Throwable]}
              :init init
              :state info
              :methods [[getInfo [] java.util.Map]
                        [addInfo [Object Object] void]]))

(import 'com.clojurebook.CustomException)

(defn- -init
  ([info message]
   [[message] (atom (into {} info))])
  ([info message ex]
   [[message ex] (atom (into {} info)))))

(defn- -deref
  [^CustomException this]
  @(.info this))

(defn- -getInfo
  [this]
  @this)

(defn- -addInfo
  [^CustomException this key value]
  (swap! (.info this) assoc key value))
```

① 异常类型是 `java.lang.Exception` 子类……

- ② 并且实现了 Clojure 的接口 `java.lang.IDeref`, 让它可以使用 `deref` 抽象, 这在 160 页的“警告”里描述过。Clojure 客户因而可以对这个异常类型使用 `deref` 和 `@` 来获得它的映射数据, 就像其他任何可 `deref` 的值一样。
- ③ 定义了一两个构造函数。我们提供的映射 (部分) 指定了 `CustomException(java.util.Map, String)` 构造函数将调用父类的 `Exception(String)` 构造函数。实际传给父类的构造函数的值是由 `:init` 函数决定的。
- ④ 异常类型将有单个 `final` 字段, 我们称为 `info`。很快就会看到这个字段是如何被使用的。
- ⑤ 我们定义两个方法：`getInfo` 和 `addInfo`。很快就会看到这些构成了这个自定义的异常很有用的 Java API。

379

- ⑥ `gen-class` 基于我们指定的签名生成构造函数。这些构造函数会用所提供的参数调用这里的 `:init` 函数，这里可以进行通常 Java 构造函数里要做的初始化。
- ⑦ `:init` 函数必须总是返回两个元素的向量：第一个是要作为参数向量传给父类的构造函数，第二个是在 `:state final` 字段设置的值。我们把含有 `info Map` 的原子储存在 `:state` 字段里，因为我们用一个原子来协调对这个 `info Map` 的改变，就把所提供的（可能是可变的）`Map` 复制到一个不可变的 Clojure 映射里。
- ⑧ `-deref` 和 `-addInfo` 函数（实现 `deref` 和 `addInfo` 方法）展示了如何与储存在 `CustomException` 的 `final info` 字段里的原子交互。

与我们在例 9-16 里看到的 `gen-class` 用法相反，`com.clojurebook.CustomException` 命名空间存在的唯一目的就是定义 `CustomException` 类。对这些情形，可以在命名空间声明中直接“内嵌”`gen-class` 配置，让生成的类名继承命名空间的名称。

380 我们来看看如何在 Clojure 里使用它，在一些模拟函数的帮助下可以很容易看到在许多大型应用程序里的结果：

```
(import 'com.clojurebook.CustomException)
;= nil
(defn perform-operation
  [& [job priority :as args]]
  (throw (CustomException. {:arguments args} "Operation failed")))
;= #'user/perform-operation
(defn run-batch-job
  [customer-id]
  (doseq [[job priority] {:send-newsletter :low
                           :verify-billings :critical
                           :run-payroll :medium}]
    (try
      (perform-operation job priority)
      (catch CustomException e
        (swap! (.info e) merge {:customer-id customer-id
                               :timestamp (System/currentTimeMillis)}))
      (throw e))))
;= #'user/run-batch-job
(try
  (run-batch-job 89045)
  (catch CustomException e
    (println "Error!" (.getMessage @e)))
; Error! Operation failed {:timestamp 1309935234556, :customer-id 89045,
;                           :arguments {:verify-billings :critical}})
;= nil
```

- ❶ `perform-operation` 抛出了一个新的 `CustomException` 异常，它的所有参数作为一个序列包含在提供的 `info` 映射里。
- ❷ 调用链里任何高层次的函数（这里的 `run-batch-job`）都可以捕获 `CustomException`，并把新数据加到异常的 `info` 映射里。由于我们在 Clojure 里不需要依赖 `gen-class` 形式创建的 `addInfo` 方法，可直接进入异常的 `info` 字段的原子里，把内容信息合并到映射里，即客户 ID 和异常发生时的时间戳。
- ❸ 我们的顶层函数也可以捕获 `CustomException`，而不仅是加入更多信息：
 - 使用我们定义的 `deref` 方法解引用（通过 `@ reader` 宏）异常，返回它所携带的累积的映射信息。
 - 通过 `.getMessage` 方法获得异常创建时提供的最初信息。记得我们并没有定义这个方法，它是由 `gen-class` 生成的类继承了基类实现的方法，就像普通 Java 一样。

像这样能随异常传递任意数据是非常强大的。你甚至可以根据问题域的情况在抛出（或由调用链中的一个函数或方法重新抛出）的异常里包括一个函数，让某个更高层代码能够调用或重试某个操作，也许使用不同的参数。^{注 30} ◀381

在 Java 里使用我们的新异常类型也很简单：

```
import com.clojurebook.CustomException;
import clojure.lang.PersistentHashMap;

public class BatchJob {
    private static void performOperation (String jobId, String priority) {
        throw new CustomException(PersistentHashMap.create("jobId", jobId,
            "priority", priority), "Operation failed");
    }

    private static void runBatchJob (int customerId) {
        try {
            performOperation("verify-billings", "critical");
        } catch (CustomException e) {
            e.addInfo("customer-id", customerId);
            e.addInfo("timestamp", System.currentTimeMillis());
            throw e;
        }
    }
}
```

^{注 30}：这样做会是一次性重启机制的弱实例。重启是状况系统的关键特性，这是对基于异常的错误处理的推广，在 Smalltalk 和一些其他 Lisp 语言里有。这些系统允许碰到异常状况的任何代码提供一个或多个重启以便更高层代码可以选择调用。如果你想试验更灵活的错误处理方法，再次推荐你看 Slingshot：<https://github.com/scgilardi/slingshot>。

```
public static void main (String[] args) {
    try {
        runBatchJob(89045);
    } catch (CustomException e) {
        System.out.println("Error! " + e.getMessage() + " " + e.getInfo());
    }
}
```

与我们在 Clojure 里使用 `CustomException` 的唯一区别是，对于存储在异常类的 `info` 字段里的原子，与尝试执行在 Java 里与 `swap!` 等价的操作相比，`.addInfo` 方法更可取，而且 `.getInfo` 要好于 `.deref` 方法，因为前者的类型是返回一个 `java.util.Map` 对象。

注解

Java 里的注解是一种静态定义的元数据，可以附着于类、方法和字段声明。这个元数据或者在编译时可供代码生成设施和其他编译时进程使用，或者在运行时通过 Java 的反射机制可用。注解是在 Java 5 引入的，让函数库和框架的使用者声明式地定义行为和语义，但与被影响的东西放在一起。这与 XML 文件和其他配置机制把有价值的元信息和元信息所要描述的东西分开形成对照。注解现在广泛用于许多 Java 环境，因而 Clojure 能够在这样的环境下无缝地契合很重要。

注解是为了集成

注解是 Clojure 的 JVM 寄主经常使得 Clojure 程序员退缩的一个领域。部分原因是注解本身甚至在 Java 里也是产生悲剧性的复杂度的根源。不过，真正的原因是与 Clojure 的元数据、宏系统和运行时编译能力的组合相比，Java 的注解并没有做多少事情，却典型地涉及大量的工作和冗余。

因而，你会发现 Clojure 函数库和应用程序很乐意使用 Clojure 提供的各种 JVM 互操作的其他方面，却很少愿意使用 Java 的注解，除非那样做是集成所必须的。

产生带注解的 JUnit 测试

Clojure 把附加在任何类生成形式的元数据识别为对所生成的类、方法或字段的注解。我们来看一个例子，使用受欢迎的 JUnit 测试框架 (<http://junit.org>) 里的 `org.junit.Test` 方法注解来指定 `gen-class` 类定义的某个方法应该看做测试。

例9-18：使用JUnit注解把gen-class方法标记为测试

```
(ns com.clojurebook.annotations.junit
  (:import (org.junit Test Assert))
  (:gen-class
    :name com.clojurebook.annotations.JUnitTest
    :methods [{^{org.junit.Test true} simpleTest [] void}
              {^{org.junit.Test {:timeout 2000}} timeoutTest [] void} ❶
              {^{org.junit.Test {:expected NullPointerException}}      ❷
               badException [] void}])) ❸

(defn -simpleTest
  [this]
  (Assert/assertEquals (class this) com.clojurebook.annotations.JUnitTest))

(defn -badException
  [this]
  (Integer/parseInt (System/getProperty "nonexistent")))

(defn -timeoutTest
  [this]
  (Thread/sleep 10000)) ❹
```

- ❶ 对生成的 gen-class 类的 simpleTest 方法的一个注解。`^{org.junit.Test true}` 与 ◀383 Java 方法上一个光杆的 `@org.junit.Test` 注解等价。
- ❷ 这里我们指定了 timeoutTest 方法 org.junit.Test 注解的 timeout 字段的值为 2000 毫秒。这与 Java 里的 `@org.junit.Test(timeout=2000)` 注解等价。
- ❸ 同样，我们指定 badException 在被调用时应该抛出一个 NullPointerException 异常，这是通过把这个类作为 org.junit.Test 注解的 expected 字段值提供。这等价于 Java 里的 `@org.junit.Test(expected=NullPointerException)`。
- ❹ 我们对 badException 方法的实现是尝试把一个并不存在的系统属性当做整数来解析，这会抛出 NumberFormatException，而不是在注解里的 expected 值所表示的 NullPointerException 异常。
- ❺ 我们对 timeoutTest 方法的实现是休眠 10 秒，比在注解里定义的可接受的 timeout 值 2 秒要长。

提前编译这个命名空间将会产生一个 com.clojurebook.annotations.JUnitTest 类，可以把它加到 JUnit 的运行器里。其中一个测试会通过 (simpleTest 的断言总会是真)，但其他两个会失败，这是由注解元数据里提供的配置导致的。JUnit 运行器的输出将会包括：

```
There were 2 failures:
1) timeoutTest(com.clojurebook.annotations.JUnitTest)
```

```
java.lang.Exception: test timed out after 2000 milliseconds
2) throwsWrongException(com.clojurebook.annotations.JUnitTest)
java.lang.Exception: Unexpected exception,
expected<java.lang.NullPointerException> but was<java.lang.NumberFormatException>
```

在 `gen-class` 的 `:methods` 声明里提供的注解完全决定了应用于生成的类的测试标准，可以直接用于一个注解驱动的 JUnit 测试环境。

实现 JAX-RS Web 服务端点

JAX-RS 是 Java 世界里较受欢迎的 Web 服务标准。它定义了一系列基于注解的 API，对使用标准的 Java 类来创建 REST 风格的服务很有用处。实现这一标准的容器使用注解来发现所请求的 URL 对应的类、基于请求的 HTTP 方法决定调用合适的类方法，并设置应答的 Content-Type 等内容。

让我们在 Clojure 里定义一个 JAX-RS 资源类，这又得使用 `gen-class`，不过这次我们试试 `deftype`，以演示使用 Clojure 对注解的支持中可以应用的一些变化：^{注 31}

384 例9-19：一个用JAX-RS注解实现的Web服务

```
(ns com.clojurebook.annotations.jaxrs
  (:import {javax.ws.rs Path PathParam Produces GET}))

(definterface Greeting
  (greet [^String visitor-name]))          ①

(deftype ^[Path "/greet/{visitorname}"] GreetingResource []
  Greeting
  (^{GET true
      Produces ["text/plain"]}
   greet
   [this ^{PathParam "visitorname"} visitor-name]        ②
   (format "Hello %s!" visitor-name)))
```

- ① 用 `definterface` 形式为要实现的 `deftype` 类定义了一个单方法的接口，称为 `Greeting`。它接受单个 `String` 类型的参数。
- ② 定义了一个 `deftype` 类，带有 `Path` 类的注解，值为 `"/greet/{visitorname}"`，这意味着对 JAX-RS 容器的请求如果与这个 URL 匹配都将路由到 `GreetingResource` 类。
- ③ `greet` 方法实现有两个注解：`GET`，这使得这个方法有权处理 `GET` 请求；`Produces`，这定义了 JAX-RS 容器在发回应答时将会指定的 `Content-Type`。这种情况下，我们只是从 `greet` 方法返回一个字符串，因而 `"text/plain"` 是合适的。

注 31：关于 `deftype` 的详情，请参考 270 页“定义你自己的类型”一节。

- ④ 在 Path 类注解里定义的 URL 模式提供了单个参数 visitorname。通过添加一个与 URL 模式里所用名称对应的 PathParam 注解，我们指定 URL 参数应该与 visitorname 方法的参数一致。

一旦提前编译，GreetingResource 就可以部署到任何的 JAX-RS 容器。使用 Grizzly 的嵌入 Web 服务器，可以在 REPL 运行一个：

```
{com.sun.jersey.api.container.grizzly.GrizzlyWebContainerFactory/create  
    "http://localhost:8080/"  
    {"com.sun.jersey.config.property.packages" "com.clojurebook.annotations.jaxrs"})
```

这会启动一个 Grizzly 实例，运行在 localhost:8080，搜索 com.clojurebook.annotations.jaxrs 包为资源处理器，我们的 GreetingResource 类会被找到，并作为候选的请求处理函数。访问 http://localhost:8080/application.wadl 将会返回 JAX-RS 容器的 WADL 描述符，从中可以看到我们的资源 URL、visitorname 参数和 text/plain 媒体类型：

```
% curl http://localhost:8080/application.wadl  
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>  
<application xmlns="http://research.sun.com/wadl/2006/10">  
  <doc xmlns:jersey="http://jersey.java.net/">  
    jersey:generatedBy="Jersey: 1.8 06/24/2011 12:17 PM"/>  
  <resources base="http://localhost:8080/">  
    <resource path="/greet/{visitorname}">  
      <param xmlns:xsi="http://www.w3.org/2001/XMLSchema"  
            type="xsi:string" style="template" name="visitorname"/>  
      <method name="GET" id="greet">  
        <response>  
          <representation mediaType="text/plain"/>  
        </response>  
      </method>  
    </resource>  
  </resources>  
</application>
```

386

GET 任何像 http://localhost:8080/greet/<some-name> 这样形式的 URL 都会产生从注解的 JAX-RS 资源类生成的结果：

```
% curl http://localhost:8080/greet/Jose  
Hello Jose!
```

在 Java 里使用 Clojure

我们暂时假设你要在 Java 里调用 Clojure 的函数库，而库里没有定义任何的类型或类。^{注32} 要使用 Clojure 的代码，需要使用 Clojure 在命名空间里定义的“原生”函数和常量值。幸好在 Java 里做这样的事很简单：

1. 载入想要使用的 Clojure 代码。这意味着着重用在 Clojure 的 `clojure.core` 命名空间里提供的标准 `require`、`use` 或 `load` 函数。
2. 获得 `var` 的引用，指向在命名空间里定义的你感兴趣的每个函数或值。
3. 按应用程序的需要调用函数、使用值。

演示 Java → Clojure 互操作所需要的就两个 `var`，其一提供一个函数，其二提供值。这个值会来自一个简单的 Clojure 命名空间：

例9-20：简单的Clojure命名空间

```
(ns com.clojurebook.histogram)
```

```
(def keywords (map keyword '(a c a d b c a d c d k d a b b b c d e e e f a a a a)))
```

我们要使用的函数是 `frequencies`，来自 `clojure.core` 命名空间，它接受任何可序列化的值，返回序列的元素及其出现频次的映射。^{注33}

这里是一个使用 `frequencies` 的 Java 类，用到了 `keywords` 值和许多其他的值。

386

例9-21：在Java里使用例9-20中的Clojure代码

```
package com.clojurebook;

import java.util.ArrayList;
import java.util.Map;

import clojure.lang.IFn;
import clojure.lang.Keyword;
import clojure.lang.RT;
import clojure.lang.Symbol;
import clojure.lang.Var;

public class JavaClojureInterop {
    private static IFn requireFn=RT.var("clojure.core","require").fn();
    private static IFn randIntFn = RT.var("clojure.core", "rand-int").fn();
    static {
        requireFn.invoke(Symbol.intern("com.clojurebook.histogram"));
    }
}
```

注 32： 这里所示的技巧可以用于任何 JVM 语言；直接把例 9-21 中的 Java 代码翻译成你所喜欢的语言即可。

注 33： 结果技术上是一个直方图；关于直方图是什么的概述见 <http://en.wikipedia.org/wiki/Histogram>。

```

private static IFn frequencies = RT.var("clojure.core", "frequencies").fn();❶
private static Object keywords = RT.var("com.clojurebook.histogram",
    "keywords").deref();❷

@SuppressWarnings({ "unchecked", "rawtypes" })
public static void main(String[] args) {
    Map<Keyword, Integer> sampleHistogram =
        (Map<Keyword, Integer>)frequencies.invoke(keywords);❸
    System.out.println("Number of :a keywords in sample histogram: " +
        sampleHistogram.get(Keyword.intern("a")));❹
    System.out.println("Complete sample histogram: " + sampleHistogram);
    System.out.println();

    System.out.println("Histogram of chars in 'I left my heart in san fransisco': " +
        frequencies.invoke("I left my heart in San Fransisco".toLowerCase()));❺
    System.out.println();
}

}
}

```

- ❶ 首先获得两个后面要用到的标准库函数 `require` 和 `rand-int`。注意，我们使用的是 `fn()` 方法，它返回一个 Clojure 函数（所有的都实现了 `IFn` 接口）。`fn()` 和 `deref()` 的唯一差别是前者为我们执行了把类型强制转换成 `IFn` 的操作。
- ❷ 在尝试访问非核心 Clojure 命名空间前，需要先载入它。这里通过 `requireFn` `IFn` 引用来使用 `require`。这与在 Clojure 里对 `(require 'com.clojurebook.histogram)` 求值是完全等价的。
- ❸ 这里我们获得指向 `clojure.core/frequencies` var 的引用，这与 Clojure 里的 `#'*clojure.core/frequencies` 等价。
- ❹ 这里用 `deref` 获得 `sample-values` var 的值，这就是例 9-20 里的关键词序列。
- ❺ 我们对样例数据调用 `frequencies` 函数，使用的是 `IFn` 的 `invoke()` 方法之一。注意，这些方法（像 `Var.deref()` 一样）返回一个 `Object`，Clojure 是一门动态语言，因而 `var` 的值可以是任何类型的。这在 Clojure 里相当自然，不过在像 Java 这样的静态类型环境里的确需要一些思考：我们知道 `frequencies` 函数返回一个 `Map` 映射，而且知道提供给它的数据是 `Keyword`，因而可以把 `frequencies` 的结果安全地转成 `Map<Keyword, Number>` 类型。

<387

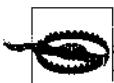
❶ Clojure 返回的值是普通的 Java 对象，我们也可以那样来用。Keyword 和 Number 的 Map 可以如同由一个 Java 方法生成的那样使用。这里我们获得内化的 :a 关键词，看看它在样本数据序列里出现的频次。

❷❸ 由于 Clojure 里对各种具体类型的通用处理，我们可以调用 frequencies 来处理源自 Java 的随机整数 List 和 String，并可以期望获得有用的结果。

编译这个 Java 类后，运行它会产生这样的输出：

```
% java -cp target/java-clojure-interop-1.0.0-jar-with-dependencies.jar  
com.clojurebook.JavaClojureInterop  
Number of :a keywords in sample histogram: 8  
Complete sample histogram: {:a 8, :c 4, :d 5, :b 4, :k 1, :e 3, :f 1}  
  
Frequencies of chars in 'I left my heart in san fransisco':  
{\space 6, \a 3, \c 1, \e 2, \f 2, \h 1, \i 3, \l 1, \m 1,  
\n 3, \o 1, \r 2, \s 3, \t 2, \y 1}  
  
Frequencies of 500 random ints [0,10):  
{0 60, 1 61, 2 55, 3 46, 4 37, 5 45, 6 47, 7 52, 8 49, 9 48}
```

388



两个潜在的绊脚石值得在这里提一提：

- 载入 Clojure 代码（如我们用 require 在例 9-21 中所做的那样）要求所涉及的命名空间对应的源文件或提前编译的 class 文件在你的 classpath 上。
- 一般来说，应该获得指向需要访问的 var 的引用一次（通常保存在一个静态引用里）。而且，如果所保存的值预计不会改变的话，注³⁴ 一次性获得 var 的值是明智的（通过 fn() 或者 deref()）。这可以避免（微小但并非微不足道的）运行时 var 查找的代价。

在理解了上述这些后，你几乎可以在 Java 里做用 Clojure 函数和数据想做的任何事。Java → Clojure 互操作的其余故事就是也能在 Java 里利用 Clojure 定义的类型和协议。

使用 deftype 和 defrecord 类

每种形式³⁵ 都生成一个 Java 可以访问的类。因此，你可以在 Java 里创建和使用这些类的实例，好像它们从开始就是用 Java 写成的。

考虑下面的命名空间。

注 34：关于 var 何时及如何随着时间流逝而改变值的详情，见 201 页“动态作用域”一节。

注 35：我们在 374 页“定义具名的类”一节里详细讨论了 gen-class、deftype 和 defrecord 在第 6 章中进行了讨论。

例9-22：用deftype和defrecord定义一些类

```
(ns com.clojurebook.classes)

(deftype Range
  [start end]
  Iterable
  (iterator [this]
    (.iterator (range start end)))))

(defn string-range
  "Returns a Range instance based on start and end values provided as Strings
   in a list / vector / array."
  [[start end]]
  (Range. (Long/parseLong start) (Long/parseLong end)))

(defrecord OrderSummary
  [order-number total])
```

在载入时，Clojure 生成两个类，`com.clojurebook.classes.Range` 和 `com.clojurebook.classes.OrderSummary`。我们可以在 Java 里使用它们，就像它们是用 Java 写成的一样；对 Clojure 生成的类所实现的每个字段和方法，你甚至可以在 IDE 里看到恰当的代码完成提示。

389

例9-23：在Java里使用在例9-22中定义的deftype类和defrecord类

```
package com.clojurebook;

import clojure.lang.IFn;
import clojure.lang.RT;
import clojure.lang.Symbol;
import com.clojurebook.classes.OrderSummary;
import com.clojurebook.classes.Range;

public class ClojureClassesInJava {
    private static IFn requireFn = RT.var("clojure.core", "require").fn();
    static {
        requireFn.invoke(Symbol.intern("com.clojurebook.classes"));
    }

    private static IFn stringRangeFn = RT.var("com.clojurebook.classes",
        "string-range").fn();

    public static void main(String[] args) {
        Range range = new Range(0, 5);
        System.out.print(range.start + "-" + range.end + ": ");
        for (Object i : range) System.out.print(i + " ");
    }
}
```

```

        System.out.println();

    for (Object i : (Range)stringRangeFn.invoke(args)) ❶
        System.out.print(i + " ");
    System.out.println();

    OrderSummary summary = new OrderSummary(12345, "$19.45"); ❷
    System.out.println(String.format("order number: %s; order total: %s",
        summary.order_number, summary.total));
    System.out.println(summary.keySet());
    System.out.println(summary.values());
}
}

```

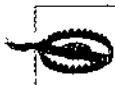
- ❶ 这里我们定义一个 Range deftype 类的实例，提供它所需要的两个参数。
- ❷ 正如在 277 页“类型”一节里详细讨论的，deftype 类并不自动实现任何接口，不过我们可以使用在 deftype 定义里指定的名称访问它的两个 final 字段，而且……
- ❸ 由于 Range 定义为实现 Iterable，可以用它获得一个 Iterator，然后在 for 循环里使用，就像其他任何 Iterable 实例一样。
- ❹ deftype 和 defrecord 类经常可能要求一些特定类型的参数，这取决于这些类是如何使用的；因而提供一个工厂函数来简化实例化过程是明智的。这里我们使用 string-range 工厂函数，它接受任何包含两个字符串的可解构的集合类为参数，基于这些字符串里解析得到的整数值返回一个 Range 实例。这有助于我们避免不得不分解、解析来自命令行的数值输入。
- ❺ defrecord 类和 deftype 类一样，不过它为某些接口提供默认的实现；这里我们创建一个 defrecord 实例，演示对 final 字段的访问以及从 java.util.Map 接口访问一两个方法的默认实现。

一旦编译 Java 类后，我们可以运行它，看看结果：

```

% java -cp
target/java-clojure-interop-1.0.0-jar-with-dependencies.jar
com.clojurebook.ClojureClassesInJava 5 10
0-5: 0 1 2 3 4
5 6 7 8 9
order number: 12345; order total: $19.45
#{:order-number :total}
(12345 "$19.45")

```



什么时候需要提前编译？

如果你在 Java 里使用任何生成类的 Clojure 形式（包括 `deftype`、`defrecord`、`defprotocol` 和 `gen-class`）的结果，必须提前编译包含这些形式的命名空间。Java 的编译器需要这些 class 文件在磁盘上可用，才能使用 Clojure 生成的类编译 Java 代码。这与在只用 Clojure 的情景里使用 `defrecord` 等形式是不同的；那时 Clojure 只需在运行时生成并载入所需的类，再把它们载入 JVM 里，不需要在磁盘上产生文件。

我们在 337 页“提前编译”一节讨论了提前编译，在 351 页“构建混合源文件项目”一节里讨论了与混合源文件项目有关的编译问题。

实现协议接口

协议让你能够在 Clojure 里简洁地创建非常灵活的领域模型。^{注 36} 虽然可以在 Clojure 里扩展协议以服务于已有的 Java 类和接口，但是有时候你可能希望一个 Java 类能使用已有的协议而不用修改 Clojure 代码。为这一目的，协议会生成一个接口，让你在 Java 类里实现。例如这里是一个 Clojure 命名空间，包含一个协议和两个实现，一个针对字符串，另一个是默认实现，后者会为所有的 `Object` 转发这个默认实现：

```
(ns com.clojurebook.protocol

(defprotocol Talkable
  (speak [this]))

(extend-protocol Talkable
  String
  (speak [s] s)
  Object
  (speak [this]
    (str (-> this class .getName) "s can't talk!")))
```

391

`Talkable` 协议定义了一个函数 `speak`，并生成一个 `com.clojurebook.protocol.Talkable` 接口，它只有单个 `speak` 方法。在 Java 里可以轻易实现这个接口。

例9-24：在 Java 里通过生成的接口实现 Clojure 协议

```
package com.clojurebook;

import clojure.lang.IFn;
import clojure.lang.RT;
import clojure.lang.Symbol;
import com.clojurebook.protocol.Talkable;

public class BitterTalkingDog implements Talkable {
```

注 36：所有关于协议的内容见第 6 章。

```

public Object speak() {
    return "You probably expect me to say 'woof!', don't you? Typical.";
}

Talkable mellow () {
    return new Talkable () {
        public Object speak() {
            return "It's a wonderful day, don't you think?";
        }
    };
}

public static void main(String[] args) {
    RT.var("clojure.core", "require").invoke(
        Symbol.intern("com.clojurebook.protocol"));
    IFn speakFn = RT.var("com.clojurebook.protocol", "speak").fn();

    BitterTalkingDog dog = new BitterTalkingDog();

    System.out.println(speakFn.invoke(5));
    System.out.println(speakFn.invoke(
        "A man may die, nations may rise and fall, but an idea lives on."));
    System.out.println(dog.speak());
    System.out.println(speakFn.invoke(dog.mellow()));
}
}

```

① 我们的类实现了 `Talkable` 的 `speak` 方法。

② 协议生成的接口就像其他任何 Java 接口一样；这里我们定义并返回实现这个协议的接口的匿名内部类的一个实例。

③ 我们需要载入 `com.clojurebook.protocol` 命名空间以便协议的 `String` 和 `Object` 扩展可用，然后查找这个协议所定义的 `speak` var 的引用。

如我们所看到的，Clojure 提供了双向的互操作。我们已经演示了 Clojure 可以使用 Java 的抽象并成为它的一部分，同样的，Java 也可以成为 Clojure 的关键抽象的一部分。

当从命令行运行时，这个类的 `main` 方法输出的结果是：

```
% java com.clojurebook.BitterTalkingDog
java.lang.Integers can't talk!
A man may die, nations may rise and fall, but an idea lives on.
You probably expect me to say 'woof!', don't you? Typical.
It's a wonderful day, don't you think?
```

乐于合作的伙伴

Clojure 提供了非常迷人的大量特性，同时毫不掩饰它是一门 JVM 语言，充分利用了这个平台的资产，包括它的成熟度、效率及可靠的运维特征。这让你利用 Java 函数库、框架和社区的巨大生态系统，也让你向这个生态系统回馈自己的贡献。

面向REPL的编程

工具的质量极为重要，可能成全也可能破坏一门语言的体验，更不用说在更大的方面你成功的程度了。Clojure 的 REPL 我们在第 1 章里已经介绍过，是 Clojure 最为基础性的工具——而且我们将会看到，可能也是 Clojure 最强大的工具。

一开始我们就已经强调过，Clojure 总是被编译的，没有解释器。而且正如在第 5 章学到的，Clojure 的编译器在运行时也是完全可用的，使得整个语言在运行时可用——因而也可用于 Clojure REPL。这意味着：

- 在 REPL 里（如开发环境下）载入和运行的代码将工作和表现得和从磁盘文件上（就像在生产环境下那样）载入的代码完全一样。
- 可以随时用 REPL 定义或重新定义任何 Clojure 结构。

这些特征的影响使得 REPL 成为每个 Clojure 程序员绝对不可或缺的工具之一，其他提供 REPL 和解释器的语言通常情况就不是这样的。这里我们将探索 REPL 催生的一些工作流，这可能改变你开发软件的方法。

交互式开发

“交互式开发”这个术语含义很多，因为大多数现代语言都提供一定程度的交互。甚至 Java 程序员也可以交互式地求表达式的值，例如在调试器里在断点中断时。当然，Ruby、Python 和其他语言提供不同复杂度的 REPL，尽管它们通常不与其他工具（像编辑器）集成，只限于在命令行下运行，而且寄主语言对在运行时如何改变或重定义及哪些代码可以改变或重定义经常有重大的限制。

394 相反, Clojure REPL 催生的交互式开发放松了所有这些限制。当用 Clojure 时,你会发现,在一个持续的REPL会话帮助下交互式地构建应用程序是你可以选择的最有成效的方式,在你的指尖和Clojure运行时与JVM里所进行的之间几乎没有阻隔。

例10-1:一个微型Swing“应用”

```
(ns com.clojurebook.fn-browser
  (:import (javax.swing JList JFrame JScrollPane JButton)
           java.util.Vector))

(defonce fn-names (->> (ns-publics 'clojure.core)          ❶
                           (map key)
                           sort
                           Vector.
                           JList.))

(defn show-info [] )          ❷

(defonce window (doto (JFrame. "Interactive Development!")
                      (.setSize (java.awt.Dimension. 400 300))
                      (.add (JScrollPane. fn-names))          ❸
                      (.add java.awt.BorderLayout/SOUTH
                            (doto (JButton. "Show Info")        ❹
                                (.addActionListener (reify java.awt.event.ActionListener
                                              (actionPerformed [_ e] (show-info)))))))
                      (.setVisible true)))
```

❶ `fn-names`是一个`JList`,它将包含指称`clojure.core`命名空间的所有公共`var`的符号,按字典顺序排列。

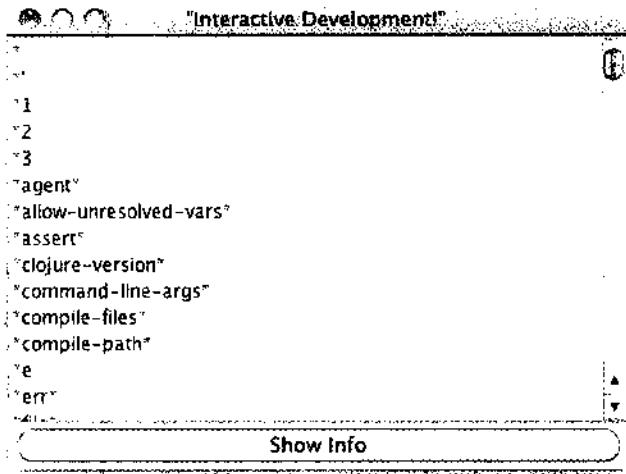
❷ `show-info`函数暂时将维持无操作。后面会改正。

❸ 这个列表组件将会放在一个可滚动的容器里,后者被加入一个可以合理重设大小的窗口。

❹ 加到窗口的还有一个按钮,它的`click listener`会调用暂时无操作的`show-info`函数。

可以把这些代码载入Clojure REPL,或保存到classpath^{注1}上的文件`com/clojurebook/fn_browser.clj`里,然后用`(require 'com.clojurebook.fn-browser)`载入。无论哪种情况,假设你不是在像服务器这样的headless环境下使用Clojure的,将会看到像这样的Swing窗口弹出:

注1: Clojure代码的组织和classpath的概念都在第8章里讲述过。



除非所使用的语言仍然要求单步地进行“编写→编译→调试”循环，能够从 REPL 调出 UI 算不上什么让人大开眼界的事。略微有意思的事是，能以任何我们方便的方式载入更多的代码对正在运行的环境进行无缝改变。

<395

演示一下，来让 Show Info 按钮做点什么，因为目前它是完全不动的。为此，只需简单重定义 show-info 函数以便 click listener 委派做一些有意思的事，如像弹出列表里被选中的 clojure.core 函数的文档：

```
(in-ns 'com.clojurebook.fn-browser)

(import '(javax.swing JOptionPane JTextArea))

(defn show-info
  []
  (when-let [selected-fn (.getSelectedValue fn-names)]
    (JOptionPane/showMessageDialog
      window
      (-> (ns-resolve 'clojure.core selected-fn)
        meta
        :doc
        (JTextArea. 10 40)
        JScrollPane.))
      (str "Doc string for clojure.core/" selected-fn)
      JOptionPane/INFORMATION_MESSAGE))))
```

- ❶ 如果在列表组件里选中了某个函数名，使用 ns-resolve 在 clojure.core 命名空间里求解，这将返回一个 var……

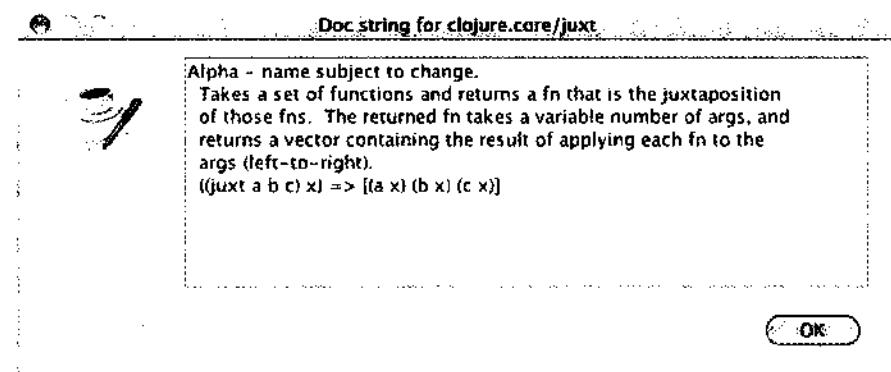
④ 我们将获得它的元数据……

396 ④ 从中得到 :doc 值，这是 var 在定义时提供给 Clojure 保存的文档字符串。关于文档字符串的更多信息，参见 199 页“文档字符串”一节。

④ 然后函数的文档成为 JTextArea 组件内容的初值，作为在 JOptionPane 弹出框里的“信息”。

可以把以上代码载入 REPL，或者修改前面创建的 *fn-browser.clj* 文件，通过 (`require 'com.clojurebook.fn-browser :reload`) 重新载入。^{注2} 无论哪种情况，一旦重新定义了 `show-info`，按钮的 click listener 将会调用新定义的函数，不需要重新创建、修改或其他方式碰那个按钮。

选择一个 `clojure.core` 里有文档的函数，单击 Get Info 按钮将弹出它的文档：



用 defonce 来避免 var 受影响

一旦代码保存到一个你正在载入和重新载入一个 Clojure 进程的文件里——不管是通过 REPL 或使用 `use` 和 `require` 的 `:reload` 或 `:reload-all` 选项——可能会有某些 `var` 的定义你不想重新求值。例如，在例 10-1 中不想对 `window` 和 `fn-names` 重新求值，那样做会导致每次重新载入这个文件时就会创建新窗口和新的列表组件。

注 2： 在后一种情况下，只需简单地把 `JOptionPane` 和 `JTextArea` 加入到文件的 ns 形式中的 `:import` 的声明里，载入文件将会使得这些额外依赖的类被引进命名空间。如果提供 `:reload-all` 给这里的 `require`，将会进一步使得所有 `:require` 或 `:use` 的命名空间被传递式地重新载入。

解决办法是用 `defonce`。正如 `def` 和其他类似的，`defonce` 在当前命名空间里定义了一个 `var` 的值，但不会重新定义已经有值的任何 `var`。所以使用 `defonce` 让我们可以把应用程序生命期里固定不变的 `var` 定义（如例子中的 `window` 或 Web 应用程序里的数据库连接池）与在开发过程中甚至可能在生产环境下可能随时准备重新定义的其他 `var` 混在一起。

这是一个特别简单的例子，但希望已经说明了 Clojure 的 REPL 交互所能提供的某些灵活性。尽可能的方便，不需重启应用程序或重建窗口就可以交互式地调整用户接口的行为，同样的灵活性和即时反馈也适合数据密集算法的开发或复杂域模型，这产生了不可估量的好处。

持续、演化的环境

对某些人来说，目前所演示的看来不过是对已有工作流的小幅改进，特别是像 Ruby、Python 和 PHP 这样的语言。毕竟，这些语言和其他语言也可以交互式地载入代码，在部署到长期运行的环境时通常也能重新定义各种结构。不过这种类比也就到此为止了。

Clojure 不要求基于文件的工作流。Python、Ruby 和 PHP 要求代码必须在磁盘上才能载入。^{注3} 对使用 Clojure 来说这是完全合理的选择——正如我们在前例中提到的，当需要一个已经载入的命名空间，可以用 `:reload` 选项——但这完全不是必须的。要载入代码到 Clojure 进程完全可以不碰磁盘，或者通过直接的 REPL 交互（即键盘输入），或者用 Clojure 开发环境提供的面向 REPL 的命令随意载入文件、命名空间或整个项目。

Clojure 的动态性是由语言和运行时明确提供的。Clojure 设计的许多基础方面明确鼓励（或者不阻止）在运行时动态重定义它的结构。命名空间和 `var` 的具体化、载入代码的重要性与要求代码在磁盘或编辑器缓冲里可用、运行时字节码生成与 `class` 文件载入、编译时间和运行时间距离的缩小——这些及 Clojure 的其他一些元特征共同使得它可作为一个可持续的画布，让你在上面画出你的愿景。

这看起来简直像炒作，除非你体验了配备有功能正常的 REPL 设施的 Clojure 开发环境。那样的环境让你轻易地把文本编辑器与一个或多个持续运行的 REPL 配对，这让你可以轻易地在一个编辑器（或 2 个，或 9 个）里写代码——从那儿可以把单个表达式和整个文件的代码从那个编辑器用一个按键发送给持续运行的 REPL——并与那个 REPL 关联的 Clojure 运行时的状态交互以检查中间结果，试验半成形的想法，不用败坏项目的“真

^{注3：} 从技术上说，Ruby 的情况不是真的，不过许多工具和围绕 Ruby 的文化规范确实推动它朝这个方向发展。

正”代码，并询问 Clojure 运行时以验证你的工作、指导后面的步骤。Clojure 程序员连续数日用同一个 JVM/Clojure 进程并非少见，渐进地修改运行时和应用程序载入的代码，直到它的行为符合预期和所定义的测试。

这样的最佳体验有时被称为“流”，这是一种状态，人的注意力、想象的清晰度和关键信息唾手可得使能力大幅提升、一切尽在掌控的感觉和视野的宽阔。Clojure 肯定没有把流的市场逼入死角，程序员描述了使用各种语言享受这一状态的情景。猜测 Clojure 比大多数语言更能做到这一点并非不合理，很多方面是由于 Clojure 丰富的 REPL 功能和体验，从而在程序与程序员之间的反馈循环更紧密。这一点在 2010 Clojure Conj 大会上有一个演讲详细讨论到了：^{注4}

在 REPL 下编程在某些方面很有点像指导一个伙伴。我们可以调查它的当前状态，机器正在发生的情况，我们的算法在底层做些什么——在 REPL 下这些都有丰富的方式。

——Tom Faulhaber, *Lisp, Functional Programming, and the State of Flow*

一旦理解了 REPL 是如何工作的，找到了在编程实践中如何使用它的感觉，鼓励这种体验的最好办法是确保所需的工具准备停当。

工具集

由于 Java 复杂而累赘，“Java 工具集”总是意味着使用真正的集成开发环境（IDE），例如 Eclipse 和 IntelliJ IDEA：全面的代码完成、重构、类层次可视化，而且其他特性并非是虽好但不必要的东西，而是多数 Java 程序员的硬性要求。相反，动态编程语言（包括 Python 和 Ruby）通常只要求有一个功能强大的编辑器和命令提示。多数 Clojure 程序员更遵循后者的模型，^{注5} 但有一个关键差别：功能强大的编辑器是必需的，可以访问 Clojure REPL——最好与所用的编辑器和其他设施很好集成——至少是同样重要，如果不是更重要。

399 幸好做到这一点很简单，因为许多广受欢迎的编辑器（如 Emacs、vim、TextMate 和 jEdit 等）都有很好的 Clojure 支持，集成开发环境（如 Eclipse、IntelliJ IDEA 和 NetBeans）也一样。从这些中选择任何一种起步都是相当容易的，^{注6} 如果其他情况一样，我们推荐使用你现

注 4： 视频和演示稿在 <http://blip.tv/clojure/tom-faulhaber-lisp-functional-programming-and-the-state-of-flow-4539472>。

注 5： 许多 Clojure 编程环境也提供了像代码完成等东西，不过在这里我们谈的是最小预期，而不是已经存在的和值得拥有的东西的开放集。

注 6： 参见 <http://dev.clojure.org/display/doc/Clojure+Tools> 列出的提示。

在使用得最得心应手、最适合你已有的风格和工作流的工具。^{注7} 为了便于比较，下面概述社区里最受欢迎的两个对 Clojure 支持的工具，分别代表对 Clojure 工具集（也许也是对一般的工具集）截然不同的方法：Eclipse 和 Emacs。

首先来算一下所有 Clojure REPL 提供的一些基本而实用的工具，这些是你需要熟悉以提升编程体验的。

最简的 REPL

你每天都会使用 Clojure REPL，这意味着使用它的基本功能。无论是使用最基本的文本编辑器与在单独的命令行里的 REPL，还是使用最大的集成开发环境与集成的 REPL 会话，这些实用工具都是可用的、不可或缺的。^{注8}

REPL 绑定的 var。有一些 var 典型地只在 REPL 会话里绑定，提供交互环境里必要的便利。

- *1、*2 和 *3 存有最近求值的表达式的值。例如，*1 对应 Ruby 和 Python 里的 _。
- *e 提供 REPL 会话里发生的最后一个未捕获的异常。这与 Python 里的三元组 sys.last_type、sys.last_value 和 sys.last_traceback 类似。

在互动探索 API 和数据时，这些 var 和它们的自动管理真是非常便利：

```
(split-with keyword? [:a :b :c 1 2 3])
;= [(:a :b :c) (1 2 3)]
(zipmap (first *1) (second *1))
;= {:c 3, :b 2, :a 1}
(apply zipmap (split-with keyword? [:a :b :c 1 2 3]))
;= {:c 3, :b 2, :a 1}
```

clojure.repl/pst 将会打印提供的任何异常的堆栈追踪，不过默认绑定的是 *e：

```
(throw (Exception. "foo"))
;= Exception foo  user/eval1 (NO_SOURCE_FILE:1)
(pst)
; Exception foo
;  user/eval1 (NO_SOURCE_FILE:1)
;  clojure.lang.Compiler.eval (Compiler.java:6465)
; ...
```

<400

^{注7} 强调一点：即使你使用 notepad.exe 最舒适，如果在旁边的终端窗口里有 REPL 可用，你就应该用它。我们认为有更好的选择，过一段时间你也许会有同感，但没有什么比同时学习一门新的编程语言和一套新的工具更令人沮丧。

^{注8} 如果不巧你是在命令行启动 REPL（即通过直接开启 java 程序），不是用 Leiningen、Counterclockwise、Emacs 或不是终端的任何其他 Clojure 工具提供的 REPL，那么几乎肯定你需要使用 JLine (<http://jline.sourceforge.net>) 或 rlwrap (<http://utopia.knoware.nl/~hlub/rlwrap/>) 之一。Clojure 内建的 REPL 不提供像命令回忆（即按向上光标键得到发送给 REPL 的前一行）或行内编辑还没发送给 REPL 的文本的功能，JLine 和 rlwrap 都会把这些功能加给内置的 REPL。

`clojure.repl`。说起 `clojure.repl`，它提供了一组在 REPL 里非常便利的工具。包括前面提到的 `pst`，还有 `apropos`，它显示在载入的命名空间里有什么函数与给定的正则表达式或字符串匹配：

```
(apropos #"\^ref")
;= (ref-max-history refer-clojure ref-set
;= ref-history-count ref ref-min-history refer)
```

`find-doc` 所做的差不多，只是它在文档里搜索，并且打印匹配的 var 所有的信息。

还有 `source`，它会打印从源文件载入的任何函数的源代码：

```
(source merge)
; (defn merge
;   "Returns a map that consists of the rest of the maps conj-ed onto
;   the first. If a key occurs in more than one map, the mapping from
;   the latter (left-to-right) will be the mapping in the result."
;   {:added "1.0"
;    :static true}
;   [& maps]
;   (when (some identity maps)
;     (reducel #(conj (or %1 {}) %2) maps)))
```

最后是 `doc`，它只是打印给定 var 的文档，以及 `dir`，它将打印给定命名空间里声明的公共 var 列表：

```
(require 'clojure.string)
;= nil
(dir clojure.string)
; blank?
; capitalize
; escape
; join
; lower-case
; replace
; replace-first
; reverse
; split
; split-lines
; trim
; trim-newline
; triml
; trimr
; upper-case
```

401

Clojure REPL 几乎都是以载入 `clojure.repl` 开始的，从而让它有用的函数总是可用，少有例外。

内省命名空间

命名空间自身是实体，这与任何数据结构一样具体而可塑。有许多函数你可在 REPL 里用来内省和修改命名空间，下面来看看其中一些。^{注9}

注意，多数时候你根本不用碰这些函数。不过如果错误地在某个命名空间里定义了一些函数或数据，可以用这些函数来找到它们，并有可能删除这些惹麻烦的定义。这能够帮你摆脱某些情景，否则就得重启应用程序或 REPL 会话，例如需要用 `deftype` 定义的类型与已经引进一个命名空间的已有 Java 类同名了。

`ns-map`、`ns-imports`、`ns-refers`、`ns-publics`、`ns-aliases` 和 `ns-interns`。这些函数都返回给定命名空间里符号到 var 或引进的类的映射，即 `refer`、`import` 和 `def` 都在命名空间里注册符号，这些函数报告这些不同类型的映射。

```
(ns clean-namespace)
:= nil
(ns-aliases *ns*)
:= {}
(require '[clojure.set :as set])
:= nil
(ns-aliases *ns*)
:= {set #<Namespace clojure.set>}
(ns-publics *ns*)
:= {}
(def x 0)
:= #'clean-namespace/x
(ns-publics *ns*)
:= {x #'clean-namespace/x}
```

`ns-unmap` 和 `ns-unalias`。前者可用于删除符号到 var 或引进的类的映射，后者会删除命名空间别名。

```
(ns-unalias *ns* 'set)
:= nil
(ns-aliases *ns*)
:= {}
(ns-unmap *ns* 'x)
:= nil
(ns-publics *ns*)
:= {}
```

`remove-ns`。这是命名空间管理的“核心选项”。`ns注10` 会创建一个命名空间，`remove-ns` 则把一个命名空间从 Clojure 的权威命名空间映射删除。

402

```
(in-ns 'user)
```

注9： `(apropos #"(ns-|ns)")` 会提供一个更完整的列表，让你自行探索。

注10： 或者不太常用的 `create-ns`。

```
;= #'Namespace user>
(filter # (= 'clean-namespace (ns-name %)) (all-ns))
;= (#'Namespace clean-namespace)
(remove-ns 'clean-namespace)
;= #'Namespace clean-namespace>
(filter # (= 'clean-namespace (ns-name %)) (all-ns))
;= ()
```

这意味着在丢弃的命名空间里内化的 var 所定义的所有代码和数据都将无法访问并等待垃圾回收。如果在其他地方还有指向在丢弃的命名空间里定义的函数、协议或数据的引用，当然就不会被回收。



结构化编辑 Clojure 源代码

对于其他人所选择的文本编辑器，我们显然是平等对待的，不过在编辑 Clojure 代码上有一个非常迷人的功能我们不得不提：paredit，这是起源于 Emacs 社区的编辑模式，简化了对符号表达式的编辑，许多 Clojure 程序员认为是必不可少的。

多数高质量的 Java 编辑器提供了各种各样的结构选择功能，如自动插入括号对、扩展选择以包括包含它的元素、表达式或范围。paredit 给 Clojure 提供了同样的，而且多数实现走得比 Java 编辑器支持的概念远得多，包括每次按表达式移动光标或选择、移动整个表达式、根据需求自动用大括号、方括号或圆括号把表达式括起来以保证源代码是结构正确的。

总之，如果发现编辑 Clojure 代码难——例如方便地选择单个符号表达式或保持圆括号、大括号或方括号——你几乎肯定会得益于使用编辑器里的 paredit 风格的特性，或者改为使用提供这些特性的编辑器（大多数都可以提供）。

403

Eclipse

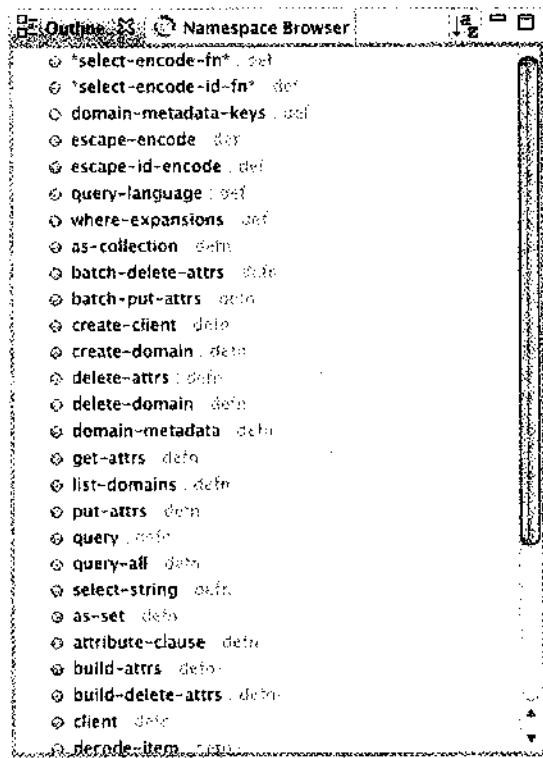
Eclipse——与在 Eclipse 里提供 Clojure 支持的插件 Counterclockwise 搭配使用^{注11}——为 Clojure 开发提供了全面的特性：编辑、代码完成、REPL 集成、内省、调试和剖析。Eclipse 肯定不能认为是轻量的，作为交换，它提供了迷人的工具集组合，对许多程序员来说，Eclipse 有他们熟悉的用户界面，在使用中可不断发现新功能。此外，如果需要不仅使用 Clojure 而且也要使用 Java，可能还是在同一个项目里，那么很难胜过（或放弃）只有像 Eclipse 这样的集成开发环境才提供的 Java 功能。

Clojure 编辑。虽然 Counterclockwise 对 Clojure 的编辑支持还不如 Emacs 提供的功能复杂，但它仍然是非常强、最好的编辑器之一，提供了 paredit 的部分实现，^{注12} 优秀的语法

注 11： 可到 <http://dev.clojure.org/display/doc/Getting+Started+with+Eclipse+and+Counterclockwise> 查看详情。
Counterclockwise 项目主页在 <http://code.google.com/p/counterclockwise/>。

注 12： 在 402 页“结构化编辑 Clojure 源代码”里描述过。

加亮，以及从 Eclipse 继承来的大量有用的文本编辑功能。每个 Clojure 编辑器也输出信息到 Eclipse 的标准“大纲”视图，这个视图维护着当前文件包含的所有顶级表达式（通常也是函数定义）的列表：



Eclipse 和 Counterclockwise 在界面里提供了很多的提示，帮助你学习诸如什么命令可用、不同情景下默认的键盘快捷方式是什么（有些在不同操作系统上会不一样）之类的东西。如果想看帮助，可以参考 Counterclockwise 加到 Eclipse 上的“Clojure”菜单（及编辑器里的上下文菜单）以及 Clojure 编辑器参考页，当打开任何 Clojure 文件时，可以从“帮助”菜单中选择“动态帮助”项找到。

REPL 集成。 Counterclockwise 使用 nREPL^{注 13}（一种 Clojure REPL 服务器和客户端函数库，用于其他 Clojure 工具集，很容易嵌入 Clojure 应用里）作为它与 REPL 集成的基础。这让你可以与任何运行 nREPL 服务器的 Clojure 应用程序连接、交互，包括经过 Eclipse 和 Counterclockwise 启动的所有 Clojure 进程。除了能够求值，Counterclockwise 的 REPL 与编辑器集成，让你从打开的文件和基于当前选中的内容载入代码，包含对命令历史和代码完成的支持。

注 13： <http://github.com/clojure/tools.nrepl>。

```
(defn max-key [f & colls]
  "the latter (left-to-right) will be the mapping in the result."
  {:added "1.0"
   :static true}
  [& maps]
  (when (some identity maps)
    (reduce1 #(conj (or %1 {}) %2) maps)))
nil

(max)
max-key - clojure.core
make-hierarchy - clojure.core
map - clojure.core
mapcat - clojure.core
macroexpand-1 - clojure.core
map? - clojure.core
macroexpand - clojure.core
max - clojure.core
map-indexed - clojure.core
make-array - clojure.core
```

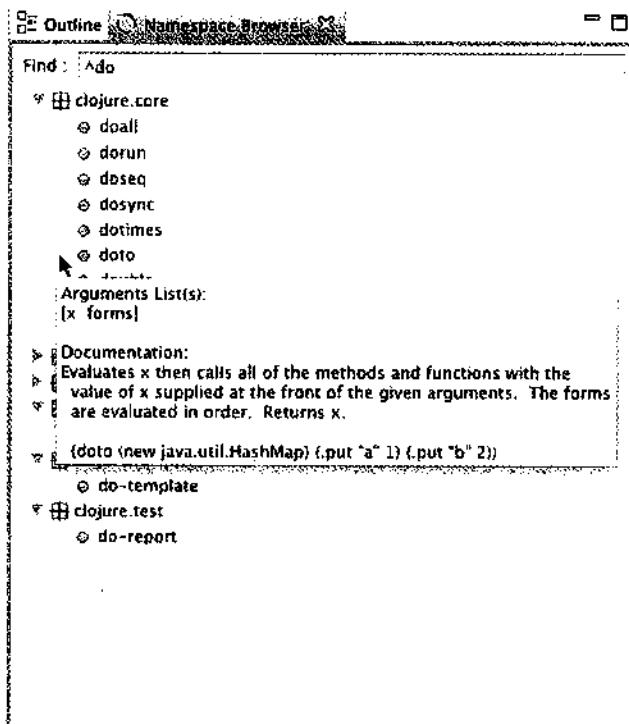
Arguments List(s):
[f & colls]

Documentation:
Returns the result of applying concat to the result of applying map to f and colls. Thus function f should return a collection.

Completion for symbols visible from current namespace

一旦REPL运行起来，你就可以在与运行REPL的项目相关联的编辑器里使用同样的代码完成和跳到定义的动作。

命名空间浏览器。Counterclockwise也提供了一个图形界面的命名空间浏览器。这个视图让你浏览和搜索当前REPL连接的Clojure环境里载入和定义的所有命名空间和var：



可以过滤 var 列表（如上所示，可选用正则表达式）。停留在一个 var 名称上一会儿将在 405 一个弹出窗口里显示它的文档，双击会打开定义这个 var 的文件并转到定义处。

Emacs

Emacs (<http://www.gnu.org/s/emacs/>) 是一个功能强大、可以扩展的编辑器，数十年来已经成为 Lisp 编程的支柱。Clojure 对 Emacs 的支持是通过组合许多模块化的工具和函数库完成的。

在 Emacs 中使用 Clojure 主要有两种方式：`inferior-lisp` 和 `SLIME`。两种方法都依赖于安装 `clojure-mode`。

Emacs 术语入门

406

Emacs 早于大多数现代软件，这一点体现在它所使用的命令和功能的一些术语上。

缓冲区	一个具名的 Emacs 对象，表示可以编辑的内容，通常是文件系统里的文件，但也可以是其他内容，例如 Clojure REPL 或调试器
窗口	Emacs 里的一块或编辑区域。Emacs 可以分割以同时看多个文件（或多个文件 + 一个 REPL 缓冲区）
M-x foo	按住键盘上的 Meta 键（通常是 Alt 或 Option），按下再松开 x 键，然后松开 Meta 键。将会出现一个提示让你输入 foo，然后按回车键
C-k	按住键盘上的 Ctrl 键（通常标为 Control 或 Ctrl），按下再松开 k 键，再松开 Control 键
C-M-x	同时按住 Ctrl 和 Meta 键，按下再松开 x 键，再松开 Ctrl 和 Meta 键
C-x C-e	按住 Ctrl 键，按下再松开 x 键，按下再松开 e 键，然后松开 Ctrl 键

更多信息可以在 Emacs 手册里找到，详情请参见 <http://www.gnu.org/software/emacs/manual/emacs.html>。

`clojure-mode` 与 `paredit`

不管是使用 `inferior-lisp` 或 `SLIME`，`clojure-mode` 和 `paredit` 都是至关重要的。

`clojure-mode` (<https://github.com/technomancy/clojure-mode>) 为 Emacs 提供了 Clojure 专用的编辑器功能，如语法高亮显示、缩进和代码导航等。也包括非常有用的 `clojure-test-mode`，这为使用 `clojure.test` 进行测试自动化提供了更多的快捷方式。设置 `clojure-mode` 的指示在项目的主页可以找到。

除了 `clojure-mode`, 前述 `paredit.el` (<http://www.emacswiki.org/emacs/ParEdit>) 是 Emacs 必有的次要模式, 为自动处理括号提供支持, 并且包含在 Emacs 里。

inferior-lisp

`inferior-lisp`^{注14} 是在 Emacs 里使用 Clojure 的最基本方式。用它可以在一个子进程里启动一个 Clojure REPL, 并在 Emacs 缓冲区里显示这个 REPL。可以像在命令行一样使用这个 REPL, 但 `inferior-lisp` 也让你交互式地从打开的源文件的缓冲区里向 REPL 发送 Clojure 代码。

- 407 ➤ `inferior-lisp` 比 SLIME 优越的一点是, `inferior-lisp` 是内置在 Emacs 里的, 设置它只需要在你的 Emacs 配置文件设置一个 `var` (或通过 `M-:`) 告诉 Emacs 如何启动 Clojure :

```
(setq inferior-lisp-program "lein repl")
```

`inferior-lisp-program` 的值可以如上调用 Leiningen、调用 Java 或其他可以启动 Clojure REPL 的命令, 如 `mvn clojure:repl` 等。设置后, 可以通过执行命令 `C-c C-z` 来启动 Clojure REPL。注意, `inferior-lisp-program` 里的命令是在 Emacs 的当前目录下运行的, 取决于是从哪里启动 Emacs 的, 在启动 `inferior-lisp` 进程前可能需要设置当前目录(通过 `M-x cd`)。这是在 Emacs 里让 Clojure 运行起来的最快也是最容易的方法:

The screenshot shows an Emacs window titled "inferior-lisp". Inside the buffer, a Clojure script is being run. The output shows the script loading a file named "url-shortener.clj", starting a REPL, and defining a counter atom. The REPL then enters a loop where it prints the current value of the counter (0) and prompts for input. The user types "1" and the counter is updated to 1.

```
(ns com.clojurebook.url-shortener
  (:use [compojure.core :only (GET PUT POST defroutes)])
  (:require [compojure.handler route]
            [ring.util.response :as response]))
(def ^:private counter (atom 0))

(def ^:private mappings (ref {}))

(defn url-for
  [id]
  (mappings id))
;; url_shortener.clj Top (5.0)  Git:master (Clojure File)
REPL started; server listening on localhost port 55657
user=> #'com.clojurebook.url-shortener/app
user=> (in-ns 'com.clojurebook.url-shortener)
#Namespace com.clojurebook.url-shortener
com.clojurebook.url-shortener=> (dosync (alter mappings assoc :a 5))
{:a 5}
com.clojurebook.url-shortener=> 1
```

注 14：这样具名不是要暗示它的品质，而是表明所用的 Lisp 是在 Emacs 启动的子进程里：<http://www.gnu.org/s/libtool/manual/emacs/External-Lisp.html>。

与 SLIME 相比，编辑支持是非常简单的，但可能你就只需要这些。在 Emacs 里编辑 Clojure 代码最常见的工作流是一个或多个含有代码的 Emacs 窗口，另有一个 Emacs 窗口是 REPL 缓冲区。代码可以在一个窗口里编辑，然后通过诸多命令之一发送给 REPL。`inferior-lisp` 和 SLIME 都提供了这一功能，不过所用的命令不一样（参见表 10-1）。

表10-1: inferior-lisp

408

快捷键	M-x 命令	描述
C-c C-z	<code>run-lisp</code>	在 <code>inferior-lisp-program</code> 里设置的命令，启动配置的 <code>inferior-lisp</code> 进程
C-M-x	<code>lisp-eval-defun</code>	对光标下的顶层形式（如 <code>defn</code> 表达式）求值
C-x C-e	<code>lisp-eval-last-sexp</code>	对光标前的符号表达式求值
C-c C-l	<code>closure-load-file</code>	把当前文件全部载入

SLIME

SLIME^{注15} 是一个 Emacs 库，为许多 Lisp 语言提供了高级编辑和 REPL 功能，这包括（但不限于）Clojure。与 `inferior-lisp` 相比，SLIME 提供了更全面的开发体验，包括持续的命令历史、代码完成、命名空间内省、调试器等。



Emacs 的扩展性既好也坏。Emacs 可以做你能想象到的一个文本编辑器可以做的任何事，但设置它却是一件令人生畏的事。

特别是，随着 Clojure 对 Emacs 的支持的演变和成熟，为 Emacs 设置 SLIME 和其他 Clojure 相关工具的指南也在不断地变化，而且变化迅速。互联网上现在充斥着在 Emacs 里设置 Clojure 的过时方法的旧博客文章和维基条目。

我们建议直接到 <http://dev.clojure.org/display/doc/Getting+Started+with+Emacs> 去看安装和设置指南，这篇维基由 Emacs 里提供 Clojure 支持的各种项目的维护者一直更新着。

为了使用 SLIME，Clojure 项目必须配置成提供一个 swank 服务器；swank 是 SLIME 的 REPL 交互协议。这样做的最简单办法是用 lein 来把 `swank-clojure` (swank 的 Clojure 实现) 作为开发时依赖加到你的 Leiningen 项目里：

```
lein plugin install lein-swank 1.4.3
```

或者把它永远加到 `project.clj` 文件的插件向量里：

```
[lein-swank "1.4.3"]
```

注15：这是 *Superior Lisp Interaction Mode for Emacs* 的首字母缩写，详情参见 <http://common-lisp.net/project/slime/>。

一旦在当前缓冲区里打开了一个你打算启动 REPL 的项目，SLIME 可以用 M-x `clojure-jack-in` 来从 Emacs 里直接启动。如果出了岔子，一定要看看 (<http://dev.clojure.org/display/doc/Getting+Started+with+Emacs>) 里的最新指示。

- 409 你可以在 SLIME 里使用 `inferior-lisp` 提供的所有键盘绑定，不过 SLIME 有更多的命令用于从缓冲区向运行中的 REPL 发送代码、在当前文件和 Clojure/SLIME 环境下执行各种操作和内省。表 10-2 列出了一些例子。

表10-2：SLIME常见命令及其键盘绑定

按键	M-x 命令	描述
C-c C-c	<code>slime-compile-defun</code>	对光标下的顶级形式求值
C-c C-k	<code>slime-compile-and-load-file</code>	全部载入当前文件
M-.	<code>slime-edit-definition</code>	跳转到光标下符号定义处
C-c C-m	<code>slime-macroexpand-1</code>	<code>macroexpand-1</code> 光标后的表达式
C-c M-m	<code>slime-macroexpand-all</code>	<code>macroexpand-all</code> 光标后的表达式
C-c I	<code>slime-inspect</code>	显示给定符号或类的值的可导航表示

查看器。要查找数据结构或 API 里的东西有时会令人畏惧，尤其是在使用了一些 Java 函数库时。SLIME 通过它的“查看器”提供方法可视化 Clojure 集合类和 Java 对象和类。直接键入 C-c I，然后再输入想要查看值的符号或表达式——或者把光标移到感兴趣的符号上再按 C-c I 键：

```
:dynamic true}
*response-code* nil)

(defn- send-body
  [^URLConnection connection data]
  (with-open [output (.getOutputStream connection)]
    (io/copy data output)
    ; make sure streams are closed so we don't hold locks on files on Windows
    (when (instance? InputStream data) (.close ^InputStream data)))))

(defn- get-response
  [^-- http-client.clj 39% (52,5) Git-master (Clojure Hi Slime[clojure] File)
  class java.net.URLConnection
  --]
  Type: class java.lang.Class
  --
  Fields:
  --
  Methods:
  public java.lang.String java.net.URLConnection.toString()
  public java.net.URL java.net.URLConnection.getURL()
  public java.lang.Object java.net.URLConnection.getContent(java.lang.Class[])
  throws java.io.IOException
  public java.lang.Object java.net.URLConnection.getContent() throws java.io.IOException
  public java.io.InputStream java.net.URLConnection.getInputStream() throws java.io.IOException
  public java.security.Permission java.net.URLConnection.getPermission() throws java.io.IOException
```

可以对包含在查看器的输出里的值和类做同样的事（或者把光标放在感兴趣的符号或值上再按回车键），因而使可视地探索复杂的 API 或大型数据结构变得很容易。

调试。当在 SLIME REPL 抛出异常时，会转存到 SLIME 调试器里。也可以在代码里手工设置断点来启动调试器：

```
(defn debug-me
  [x y]
  (let [z (merge x y)]
    (swank.core/break)))
```

运行 (debug-me {:a 5} {"b" 5/6}) 将会给你像图 10-1 所示的结果：

```

user> (require 'swank.core)
nil
user> (defn debug-me
  [x y]
  (let [z (merge x y)]
    (swank.core/break)))
#'user/debug-me
user> (debug-me {:a 5} {"b" 5/6})
[]

U:***- [*slime-repl nil*  All 19      (REPL)

Restarts:
  0: [QUIT] Quit to the SLIME top level
  1: [CONTINUE] Continue from breakpoint

Backtrace:
  0: user$debug_me.invoke(NO_SOURCE_FILE:1)
    locals:
      debug-me = #<user$debug_me user$debug_me@4be07f4b>
      x = {:a 5}
      y = {"b" 5/6}
      z = {"b" 5/6, :a 5}
  1: user$eval12819.invoke(NO_SOURCE_FILE:1)
U:***- [*slip clojure/2*  9% 13      (slip[1])
```

图 10-1：运行中的 SLIME 调试器

点击 Backtrace 部分会让本地量及其值在查看器里显示，可以同样探索这些值。

当然，在调试器运行时，Clojure 仍然是可用的，而且 REPL 仍然是活跃的，因而任何表达式可以被求值；程序当前的状态可以查看或修改，可以定义或重定义函数，等等。当然 swank.core/break 调用范围内的本地量可以在 SLIME REPL 里访问、操纵，而到达断点的线程会保持中止执行状态。这是在交互式环境里编程的一大优势。

在 REPL 里调试、监测和打补丁

在 397 页“持续、演化的环境”一节里，我们谈到了 REPL 在开发环境下提供的“持续、演化的环境”。还没说到的是，REPL 连接的那些环境并非局限于开发环境。没有技术上的原因说 REPL 不能在“部署的”环境——包括产品——下利用提供在开发环境下享受的动态性。

记住，你和一个 Clojure 环境之间交互的根本单元是“载入代码”：非常简单，对来自磁盘文件或作为输入提供给 REPL 的表达式求值，这些表达式定义了函数和值并给环境带来变化。不需要其他特别的要求。虽然有些 REPL 与本地的操作系统进程和控制台联系在一起（包括例 1-1 里介绍的由 Clojure 提供的默认 REPL），但决不都是那样的。事实上，许多（如果不是多数）Clojure 开发工具——包括 Eclipse 上的 Counterclockwise 和 Emacs 上的 SLIME——只使用“远程”REPL 进程。

当启动这些工具里的 REPL 时，会如你预期的那样创建新的 JVM 进程，但那个进程在初始化后会启动一个 REPL 服务器，然后工具与它连接：从这些工具载入的所有代码、使用 REPL 用户接口获得的所有交互全都发生在网络管道上，尽管这是连接到 localhost 上的。非常简单，这些 REPL 服务器接受以文本格式发来的代码，像其他 REPL 那样对代码应用同样的读取、求值、打印过程，打印的结果再发送回工具的 REPL 用户接口，而不是 *out*。



Counterclockwise 和 SLIME 使用的 Clojure REPL 服务器分别是 nREPL 和 swank-clojure，可以轻松地集成进你的应用程序，让你可以打开一个 REPL，连接到部署的应用程序实例，不管是在哪里运行。两者都是使用容易获得的 Clojure 函数库作为 Maven 依赖。详情请查项目有关嵌入的文档：<http://github.com/clojure/tools.nrepl> 和 <https://github.com/technomancy/swank-clojure>。

总之，只要你可以让部署的应用程序启动一个 REPL 服务器并把你的 Clojure 工具集与它相连，就可以像在自己桌面上运行它一样使用 REPL 与远程 Clojure 环境交互。不过保险地说，不应该用 REPL 连接一个部署的、非本地的开发运行时来进行新的开发，除非在非常特殊的情形下。与生产环境相连的 REPL 连接可以是你能获得的最有效的监测、调试、偶尔的打补丁工具。

- 412 ➤ 监测与运行时分析。因为 REPL 提供了高保真的连接到远程环境，就像连接到本地 Clojure 环境一样，所以有机会用其他途径办不到的方式收集和监测运行时数据和事件。例如以在 REPL 里能够访问、操纵、分析的方式捕获按键事件数据是非常强大的。下面是催生这种用法的一两个简单函数：

```

(let [log-capacity 5000
      events (agent [])]
  (defn log-event [e]
    (send events #(if (= log-capacity (count %))
                     (-> % (conj e) (subvec 1))
                     (conj % e)))
    e)
  (defn events [] @events))

```

- ❶ 定义“日志”的容量——这里把大小写死了，不过这可以很容易地用一个配置设置来决定。
- ❷ “日志”开始时是一个内容是空的向量的代理。使用代理的目的是可以更新日志向量，不需要潜在地导致调用 `log-event` 的线程阻塞，例如像更新一个原子那样。
- ❸ 在记录日志时，用 `conj` 函数把日志附加到日志向量的末尾。如果新事件会超过定义的容量，就去掉日志向量的头部。
- ❹ 返回记录的事件，以便 `log-event` 函数可以轻易地用于线索形式里或用 `comp` 与其他函数组合。
- ❺ 定义一个简单的访问函数，以便事件向量可以从闭包里提取出来。

看看用这个事件记录的东西能做点什么。设想有一个 Web 应用，你想要追踪流量的来源，特别是注意新的“热”引用。为此可以用 `log-event` 来捕获每次请求数据的子集，这里我们将伪造一些数据，模拟从一些引用域来的请求日志：

```

(doseq [request (repeatedly 10000 (partial rand-nth [{:referrer "twitter.com"}
                                                       {:referrer "facebook.com"}
                                                       {:referrer "twitter.com"}
                                                       {:referrer "reddit.com"}]))]
  (log-event request))
:= nil
(count (events))
:= 5000

```

`log-event` 工作正常，保留最多 5000 个事件，前 5000 个事件从向量头滚出去了。现在通过 REPL 连接到远程 Web 应用时，可以用这些数据做你想做的任何事情，这里可以执行简单的求和，看看哪个引用域在最后 5000 次请求中带来的流量最大：

```

(frequencies (events))
:= {{:referrer "twitter.com"} 2502,
   {:referrer "facebook.com"} 1280,
   {:referrer "reddit.com"} 1218}

```

413

在最后 5000 次请求里，Twitter 是我们站点最大的引用者（凭借着模拟数据时把源于 Twitter 的“请求”加倍的把戏）。当然，可以做更多的事，马上可做的改进包括不仅捕获引用者的域名、或许基于时间戳来截短 `events` 向量从而只保留最后 10 分钟而不是固定数目的请求。

无论如何，结果是由于使用 REPL 能够进入远程 Clojure 应用，你可以拥有 Clojure 的全部设施来探查、摆弄、分析应用程序的状态。应对这类需求的典型方法是把数据转存到磁盘或数据库并从那儿分析，或者在应用程序里建立监控台和其他状态指示——都是已知量，但要求可能不现实或不可能的资源和规划。无论如何，没有什么可以胜过随时有 REPL 可以做任何即时的分析。

打补丁。一旦发现错误的原因并确定了解决问题的方案（可能在某种测试环境下验证了解决方案），把补丁弄进已经部署的应用而不引起停机有可能是很棘手的事情，这有赖于应用程序的具体情况和相关的本地政策。不过有了远程 REPL 会话，给已部署的应用打补丁很简单，就是把更新的代码载入即可。

不过，这需要小心处理、精心准备。特别是对于在 REPL 里载入代码时什么是可以更新的有一些限制，正如在 415 页“重定义结构的限制”一节里所讨论的。此外，还可能有组织工作的问题：如果补丁要求对多个文件进行修改，需要按合适的顺序载入这些文件。这是因为你所连接的 REPL 还保留着它在远程 classpath 所包含的所有旧代码，`require` 和 `use` 声明不会奇迹般地找到你机器上已更新的代码。

在小心给已部署的应用程序打补丁这个概念之外，还有快速迭代、更新“现场”环境的用例。有的情况下，用户的需求改变太快，方案的周转期至关重要，把“产品”作为一个开发环境对待是有优势的。^{注16} 在这种情况下，能够把本地的工具通过 REPL 加入到已经部署的应用程序并极快速地推送新代码是难以忽视的优势。

414 > “已部署” REPL 的特别考虑

如果说有一个交互式的连接到一个部署到生产环境下的应用这个概念令人不安，我们要提醒你，JMX（Java 管理扩展）多年来一直在提供一种广为使用但相对笨拙的机制来动态改变运行中的 Java 程序。之外，从一个生产环境使用或提供 REPL 时有一些东西应该考虑。

所有的改变都是临时的。与在开发环境下的 REPL 一样，任何在已部署的 Clojure 环境下做出的任何修改，严格来说都是暂时的。通过 REPL 载入代码和改变数据结构会影响到当前的 JVM/Clojure 进程，但都会消失，例如当重启所部署的 Clojure 应用时。这与在

注 16：最常见的例子是“用户接受测试”情景，你部署更新越快，你的委托人、客户和合作伙伴越高兴。不过这不存在于正常的生产环境。

开发环境中的某些工作流形成对照，在后者可以选择把项目的修改保存到项目的源文件里，再载入所修改的代码——下次为这个项目启动 REPL 时，修改的代码就会是最初载入的，从而使影响持久。

这一点是需要你在使用 REPL 时就明白它是连接到一个已经部署的程序上的，或者是做好计划规避的。一个办法是确保载入到已部署的程序的任何新代码同时以部署一个新构件或执行程序到远程环境来体现，从而在程序重启时，通过 REPL 动态载入的修改就总是也出现在 `.war` 或 `.jar` 等文件里，与被修改的代码对应。

网络安全与访问控制。 Clojure REPL 的网络实现没有在安全方面提供什么，包括鉴别和传输加密。这些事通常认为是可以相互独立地考虑。

在短时间内解决这些问题的一个简便方法是确保在已部署的程序里运行的 REPL 服务器与在操作系统防火墙的安全保护下的一个网络端口绑定。^{注 17} 然后应该建立一个 SSH 隧道或使用 VPN 来获得对远程系统的访问，通过安全的途径连接到正在运行的 REPL 服务器。

同样的，REPL 服务器也没有强制要求任何访问控制：一旦有了一个 REPL 连接，就有了 Clojure 运行时和它所处环境的无限制的访问权限。从安全的角度看，对待 REPL 连接应该像对待 SSH 会话一样小心。不过通过使用求值沙箱有一些办法可以限制 REPL 的功能。^{注 18}

重定义结构的限制

415

交互式地重定义 Clojure 部分程序的能力几乎没有限制：从函数到顶层数据结构到通过 `deftype` 定义的类型和通过 `defrecord` 定义的协议和多重方法的一切，都可以在运行时（当然是在每个结构的契约范围内）重新定义和修改，就像是载入新的或更新的代码一样。不过这一能力是几乎没有限制的，这里的“几乎”主要是由寄主引起的限制。

固定的 classpath。 JVM 的 classpath 通常不能在运行时修改或扩展。这意味着不能载入和使用在 JVM/Clojure 进程启动时没有预期或不存在的新依赖。在 JVM 和 Clojure 社区里有许多绕过这一限制的办法。^{注 19}

`gen-class` 从来都不是动态的。正如在 374 页“定义具名的类”一节里描述的，`gen-class` 严格在提前编译时生成静态 Java 类。这个 class，根据定义，是不能在运行时更新

注 17： 部署的环境典型地只暴露常用网络服务，如 HTTP、HTTPS 和 SSH 所用的端口，这可能是默认情况。

注 18： `clojail`——<https://github.com/flatland/clojail>，就是这样的沙箱之一，它已经通过了实战考验，在各种 Clojure IRC 频道里和难题网站 4clojure.com 上用于对各种不受信任的代码片段求值。

注 19： 其中一个是 `pomegranate`（详情见 <https://github.com/cemerick/pomegranate>），它会通过直接从磁盘或解决 Maven 依赖的方式把 `.jar` 文件加到 JVM 的 classpath 里。

的。JVM 生态系统已经演化出了一些方法来重新载入和更新静态类（像 gen-class 所生成的那样），但这些方法超出了在这里要讨论的范围。

类实例永远保留 inline 实现。由 deftype 和 defrecord 定义的类的 Inline 接口和协议实现不能动态更新这些类的已有实现。这里的解决办法包括把实现委派给单独的函数以便可以根据需要来重新定义（在例 10-1 里基于 reify 实现的 ActionListener 就是这样做的），在用 extend 实现协议时不是提供给 var 命名的函数，而用 #' 写法直接提供 var。一旦更新后者所包含的函数，新函数就会用于支持协议的实现。

除了 JVM 隐含的这些限制之外，还应该记住 Clojure 特有的两个问题：

重定义宏不会重新扩展对宏的使用。如果定义了一个宏，并在一个函数的定义里使用了宏，重定义宏并不会更新这个函数的定义。记住，宏只是在编译时使用的：为了“应用”新的宏实现，宏的所有使用也都必须重新载入（并因而重新编译）。

重新定义多重方法不会更新多重方法的转发函数。如在 305 页“重新定义一个多重方法不会更新这个多重方法的转发函数”所指出的，defmulti 有 defonce 的语义，因而转发函数不会因为载入被修改的 defmulti 形式而被更新。解决的办法是用 ns-unmap 来解除多重方法的 var——不幸的是，这会要求重新载入多重方法的每个方法的实现。

理解何时捕获一个值而不是解引用一个 var。别忘了 def 及类似的宏在当前命名空间内化一个 var，而 var 包含了你要定义的实际值。var 的符号求值得到这个 var 在那一时间点的值。如果只是一个简单的函数调用，这正是你想要的；不过，如果 var 是另一个函数的名称，你把 var 的值作为参数传递，而不是 var 自身。因而，如果要重新定义 var，新值就不会被考虑：

```
(defn a [b] (+ 5 b))
:= #'user/a
(def b (partial a 5))      ❶
:= #'user/b
(b)
:= 10
(defn a [b] (+ 10 b))      ❷
:= #'user/a
(b)
:= 10
```

❶ 把 b 定义为一个函数，由 var a 里的函数部分应用于一个参数得到的。

❷ 重新定义 a……

❸ 但因为 b 已经捕获了最初的函数 a，重新定义不会有影响。

这里的解决办法也是把 var 自身作为参数来提供：

```
(def b (partial #'a 5))      ❶
:= #'user/b
(b)
:= 15
(defn a [b] (+ 5 b))        ❷
:= #'user/a
(b)
:= 10
```

- ❶ 把 #'a 提供给 partial，而不仅仅是 a，这样的结果是 partial 返回的函数捕获了 var a，而不是当时的值。
- ❷ 现在重新定义 a……
- ❸ 现在调用 b 使用的是 #'a 保存的被重新定义的函数。

小结

<417

Clojure REPL 是一个工具，可让编程的每个小时平稳度过，加快发现每个新错误的原因，在生产环境下转败为胜。有效地使用它才能获得其他高品质的 Clojure 工具的全部好处，完全理解它的潜力是 Clojure 编程特有体验的一部分。

实战

数字与数学

许多类型的应用程序可以安全地忽视数学的细节和微妙之处，无论所用语言或运行时是什么。在这些情况下，经常是 I/O 开支、数据库查询和其他因素决定了应用程序的瓶颈。

不过也有一些领域，数值效率和正确性至关重要，而且这样的情况似乎在不断增加：大规模数据处理、可视化、统计分析以及相似类别的应用程序经常要求有其他地方所没有的数学严谨程度。Clojure 让你选择如何优化应用程序的数值用法，以满足两种不同轴向的要求。不需要牺牲简练、表达力或运行时的动态性，你可以选择：

1. 用原始数字类型获得这些原始类型可用范围内的最大效率。
2. 用封装的数字类型可靠地获得任意精度操作的效率。

在这一章，我们会深入探讨 Clojure 是如何建模数字并实现对数字的操作的。

Clojure 中的数字

第一步必须是理解手头的原始材料，即 Clojure 的数字表达，如表 11-1 所示。^{注 1} 从比较 Clojure 的数字与 Ruby 和 Python 的数字（熟悉 Java 的人应该马上感到放松，只要 Clojure 重用 Java 的数字表达）是不错的开始。

^{注 1} 关于与每种数字类型对应的 reader 语法，见表 1-2。

422 表11-1：Clojure的数字类型，并与Python和Ruby的表示比较

数值类型	Clojure 的表示	对应的 Python 表示	对应的 Ruby 表示
原始的 64 比特整数	<code>long</code>	没有，Python 和 Ruby 没有提供原始的数字类型（两个语言的所有表示都是封装的）	
原始的 64 比特	<code>double</code>		
IEEE 浮点数			
封装的整数	<code>java.lang.Long</code>	<code>int^a</code>	<code>Fixnum^b</code>
封装的小数	<code>java.lang.Double</code>	<code>float</code>	<code>Float</code>
“大”整数（无限的任意精度整数）	<code>java.math.BigInteger</code> 和 <code>closure.lang.BigInt</code>		<code>Bignum</code>
“大”小数（无限的任意精度小数）	<code>java.math.BigDecimal</code>	<code>decimal.Decimal</code>	<code>BigDecimal</code>
有理数（经常也称“为分数”）	<code>closure.lang.Ratio</code>	<code>fractions.Fraction</code>	<code>Rational</code>

^a Python 的 `int` 是 32 比特宽，因而与 JVM 更接近的等值类型是 `java.lang.Integer`，不过如下一节所讲，Clojure 扩宽所有的数字为 64 比特宽的表示，因而 `long`（这里是 `Long`）总是优选的。

^b Ruby 的 `Fixnum` 的范围依赖于实现和机器类型，通常是 31 或 63 比特宽（最后 1 比特是为实现它的 `fixnum` 语义保留的，关于 `fixnum` 的简短讨论，见 433 页“对象相同（identical?）”一节）。

下面我们解析几个术语，进一步讨论表中的一些关键事实。

Clojure 首选 64 比特（或更大）的表示

JVM 支持许多更小的表示——例如 32 比特的浮点数和 32 比特和 16 比特的整数（分别为 `float`、`int` 和 `short` 类型）——Clojure 的 reader 形式和数值操作产生 64 比特（或更大）的数字表示。所有的数学操作都可以用于这些更窄的类型，但 Clojure 会把返回值扩大到对应的 64 比特表示。例如扩大 32 比特的整数产生一个 64 比特的 `long`：

```
(class (inc (Integer. 5)))
;= java.lang.Long
```

与 Java 相比，这简化了 Clojure 的数字模型，Java 有三种不同的整数表示和两种不同的小数表示。

Clojure 的混合数字模型

与多数动态语言（包括 Ruby 和 Python）不同，Clojure 支持原始的数字类型（即 `long` 和 `double`）和封装的数字类型。

原始类型不是对象，而是直接与机器层的类型对应的值类型，从而某些对原始类型的数值操作是直接在硬件里实现的。Clojure 重用了 JVM 的 `long` 和 `double` 等原始类型，这

相反，封装的数字是由类定义的对象，`java.lang.Long` 就是一个封装的类，它的唯一目的就是作为 `long` 的容器，而 `java.lang.Double` 则是封装原始类型 `double` 的结果。^{注 2} 作为对象，它们会因每次分配存储产生代价，因而对它们的操作必然会更慢：它们必须经常首先被解封（以获得里面的原始数值），然后每次操作的结果可能还需要封装起来（要求分配与返回值类型匹配的封装类）。

Long 和 Double 既然并不提供比对应的原始类型更多的语义优势，为什么它们还存在？

孤立来看，似乎 `Long` 和 `Double` 封装类没什么用处：用在数学操作里意味着更多的代价，而且也不提供在范围或精度上像 `BigInteger` 和 `BigDecimal` 那样的优势。`Long` 和 `Double`（以及对应 JVM 原始类型的其他 Java 封装类，如 `Boolean` 和 `Short`）存在的原因是让 JVM 数字可以用于对象可用的所有情景。

例如，如果没有这些封装类，将不可能把数字存于哈希映射和其他集合类里。Java 的集合类 API 只能处理对象。

在 436 页“优化数字效率”一节里还会更多地讨论封装的数字，特别是它们对 Clojure 的影响。

由于原始类型与机器类型的对应关系、与封装数字相关的分配代价，使用原始类型总是更快——依赖于算法的具体情况有时快几个数量级。另一方面，JVM 原始提供的 64 比特的数字类型在范围和精度上的确有限制。为了填补这个缺口，Clojure 重用 `BigDecimal`，也可用 `BigInteger`——Java 中的两个无限的数值表示——而且提供自己的 `BigInt`。这些类型是由类定义的，因而与封装的数字一样有代价，但让你可以安全地使用任意大或任意精度的数字。

因而，可以从两个维度来理解 Clojure 的数字类型：特定表示是原始类型的或封装的，它的范围或精度是有限的或任意的。整数和小数不同的具体表示在表 II-2 中的矩阵中基于这两个轴显示出来。

^{注 2}：原始类型总是由全小写的名称表示（如 `double`），而封装的表示总是由大写的类名（如 `Double`）表示。

424 表11-2: Clojure里数字表示的矩阵比较

有限的范围 / 精度	任意范围 / 精度
原始类型 long、double	N/A
对象类型 java.lang.Long、java.lang.clojure.lang.BigInt、java.math.Double	BigDecimal、java.math.BigInteger

虽然 Clojure 使用、提供许多不同的数字类型，但是数学操作的语义在不同类型之间以及操作不同类型的数字时都是一致的。例如，dec 对它的参数减 1，不管参数的类型，并总是返回一个同样具体类型的数：

```
(dec 1)
;= 0
(dec 1.0)
;= 0.0
(dec 1N)
;= 0N
(dec 1M)
;= 0M
(dec 5/4)
;= 1/4
```

同样，可以自由地在一个操作里混合使用不同类型的数字：^{注3}

```
(* 3 0.08 1/4 6N 1.2M)
;= 0.432
(< 1 1.6 7/3 9N 14e9000M)
;= true
```

当算术操作的参数是混合类型时，结果的类型由 Clojure 的类型扩展规则决定。这在 425 页“数字传播规则”一节进行讨论。

有理数

有理数是可以表示为两个整数的分数的数字的集合，例如， $1/3$ 和 $3/5$ 都是有理数（分别等于 $0.333\dots$ 和 0.6 ）。多数语言——包括 Java、Ruby 和 Python——只支持整数和浮点数的表示法和算术。因而，当遇到一个有理数时，就马上把它“平整”为对应的最接近的浮点数：

```
# Ruby
>> 1.0/3.0
```

注 3：这项对 Java 的改进令人高兴，这是因为 Java 的数学操作并不能用于任意精度的数字类型尤其不便。

```
0.3333333333333333
```

```
# Python
>>> 1.0/3.0
0.3333333333333331
```

浮点数表示法在各种计算中的危害已经相当清楚（虽然很少被好好理解）。下面是一个常见的例子，用 Clojure 来示例：

```
(+ 0.1 0.1 0.1)
;= 0.3000000000000004
```

引入的误差只不过是浮点数表示法的结果。^{注4}Clojure 通过 (a) 允许有理数字面量和 (b) 不强制有理数为不精确的浮点数来避免这一点：

```
(+ 1/10 1/10 1/10)
;= 3/10
```

这样做的负面影响是，当它是可以“平整”一个有理数为整数而不损失精度时，Clojure 会那样做：

```
(+ 7/10 1/10 1/10 1/10)
;= 1
```

比例数可以明确地强制转换为浮点表示：

```
(double 1/3)
;= 0.3333333333333333
```

而浮点数也可以用 `rationalize` 函数转换为一个比例数：

```
(rationalize 0.45)
;= 9/20
```

数字传播规则

当一个算术操作涉及不同类型的数字时，操作返回值的类型由一个固定的层级决定。每种数字类型传播的程度各不相同，操作的参数传播度最高的一个决定返回值的类型（参见图 11-1）。

注 4：与许多其他运行时一样，JVM 依据 IEEE 754 规范来表示浮点数。如果对浮点数在内存里所占的比特数表示感兴趣，这里 (http://en.wikipedia.org/wiki/IEEE_754-2008) 对这个规范的概览是不错的起点。

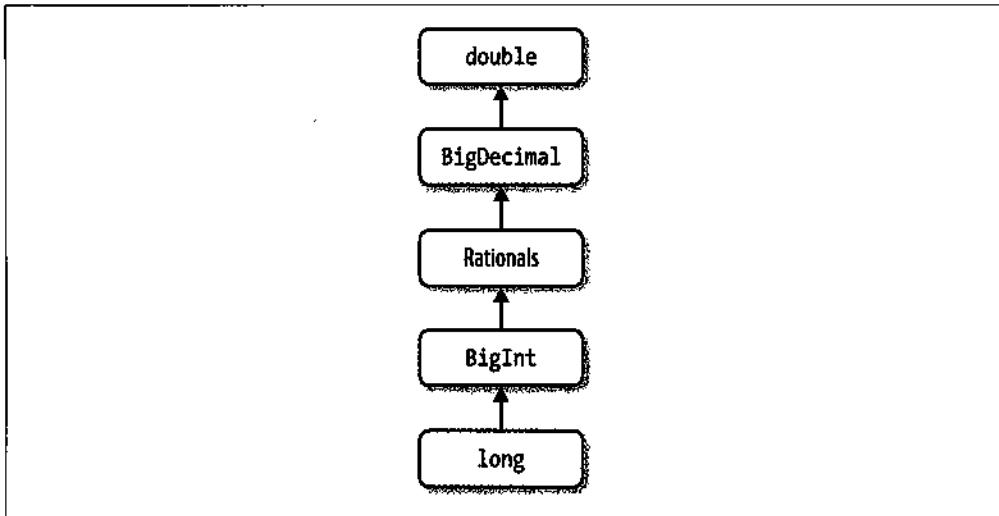


图11-1：Clojure的数字类型，按相对传播度排列

426> 每个数学操作都必须返回某个具体类型的值，并且当操作涉及不同类型的参数时，必须定义一个方法来选取操作的类型。Clojure 定义的具体层级是有顺序的，层级的实现永远不会让返回值的类型强制是“有损的”，例如每个 long 可以强制为 big int 或 rational 或 decimal 而不影响值的语义，相反却不成立。

这演示如下：

```

(+ 1 1)          ❶
:= 2
(+ 1 1.5)        ❷
:= 2.5
(+ 1 1N)         ❸
:= 2N
(+ 1.1M 1N)      ❹
:= 2.1M

```

- ❶ 操作于同一类型的参数返回那个类型的结果。
- ❷ 操作涉及 long 和 double 总会返回一个 double 类型。
- ❸ 操作涉及 long 和 BigInteger 总会返回一个 BigInteger 类型。
- ❹ 操作涉及 BigDecimal 和 BigInteger 总会返回一个 BigDecimal 类型。

427> 这里唯一的问题是任何涉及 double 的都会返回 double 类型，即使 double 不能正确地表示其他数值表示的全部范围。这是因为：

1. `double` (按照 IEEE 浮点数规范) 定义了一些特别值, 这些值不能表示为 `BigDecimal` (特别是 `Infinity` 和 `NaN` 两个值)。
2. `double` 是唯一一个天生就不精确的表示——一个涉及不精确数字的操作返回一个暗示精确但实际并不精确的类型值是有问题的。

类型传播的概念扩展到 Clojure 提供的算术操作以外。算术操作只是函数, 因而同样的规则应用于你写的能接受数字为参数、用 Clojure 的操作符实现的函数。考虑下面这个简单函数, 计算平方和:

```
(defn squares-sum
  [& vals]
  (reduce + (map * vals vals)))
 ;;= #'user/squares-sum
 (squares-sum 1 4 10)
 ;;= 117
```

如果加一个 `double`、一个 `big int`、一个 `big decimal` 或一个 `rational` 到这些被求和的数中, 将会怎么样? 我们将分别得到一个 `double`、一个 `big int`、一个 `big decimal` 或者一个 `rational`, 不管被求和的其他数值类型是什么, 这与我们在图 11-1 看到的类型传播层级是一致的:

```
(squares-sum 1 4 10 20.5)
 ;;= 537.25
 (squares-sum 1 4 10 9N)
 ;;= 198N
 (squares-sum 1 4 10 9N 5.6M)
 ;;= 229.36M
 (squares-sum 1 4 10 25/2)
 ;;= 1093/4
```

Clojure 中的数学

知道了可用的数字类型和表示法还不足以理解数学在 Clojure 里是如何建模的。Clojure 中的各种算术和相等操作符提供了更多的语义保证, 以支持在所有的数字类型上和在计算数学中经常遇到的许多问题都能获得合乎情理的结果, 如像溢出和向下溢出处理、算术结果升级的控制和集合类相等。

有限与任意精度

64 比特的 `long` 和 `double` 数字范围是相当大的, 在 $\pm 2^{63}-1$ 范围内的整数都是可以表示和操作的, $\pm 1.7976931348623157^{308}$ 范围内的小数也一样。大多数程序不需要更大的范围或精度, 因而 `long` 和 `double` 类型通常是完全够用了。

428

对于那些的确需要更大范围和 / 或精度的应用程序，“任意精度”的数字表示也是有的。这些表示总是封装的，Clojure 用 `clojure.lang.BigInt` 和 `java.math.BigInteger` 来表示任意范围的整数，用 `java.math.BigDecimal` 来表示任意精度的小数。

Java 已经提供了一个 BigInteger，Clojure 为什么还要有自己的 BigInt 类？

这一实现细节有两个起源。

第一点，`BigInteger` 有点问题，它的 `.hashCode` 实现与 `Long` 的不一致：

```
(.hashCode (BigInteger. "6948736584"))
;= -1641197977
(.hashCode (Long. 6948736584))
;= -1641198007
```

这个情况真的很糟糕，导致出现相等的值仅仅因为这些值的具体类型的实现细节不同就可能在集合中出现两次（或在哈希映射中被映射到两个不同的值上）的情况。^{注5}

第二点，虽然涉及 `BigInteger` 的所有操作都必须使用（更慢的）基于软件的实现，Clojure 优化了涉及 `BigInt` 的数学，如有可能使用原始（快得多）的操作，只要涉及的值在原始 64 比特 `long` 类型的范围之内。这意味着经常只是在数据实际需要时你才承担任意精度的整数算术操作的代价。

最后，你一般不需要关心有两个任意精度的具体整数类型，不管涉及什么类型，Clojure 所有数字操作的语义都不会受涉及的特定类型影响，而且结果永远不会是 `BigInteger` 类型的。

为了让所有关于精度和范围的讨论落到实处，我们来看看用这些任意大的整数能做什么，而用 `long` 类型是不能做到的。假设程序里一个值碰巧是 `long` 类型，能表示的最大值是多少：

```
429 ➤ (def k Long/MAX_VALUE)
;= #'user/k
k
;= 9223372036854775807
```

这是一个很大的数，但有时候它还不够大：

注5： 这方面的例子见 435 页“等值让你不致发疯”一节。

```
(inc k)
;= ArithmeticException integer overflow
```

Clojure 的基本数学操作符在整数溢出时会抛出异常，不会默默地“环绕”值，正如在 430 页“无检查的操作”一节里讨论的。但有时候，我们的确需要使用比 long 和 double 类型允许的最大值还要大的值，这时有以下一些选择。

明确地使用任意精度的数值。依赖本地的需求和上下文，可以使用 bigint 和 bigdec 类型强制函数来达到目的：

```
(inc (bigint k))
;= 9223372036854775808N
(* 100 (bigdec Double/MAX_VALUE))
;= 1.797693134862315700E+310M
```

或者使用合适的字面量写法。别忘了表 1-2 中讲到的，N 后缀产生一个 BigInt，而 M 后缀产生一个 BigDecimal：

```
(dec 10223372636454715900N)
;= 10223372636454715899N
(* 0.5M 1e403M)
;= 5E+402M
```

此外，整数字面量在超过 64 比特 long 类型的限度时会自动提升为 BigInt 类型：

```
10223372636454715900
;= 10223372636454715900N
(* 2 10223372636454715900)
;= 20446745272909431800N
```

为整数计算使用自动提升操作符。如果我们可以控制对特定函数的输入，明确使用任意精度是好的，那让我们使用 Clojure 的基本算术操作符，并在必要时依赖它的（快速）数字类型传播语义来引发返回值类型提升。如果知道某个计算会要求无限精度或范围，可以提供任意精度的输入。

另一方面，如果在实现一种计算或算法，只是有可能超过 long 整数的限度，而类型传播不足以保证正确结果时，可以使用 Clojure 里带撇号⁶ 的自动提升变体的算术操作符来自动地提升 long 类型的结果为 BigInt 类型，否则有可能溢出：

```
(inc' k)
;= 9223372036854775808N
```

430

不过这些操作符只在必要时提升结果，这保证了结果在 64 比特 long 类型范围内时不会

注 6：“带撇号”这里指用撇号（典型地是用一个 ' 表示）作为操作符名称后缀以表示基准操作符变体的约定。关于在 Clojure 外使用这一写法的详情，见 [http://en.wikipedia.org/wiki/Prime_\(symbol\)](http://en.wikipedia.org/wiki/Prime_(symbol))。

被提升：

```
(inc' 1)
:= 2
(inc' (dec' Long/MAX_VALUE))
:= 9223372036854775807
```

带撇号的变体可用于 Clojure 里可能导致整数溢出或向下溢出的所有数学操作符：`inc'`、`dec'`、`+`、`-` 和 `*`。

使用带撇号的操作符有轻微的代价，每个涉及有限表示的操作都必须检查结果，并可能需要用 `BigInt` 类型的操作代替 `long` 类型的操作重做。这是以最小的努力获得有保障的结果所不得不付出的代价。

无检查的操作

溢出和向下溢出是指对整数的操作结果超出了整数的表示法所能支持的限度。溢出和向下溢出的效果可能对每个曾在 Java 里处理过数字数据的人都不陌生，^{注7} 例如下面的 Java 代码：

```
System.out.println(Long.MAX_VALUE);
System.out.println(Long.MAX_VALUE + 1);
```

会产生如下输出：

```
9223372036854775807
-9223372036854775808
```

哎哟，出错了。当然，没有谁会故意给最大值常数加 1，不过如果我们的计算（出乎意料地）涉及到某个数值表示所定义的界限时，这样的事就会发生。

幸亏 Clojure 所有的数学操作都会检查溢出和向下溢出，并在必要时抛出异常：

```
Long/MIN_VALUE
:= -9223372036854775808
(dec Long/MIN_VALUE)
:= #<ArithmaticException java.lang.ArithmaticException: integer overflow>
```

这肯定比不得不追踪应用程序的怪异行为要好得多了，因为应用程序会由于一个操作在某个地方溢出而返回一个语义上是错误的“环绕”结果。

431 ➤ 不过在一些有限的情况下你可能想要保留 Java 操作符的无检查行为：

注7： 向下溢出和溢出在像 Python 和 Ruby 这样的语言中不是问题，这两门语言总是为所有的数学操作符应用自动提升以避免出现这种情形。

- 你想保留与 Java 在这方面不幸语义完全兼容——例如，假设你要用 Clojure 来实现以前用 Java 写的某个功能的新版本。
- 你想避免 Clojure 的溢出 / 向下溢出检查相关的（非常小的）代价。

Clojure 的所有操作符都有 `unchecked-*` 变体，这些操作符不进行溢出 / 向下溢出检查：

```
(unchecked-dec Long/MIN_VALUE)
;= 9223372036854775807
(unchecked-multiply 92233720368547758 1000)
;= -80
```

这些变体有些冗长 (`unchecked-multiply` 与 `*` 相比相当难用)。另一种办法是用 `set!` 把 `*unchecked-math*` 设置成真值，再执行数学操作都不需检查的顶层形式：

```
(inc Long/MAX_VALUE)
;= #<ArithmaticException java.lang.ArithmaticException: integer overflow>
(set! *unchecked-math* true)
;= true
(inc Long/MAX_VALUE)
;= -9223372036854775808
(set! *unchecked-math* false)
;= false
```



如果要使用 `*unchecked-math*`，最常见的做法是在应该使用无检查操作的形式前在源文件顶部用 `set!` 来设置它，可选在这些形式之后用 `set!` 把它设置回 `false`。`*unchecked-math*` 在载入每个源文件时重新设置，因而需要在每个使用它的文件里设置它，即不可能在一个顶层命名空间里设置它后，就可以期望它会在随后载入的文件里生效。如果不是这样的话，忘记在文件末尾把 `*unchecked-math*` 重设回 `false` 将会在你的（以及其他人的）代码里“渗透”无检查操作这种不幸的语义。

注意，通常不能用 `binding` 来设置 `*unchecked-math*`：

```
(binding [*unchecked-math* true]
  (inc Long/MAX_VALUE))
;= #<ArithmaticException java.lang.ArithmaticException: integer overflow>
```

这是因为 `*unchecked-math*` 控制的是编译器的操作，但 `binding` 形式直到它被求值时才会生效。这发生在编译器已经处理完某个顶层形式之后（有时是很久之后）。^{注8}

<432>

任意精度的小数操作的刻度和取整模式

在 Java 里使用 `BigDecimal` 最令人沮丧的事情之一是许多操作默认就会失败：

注8： 把这样的 `binding` 形式提供给 `eval` 的表达式会使用无检查的操作符编译、求值，因为编译是在动态作用域建立起来之后执行的……虽然我们无法想象你为什么要这样做。

```
new BigDecimal(1).divide(new BigDecimal(3));  
= java.lang.ArithmaticException:  
= Non-terminating decimal expansion; no exact representable decimal result.
```

这个代码本身已经冗长得令人痛苦，要让它能暂且工作起来而指定取整模式和最大刻度甚至更糟：

```
new BigDecimal(1).divide(new BigDecimal(3), new MathContext(10, RoundingMode.  
HALF_UP));  
  
= 0.3333333333
```

Ruby 的任意精度小数实现也有大致同样的代价，不过所平衡的东西是不一样的：不是强制你在每个操作中传递一个数学环境，而是在 `Decimal` 类里设置全局的数学环境参数。

Clojure 提供了一个宏 `with-precision`，这显著地简化了这一点。你提供想要的刻度（和一个可选的取整模式），那个信息就会为你传递给在 `with-precision` 的范围内执行的所有 `BigDecimal` 操作：

```
(/ 22M 7)  
;= #<ArithmaticException java.lang.ArithmaticException:  
;= Non-terminating decimal expansion; no exact representable decimal result.>  
(with-precision 10 (/ 22M 7))  
;= 3.142857143M  
(with-precision 10 :rounding FL00R  
(/ 22M 7))  
;= 3.142857142M
```

而且，如果你真的想那样做，也可以不用 `with-precision` 而通过设置 `*math-context*` 为合适的 `java.math.MathContext` 实例来设置刻度和取整模式：^{注9}

```
433 > (set! *math-context* (java.math.MathContext. 10 java.math.RoundingMode/FL00R))  
;= #<MathContext precision=10 roundingMode=FL00R>  
(/ 22M 7)  
;= 3.142857142M
```

相等与等值

Clojure 提供三种方式确定值是否相等，体现为三种不同的谓词函数。

注9：既然 `*math-context*` 是一个动态 var，可以通过使用 `set!` 或 `alter-var-root` 来改变它的值。至于选择用哪种，这取决于你想要的改变是在线程内还是在全局范围内生效。关于动态作用域，可以阅读 201 页“动态作用域”一节。

对象相同 (`identical?`)

对象相同, Clojure 里是用 `identical?` 实现的, 用来确定两个 (或多个) 对象完全是同一实例。这直接对应于 Java 里的 `==` (当用于比较对象引用时), Python 里的 `is` 和 Ruby 里的 `equal?` :

```
(identical? "foot" (str "fo" "ot"))
;= false
(let [a (range 10)]
  (identical? a a))
;= true
```

一般来说, 数字从来不会是 `identical?` 的, 即使提供字面量来比较 :

```
(identical? 5/4 (+ 3/4 1/2))
;= false
(identical? 5.4321 5.4321)
;= false
(identical? 2600 2600)
;= false
```

例外的是, JVM (因而 Clojure 也) 提供有限范围的 fixnum。Fixnum 是封装的整数值的池子, 总是优先使用池子里的数值而不是分配新的整数。Ruby 的 fixnum 语义涵盖它全部范围的整数, 而 Python 的 fixnum 只是在 -5 和 256 之间。^{注10}Oracle JVM 的 fixnum 范围是 ± 127 ^{注11}, 因而只有操作的返回结果是这个范围内的整数时, 它与等值的整数比较 `identical?` 才会成立 :

```
(identical? 127 (dec 128))
;= true
(identical? 128 (dec 129))
;= false
```

总之, 比较数字时不要用 `identical?` 是明智的。

引用相等 (=)

434

这是最常被用来指称“相等”的: 一种 (潜在可能) 类型敏感的、深度比较, 以确定值的结构是相同的。Clojure 中的 `=` 谓词采用了 Java 的引用相等的相等语义, 由 `java.lang.Object.equals` 定义。^{注12}Clojure 里的 `=` 在功能上等价于 Python 和 Ruby 中的 `==`。

注 10: 这一不同寻常的范围是 CPython 的一个实现细节。

注 11: JVM 的 fixnum 语义定义为原始值封装转换的一部分 (在 Java Language Spec 的第 5.1.7 节中讨论), 并且在其他版本和实现中可能会不同。

注 12: 在比较非数字时, Clojure 中的 `=` 只是委托给参数的 `Object.equals` 实现。`Object.equals` 的具体情况比我们在这里总结的要复杂得多。花时间看看相应的 javadocs 并完全吸收是非常值得的: <http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals%28java.lang.Object%29>。

这通常产生合乎直觉的、方便的结果，特别是在比较不同具体类型的集合类：

```
(= {:a 1 :b ["hi"]}
  (into (sorted-map) [[:b ["hi"]]] [:a 1]))
(doto (java.util.HashMap.)
  (.put :a 1)
  (.put :b ["hi"])))
:= true
```

注意，不同类型的集合类相互间从不`=`。例如，任何顺序性集合类（如向量、列表或序列）可能与另一种具体类型的顺序性集合类相等，但一个顺序性集合类决不会与一个集合或映射相等。

同样的机制也适用于数字相等。准确地说，`=`只会在比较同类的数字时才会为真，即使所表示的数值是等值的。例如，所有类型的整数都可以用`=`来比较（即使整数类型比Clojure偏爱的64比特`long`类型要小），不同宽度的有限精度的小数也相处得很好：

```
(= 1 1N (Integer. 1) (Short. (short 1)) (Byte. (byte 1)))
:= true
(= 1.25 (Float. 1.25))
:= true
```

不过，不同类的等值数字的比较，`=`永远不会返回真：

```
(= 1 1.0)
:= false
(= 1N 1M)
:= false
(= 1.25 5/4)
:= false
```

Clojure中的`=`有可能消除这些类型差异（就像Ruby和Python所做的那样），不过那样会带来一些运行时代价，是不会被那些处理异质数字数据、需要最大化程序的效率的人认可的。因而，Clojure选择了提供第三种意义上的相等，专门解决类型不敏感的等值测试。

数字等值（`==`）

Clojure的`==`谓词实现了数字等值，一种与我们的直觉理解相符、不受我们用来划分数字表示的人为类别影响的相等度量。在比较不同实现类的数字时`=`返回假，`==`则可能返回真，如果这些数字所代表的值在数值上是相等的：

```
(== 0.125 0.125M 1/8)
:= true
(== 4 4N 4.0 4.0M)
:= true
```

注意，即使是有理数在用 `=` 与对应的小数表示比较也是对的。

`=` 要求所有的参数都是数字，否则会抛出异常。这意味着如果不能肯定参数的类型，需要：

- 使用 `=`。
- 如果需要 `=` 的相等语义，需要检查参数（这里用 `number?` 就够了）来保护它。

```
(defn equiv?
  "Same as `=`, but doesn't throw an exception if any arguments are not numbers."
  [& args]
  (and (every? number? args)
       (apply == args)))
 ;;= #'user/equiv?
 (equiv? "foo" 1)
 ;;= false
 (equiv? 4 4N 4.0 4.0M)
 ;;= true
 (equiv? 0.125 0.125M 1/8)
 ;;= true
```

等值让你不致发疯

在异质类型的数字用于集合类时，Java 的数字相等的概念是真的难受。因为 Java 的集合类实现依赖于每个成员对相等的定义（用于确定集合归属和用一个键定位映射条目等事情），而 Java 数字定义为基于实现类来限制相等，结果可能出现像下面这样的情况：

```
java.util.Map m = new java.util.HashMap();
m.put(1, "integer");
m.put(1L, "long");
m.put(java.math.BigInteger.valueOf(1), "bigint");
System.out.println(m);

>> {1 bigint, 1 long, 1 integer}
```

436

哎哟：映射里的三个键我们肯定希望哈希到同一条目——这肯定会导致不可思议的微妙的错误。相比之下，Clojure 的集合类对数字键和成员归属使用它的数字等值定义来确定：

```
(into #{} [1 1N (Integer. 1) (Short. (short 1))])
 ;;= #{1}
 (into {}
  [[1 :long]
   [1N :bigint]
   [(Integer. 1) :integer]])
 ;;= {1 :integer}
```

小心混合浮点相等测试

浮点小数总是这样的，必须小心解释浮点数表示的细节。简单的操作有时也能产生“错误的”结果：

```
(+0.1 0.2)
;=0.3000000000000004
```

不同类型的小数相等比较可以产生出乎意外的结果。看这个：

```
(==1.1 (float 1.1))
;=false
```

注意，尽管用的是类型不敏感的`==`等值谓词结果也不为真。问题是`float`（32 比特表示）需要扩展为`double`（64 比特表示），而这是无法对 1.1 做到精确的：

```
(double (float 1.1))
;= 1.100000023841858
```

同样的行为也出现在 Java 里，返回的结果也是`false`：

```
1.1f == 1.1d
```

这一问题的根源在于 IEEE 浮点数规范。像 Ruby 和 Python 这样的语言只提供一种有限精度的数值表示，从而避开了这一问题。

优化数字效率

Clojure 提供了清晰而简洁地建模、实现数据和数学密集算法需要的所有原材料。不过，虽然表达力很重要，但是在这些情形下效率经常是压倒性的考虑——它是驱使各地的程序员用 C 和 Fortran 之类语言来实现程序里效率至关重要的部分，本来他们是要首选更高级语言的。这是不错的应对机制，并且总是可以作为最后选择的手段，^{注 13} 但经常是集成、构建和部署复杂度的来源（更不用说对开发者生产效率的削弱）。因而，在我们的高级语言里能做的事越多，我们就会越开心、高效。

那么，如何能够在 Clojure 里尽可能地提高数值密集代码的效率呢？

使用原始类型。如我们在 422 页“Clojure 的混合数字模型”一节里讨论的，原始类型不像封装的数字那样有分配和垃圾回收的代价，而且针对原始类型的多数操作是在非常低

^{注 13}：你可以很容易地在 JVM 和 Clojure 里调用原生函数库。参见 <https://github.com/Chouser/clojure-jna>，这是一个 Clojure 函数库，它提供了简洁的方式使用 JNA (<https://github.com/twall/jna>)。

层（经常是硬件自身）实现的。其他情况一样的话，同样的算法¹⁴用原始类型实现经常会比使用封装的数字要快一个数量级。在 Clojure 里这样做会让你实现的算法接近用 Java 实现相同功能时的运行效率。

避免集合类和序列。在需要考虑效率时尽可能使用原始类型的理念的必然结果是应该在这些情形下避免使用集合类和序列。正如我们在 423 页“Long 和 Double 既然并不提供比对应的原始类型更多的语义优势，为什么它们还存在？”里提到的，集合类只能处理对象，一般不能储存原始类型的值。当一个原始类型的值加到一个集合类时，它自动被“提升”为对应的封装类的一个实例——这种把原始类型和基于对象的类型系统嫁接起来的方法被称为自动封装，这是 Clojure 从 JVM 继承来的动态特性。

例如，把一个 `double` 添加到一个列表结果是它的封装类(`java.lang.Double`)的一个实例，用它的原始类型 `double` 值分配和初始化的。虽然相关的分配和垃圾回收在 JVM 上非常快，最快的分配和垃圾回收还是根本就不要这样做。因而，如果你真的想尽可能地提升代码最火热部分的效率，避免集合类和序列。

当放弃使用集合类和序列时，自然的退路就是使用原始类型的数组。

声明函数接受和返回原始类型

◀ 438

当 Clojure 编译函数时，它会生成一个对应的类，实现 Clojure 的 Java 接口之一 `clojure.lang.IFn`。`IFn` 定义了许多 `invoke` 方法，这些是你在调用 Clojure 函数时在底层调用的。¹⁵

所有参数和返回值在（无装饰的）函数边界都是对象。这些 `invoke` 方法所接受的参数和返回的值的根类型都是 `java.lang.Object`。这使得 Clojure 的动态类型默认值（即你的函数的实现决定了可接受参数类型的范围，而不是由语言强制的静态类型声明决定的）成为可能，但产生的副作用是迫使 JVM 封装这些函数作为参数传来的或作为结果返回的任何原始类型。因而，如果用一个原始类型参数调用一个 Clojure 函数，例如一个 `long` 类型——那个参数会被封装成一个 `Long` 对象以符合 Clojure 函数的底层的 `invoke` 方法的类型签名。同样，如果函数的返回值是一个原始类型的值，底层的 `Object` 返回类型确保这些原始类型在调用者接收到结果前封装好。

毫无疑问，Clojure 是一门动态编程语言，但它也认识到可选地提供类型信息以优化运行时的效率是非常值的平衡。基于类的类型提示¹⁶使得 Clojure 编译器在执行 Java 互操作

注 14：一个指数时间复杂度的算法不会在效率上超过一个多项式时间复杂度的算法，不管前者是否使用原始类型。也就是说：首先确保使用正确的算法，然后再考虑如何在必要时优化它。

注 15：以及碰巧需要在 Java 里调用 Clojure 函数所使用的方法；要学习在 Java（或其他基于 JVM 的语言也一样）里调用 Clojure 的一些细微之处，见第 9 章。

注 16：我们在 366 页“为了效率进行类型提示”一节广泛探索类型提示。

(如调用涉及 Object 的方法等) 时能够避免反射, 但提示并不改变实现这些方法的函数签名; 这些 invoke 方法继续接受所有的 Object 参数。相反, Clojure 也提供静态声明这些函数的参数和返回值为原始类型: 确切地说, 是 double 或 long 类型。

查看函数生成的类的方法就可以看到这一点。作为基准, 这里是一个接受单个参数、没有提示或类型声明的函数:

```
(defn foo [a] 0)
:= #'user/foo
(seq (.getDeclaredMethods (class foo)))
:= (#<Method public java.lang.Object user$foo.invoke(java.lang.Object)>)
```

注意, 只声明了单参数的 invoke; 正如所预期的, 它接受一个 Object 类型的参数, 也返回同样类型的值。还可以看到, 使用基于类的提示不会影响到函数底层的 invoke 方法签名:

```
(defn foo [^Double a] 0)
:= #'user/foo
(seq (.getDeclaredMethods (class foo)))
:= (#<Method public java.lang.Object user$foo.invoke(java.lang.Object)>)
```

即使 Double 在语义上是一个数字, 它仍然是一个类, 因而参数类型是 Object 的。

我们再试试可供使用的原始类型声明: ^long 和 ^double:

```
(defn round ^long [^double a] (Math/round a))
:= #'user/round
(seq (.getDeclaredMethods (round foo)))
:= (#<Method public java.lang.Object user$round.invoke(java.lang.Object)>
:= #<Method public final long user$round.invokePrim(double)>)
```

有变化的是 Clojure 使用了原始类型声明来编译了一个原始类型安全的快速路径进入生成的类, 采用 invokePrim 方法的形式, 它接受一个原始类型的 double, 返回一个原始类型的 long。用一个 double 来调用这个函数将会像你自己用 Java 写了这个 invokePrim 方法一样快。

不过, 它不会接受一个类型不符合的参数:

```
(round "string")
:= #<ClassCastException java.lang.ClassCastException:
:=     java.lang.String cannot be cast to java.lang.Number>
```

注意, 这个异常抱怨 "string" 不是 Number 类的。的确可以传任何封装的数字, 只要在预期的范围内:

```
(defn idem ^long [^long x] x)
;= #'user/long
(idem 18/5)
;= 3
(idem 3.14M)
;= 3
(idem 1e15)
;= 1000000000000000
(idem 1e150)
;= #<IllegalArgumentException java.lang.IllegalArgumentException:
;= Value out of range for long: 1.0E150>
```

另一方面，你可能注意到了，通常的 `invoke` 方法（带 `Object` 类型的参数和返回值）仍在；这是为了支持用封装的数字参数调用函数的情况。这意味着你可以继续与高阶函数^{注17}如 `map` 和 `apply` 一起使用带声明为原始类型参数的函数：

```
(map round [4.5 6.9 8.2])
;= (5 7 8)
(apply round [4.2])
;= 4
```

类型提示和类型声明都可以用在 `deftype` 和 `defrecord` 类型的字段说明（因而包括构造函数）中，详细讨论在 270 页的“定义你自己的类型”一节。◀ 440



支持原始类型的函数限于 4 个参数

前面的代码就像免费的午餐：Clojure 所有的表达力与 JVM 的原始类型数值高效组合在一起。不幸的是有一个缺点：声明为接受或返回原始类型的任何 Clojure 函数限制在 4 个参数内：

```
(defn foo ^long [a b c d e] 0)
;= #<CompilerException java.lang.IllegalArgumentException:
;= fns taking primitives support only 4 or fewer args>
```

这是由于实现上的一个细节：为了开发上面提到的效率而不加入严格静态的编译过程，每个可能接受原始类型的函数签名必须在单独的接口中定义。参数和返回值有三种类型（即 `double`、`long` 和 `Object`），即使允许最大 4 个参数也会产生数百种不同的接口。

类型错误与警告

声明参数和返回值为原始类型的能力可能导致明显不一致的情景。尽管 Clojure 是动态语言，在能够侦察到问题时，无论是通过直接分析或者类型推导，它的编译器仍然会抛

注 17： 如果需要复习一下高阶函数，参见 59 页“作为头等公民的函数以及高阶函数”一节。

出编译错误。

```
(defn foo ^long [^int a] 0)
:= #<CompilerException java.lang.IllegalArgumentException:
:= Only long and double primitives are supported>
(defn foo ^long [^double a] a)
:= #<CompilerException java.lang.IllegalArgumentException:
:= Mismatched primitive return, expected: long, had: double>
```

同样，如果 *warn-on-reflection* 绑定为 true 时，在试图用需要封装的值来 recur 时，编译器会发出警告，因为值的类型不会与绑定的声明（或推理得到的）类型匹配。考虑下面这个简单的循环，从 5 倒数到 0：

```
(set! *warn-on-reflection* true)
:= true
(loop [x 5]
  (when-not (zero? x)
    (recur (dec x))))
:= nil
```

x 在这里基于所提供的字面量推理得到一个 long，recur 的参数 ((dec x) 的结果) 也将是 long，因而这里没有不一致的地方。不过，如果使用自动提升的 dec' 操作符，recur 的参数就有可能是 BigInt，Clojure 会捕获到这一点的：

```
441 > (loop [x 5]
           (when-not (zero? x)
             (recur (dec' x))))
; NO_SOURCE_FILE:2 recur arg for primitive local:
;                   x is not matching primitive, had: Object, needed: long
; Auto-boxing loop arg: x
:= nil
```

如果试图用一个不兼容的原始类型 recur 到一个 long 类型的绑定，例如 double，也会发生同样的事：

```
(loop [x 5]
  (when-not (zero? x)
    (recur 0.0)))
; NO_SOURCE_FILE:2 recur arg for primitive local:
;                   x is not matching primitive, had: double, needed: long
; Auto-boxing loop arg: x
:= nil
```

这些警告并不限于在函数体内检查原始类型。下面这个函数返回 double，当试图用返回值与一个 long 绑定来 recur 时，也会引发警告：

```
(defn dfoo ^double [^double a] a)
```

```
;= #'user/dfoo
(loop [x 5]
  (when-not (zero? x)
    (recur (dfoo (dec x)))))

; NO_SOURCE_FILE:2 recur arg for primitive local:
;           x is not matching primitive, had: double, needed: long
; Auto-boxing loop arg: x
:= nil
```

在这种情景下，为了保证使用正确类型的原始类型值，可以使用 `long`，原始类型强制函数之一来避免封装和警告：

```
(loop [x 5]
  (when-not (zero? x)
    (recur (long (dfoo (dec x)))))

:= nil
```

Clojure 的类型强制转换函数——`short`、`int`、`long`、`double`、`float` 和 `boolean`——只在需要用它来除掉与反射或自动封装有关的编译器警告时才有用处。例如上面使用 `long` 清除了可能的自动封装。下面用 `double` 来清除一个反射调用：

```
(defn round [v]
  (Math/round v))
; Reflection warning, NO_SOURCE_PATH:2 - call to round can't be resolved,
;= #'user/round
(defn round [v]
  (Math/round (double v)))
;= #'user/round
```

当用于其他情景时，类型强制转换函数不会返回一个类型强制转换函数所指示的类型对应的值：

```
(class (int 5))
;= java.lang.Long
```

442

在这方面，类型强制转换函数与类型提示最为相似（在 366 页“为了效率进行类型提示”一节讨论），应该据此使用。

合理使用原始类型的数组

在 370 页“数组”一节已经大概描述了 Java 数组一般在 Clojure 里可能如何使用，在使用原始类型的数组时，有几点应该特别加以考虑。

孤立地修改局部数组是可以的。“如果一棵树在森林里倒下，没人听见，那么它出了声音吗？”照着这样的思路，绝对在有些时候使用可变数组不仅是可以接受的，而且是合理的，完全与 Clojure 的惯用法一致。

Clojure 强烈鼓励应用函数式编程——包括使用不可变数据结构、序列抽象、保持状态和身份的分离，以及在第 1 部分描述的其他一切，但它从根本上是一门实用的语言。如果你真的需要在可变的数组里“倒腾”以获得需要的效率特征，Clojure 也不会挡你的路。^{注18}而且只要你对可变数组的使用是孤立的（即不让可变数组从需要它的热点代码逃脱）、局部的（即你所改动的数组并不是全局可用的或不是传给你函数的参数），你可以很坚定地声称你的代码仍然在本质上是函数式编程，因为它仍保留着幂等语义。

这种方法是合理的，一个基本例子是构造一个数据集的直方图。在例 9-20 中我们用 Clojure 的 `frequencies` 函数来获得各种数据集的直方图，不过这里我们来看看自己实现生成直方图的函数需要做些什么。先假设我们处理的是有限整数数据，`var` 可能是最合适的表示：

```
(defn vector-histogram
  [data]
  (reduce (fn [hist v]
            (update-in hist [v] inc))
          (vec (repeat 10 0))
          data))
```

大概看看它的效率像什么样的：

```
(def data (doall (repeatedly 1e6 #(rand-int 10))))
; #'user/data
(time (vector-histogram data))
; "Elapsed time: 505.409 msecs"
;= [100383 100099 99120 100694 100003 99940 100247 99731 99681 100102]
```

④ ① 数据集只是随机的 Long 序列（用 `doall` 函数来保证完全实现）。

`vector-histogram` 使用了 Clojure 的不可变向量数据结构——非常高效，不过可能还不够高效。^{注19} 再试试用原始类型 long 的数组来维护直方图的计数：

例11-1：用临时数组来建直方图

```
(defn array-histogram
  [data]
  (vec
    (reduce (fn [^longs hist v]
              (aset hist v (inc (aget hist v)))
              hist)
            (long-array 10)
            data)))
```

注 18：同样的思想产生了易变集合，在 130 页“易变集合”一节中进行了描述，还有 `deftype` 的可变性，在 277 页的“类型”一节中讨论。

注 19：使用临时向量的确可以显著提升效率，但只及下面要谈的基于数组的实现一半。

这快多了，快大约 20 倍：

```
(time (array-histogram data))
; "Elapsed time: 25.925 msecs"
;= [100383 100099 99120 100694 100003 99940 100247 99731 99681 100102]
```

这是使用原始类型数组的完美地方：

1. 它与我们需要的模型和用法匹配，对于它能处理的输入类型也没有限制（`reduce` 可用于任何顺序性集合类，包括其他数组）。
2. 可变数组只在局部使用，从不跑到函数外。因而 `array-histogram` 是纯的，与 Clojure 里其他这样的函数相处完全自如。
3. 使用数组没有改变用户预期的语义，包括返回与 `vector-histogram` 相同的具体数据结构（向量）。

原始类型数组的机制

如果想从已有的集合类创建一个数组，可以使用 `into-array` 或 `to-array`。后者总是返回对象的数组，前者会返回集合类提供的第一个值的类型的数组，或者返回指定的超类型的数组：

```
(into-array ["a" "b" "c"])
;= #<String[] [Ljava.lang.String;@4413515e>
(into-array CharSequence ["a" "b" "c"])
;= #<CharSequence[] [Ljava.lang.CharSequence;@5acad437>
```

- 显式产生集合类值的超类型的数组，在与某些要求这样的数组的 Java API 互操作时这特别方便。

提供与所需原始类型对应的 `Class` 和要封装的值的集合类，也可以使用 `into-array` 来生成原始类型的数组：

```
(into-array Long/TYPE (range 5))
;= #<long[] [J@21e3cc77>
```

Clojure 提供了一些辅助函数用于创建原始类型的数组或引用数组：`boolean-array`、`byte-array`、`short-array`、`char-array`、`int-array`、`long-array`、`float-array`、`double-array` 和 `object-array`。这些函数都需要一个参数，或者是所需的大小，或者一个集合类：

```
(long-array 10)
;= #<long[] [J@12ee6d57>
(long-array (range 10))
;= #<long[] [J@676982f8>
```

或者，可以同时提供大小和序列两个参数，如果序列太短不够初始化整个数组，它会用对应类型的默认值来填充，`object-array`除外：

```
(seq (long-array 20 (range 10)))
;= (0 1 2 3 4 5 6 7 8 9 0 0 0 0 0 0 0 0 0 0)
```

`make-array` 用于创建任何大小或维度的新空数组，用所给类型的默认值来初始化（对对象类型是 `nil`，对布尔数组是 `false`，对原始数字数组是 `0`）：

```
(def arr (make-array String 5 5))
:= #'user/arr
(aget arr 0 0)
:= nil
(def arr (make-array Boolean/TYPE 10))
:= #'user/arr
(aget arr 0)
:= false
```

数组类。虽然 `make-array` 让你可以创建任何维度的数组，但有的情况下需要获得对应一个数组类型的 `Class`。例如为了把一个协议扩展为特定类型的数组，如例 6-2 所示。你用 `class` 总能得到一个数组类型的 `Class`：

```
(class (make-array Character/TYPE 0 0 0))
:= [[[C
```

不过创建一个数组只是为了得到一个 `Class` 实例似乎有些不对，如同创建某个长度的数组以得到指向这个长度数字的一个引用。

注意到上面的数组 `Class` 的打印表示使用了不同寻常的写法吗？这是 JVM 命名与数组类型对应的类的方法。每个封装类提供一个静态 `TYPE` 字段与它的原始类型的 `Class` 对应，不过没有为嵌套数组预定义的 `Class`。这时需要用 `Class.forName` 和 JVM 的写法来查找这种类型的 `Class`：

445 >

```
(Class.forName "[[Z")
:= [[Z
(.getComponentType *1)
:= [Z
(.getComponentType *1)
:= boolean
```

写法是有一点晦涩，但是规则的。每个前缀的方括号表示一层深度的数组，因而 "`[Z`" 对应于一维的 `boolean` 数组，"`[[Z`" 对应于二维的 `boolean` 数组，等等。对于每种原始类型都有一个保留字符：

- Z—`boolean`
- B—`byte`

- C—char
- J—long
- I—int
- S—short
- D—double
- F—float

数组特有的类型提示。总的说来，除非 Clojure 知道你所用的数组的类型，否则访问和变更操作结果会出现反射，因而与数组作为局部优化的用处相悖。有一组类型提示是专门为使用数组准备的。

```
^objects
^booleans
^bytes
^chars
^longs
^ints
^shorts
^doubles
^floats
```

在例 11-1 中你可以看到这在 `array-histogram` 里的使用情况。如果没有给 `reduce` 函数提示 `hist` 参数的类型，`aget` 和 `aset` 操作都会变成反射调用，结果是运行时比使用不可变向量的基准直方图实现还差 88 倍！

访问和变更。`aget` 和 `aset` 分别提供数组的访问和变更操作：

```
(let [arr (long-array 10)]
  (aset arr 0 50)
  (aget arr 0))
:= 50
```

两种操作都要求数组的类型是已知的以避免反射，通常由合适的类型提示来提供。在这
个例子里，`arr` 已知是 `long` 类型的数组，因为它在局部声明了。446

在访问和修改多维数组的值时有一些特别的考虑，我们很快就会谈到。

对数组使用 `map` 和 `reduce`。`map` 和 `reduce` 使用起来很令人愉快，但与泛型集合类一样，只能用于容纳对象的序列。因而，当应用于处理数组时，`map` 和 `reduce` 会引起对原始类型值的自动封装。

用 `map` 和 `reduce` 能做的任何事都可以展开成 `loop` 表达式，而这倒是提供了完全的原始类型支持。不过写 `loop` 形式很容易犯错误，因为你必须跟踪、管理所操作的数组的下标。

为了省去这些麻烦事，Clojure 提供了 `amap` 和 `areduce`，这两个宏仿照对应的函数式宏，但专门用来操作数组同时避免自动封装：

```
(let [a (int-array (range 10))]
  (amap a i res
        (inc (aget a i))))
  ;;= #<int[] [I@eaf261a>
  (seq *1)
  ;;= (1 2 3 4 5 6 7 8 9 10)
```

`amap` 预期 4 个参数：“源”数组用来映射一个表达式、给下标命名的名称（这里是 `i`）、给结果数组命名的名称（初始化为源数组的一个拷贝，这里是 `res`）以及一个表达式，它的结果将会设置为结果数组 `res` 在下标 `i` 的值。

`areduce` 以同样的方式工作：

```
(let [a (int-array (range 10))]
  (areduce a i sum 0
           (+ sum (aget a i))))
  ;;= 45
```

`a` 和 `i` 所起的作用与在 `amap` 里一样。`sum` 是累加器的名称（与提供给 `reduce` 的函数的第一个参数对应），然后是累加器的初始值，然后是一个表达式，它的值将成为下次迭代中累加器的值——或者在归约完成后 `areduce` 形式的结果。

多维的考虑。虽然 `aset` 和 `aget` 容易用于一维数组，但用于多维数组时需要额外小心。虽然多维数组“最终的”值是你所指定的原始类型，中间层是对象的数组（其他数组）。此外，因为 `aget` 和 `aset` 不可能为所有可能数组维度提供函数变体（只是参数个数不同），它们通过递归使用 `apply` 来支持多维操作、获取或设置多维数组的每一层。

447 所有这些累积起来对多维数组的简单操作也可能产生可怕的效率结果：

```
(def arr (make-array Double/TYPE 1000 1000))
 ;;= #'user/arr
 (time (dotimes [i 1000]
   (dotimes [j 1000]
     (aset arr i j 1.0)
     (aget arr i j))))
 ; "Elapsed time: 50802.798 msecs"
```

因为 `aset` 不为 N 维数组提供直接的函数变体，在使用 `apply` 传播到其他参数时，`1.0` 被 `aset` 封装起来。而且我们没有办法提示或声明 `arr` 是一个原始类型的数组，因而涉及的所有操作都反射地执行。`aget` 和 `aset` 唯一的快速通道就是操作合理提示的一维数组。

“修正”的办法是手工展开多维数组，提供必要的类型提示：

```

(time (dotimes [i 1000]
    (dotimes [j 1000]
        (let [^doubles darr (aget ^objects arr i)]
            (aset darr j 1.0)
            (aget darr j))))))
; "Elapsed time: 21.543 msecs"
;= nil

```

是的，在效率上这比前面对多维数组使用 `aset` 和 `aget` 的幼稚作法相比相差 2600 倍，而且与等价的 Java 代码一样快。^{注 20}

多维数组操作的自动类型提示

现在我们知道如何获得最佳效率，但那需要一些可能容易出错的类型提示、写更多我们不愿写的代码：数组每深入一层就需要单独的 `let` 绑定。这要求一两个宏来对多维数组自动完成解包和合理提示类型：^{注 21}

例 11-2: `deep-aget`

```

(defmacro deep-aget
  "Gets a value from a multidimensional array as if via `aget`,
  but with automatic application of appropriate type hints to
  each step in the array traversal as guided by the hint added
  to the source array.

  e.g. (deep-aget ^doubles arr i j)"
  ([array idx]
   `(aget `array `idx))
  ([array idx & idxs]
   (let [a-sym (gensym "a")]
     `(let [~a-sym (aget ~(vary-meta array assoc :tag 'objects) `idx)]
        (deep-aget ~(with-meta a-sym {:tag (-> array meta :tag)}) `@idxs)))))

```

448

- ① 如果我们访问的是一维数组（由单一下标表明），那么就直接用 `aget`，并且假设符号 `array` 已经合理地提示了类型。
- ② 如果我们还在最终数组“之上”，就获得下一层的数组，确保提示 `aget` 的参数 `array` 是 `^objects`。
- ③ 我们用 `a-sym`（下一层的数组）递归调用 `deep-aget`，万一是最终数组就重新应用最终数组类型提示；如果是那样的话，在第 1 步的单下标函数变体的 `deep-aget` 会接收这个调用，并执行最终的 `aget`。

^{注 20} 等值的 Java 代码实际上基本做了同样的工作，不过 Java 的语法为支持简洁的数组访问和变更做了优化，而且它的静态编译模型可以很容易地推导出涉及的中间类型。

^{注 21} 这一方法最初是在 <http://clojure-me.cgrand.net/2009/10/15/multidim-arrays> 里描述的。

deep-aset 不得不采取与 deep-aget 不同的方法，因为它需要使用 deep-aget 来高效地遍历多维数组以到达最后它能对最终数组的值应用 aset 的地方：

例11-3：deep-aset

```
(defmacro deep-aset
  "Sets a value in a multidimensional array as if via `aset`,
  but with automatic application of appropriate type hints to
  each step in the array traversal as guided by the hint added
  to the target array.

  e.g. (deep-aset ^doubles arr i j 1.0)"
  [array & idxsv]
  (let [hints '{booleans boolean, bytes byte
               chars char, longs long
               ints int, shorts short
               doubles double, floats float}
        hint (-> array meta :tag)
        [v idx & sxdi] (reverse idxsv)
        idxs (reverse sxdi)
        v (if-let [h (hints hint)] (list h v) v)
        nested-array (if (seq idxs)
                         `(~(deep-aget ~(vary-meta array assoc :tag 'objects) ~@idxs)
                            array)
                         a-sym (gensym "a"))
        `(let [~a-sym ~nested-array]
           (aset ~(with-meta a-sym {:tag hint}) ~idx ~v))))
```

- ① 在数组提示符号和原始类型强制转换函数名称之间维持着一个映射；如果 array 用前者之一来提示，那么对应的类型强制转换表达式将会与 aset 和传给 deep-aset 的值一起使用。举例来说，这让用户可以，用 long 来设置 double 类型的多维数组的值，因为 long 会通过 (double v) 在 aset 调用中被强制转换。

449 使用 deep-aget 和 deep-aset，可以得到的效率如同手动解包和提示 N 维数组：

```
(time (dotimes [i 1000]
  (dotimes [j 1000]
    (deep-aset ^doubles arr i j 1.0)
    (deep-aget ^doubles arr i j)))
; "Elapsed time: 25.033 msecs"
```



当 *warn-on-reflection* 设置为 true 时，如果数组的类型不能被推理得到，调用 aget 和 aset 会抛出警告，除非是多维的函数变体！

用 Clojure 可视化芒德布罗集

用得很滥的斐波那契数和素数生成器经常用于微基准测试数值效率，让我们选择一个比这略为有趣一些的例子。可视化芒德布罗集²²（或者其实是任何分形的可视化）一直是常见的实习科目，在这里它可以很好地展示如何在 Clojure 里优化数值算法。

芒德布罗集由一个复数多项式迭代应用定义的：

$$z_{k+1} = z_k^2 + c$$

其中 c （一个复数）是芒德布罗集的一员，如果 z_0 的初始值为 0，随着 k 的增长， z_{k+1} 是有界的。从这一计算产生无界结果的 c 称为向无穷逃逸。

首先，我们先来看看用 Clojure 幼稚地实现芒德布罗集，²³ 包括渲染这一实现结果的一两个实用函数：

例11-4：用Clojure实现的芒德布罗集

```
(ns clojureprogramming.mandelbrot
  (:import java.awt.image.BufferedImage
           (java.awt Color RenderingHints)))

(defn- escape
  "Returns an integer indicating how many iterations were required
  before the value of z (using the components `a` and `b`) could
  be determined to have escaped the Mandelbrot set. If z
  will not escape, -1 is returned."
  [a0 b0 depth]
  (loop [a a0
         b b0
         iteration 0]
    (cond
      (< 4 (+ (* a a) (* b b))) iteration
      (>= iteration depth) -1
      :else (recur (+ a0 (- (* a a) (* b b)))
                   (+ b0 (* 2 (* a b)))
                   (inc iteration)))))

(defn mandelbrot
  "Calculates membership within and number of iterations to escape
  from the Mandelbrot set for the region defined by `rmin`, `rmax`,
  `imin` and `imax` (real and imaginary components of z, respectively)."
```

450

注 22：关于芒德布罗集、背后的数学以及如何生成可视化，请参见 http://en.wikipedia.org/wiki/Mandlebrot_set 里的简介。不得不提的还有，Jonathan Coulton 有关芒德布罗集和它的创造者 / 发现者 Benoit Mandelbrot 的歌和音乐视频在 <http://www.youtube.com/watch?v=ES-yKOYaXq0>，非常棒。

注 23：简单一点的实现是可能的，例如使用 iterate 函数来惰性计算复数多项式的结果，按最大迭代次数从惰性序列的头部只 take 足够多的结果。这样的方法简洁许多，但因为要使用惰性序列和集合类，计算结果将会是封装的——因而慢许多。

Optional kwargs include `:depth` (maximum number of iterations to calculate escape of a point from the set), `:height` ('pixel' height of the rendering), and `:width` ('pixel' width of the rendering).

Returns a seq of row vectors containing iteration numbers for when the corresponding point escaped from the set. -1 indicates points that did not escape in fewer than `depth` iterations, i.e. they belong to the set. These integers can be used to drive most common Mandelbrot set visualizations."

```
[rmin rmax imin imax & {:keys [width height depth]}
                         :or {width 80 height 40 depth 1000}]

(let [rmin (double rmin)
      imin (double imin)
      stride-w (/ (- rmax rmin) width)
      stride-h (/ (- imax imin) height)]
  (loop [x 0
         y (dec height)
         escapes []]
    (if (= x width)
        (if (zero? y)
            (partition width escapes)
            (recur 0 (dec y) escapes))
        (recur (inc x) y (conj escapes (escape (+ rmin (* x stride-w))
                                                (+ imin (* y stride-h))
                                                depth))))))

(defn render-text
  "Prints a basic textual rendering of mandelbrot set membership,
  as returned by a call to `mandelbrot`."
  [mandelbrot-grid]
  (doseq [row mandelbrot-grid]
    (doseq [escape-iter row]
      (print (if (neg? escape-iter) \* \space)))
    (println)))

(defn render-image
  "Given a mandelbrot set membership grid as returned by a call to
  `mandelbrot`, returns a BufferedImage with the same resolution as the
  grid that uses a discrete grayscale color palette."
  [mandelbrot-grid]
  (let [palette (vec (for [c (range 500)]
                       (Color/getHSBColor 0.0 0.0 (/ (Math/log c) (Math/log 500))))))
        height (count mandelbrot-grid)
        width (count (first mandelbrot-grid))
        img (BufferedImage. width height BufferedImage/TYPE_INT_RGB)]
```

451

```
^java.awt.Graphics2D g (.getGraphics img)
(doseq [[y row] (map-indexed vector mandelbrot-grid)
        [x escape-iter] (map-indexed vector row)]
  (.setColor g (if (neg? escape-iter)
                  (palette 0)
                  (palette (mod (dec (count palette)) (inc escape-iter)))))
  (.drawRect g x y 1 1))
(.dispose g)
img))
```

`mandelbrot` 函数返回一个网格，每个成员是在对应点逃逸到无穷前多项式需要的迭代数字，从不逃逸到无穷的点因而是由 `a-1` 表示的迭代计数的集合成员。芒德布罗集最简单（而且可以承认是较粗糙的）的可视化可以通过打印星号和空白到控制终端获得，这是由 `render-text` 函数实现的：^{注24}

不过，如果要放大（即改变可视化的域聚焦在芒德布罗集更小、更有趣的子特征）需要比这里执行的 100 次多得多次数的 `escape` 函数迭代。快速计时所需要的——例如产生 1600×1200 的图形——显示上述实现可能不如我们所希望的那样快：

452

注 24：文本的性质——字符的高度大于它的宽度——需要使用扭曲的长宽比（这里是 80×40 ）以便结果与通常用 1:1 的长宽比渲染的图像看起来相似。

```
(do (time (mandelbrot -2.25 0.75 -1.5 1.5
                      :width 1600 :height 1200 :depth 1000))
      nil)
    ; "Elapsed time: 82714.764 msecs"
```

如果要创建一个应用程序交互式地探索芒德布罗集（或者执行某些自动的探索），这样的效率将会完全不可接受。

代码里最热的部分显然是 `escape` 函数里的 `loop` 部分。它所执行的计算复杂度我们是无能为力的——这是由芒德布罗集定义的多项式所决定的，但可以消除数值封装。

正如在 438 页“声明函数接受和返回原始类型”一节里讨论的，Clojure 的函数是用 Java 方法实现的，接受 `java.lang.Object` 类型的参数，因而 `mandelbrot` 函数调用 `escape` 函数所用的（芒德布罗集多项式里的 `z` 实数和虚数部分）`a0` 和 `b0` `double` 原始类型最后被封装成为 `Double` 对象。这像瀑布那样传递到 `loop` 形式里绑定的类型，从而毁掉了整个函数的效率：`a` 和 `b` 分别用 `a0` 和 `b0` 初始化也隐式地把前者绑定的类型设置为 `Double`，因而 `escape` 里的每个算术操作最后都不得不了解封装这些值，其结果在 `recur` 形式里与 `a` 和 `b` 绑定时必然再被封装。

解决的办法是消除第一步的无类型绑定。注意在更新后的 `escape` 函数里实现 `a0` 和 `b0` 参数的 `^double` 类型声明，其他方面保持不变。

例11-5：更新的`escape`函数声明数字参数为原始类型的

```
(defn- escape
  [^double a0 ^double b0 depth]
  (loop [a a0
         b b0
         iteration 0]
    (cond
      (< 4 (+ (* a a) (* b b))) iteration
      (>= iteration depth) -1
      :else (recur (+ a0 (- (* a a) (* b b)))
                    (+ b0 (* 2 (* a b))))
                  (inc iteration))))
```

现在，`escape` 函数生成的类接受原始类型的 `doubles` 为它的前两个参数。Clojure 的类型推演完成其余的事：`loop` 里的 `a` 和 `b` 绑定也是类型为 `double` 的，这让所有的算术计算都只用原始类型的参数执行，当 `recur` 的参数重新与 `loop` 里对应的名称绑定时避免封装结果。

453

实际的提升有多少呢？一个数量级的改进：

```
(do (time (mandelbrot -2.25 0.75 -1.5 1.5
                      :width 1600 :height 1200 :depth 1000))
    nil)
  ; "Elapsed time: 8663.841 msecs"
```

有了这一改进，就可以继续下去，开始探索芒德布罗集，用 `render-image` 函数生成一个栅格化的芒德布罗集成员网格，见图 11-2 和图 11-3。

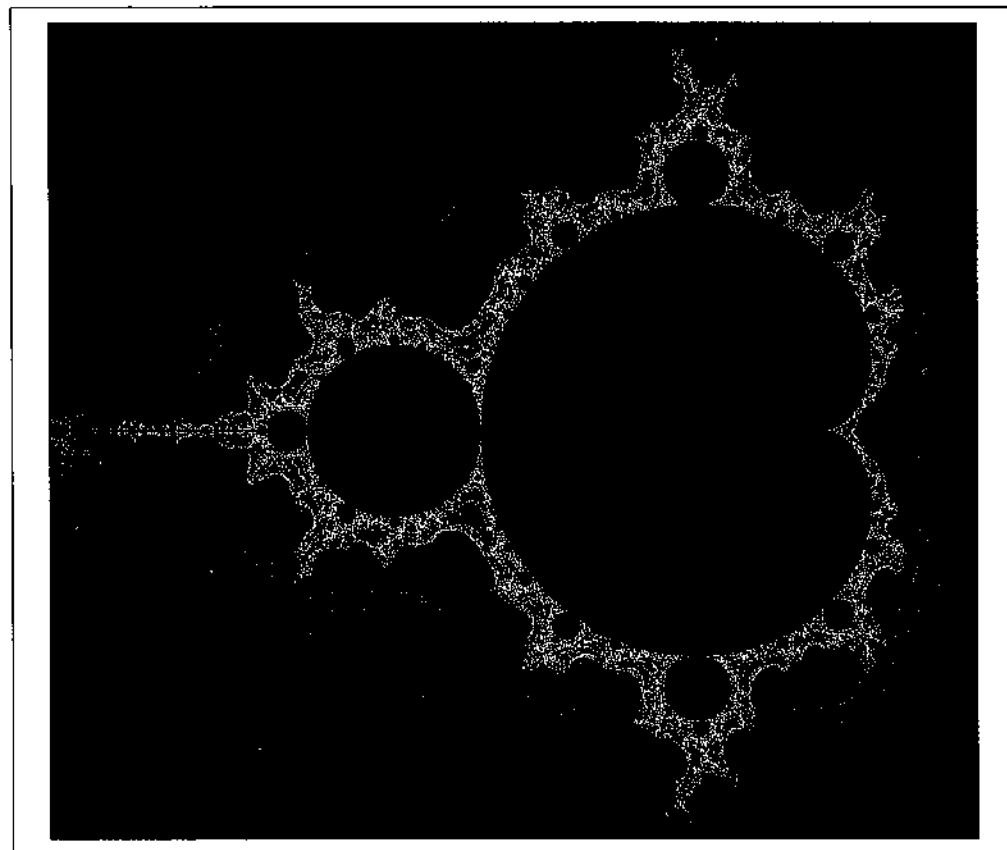


图 11-2：(render-image (mandelbrot -2.25 0.75 -1.5 1.5 :width 800 :height 800 :depth 500))

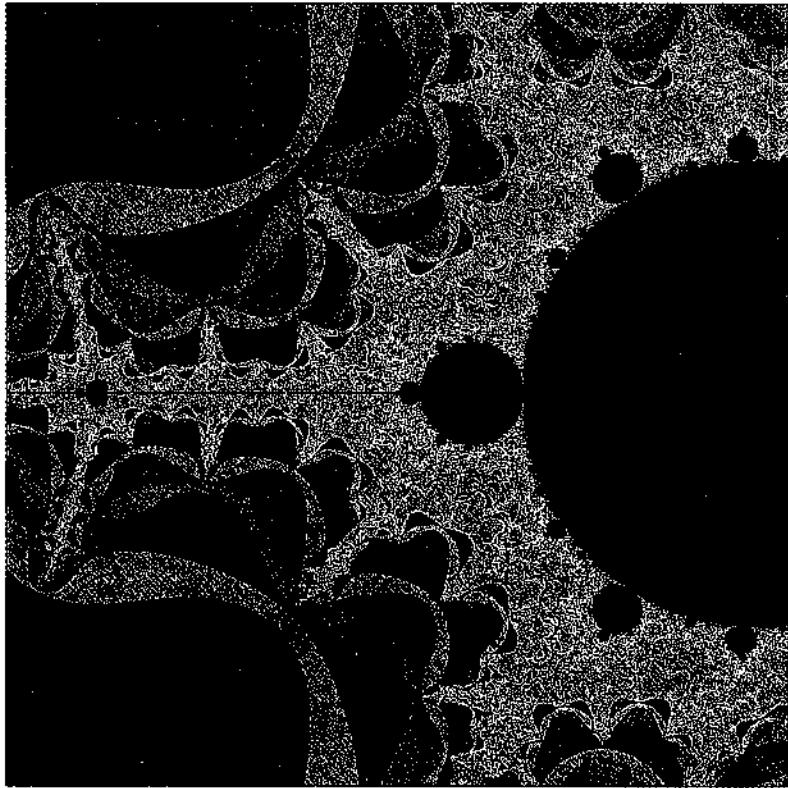


图11-3: (render-image (mandelbrot -1.5 -1.3 -0.1 0.1 :width 800 :height 800 :depth 500))

454 ➤ render-image 使用一个固定的、离散的灰度调色板。把它改成使用彩色的或连续的调色板留作练习。

455 ➤  如果你使用的是一个典型的文本 REPL,^{注25} 对上述表达式求值将会返回一个 BufferedImage 实例。可以用 JDK 的 ImageIO 类和它的 write 方法把图形保存到磁盘里：

```
(javax.imageio.ImageIO/write *1 "png" (java.io.File. "mandelbrot.png"))
```

这将会把图移绑定到 *1^{注26} 作为 PNG 保存到你当前目录的 *mandelbrot.png* 文件里。写一个函数封装这个调用应该是很容易的。

注 25： 支持混合内容的 Clojure REPL 开始出现，在这样的 REPL 里，求值 render-image 将立即显示所生成的图形，不需你先把它保存到磁盘里或用其他方式从 REPL 获得图形数据。关于这方面的例子，请参见 <http://cemerick.com/2011/10/26/enabling richer-interactions-in-the-clojure-repl>。

注 26： 记得 *1 总是与 REPL 里最后求值的表达式的值绑定。关于详情和好用的类似 var 见 399 页“REPL 绑定的 var”一节。

有各种各样加快芒德布罗集计算的高级算法和技巧，可以大大改进 `mandelbrot` 的运行时间，不过那超出了本书的讨论范围。一些 Clojure 特性的优化包括：

- 不在一个命令式的 `loop` 里进行所有的计算，例如用 `for` 把它重新塑造成惰性序列，这让你很容易用 `pmap` 把计算并行化。
- 从 `mandelbrot` 返回一个网格（一排排点逃逸迭代数的序列）提供了很大的灵活性。我们已经看到，可以使用同样结果的数据结构生成文本表示或位图，同样，也可以换入 3D 渲染函数而不用改变 `mandelbrot` 或 `escape` 函数。产生那样的网格的确有一些代价，因而可能值得通过 `mandelbrot` 向 `escape` 传入指向回调函数的引用，因而每个点的动作（不管那是向终端打印一个字符或在位图上画一个像素）不需要集合类和序列相关的任何分配就可以执行。当然，这个回调函数应该把它的参数像 `escape` 那样声明为原始类以避免封装。



设计模式

设计模式是对反复出现的问题的通用的、可重复使用的解决方案。作为面向对象编程的标志，设计模式提供了 Java 和 Ruby 世界许多人都很熟悉的共同词汇。

另一方面，模式也可能成为冗长和样板的来源。对于这一点，Paul Graham 评论道：在一门语言里存在和使用设计模式表明了语言自身的弱点，而不是解决手头问题的结果：

在我看到自己程序里的模式时，我认为它是麻烦的征兆。程序的形态应该只反映它需要解决的问题。代码里任何其他规则性，至少对我来说，是我所用的抽象还不够强大的一个征兆……

——Paul Graham, <http://www.paulgraham.com/icad.html>

Graham 远不是做出这样评论的第一人。Peter Norvig 在之前已经具体展示了 (<http://www.norvig.com/design-patterns/>) Lisp 语言或者简化了设计模式或者使得多数设计模式无足轻重不为人注意。Clojure 延续了这一传统：由于有功能强大的头等函数、动态类型和不可变值等构造，许多最常见的设计模式消失了。而有了宏，Clojure 给了你避免自己代码重复所需的工具。

Clojure 代码纳入常见设计模式的例子散见于本书：

监听者、观察者。作为头等函数和动态类型的牺牲品，这些不过是一个相关操作发生时被调用的函数，在 176 页“观察器”一节对引用类型使用观察器可以看出这一点。在引用类型之外，偏爱不可变的值意味着你需要跟踪的可变的东西的范围大大缩小了。

抽象工厂、策略、命令。如果任何种类的功能你有多重实现——不管它是产生不同类型或配置的值还是实现算法的不同变化——都不需要产生一个 FactoryFactory^{注1} 或一个环境来调用你的算法实现。两种情况下，都只需要另一个函数即可。

迭代器。迭代器完全被序列取代，已在 89 页“序列”一节描述过，通过像 map 这样的函数使用序列是声明式的、毫不费力。

适配器、包装器、代理。这些是因其他地方不灵活的类型层级所需，协议令这些都没有必要，不需要继承、调整或包装，就可以为已有类型正交地定义新行为。见第 6 章。

备忘录。在对象变化的上层选一个共同的 API 解决了问题最机械化的部分（API 通用性），但没有做什么去解决可变状态自身显眼的复杂度。不可变集合类和记录反转了这一策略：可变状态排除在值之外，保留值之前的版本既代价低廉又容易做到，状态转换留给引用类型（每种提供不同的 API 来适应各自特别的并行和变化语义）。

模板方法。类继承的局限已经广为人知，数百万的人每天深受其害，但仍然使用广泛，在许多语言中经常作为组合功能唯一的办法。关注的混合令人生疑，最好重塑为一个高阶函数，它可以接受实现可变行为的函数作为参数，同时提供共享功能的标准实现。例如，如果需要基于应用程序内部的某些数据结构调用不同提供者提供的等价 HTTP API，高阶函数可以让我们定义提供者特有的功能为单独的函数，而把通用的 HTTP 功能保持在一个地方：

```
(defn- update-status*
  [service-name service-endpoint-url request-data-fn]
  (fn [new-status]
    (log (format "Updating status @ %s to %s" service-name new-status))
    (let [http-request-data (request-data-fn new-status)
          connection (-> service-endpoint-url java.net.URL. .openConnection)]
      ;;; ...set request method, parameters, body on `connection'
      ;;; ...perform actual request
      ;;; ...return result based on HTTP response status
      )))

(def update-facebook-status (update-status* "Facebook" "http://facebook.com/apis/...")
  (fn [status]
    {:params {:_a "update_status"
              :_t status}
     :method "GET"}))

(def update-twitter-status ...)
(def update-google-status ...)
```

注 1： 这些事留给创建工厂的专家：<http://discuss.joelonsoftware.com/default.asp?joel.3.219431.12>。

让我们来深入发掘某些其他更常见的模式，个别的已经在其他语言中深深地植入了共通的编程实践，可能需要花些工夫把它们再过滤出来。 ◀ 459

依赖注入

在许多面向对象的编程语言里，依赖注入是把一个类从它所依赖的其他对象解耦的一种方法。一个对象不是在内部初始化其他对象，而是接受经常由运行时或托管你的程序的应用程序容器自动提供的那些对象为参数。

在一门像 Java 这样的静态语言中，通过面向接口而不是具体类编程来实现。考虑随处可见的宠物商店概念的这个实现：

```
interface IDog {
    public String bark();
}

class Chihuahua implements IDog {
    public String bark() {
        return "Yip!";
    }
}

class Mastiff implements IDog {
    public String bark() {
        return "Woof!";
    }
}

class PetStore {
    private IDog dog;
    public PetStore() {
        this.dog = new Mastiff();
    }

    public IDog getDog() {
        return dog;
    }
}

static class MyApp {
    public static void main(String[] args) {
        PetStore store = new PetStore();
        System.out.println(store.getDog().bark());
    }
}
```

上面这个宠物商店只与 mastiff 工作正常。如果要想把它改成支持其他不同类型的狗，需要修改并重新编译 PetStore 类。为了使 PetStore 更可重用，可能把 PetStore 改写成这样：

```
460 ➤ class PetStore {
    private IDog dog;
    public PetStore(IDog dog) {
        this.dog = dog;
    }

    public IDog getDog() {
        return dog;
    }
}

class MyApp {
    public static void main(String[] args) {
        PetStore store = new PetStore(new Chihuahua());
        System.out.println(store.getDog().bark());
    }
}
```

现在商店的狗的类型已经参数化了。某种狗通过构造函数被“注入”到我们的 PetStore 里，在每次想使用不同的狗时，PetStore 不需要重新编译。甚至可以在 PetStore 编写时还不存在的类也可以与 PetStore 一起使用，只要实现 IDog 接口就可以了。例如，这让我们轻松地创建模仿的狗类型供测试用。

依赖注入通常由一个“容器”完成，使用运行时配置自动初始化关键对象，接口的实现就在 classpath 或配置里指定的位置自动发现。依赖于所涉及的容器实现，配置通常采取分别管理的配置代码集或 XML 文件。

Clojure 把这个问题反转过来。在 Java 里，bark 是一个由 IDog 类定义的方法，Clojure 代码则把 bark 作为一个协议方法定义，脱离任何具体的类型。

```
(defprotocol Bark
  (bark [this]))

(defrecord Chihuahua [weight price]
  Bark
  (bark [this] "Yip!"))

(defrecord Mastiff []
  Bark
  (bark [this] "Woof!"))
```

现在，我们的宠物商店将像下面的样子：

```
(defrecord PetStore [dog])  
  
(defn main  
  [dog]  
  (let [store (PetStore. dog)]  
    (println (bark (:dog store))))  
(main (Chihuahua.))  
;= Yip!  
  
(main (Mastiff.))  
;= Woof!
```

461

是的，这样就行了！我们的 PetStore 现在只是一行很短的代码。

在 Java 里，PetStore 局限于实现了 IDog 接口的对象。在 Clojure 里，PetStore 没有这样的限制：它接受任何类型的参数，只要是这种类型实现了 Bark 协议，一切都会工作得很好。这会让我们做一些 Java 不可能允许的事，如扩展 Bark 到我们并不控制的类。用下面这样简单的代码：

```
{extend-protocol Bark  
  java.util.Map  
  (bark [this]  
    (or (:bark this)  
        (get this "bark"))))
```

我们现在可以把任何 Map 当狗来用，包括 Clojure 的 IPersistentMap 或 Java 的 HashMap，其中的 bark 字符串可以映射到 :bark 或 "bark"：^{注2}

```
(main (doto (java.util.HashMap.)  
             (.put "bark" "Ouah!")))  
;= Ouah!  
  
(main {:bark "Wan-wan!"})  
;= Wan wan!
```

如果使用的是记录类型，所有字段（除非声明为原始类型）都是动态的，所以没有类型来驱动自动神奇配置作为依赖注入的容器（是好是坏看你的视角）。有利的是，没有容器，^{注3} 可用于配置的选项扩展了。例如这里是一个合适的“配置文件”（其实只是包含 Clojure 可读的数据的文件，我们只是读取，并不求值）：

例12-1: petstore-config.clj
{:dog #user.Chihuahua{:weight 12, :price "\$84.50"}}

注2： 其他狗叫拟声词由 https://en.wikipedia.org/wiki/Bark_%28utterance%29#Representation 提供。

注3： 注意，对于 Clojure 的类型和记录，你绝对可以选择使用“遗留”的依赖注入容器（像 Spring 和 Guice 等）。

记得在 274 页“可读入的表示法”一节里讲到，`#user.Chihuahua{...}`写法是具名的记录类型的可读表示。我们相信这比其他的方法好很多，如使用 XML 配置 Spring beans，`@Bean` 写法，或者其他依赖注入容器提供的类似办法。

462> 一个扩展的 PetStore 工厂函数可以为特定的环境读取配置：

例12-2：configured-petstore

```
(defn configured-petstore
  []
  (-> "petstore-config.clj"
       slurp
       read-string
       map->PetStore))
```

现在我们将总会得到合适“注入”的 PetStore：

```
(configured-petstore)
:= #user.PetStore{:dog #user.Chihuahua{:weight 12, :price "$84.50"}}}
```

策略模式

另一个常见的设计模式是策略模式。这种模式允许动态选择一种方法或算法。假设想在运行时选择一种排序算法：

```
interface ISorter {
    public sort (int[] numbers);
}

class QuickSort implements ISorter {
    public sort (int[] numbers) { ... }
}

class MergeSort implements ISorter {
    public sort (int[] numbers) { ... }
}

class Sorter {
    private ISorter sorter;
    public Sorter (ISorter sorter) {
        this.sorter = sorter;
    }

    public execute (int[] numbers) {
        sorter.sort(numbers);
    }
}
```

```

class App {
    public ISorter chooseSorter () {
        if (...) {
            return new QuickSort();
        } else {
            return new MergeSort();
        }
    }
    public static void main(String[] args) {
        int[] numbers = {5,1,4,2,3};

        Sorter s = new Sorter(chooseSorter());

        s.execute(numbers);

        //... now use sorted numbers
    }
}

```

◀ 463

在这个案例中，Clojure 相比 Java 优势非常明显。在 Java 里，需要把方法放在类里，但函数在 Clojure 里是头等对象。如果直译，我们的 Clojure 代码可能看起来像这样：

```

(defn quicksort [numbers] ...)
(defn mergesort [numbers] ...)
(defn choose-sorter
  []
  (if ...
    quicksort
    mergesort))

(defn main
  []
  (let [numbers [...]]
    ((choose-sorter) numbers)))

```

看不到什么类。实现算法语义的每种函数都可以直接调用，没有类定义来妨碍我们指定行为的方式。

你甚至不需要给自己的排序算法命名——匿名函数工作得同样好。例如组合 Clojure 内置的 sort 和 reverse 来反转排序的顺序就是一个匿名函数：

```
((comp reverse sort) [2 1 3])
:= (3 2 1)
```

责任链

Clojure 的设施使得许多模式不需要或无足轻重不为人注意，少数几个仍然有效并继续影响我们设计和实现 Clojure 程序。其中之一是常见的控制流形式，称为责任链。在这一模式里，事件被发出，再由某些已设置的处理函数处理。处理函数可以处理这个事件，或者把它传给下一个处理函数。这些处理函数形成一个链，事件从这个链往下传，直到其中一个处理函数决定这个事件不应再往下传。

命令链是有用的，因为它允许把一个过程定义成多个部分，可以组合。没有哪一步需要知道过程中的其他步骤的事，除了如何往链下传递控制。

这一概念在许多地方出现。UNIX 的管道也是一例，文本数据从一个进程传递到下一个进程。Java Servlet 的过滤器是另一例子，Web 请求从一系列的过滤器通过，直到生成一个回应。

在 Java 里，可以通过定义一系列的“处理器”对象来构造责任链，然后每个用指向下一个处理器的指针来初始化。

```
abstract class Processor {  
    protected Processor next;  
    public addToChain(Processor p) {  
        next = p;  
    }  
    public runChain(data) {  
        Boolean continue = this.process(data);  
        if(continue and next != null) {  
            next.runChain(data);  
        }  
    }  
    abstract public boolean process(String data);  
}  
  
class FooProcessor extends Processor {  
    public boolean process(String data) {  
        System.out.println("FOO says pass...");  
        return true;  
    }  
}  
  
class BarProcessor extends Processor {  
    public boolean process(String data) {  
        System.out.println("BAR " + data + " and let's stop here");  
        return false;  
    }  
}
```

```
class BazProcessor extends Processor {  
    public boolean process(String data) {  
        System.out.println("BAZ?");  
        return true;  
    }  
}  
  
Processor chain = new FooProcessor().addToChain(  
    new BarProcessor).addToChain(new BazProcessor);  
chain.run("data123");
```

在这个例子中，我们用一个简单的协议即返回布尔值 `true` 来表示数据没有被处理，用 `false` 表示继续通过链来传递数据。

在 Clojure 里，执行的基本单元是函数而不是类或者方法。这意味着实现像责任链这样 ◀ 465 的东西只需用简单的函数组合就可以。

```
(defn foo [data]  
  (println "FOO passes")  
  true)  
  
(defn bar [data]  
  (println "BAR" data "and let's stop here")  
  false)  
  
(defn baz [data]  
  (println "BAZ?")  
  true)  
  
(defn wrap [f1 f2]  
  (fn [data]  
    (when (f1 data)  
      (f2 data))))  
  
(def chain (reduce wrap [foo bar baz]))
```

这里定义了一个函数 `wrap`，它接受两个函数，用与 Java 例子相同的方式把它们组合起来：`f1` 先运行，如果返回 `true`，下一步运行 `f2`。用这种方式可以建立函数的一个管道线，它会运行直到有一个函数结束了链或链用光了。我们可以用一个简单的 `reduce` 来组合一系列的函数。

`Ring`。Clojure 里命令链更现实的一个例子是函数库 `Ring`，我们会在第 16 章详细讨论。`Ring` 是处理向 Web 服务器发出的请求的一个函数库。请求的处理函数是以 Clojure 映射的形式接受请求对象的一个函数，返回一个应答对象，也是一个 Clojure 映射。

修改和处理这些应答和请求的函数称为“中间件”。中间件可用上例中差不多同样的方

式组合。假设我们以一个简单的处理函数开始：

```
(defn my-app
  [request]
  {:status 200
   :headers {"Content-type" "text/html"}
   :body (format "<html><body>You requested: %s</body></html>"
                 (:uri request))})
```

然后可以用执行各种其他函数的中间件“包装”这个处理函数。一种中间件可能解析请求里的 Web cookie 数据、把一个 :cookie 键加到请求或应答映射里。另一种中间件可能为 :session 数据做同样的事。这些类型的中间件已经由 Ring 提供给你使用，但它没有提供日志中间件把请求记录到一个日志文件里，定义这个中间件可能像这样：

```
466 (defn wrap-logger
      [handler]
      (fn [request]
        (println (:uri request))
        (handler request)))
```

取决于一个中间件的目标，它可以在把 `request` 处理传给处理链的下一个 `handler` 之前或之后执行动作——或者如果合适，根本就不把 `request` 往下传。这个简单的日志中间件只是简单地输出每个 `request :uri` 到 `stdout`，然后再调用下一个处理函数。

然后我们就可以与自定义的日志中间件一起使用 Ring 提供的 `cookies` 和会话中间件：

```
(require '[ring.middleware cookies session])

(def my-app (-> my-app
                  wrap-cookies
                  wrap-session
                  wrap-logger))
```

这种组合方式和 `comp` 非常相似，我们已经在 68 页“函数（功能）的组合”一节探索过了，只要它产生一个函数处理中间件的返回结果，处理顺序与包含在 `->` 形式的参数正好相反。因为 `wrap-logger` 是调用的最后一个中间件函数，它所返回的形成了组合函数的最外层，而 `my-app` 是最内层的。

面向方面的编程

面向方面的编程（AOP）是允许把横切的关注分离开的一种方法。在面向对象的代码里，一个行为或过程经常在多个类里重复，或分布在多个方法里。面向方面的编程是抽象这种行为并把它应用于类和方法而不需使用继承的一种途径。

常见的例子是计量。经常想要为运行的代码产生和记录计时数据或其他调试信息。在面向对象的代码里没有方便的方法可以把这种行为组合到已有的功能里，因而计时代码最后被加到感兴趣的特定方法里，然后（希望如此）在不再需要时注释掉：

```
public class Foo
    public void expensiveComputation () {
        long start = System.currentTimeMillis();
        try {
            // do computation
        } catch (Exception e) {
            // log error
        } finally {
            long stop = System.currentTimeMillis();
            System.out.println("Run time: " + (stop - start) + "ms");
        }
    }
}
```

467

为某一方法记录运行时要求改变这个方法，为了测量多个方法我们不得不重复做编辑。我们真正想要的是一个可以通用的办法，把某些行为与一个方法“包装”起来。达到这一目的的函数或方法有时称为建议。

如果能够把定义建议和将会应用建议的方法分开，将会获得很大的灵活性：

- 不需修改方法本身就可以对具体方法有选择地应用^{注4}建议；调试或其他开发时的建议（如性能测试或追踪）可以通过改变单个配置来完全禁用，或者选择完全不用面向方面编程的转换。
- 可以在一个地方而不是在多个方法里改变建议的行为，因而，举例来说，用于提供计时或其他计量的增强代码只需要做一次。

面向方面编程在 Java 里的一个实现是 AspectJ (<http://www.eclipse.org/aspectj/>)，这是对 Java 的一个扩展，为面向方面的编程提供了访问更多类似 Java 的构造。AspectJ 提供了自己的编译器，把 AspectJ 代码编译成 Java 类，根据什么方法和类应该受影响（称为切点）的说明织入你的应用程序的类里。我们来看一个 AspectJ 的例子，以一个类开始，包含了我们可能想要获得计量的方法：

```
public class AspectJExample {
    public void longRunningMethod () {
        System.out.println("Starting long-running method");
        try {
            Thread.sleep((long)(1000 + Math.random() * 2000));
        } catch (InterruptedException e) {
```

注4： 建议或其他方面的转换经常称为“织入”。

```
        }
    }
}
```

然后定义一个 Timing 方面，这将会应用于这个类的所有方法：

```
public aspect Timing {
    pointcut profiledMethods(): call(* AspectJExample.*(..));
    long time;

    before(): profiledMethods() {
        time = System.currentTimeMillis();
    }

    after(): profiledMethods() {
        System.out.println("Call to " + thisJoinPoint.getSignature() +
            " took " + (System.currentTimeMillis() - time) + "ms");
    }
}
```

468

- ① AspectJ 有自己的语法匹配包、类和方法签名，这个切点匹配 AspectJExample 类里所有的方法，我们将用它来指引 before 和 after 建议的应用。
- ② 在受影响的方法前运行的方面代码获得当前的时间，因而可以与这个方法返回后的时间进行比较。
- ③ 在受影响的方法之后运行的代码，这里只是简单地把方法的运行时间输出到标准输出。

如果我们运行一个调用 AspectJExample.longRunningMethod 的程序，就会看到与这个类似的输出：

```
Starting long-running method
Call to void com.clojurebook.AspectJExample.longRunningMethod() took 1599ms
```

为了改变“部析”的设置或完全省略“剖析”，不是在我们的代码里到处修改，而是可以在一个地方完成改变 AspectJ 建议的行为或者通过修改切点全部中止，或者一开始就不把方面织入我们的方法或应用。

Robert Hooke。在 Clojure 里，由于有 var 和头等函数，可以轻易得到面向方面编程功能的一个超集。函数可以轻易作为参数传递给其他函数，而且 Clojure 允许 var 在运行时被重新定义，这两个特性组合起来允许轻易用其他函数“包装”函数以按照自己喜欢的方式改变行为或结果。

Robert Hooke 函数库 (<https://github.com/technomancy/robert-hooke>) 提供了一个简单而

强大的方法为函数定义建议（称为钩子）。回到上面的例子，设置一个计时函数的钩子很容易：

```
(defn time-it [f & args]
  (let [start (System/currentTimeMillis)]
    (try
      (apply f args)
      (finally
        (println "Run time: " (- (System/currentTimeMillis) start) "ms")))))
```

time-it是一个通常的Clojure函数。它接受两个参数、另一个函数f和一个参数集args，那是调用钩子的函数计划传给f的参数。我们的建议可以在调用f（如果它决定调用f的话）之前或之后做任何事，这里只是记录运行时间。

不需要间接地允许time-it调用f，它只是简单地通过apply来调用，在args里把参数传递给它。注意，由于参数数量可变，time-it可以用于任何函数，不论函数需要多少参数。

这个建议可以像这样用于一个函数：

◀ 469

```
(require 'robert.hooke)

(defn foo [x y]
  (Thread/sleep (rand-int 1000))
  (+ x y))

(robert.hooke/add-hook #'foo time-it) ❶
```

- ❶ robert.hooke调用把time-it钩子加到var #'foo上，这会影响到这个函数所有的调用。

现在当调用foo时，可以看到time-it代码在运行：

```
(foo 1 2)
; Run time: 772 ms
;= 3
```

Robert Hooke也提供了暂时或长期禁用或删除钩子的能力：

```
(robert.hooke/with-hooks-disabled foo (foo 1 2)) ❷
;= 3

(robert.hooke/remove-hook #'foo time-it) ❸
;= #<user$foo user$foo@4f13f501>
(foo 1 2)
;= 3
```

- ❶ with-hooks-disabled允许暂时停止调用某个var上的钩子。

- `remove-hook` 会从一个 var 上删除某个钩子。

注意，这都是在 REPL 里完成的。增加、删除和暂时停止一个 var 上的钩子调用完全可以在运行时决定，就看什么标准最适合你的应用程序。

最后，因为增加和删除钩子都是对普通的 var 使用普通函数（即 `add-hook` 和 `remove-hook`）完成的，所以不需要我们前面看到的任何像 AspectJ 切点语法那样独特的东西。在 Clojure 环境里有优秀的工具可用于内省，因而可以做些像把一个钩子加到一个命名空间里的所有 var 这样的事情。

```
(require 'clojure.set)
:= nil
(doseq [var (-> (ns-publics 'clojure.set)
                  (map val))]
  (robert.hooke/add-hook var time-it))
:= nil
(clojure.set/intersection (set (range 100000))
                           (set (range -100000 10)))
; Run time: 97 ms
:= #{0 1 2 3 4 5 6 7 8 9}
```

- 470> 完成所有这些都不需要任何特别的语言或编译器支持，而 AspectJ 是需要的。实际上，Robert Hooke 函数库只有 100 行纯 Clojure 代码，如果它不存在，你自己也可以写一个。

最后的思考

正如我们所看到的，许多常见的设计模式在 Clojure 里是微不足道的，多数情况下完全消失在语言和它的函数库里了。设计模式的目标通常在于管理和减弱以 Java 和 Ruby 为代表的面向对象的方法的复杂度，由于关注函数和数据的分离，加上头等函数、合并与组合函数的工具，在 Clojure 里典型的情况是不需借助设计模式来完成设计和架构。

测试

Clojure 里的测试在概念上和 Java、Python 或 Ruby 里的大致相同。不管使用哪种语言，目的总是：

1. 构造合适的环境
2. 运行一些代码
3. 证实代码按预期表现或返回

当然，这一任务的每一部分的细节在语言间和测试框架间差别可以很大。在本章里，我们将概述在 Clojure 中做测试的几种方法，重点关注 `clojure.test`，它是包含在 Clojure 标准函数库里的测试框架。

不可变值与纯函数

在面向对象的语言如像 Java、Python 或 Ruby 里，测试可能是一件复杂的事情。对象之间倾向于有大量、微妙的交互。变更一个对象可能会改变任何数目的其他对象，或者一个对象的行为可能隐式依赖于其他对象的状态。这些交互经常是组合性的，使得很难可靠地描述可以改变我们的程序行为的环境特征，因而更难测试。

如在第 2 章中详细描述的，Clojure 鼓励使用不可变值和纯函数。以这样的感悟写出的代码从测试的角度来看简单多了：如果函数的结果只是由它的参数决定，那么简单的单元测试就足以确保测试的覆盖度。当然，对于应用程序包含了不是纯函数的部分，集成测试和功能测试还是需要的。

在面向对象的语言里，常见的测试辅助手段是使用模拟对象，模仿某些代码依赖的真实对象或服务的行为。构造了合适的模拟对象后，要测试的代码可以更有效地分离出来，防止依赖中可能的错误或可变的行为不当地影响测试结果。

模拟对象可能是伴生复杂度的一个教训，特别是在静态类型语言里。仅仅为了能够创建模拟对象，可能需要创建接口，而且经常需要为模拟设计数据类型，以便提供统一的 API，处理从实际服务或模拟服务获得的数据。

在 Clojure 里做测试时，模拟对象经常没有必要。有了不可变集合类和记录类型，根本就没有任何理由去模拟数据本身，你甚至完全可以从生产环境中捕获数据、持续化，然后重用做测试数据，不需要任何包装、转换或类型“体操”^{注1}。“模拟数据”在 Clojure 里只是更多的数据而已。

在其他语言中模拟的一个相关推论是需要模拟 Clojure 函数。考虑给定一个用户名查找地址的函数：

```
(defn get-address
  [username]
  ;; access database
)
```

在源数据库不可用或者没有配置好的任何环境里，这个函数都会失败，如果数据库是可用的，但包含与测试预期不同的数据，涉及这个函数的测试也会失败。

`with-redefs` 是解决这个问题的一种办法。它用某些其他值暂时替代一组具名 var 的根值，执行主体代码，然后重置这些 var 回原始根值，实际效果就是模拟了这些 var：

```
(with-redefs [address-lookup (constantly "123 Main St.")]
  (println (address-lookup)))
; 123 Main St.
```

可以同样方式使用 `binding`——在 201 页“动态作用域”一节里描述的——但是 `with-redefs` 在许多情况下更合适，尤其是测试时：

- `with-redefs` 对可以受影响的 var 没有限制，这与 `binding` 不同，那只能用于标记为 `^:dynamic` 的 var。

注 1：一些测试情景对生成的测试数据有特别的需要——这一概念由于 Haskell 的 QuickCheck 已经得到一定关注，不过那样的数据仍然不可变，可用于 Clojure 里所有通用的数据操作功能。有许多 Clojure 函数库提供各种测试数据生成功能，包括 `test.generative` (<https://github.com/clojure/test.generative/>)、`ClojureCheck` (<https://bitbucket.org/kotarak/clojurecheck>) 和 `re-rand` (<https://github.com/weavejester/re-rand>)。

- `with-redefs` 没有动态绑定所含的线程局限的语义，即所有线程、代理、`future` 等能看到受影响的 `var` 的临时根值。
- 受 `binding` 影响的动态 `var` 可以用 `set!` 来设置绑定的值。这不会阻止 `alter-var-root` 用于被 `with-redefs` 暂时修改的 `var`，不过在不需要时，`set!` 不可用是一个合理的预防措施。

简而言之，动态 `var` 和 `binding` 所提供的经常比你在测试时需要的更强大，能使你有这样的环境里做出一些“聪明过头”的事。

无论如何，`with-redefs` 和 `binding` 都可与 `clojure.test` 提供的 `fixture` 功能一起使用，以确保关键 `var` 在测试的范围内按需要模拟，或者让函数返回常量值，或者重定位操作，例如基于本地测试配置访问特别的数据库。在 479 页“Fixtures”一节我们将进一步讨论 `fixture`。

`clojure.test`

`clojure.test` 是“官方的”Clojure 测试框架。这是一个简单的函数库，但对许多任务已经够用了。



还有其他一些受欢迎的 Clojure 测试框架，提供更复杂的语义和功能。其中最受欢迎的是 Midje，可从 <https://github.com/marick/Midje> 获得。

`clojure.test` 里的断言使用 `is` 宏。`is` 对单个表达式求值，测试结果是否逻辑上为真，并返回这个值。如果有的话，`is` 会报告任何失败，包括我们获得的实际值以及断言提供的（可选）信息：

```
(use 'clojure.test)

(is (= 5 (+ 4 2)) "I never was very good at math...")
; FAIL in clojure.lang.PersistentList$EmptyList@1 (NO_SOURCE_FILE:1)
; I was never very good at math...
; expected: (= 5 (+ 4 2))
;   actual: (not (= 5 6))
;= false

(is (re-find #"foo" "foobar"))
;= "foo"
```

`is` 定义了多个特别的断言，可以在表达式里使用。^{注2} 例如，`thrown?` 测试某一类型的错

注2：通过为 `clojure.test/assert Expr` 多重方法定义新的方法，也可以把自己特殊的断言加到 `is` 里。有关详情见 `clojure.test` 命名空间的文档：<http://clojure.github.com/clojure/clojure.test-api.html>。

误是否会在表达式的求值过程中抛出来：

```
(is (thrown? ArithmeticException (/ 1 0)))          ●  
;=> #<ArithmetricException java.lang.ArithmetricException: Divide by zero>  
(is (thrown? ArithmeticException (/ 1 1)))  
; FAIL in clojure.lang.PersistentList$EmptyList@1 (NO_SOURCE_FILE:1)  
; expected: (thrown? ArithmeticException (/ 1 1))  
;   actual: nil  
;= nil
```

① 当 thrown? 断言通过时，is 返回这个异常。

thrown-with-msg? 是相似的，不过同时还测试正则表达式是否能够在错误的信息里找到：

```
(is (thrown-with-msg? ArithmeticException #'zero" (/ 1 0)))  
;=> #<ArithmetricException java.lang.ArithmetricException: Divide by zero>  
(is (thrown-with-msg? ArithmeticException #'zero" (inc Long/MAX_VALUE))) ①  
; FAIL in clojure.lang.PersistentList$EmptyList@1 _SOURCE_FILE:1)  
; expected: (thrown-with-msg? ArithmeticException #'zero" (inc Long/MAX-VALUE))  
;   actual: #<ArithmetricException java.lang.ArithmetricException: integer overflow>  
;= #<ArithmetricException java.lang.ArithmetricException: integer overflow>
```

① 当非自动提升的操作符产生溢出状况时，Clojure 抛出一个 ArithmeticException，这个异常的信息里没有包含 #'zero"，因而断言失败。

你可以用 testing 宏给测试添加文档、扩展失败报告，让没通过的测试的范围描述包含在错误报告里：

```
(testing "Strings"  
  (testing "regex"  
    (is (re-find #"foo" "foobar"))  
    (is (re-find #"foo" "bar")))  
  (testing ".contains"  
    (is (.contains "foobar" "foo"))))  
; FAIL in clojure.lang.PersistentList$EmptyList@1 (NO_SOURCE_FILE:1)  
; Strings regex  
; expected: (re-find #"foo" "bar")  
;   actual: (not (re-find #"foo" "bar"))
```

定义测试

有两种定义测试的方式。第一种也可能是最有用的，是用 deftest 宏把测试定义成单独的函数。这个宏只定义零参数的函数（像通过 defn 那样），var 附加一些元信息使它成为一个测试。deftest 定义的测试在其他方面与正常函数无异，可以在 REPL 里调用，等等：

```
(deftest test-foo
  (is (= 1 1)))
  ;;= #'user/test-foo
  (test-foo)
  ;;= nil
```

通常在你的函数库或应用程序之外的一组命名空间里定义测试，经常是在项目的 `test` 子目录里。正如在第 8 章里描述的，Clojure 最常用的构建工具会在这些目录里查找并运行测试。

所有的 `clojure.test` 测试实际上都定义了一些 `var`，每个 `var` 元数据的 `:test` 槽里都有一个函数。这一点我们可以在上面定义的测试里看到：

```
(:test (meta #'test-foo))
 ;;= #<user$fn_366 user$fn_366@4e842e74>
```

通过 `(test-foo)` 调用的函数只是委托给这个 `:test` 函数。这也许看起来像是个傻气的细节，但这使得把测试与被测试的函数绑到一起成为可能，是 `with-test` 正帮你做的这件事。^{注3} 它接受定义一个 `var` 的任何形式作为第一个参数，任何数量的其他形式作为其他参数组成测试函数的主体，保存在 `var` 的元数据里：

```
(with-test
  (defn hello [name]
    (str "Hello, " name))
  (is (= (hello "Brian") "Hello, Brian"))
  (is (= (hello nil) "Hello, nil")))
  ;;= #'user/hello
```

这样定义后，就可以在应用程序里任意调用和使用 `hello`，不用运行任何测试：

```
(hello "Judy")
 ;;= "Hello, Judy"
```

在 `var` 定义形式之后提供给 `with-test` 的主体被打包在一个函数里，储存在 `var` 元数据的 `:test` 槽，这就是测试函数：

```
((:test (meta #'hello)))
 ; FAIL in clojure.lang.PersistentList$EmptyList@1 (NO_SOURCE_FILE:5)
 ; expected: (= (hello nil) "Hello, nil")
 ;   actual: (not (= "Hello, " "Hello, nil"))
 ;;= false
```

有帮助的是，`clojure.test/run-tests` 函数使用这个元数据动态找到一个或多个命名空间里的测试并运行所有的测试。因而它能找到载入的源文件里的测试和在 REPL 里定义

◀ 476

注3：可以把这种方法看做比 Python 的 `doctest` 模块或 Ruby 的 `rubydoctest` 更为结构化的版本，共同点是测试与被测试的函数定义在一起。

的测试：

```
(run-tests)          ❶
; Testing user
;
; FAIL in (hello) (NO_SOURCE_FILE:5)
; expected: (= (hello nil) "Hello, nil")
;   actual: (not (= "Hello, " "Hello, nil"))
;
; Ran 2 tests containing 3 assertions.
; 1 failures, 0 errors.
;= {:type :summary, :pass 2, :test 2, :error 0, :fail 1}
```

❶ 如果没有指定命名空间，`run-tests` 在 `*ns*` 里搜索带有 `:test` 函数的 var。

与许多语言里“构建 - 编译 - 测试”循环不同的是，Clojure 使得在写函数时通过像 `run-tests` 这样的工具轻易地测试函数。

这一方法有一点需要注意的是，一旦定义了测试，只要 JVM 还活着，测试就会一直保持着。从磁盘载入一个文件或申请包含测试的命名空间都不会丢弃这些测试，所以不想要的测试会持续存在，`run-tests` 会继续找到并运行它们，直到重启 JVM——而那是我们尽可能避免的事。幸好你可以用 `ns-unmap` 取消定义一个 var（及其测试）：^{注4}

```
(ns-unmap *ns* 'hello)
;= nil
(run-tests)
; Testing user
;
; Ran 1 tests containing 1 assertions.
; 0 failures, 0 errors.
;= {:type :summary, :pass 1, :test 1, :error 0, :fail 0}
```

或者可以改变 `hello` 上的元数据，让它不再包含 `:test` 函数，对于用 `with-test` 定义的 var，可以继续使用那些函数：

```
(with-test
  (defn hello [name]
    (str "Hello, " name))
  (is (= (hello "Brian") "Hello, Brian"))
  (is (= (hello nil) "Hello, nil")))
;= #'user/hello
(alter-meta! #'hello dissoc :test)
;= {:ns #<Namespace user>, :name hello, :arglists ([name]),
;= :line 2, :file "NO_SOURCE_PATH"}
(run-tests *ns*)
```

注 4： 在 322 页“定义与使用命名空间”一节和 401 页“内省命名空间”一节里更多讨论了 `ns-unmap` 及其他命名空间内省和操纵函数。

```
; Testing user
;
; Ran 1 tests containing 1 assertions.
; 0 failures, 0 errors.
{:type :summary, :pass 1, :test 1, :error 0, :fail 0}
(hello "Rebecca")
:= "Hello, Rebecca"
```

测试“套件”

可以在测试里调用其他测试，这毫无问题：

```
(deftest a
  (is (= 0 (- 3 2))))
:= #'user/a
(deftest b (a))
:= #'user/b
(deftest c (b))
:= #'user/c
(c)
; FAIL in (c b a) (NO_SOURCE_FILE:2) ❶
; expected: (= 0 (- 3 2))
;   actual: (not (= 0 1))
```

❶ 失败报告很有帮助地包含了导致失败的测试函数“栈”，这里的栈是列表 (c b a)。^{注5}

定义这样的测试“套件”是方便的，但与 `run-tests` 的默认行为（因而与 Clojure 构建工具的测试运行功能）相冲突：因为它会调用所有 var 元数据里的 `:test` 函数，直接运行子测试，即使已经作为“套件”的一部分调用过了。好的话，这会导致测试运行时间更长，因为有些测试的调用是多余的；不好的话，你会得到大量重复的失败报告：

```
(run-tests)
; Testing user
;
; FAIL in (b a) (NO_SOURCE_FILE:2)
; expected: (= 0 (- 3 2))
;   actual: (not (= 0 1))
;
; FAIL in (c b a) (NO_SOURCE_FILE:2)
; expected: (= 0 (- 3 2))
;   actual: (not (= 0 1))
;
; FAIL in (a) (NO_SOURCE_FILE:2)
; expected: (= 0 (- 3 2))
;   actual: (not (= 0 1))
;
```

^{注5} 这里不幸的写法提示，调用链——c 调用 b 调用 a——不是单个调用，即对 (c b a) 求值会失败，因为通过 `deftest` 定义的函数不带参数，包括上面命名的 c。

478 ; Ran 6 tests containing 3 assertions.
; 3 failures, 0 errors.
;= {:type :summary, :pass 0, :test 6, :error 0, :fail 3}

有一两个避免这种情况的办法。第一个办法是，可以为每个命名空间定义一个基本的 run-tests 入口；这个入口必须是名为 test-ns-hook 的函数，而且带零个参数。当 test-ns-hook 存在时，它会成为这个命名空间里 run-tests 调用的唯一函数。这让你可以完全控制运行什么测试及如何组合这些测试：

```
(defn test-ns-hook [] (c))  
;= #'user/test-ns-hook  
(run-tests)  
; Testing user  
;  
; FAIL in (c b a) (NO_SOURCE_FILE:2)  
; expected: (= 0 (- 3 2))  
; actual: (not (= 0 1))  
;  
; Ran 3 tests containing 1 assertions.  
; 1 failures, 0 errors.  
;= {:type :summary, :pass 0, :test 3, :error 0, :fail 1}
```

另一种办法是，可以把子断言放在常规函数里。这些函数不会有 :test 元数据，因而不会被 run-tests 找到和运行：

```
(ns-unmap *ns* 'test-ns-hook) ●  
;= nil  
(defn a  
[]  
  (is (= 0 (- 3 2))))  
;= #'user/a  
(defn b [] (a))  
;= #'user/b  
(deftest c (b))  
;= #'user/c  
(run-tests)  
; Testing user  
;  
; FAIL in (c) (NO_SOURCE_FILE:3) ●  
; expected: (= 0 (- 3 2))  
; actual: (not (= 0 1))  
;  
; Ran 1 tests containing 1 assertions.  
; 1 failures, 0 errors.  
;= {:type :summary, :pass 0, :test 1, :error 0, :fail 1}
```

❶ 我们先把前面定义的 test-ns-hook 删掉。

- 注意，这里失败的测试“栈”只是 (c)，不是之前的 (c b a)。这是因为 `run-tests` 只跟踪测试范围，不跟踪像 a 和 b 这样的常规函数。

Fixtures

`fixtures` 提供一种方法来设置和撤除服务、数据库状态、模拟函数和测试数据，从而确保一个命名空间的所有测试在受控的环境里调用所有测试。这与 xUnit 单元测试函数库里 `setUp` 和 `tearDown` 方法（或 `@Before` 和 `@After` 注释）是相似的，不过对测试环境提供了基本无限的控制。

`fixture` 只是一个高阶函数，接受一个函数为单个参数，那是在由 `fixture` 建立并撤除的环境里要被调用的测试或一组测试。`fixtures` 可以在任何地方定义、跨测试命名空间重用，每个命名空间可以使用任何数量的 `fixtures`。

有两种方式可以对一个命名空间应用 `fixtures`：

- 可以为命名空间里找到的每个测试调用 `fixtures`。对一个包含 n 个测试的命名空间，你的 `fixture` 可以被调用 n 次，每次一个函数对应单个测试。
- 或者，可以为整个命名空间调用一次 `fixtures`，所建立的环境应用于命名空间里的所有测试函数。对于包含 n 个测试的命名空间，你的 `fixture` 只会被调用一次，一个函数会调用这个命名空间的所有测试函数。

在这两种情形下，`fixture` 的实现都采取这样的通用形式：

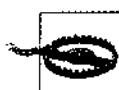
```
(defn some-fixture
  [f]
  (try
    ;; set up database connections, load test data,
    ;; mock out functions using `with-redefs` or `binding`, etc.
    (f)
    (finally
      ;; clean up database connections, files, etc.
    )))
```

无论你为某个 `fixture` 选择的是“每次”还是“一次”中的哪一选项，`fixture` 的实现高度依赖于你所要测试的和测试环境的细节。

“每次”或“一次”的选项覆盖大多数需求，不过确实意味着 `clojure.test` 里的 `fixtures` 只提供在定义自己的 `test-ns-hook` 时所能得到的部分灵活性。虽然 `fixtures` 定义有两种“生命周期”，但在一个 `test-ns-hook` 实现里，你可以完全控制运行什么、什么时候运行以及每个测试之前和之后执行什么。碰巧大多数时候 `fixtures` 更方便一些；尤其是使用它不会强迫你显式地管理在一个命名空间里如何搜索和调用测试函数，这是在使用

test-ns-hook 时你隐含需要负责的簿记工作。

480



fixtures 和 test-ns-hook 两者是互斥的，如果你定义了后者，前者就不会被使用。

为了尝试一下 fixtures，我们来看看前面的 configured-petstore，这是在例 12-2 里定义的一个函数，包括起支持作用的记录和协议重录如下：

```
(defprotocol Bark
  (bark [this]))  
  
(defrecord Chihuahua [weight price]
  Bark
  (bark [this] "Yip!"))  
  
(defrecord PetStore [dog])  
  
(defn configured-petstore
  []
  (-> "petstore-config.clj"
    slurp
    read-string
    map->PetStore))
```

给 configured-petstore 写一个测试很容易，只需要比较用它生成的 PetStore 与一个已知的值：

```
(def ^:private dummy-petstore (PetStore. (Chihuahua. 12 "$84.50")))  
  
(deftest test-configured-petstore
  (is (= (configured-petstore) dummy-petstore)))
```

configured-petstore 从磁盘上特定文件即当前目录里的 "petstore-config.clj" 载入一个记录。因而，如果那个文件不存在或没有适当地填入与我们预期的 dummy-petstore 值对应的数据，测试会失败：

```
(run-tests)
; Testing user
;
; ERROR in (test-configured-petstore) (FileInputStream.java:-2)
; expected: (= (configured-petstore) dummy-petstore)
;   actual: java.io.FileNotFoundException: petstore-config.clj
;           (No such file or directory)
;   at java.io.FileInputStream.open (FileInputStream.java:-2)
;   ...
```

```
;  
; Ran 1 tests containing 1 assertions.  
; 0 failures, 1 errors.  
:= {:type :summary, :pass 0, :test 1, :error 1, :fail 0}
```

我们需要确保保存在合适的文件以便调用 `configured-petstore` 的结果可以证实，这需要一个 fixture。我们已经有一个预期的 PetStore 实例（保存在 `dummy-petstore` 里）；让我们的 fixture 输出那个记录和字段的可读表示到 `configured-petstore` 指定的文件里是一件简单的事：

```
(defn petstore-config-fixture  
  [f]  
  (let [file (java.io.File. "petstore-config.clj")]  
    (try  
      (spit file (with-out-str (pr dummy-petstore))) ❶  
      (f) ❷  
      (finally  
        (.delete file)))) ❸
```

- ❶ 在这里把 `dummy-petstore` 的可读表示写到 "`petstore-config.clj`" 文件里。注意，我们这里使用的是 `pr` 而不是 `print` 或 `println`，后两个函数产生人可读的输出，而 `pr` 和 `prn` 产生 Clojure 可读的输出。
- ❷ `f` 在这里是一个测试函数，或者是一个会调用这个命名空间里所有测试的函数。到底是哪一种取决于 fixture 是注册为在命名空间运行 `:once` 还是为命名空间里的 `:each` 测试函数运行一次。
- ❸ 尽量清除 fixture 的效果是好形式。如果不这样做，那么 "`petstore-config.clj`" 可能在多次测试运行间持续，也许会错误地使得其他测试通过。

定义 fixture 之后，^{注6} 剩下的就是注册它了。我们将用 `:once` 生命周期：

```
(use-fixtures :once petstore-config-fixture)
```

注册 fixture 之后，可以确定 `configured-petstore` 在调用时数据已经准备到位，因而让测试通过：

```
(run-tests)  
; Testing user  
;  
; Ran 1 tests containing 1 assertions.  
; 0 failures, 0 errors.  
:= {:type :summary, :pass 1, :test 1, :error 0, :fail 0}
```

注6： 技术上不需要在顶层 var 里定义 fixtures，你可以轻易地把 fixtures 作为函数参数传给 use-fixtures。

也可以轻易地把 `petstore-config-fixture` 注册为 `:each fixture`，这样它会应用于命名空间里的每个测试函数一次。

HTML DSL 的成长

作为开发中测试的一个活生生的例子，我们来看看使用测试是如何帮助我们建立起一个生成 HTML 的函数库的。结果将会是一个简单版的 Hiccup (<https://github.com/weavejester/hiccup>)，这个函数库生成的 HTML 与被定义的结构或生成的 Clojure 集合类是镜像的。

我们的目标是能够像下面这样写 HTML 片段：

```
[::html
  [:head [:title "Propaganda"]]
  [:body [:p "Visit us at "
    [:a {:href "http://clojureprogramming.com"}]
    "our website"]
   "."]]]
```

然后把它编译成像这样的 HTML：

```
<html>
  <head><title>Propaganda</title></head>
  <body>
    <p>Visit us at <a href="http://closurebook.com">our website</a>.</p>
  </body>
</html>
```

这里的向量（其实是任何的顺序性集合类）代表 HTML 元素，第一个值是元素名，第二个值是它的属性的映射，这是可选的，其余值是元素的内容，可以是字符串或代表子元素的向量。

我们先写几个测试来开始。我们知道最终的 HTML 生成函数会是纯函数，它将接受某些输入，产生某些结果，没有外部的依赖或副作用。我们不喜欢一遍又一遍地写 `(is (= expected (f input)))`，因而大多数测试会使用 `are`：

```
(deftest test-addition
  (are [x y z] (= x (+ y z))
    10 7 3
    20 10 10
    100 89 11))
```

`are` 是 `clojure.test` 里的一个辅助宏，让断言使用模板。例如上面的 `are` 形式经过宏扩展成：

```
(do
  (clojure.test/is (= 10 (+ 7 3)))
  (clojure.test/is (= 20 (+ 10 10)))
  (clojure.test/is (= 100 (+ 89 11))))
```

are有助于减少每个断言的重复，不过我们仍需要重复这个转换，例如`(= expected (f input))`。一个简短的宏可以帮助我们连那也避免了：

```
(defmacro are* [f & body]
  `(are [x# y#] (= (f x#) y#)
        ~@body))
```

现在可以像这样写测试了：

◀ 483

```
(deftest test-tostring
  (are* str
    10 "10"
    :foo ":foo"
    "identity" "identity"))
```

假设有一个函数html，它接受一个顺序性集合类，返回一个包含HTML的字符串。进一步假设有一个辅助函数 attrs，它产生用于包含到HTML元素里的属性字符串。我们将渐进地实现这些函数，以符合在测试中不断修改的预期。

这些测试将是一个合理的起点：

```
(require 'clojure.string)

(declare html attrs)

(deftest test-html
  (are* html
    [:html]
    "<html></html>

    [:a [:b]]
    "<a><b></b></a>

    [:a {:href "/"} "Home"]
    "<a href=\"/\">Home</a>

    [:div "foo" [:span "bar"] "baz"]
    "<div>foo<span>bar</span>baz</div>"))

(deftest test-attrs
  (are* (comp clojure.string/trim attrs)      ●
    nil ""
    {:foo "bar"})
```

```
"foo=\"bar\""  
(sorted-map :a "b" :c "d")  
"a=\"b\" c=\"d\"")
```

- ❶ 我们这里耍了一点小聪明：trim将让我们忽略 attrs 发出的开头和结尾的空白，我们知道会在某些边缘情况有用。

现在来看我们的代码。不错的第一次尝试：

```
(defn attrs  
  [attr-map]  
  (->> attr-map  
    (mapcat (fn [[k v]] [k " =\" " v "\"]))  
    (apply str)))  
  
(defn html  
  [x]  
  [484] (if-not (sequential? x)  
    (str x)  
    (let [[tag & body] x  
          [attr-map body] (if (map? (first body))  
            [(first body) (rest body)]  
            [nil body]))  
      (str "<" (name tag) (attrs attr-map) ">"  
            (apply str (map html body))  
            "</" (name tag) ">"))))
```

用我们的测试来看看它有什么结果。

```
(run-tests)  
; Testing user  
;  
; FAIL in (test-html) (NO_SOURCE_FILE:6)  
; expected: (= (html [:a {:href "/"} "Home"])) "<a href=\"/\">Home</a>"  
;   actual: (not (= <a:href =\"/\">>Home</a> "a href=\"/\">Home</a>"))  
;  
; FAIL in (test-attrs) (NO_SOURCE_FILE:20)  
; expected: (= ((comp clojure.string/trim attrs) {:foo "bar"}) "foo=\"bar\"")  
;   actual: (not (= :foo =\"bar\" "foo=\"bar\""))  
;  
; FAIL in (test-attrs) (NO_SOURCE_FILE:20)  
; expected: (= ((comp clojure.string/trim attrs)  
;               (sorted-map :a "b" :c "d"))  
;               "a=\"b\" c=\"d\"")  
;   actual: (not (= :a =\"b\":c =\"d\" "a=\"b\" c=\"d\""))  
;  
; Ran 2 tests containing 7 assertions.  
; 3 failures, 0 errors.  
;= {:type :summary, :pass 4, :test 2, :error 0, :fail 3}
```

哟，糟了，怎么了呢？看起来 `attrs` 可能是麻烦的根源。结果里有看起来像 Clojure 关键字的，例如 `(attrs {:foo "bar"})` 输出成 "`:foo =\":bar\"`" 了。应该对关键字调用 `name` 函数以得到属性名的字符串。我们来修订这一点，顺便处理掉属性名和 `=` 之间多余的空格：

```
(defn attrs
  [attrs]
  (->> attrs
    (mapcat (fn [[k v]] [(name k) "=\" " v "\""]))
    (apply str)))
```

现在可以只重新运行 `test-attrs`，以看看它工作是否正常，因为我们正集中精力修订 `attrs` 函数：

```
(test-attrs)
; FAIL in (test-attrs) (NO_SOURCE_FILE:20)
; expected: (= ((comp clojure.string/trim attrs)
;                 (sorted-map :a "b" :c "d"))
;                  "a=\"b\"" "c=\"d\"")
; actual: (not (= ":a =\"b\";c =\"d\"" "a=\"b\"" "c=\"d\"")))
```

好一些了，不过看来我们忘了每个属性前的空白。简单修改如下：

◀ 485

```
(defn attrs
  [attrs]
  (->> attrs
    (mapcat (fn [[k v]] [\space (name k) "=\" " v "\""]))
    (apply str)))
```

现在怎么样呢？

```
(test-attrs)
;= nil
(run-tests)
; Testing user
;
; Ran 2 tests containing 7 assertions.
; 0 failures, 0 errors.
;= {:type :summary, :pass 7, :test 2, :error 0, :fail 0}
```

很好，看来代码干净了。看看对我们最初的例子会输出什么结果：^{注7}

```
(html [:html
        [:head [:title "Propaganda"]]
        [:body [:p "Visit us at "
                [:a {:href "http://closurebook.com"}
                 "our website"]]
```

注7： `html` 没有提供漂亮的输出，我们对结果的字符串字符串化了一下，让它适合在页面上显示。

```

    ".")]
;
:= "<html>
;:=  <head><title>Propaganda</title></head>
;:=  <body>
;:=      <p>Visit us at <a href=\"http://closurebook.com\">our website</a>.</p>
;:=  </body>
;:= </html>"
```

像这样的模板 DSL 的美妙之处在于，由于 HTML 是由常规的 Clojure 数据结构一一对应表示的，我们可用所有能用于这些数据结构的伟大工具——因而也能用于 HTML：

```

(html (list* :ul (for [author ["Chas Emerick" "Christophe Grand" "Brian Carper"]]
                      [:li author]))
;= "<ul><li>Chas Emerick</li><li>Christophe Grand</li><li>Brian Carper</li></ul>"
```

与使用典型的被当做 HTML DOM 的 API 的东西相比，或者与把 HTML 字符串放在代码里再把数据插入其中所犯下的“暴行”相比，这要有吸引力多了。在用 Clojure 输出 HTML 的选项中，像这样的解决办法（Hiccup 是其中的典型实现）代表一个极端，而像 Enlive 的其他一些选项（我们在第 16 章的例子中将用到）代表更传统的模型，在后者中，HTML 内容一般始于模板资产，然后编程填充数据。

486 依赖断言

HTML 生成函数是不错的，不过不可避免有人会在某些地方试图滥用这个 API。例如 XML 的 Clojure 标准表示（如由 *closure.xml* 所输出的）看起来像这样的：

```
{:tag :a, :attrs {:href "http://closure.org"}, :content ["Closure"]}
```

那与基于向量的 HTML 表示有相当的不同。不奇怪的是，`html` 会对这个映射做一些奇怪的事，因为这不是它所预期的：

```
(html {:tag :a, :attrs {:href "http://closure.org"}, :content ["Closure"]})
;= "{:content [\"Closure\"], :attrs {:href \"http://closure.org\"}, :tag :a}"
```

天，它返回一个字符串！那是合适的，如果我们惰性地递归使用 `html` 来处理元素向量的内容，不过这个具体结果显然不是太有用，如果我们假设一个用户（也许是合理的）以为他能够把一个 *closure.xml* 数据结构用 `html` 函数序列化到一个 HTML 字符串里。这一情况是我们想要尽快报错退出。

想到了断言。断言测试一个条件，如果条件不为真就抛出错误：

```
(defn attrs
  [attrs]
  (assert (or (map? attr-map)
              (nil? attr-map)) "attr-map must be nil, or a map")
```

```
(-> attrs
  (mapcat (fn [[k v]] [\space (name k) "=\" v "\"])))
  (apply str)))

(attrs "hi")
;= #<AssertionError java.lang.AssertionError:
;=   Assert failed: attr-map must be nil, or a map
;=   (or (map? attr-map) (nil? attr-map))>
```

- 我们用 `assert` 来验证参数的类型，不过它可以用在你代码的任何地方，以强制执行你可以表达的任何不变情形。

因为断言抛出错误，所以可能不是用于产品代码的最佳工具，因为你可能不想承担与验证断言相关的运行时代价。Clojure 让我们通过把 `*assert*` var 分别设置为 `true` 或 `false` 来启用或禁用断言。

因为 `assert` 是一个宏，在你“编译”用它的代码前应该设置 `*assert*`。在 `*assert*` 为 `false` 时，对 `assert` 的调用会从编译的函数中全部删除，不产生什么运行时代价。

```
(set! *assert* false)
;= false
(defn attrs
  [attr-map]
  (assert (or (map? attr-map)
              (nil? attr-map)) "attr-map must be nil, or a map") <487>
  (-> attr-map
    (mapcat (fn [[k v]] [\space (name k) "=\" v "\"])))
    (apply str)))
;= #'user/attrs
(attrs "hi")
;= #<UnsupportedOperationException java.lang.UnsupportedOperationException:
;=   nth not supported on this type: Character>
(set! *assert* true)      ●
;= true
```

- `*assert*` 的状态可以随时设置：对某个命名空间设置，基于某个系统属性或环境变量设置整个应用，或者像我们在这里所做的，在 REPL 里设置。

- 我们把 `*assert*` 重设回 `true` 以支持后面的例子。

上面的 `attrs` 编译时禁用了断言，因而现在得到的是不那么有帮助的 `nth not supported on this type` 错误，因为传递给 `mapcat` 的函数试图顺序地解构字符串中的字符。另一方面，在一个已经很好测试的应用里——也许是混合使用断言、单元测试和功能测试——在生产环境中禁用断言将会解除前者的运行时代价。

前条件和后条件

断言最常用的两个用例是测试函数的输入和输出。由于 Clojure 对使用纯函数的强调，许多函数的输入和输出可能是所有真正需要测试的。

`fn`（以及因此像 `defn` 那样派生出来的东西）直接支持写断言测试输入的前条件和测试输出的后条件。前条件在函数的主体前求值；后条件在函数主体执行后但在返回值发送到调用者前求值。如果任何条件求值为逻辑假，那么就会抛出一个错误。这一功能可用于确保函数参数和返回值在运行时符合定义的标准，不致让检查和不一致时抛出的异常把函数主体弄得杂乱不堪，并且有可能在不同代码间重用校验功能。

Clojure 函数主体的第一个值如果是一个带有 `:pre` 或 `:post` 键的映射，这个映射就被看做前后条件表达式的映射，在函数被编译时会被扩展成对 `assert` 的调用。^{注 8}

488 `:pre` 和 `:post` 的值应该是向量，向量的每个项目是单个的断言。函数的参数在前条件下可以被引用。函数的返回值在后条件下与 % 绑定，这与函数字面量里第一个参数的表示法是相似的。我们用合理的前后条件重新定义 `attrs` 和 `html`：

```
(defn attrs
  [attr-map]
  {:pre [(or (map? attr-map)           ❶
             (nil? attr-map))]}
  (->> attr-map
    (mapcat (fn [[k v]] [\space (name k) "=" v "\"])))
    (apply str)))
```



```
(defn html
  [x]
  {:pre [(if (sequential? x)           ❷
              (some #(=(> x first %) [keyword? symbol? string?])))
         (not (map? x)))]}           ❸
  :post [(string? %)]}           ❹
  (if-not (sequential? x)
    (str x)
    (let [[tag & body] x
          [attr-map body] (if (map? (first body))
                            [(first body) (rest body)]
                            [nil body])]
      (str "<" (name tag) (attrs attr-map) ">"
            (apply str (map html body))
            "</" (name tag) ">")))))
```

❶ 跟以前一样，我们将要求 `attrs` 只接受 nil 或映射为参数。

注 8： 如果一个映射是函数体里唯一的表达式，它就被视做函数的返回值，而不是视做前后条件的映射。

- ❸ `html` 的前条件复杂一点。如果它的参数是一个顺序性的集合类，它的第一个参数必须是一个字符串、关键字或符号。否则……
- ❹ 它可以是除映射外的任何东西。
- ❺ `html` 的后条件只是要求返回值是一个字符串。这可以更花哨些，做些事情像验证所生成的 HTML 可以干净地解析，符合某个 DTD，等等。

这些条件可以帮助标出一些常见错误。回头来看有人可能使用一个 `closure.xml` 元素映射用我们的函数来生成 HTML 的情况：

```
(html {:tag :a, :attrs {:href "http://closure.org"}, :content ["Closure"]})  
:= #<AssertionError java.lang.AssertionError:  
:=   Assert failed: (if (sequential? x)  
:=     (some (fn* [p1__843#] (-> x first p1__843#))  
:=       [keyword? symbol? string?])  
:=     (not (map? x)))>
```

注意，前后条件在它们依赖的主函数体内编译成 `assert` 语句。这意味着条件同样受到 489
`*assert*` 的状态的影响，在函数编译后，不可能增加、删除或改变函数的条件。^{注9}

注9：一个解决办法是 Trammel，这个函数库关注于在 Clojure 中为进行动态契约编程提供支持，大量使用条件和记录与类型不变约束以确保程序的正确性：<https://github.com/fogus/trammel>。

使用关系数据库

关系数据库是软件开发的一大支柱，这样已经有数十年了。机构不在某个地方使用 RDBMS 的很少，程序员至少偶尔需要写入数据和从中取得数据，不这样做的也很少。Java 有悠久历史通过 JDBC 对关系数据库进行高质量的支持；通过 JVM 的密切关系，Clojure 能够轻易、完全利用这一历史的优势。

有多种可选方案在 Clojure 里与关系数据库交互。`clojure.java.jdbc` 是一个简单而功能强大的函数库，它在 Clojure 和 JDBC 间起着薄薄的中间层的作用。`Korma` 是另一个 Clojure 函数库，提供更多 Clojure 原生的接口。而最后，如果 Clojure 的函数库不适合你的风格，或者你想要把 Clojure 混合进一个已有的基于 Java 的应用程序，你总能回头使用许多成熟、健壮的 Java 数据库函数库或框架。在本章里，我们将探索在 Clojure 里设置和使用 Hibernate。

`clojure.java.jdbc`

不管是使用原生的 Java 函数库或是 Clojure 的许多函数库之一，在 Clojure 里连接到一个数据库总会使用 JDBC，这是在 Java 里与关系数据库交互的最底层 API 抽象。`clojure.java.jdbc` (<https://github.com/clojure/java.jdbc>) 函数库包装了 JDBC，从而让它在 Clojure 里使用起来更容易：

```
{org.clojure/java.jdbc "0.1.1"}
```

关于依赖，需要把 JDBC 和 `clojure.java.jdbc` 与一个你所使用的数据库对应的 JDBC 驱动程序配成对。有数百个 JDBC 驱动程序针对几乎现存的每个数据库，因而找到一个你需要的应该不难。这里是一些常用数据库的 JDBC 驱动程序的 Leiningen 坐标：

```
[org.xerial/sqlite-jdbc "3.7.2"]          ; SQLite
[mysql/mysql-connector-java "2.0.14"]        ; MySQL
[postgresql "9.0-801.jdbc4"]                 ; PostgreSQL
```

492 我们所有的例子都将使用 SQLite (<http://sqlite.org>) 驱动程序。因为 SQLite 是一个“嵌入式”数据库——即数据库引擎在进程内运行，数据库文件储存在你所指定的位置，而不是连接到一个可能是远程的数据库服务器——我们所显示的所有代码不需要设置或运行单独的数据库就可以工作。因为有 JDBC 提供的抽象，只需要最小的修改，同样的代码就可以用于 MySQL、PostgreSQL、Oracle 及其他数据库。

把 `closure.java.jdbc` 和 `org.xerial/sqlite-jdbc` JDBC 驱动程序 (<http://www.xerial.org/trac/Xerial/wiki/SQLiteJDBC>) 函数库加到项目里后，就可以在 Clojure REPL 里深入挖掘如何使用数据库了。

所有的 `closure.java.jdbc` 操作都要求一个 “spec” 才能操作：

```
(require '[closure.java.jdbc :as jdbc])
;= nil
(def db-spec {:classname "org.sqlite.JDBC"
              :subprotocol "sqlite"
              :subname "test.db"})
;= #'user/db
```

这里，`db-spec` 是配置数据的一个映射，`closure.java.jdbc` 将用它来：

1. 定位我们的 JDBC 驱动程序（通过 `:classname` 值）。
2. 配置这个驱动程序以及驱动程序按我们的要求产生的连接。

每个 JDBC 驱动程序会要求略有不同的 spec。例如一个 MySQL 会话的 spec 可能看起来像这样的：

```
{:classname "com.mysql.jdbc.Driver"
  :subprotocol "mysql"
  :subname "//localhost:3306/databasename"
  :username "login"
  :password "password"}
```

或者 `closure.java.jdbc` spec 可以创建成直接指定一个 `javax.sql.DataSource`：

```
{:datasource datasource-instance
  :username "login"
  :password "password"}
```

或者通过 JNDI 找到一个：

```
{:name "java:/comp/env/jdbc/postgres"
```

```
:environment {} ; optional JNDI parameters for initializing  
javax.naming.InitialContext
```

或者对许多常用数据库，^{注1} 可以不用映射这个规约，而是像这样提供一个 URI 风格的连接字符串：

```
"mysql://login:password@localhost:3306/databasename"
```

如果你已经熟悉 JDBC，这些方法任何一种看起来应该都是熟悉的；从一个 JDBC 风格 493 的连接字符串推导出一个合适的 URI 连接字符串或数据库 spec 应该是小事一桩。

一旦你的数据库 spec 确定下来，就可以开始使用 `closure.java.jdbc` API 了：

```
(jdbc/with-connection db-spec)  
:= nil
```

`with-connection` 用于打开到数据库的一个连接。如果没有给 `with-connection` 主体，连接会在打开后又立即关闭；这在 REPL 里是一个有用的健康检查，因为不正确的用户名、密码或数据库 URL 都将产生异常。

所有在 `with-connection` 作用范围内的表达式可以在一个活生生的、开放的数据库连接的环境里执行。在控制从那个范围退出时，连接会自动被关闭或释放。^{注2}

可以使用 `create-table` 来创建一个名为“authors”的表，并定义一些栏。关键字和字符串可以用做表和栏的名称；后者最适合于表名或栏名包含了不能表示为关键字字面量的字符：

```
(jdbc/with-connection db-spec  
(jdbc/create-table :authors  
  [:id "integer primary key"]  
  [:first_name "varchar"]  
  [:last_name "varchar"]))  
:= ()
```

`insert-records` 是一个简单的函数，把数据插入到数据库，返回包含为每个插入的记录生成的键的映射序列。映射中的键对应于表中的栏：

```
(jdbc/with-connection db-spec  
(jdbc/insert-records :authors  
  {:first_name "Chas" :last_name "Emerick"}  
  {:first_name "Christophe" :last_name "Grand"}  
  {:first_name "Brian" :last_name "Carper"}))
```

注1： 在本书写作时，连接字符串可用于 PostgreSQL、MySQL、SQLite、HSQLDB 和 Derby。

注2： Clojure 里常用的习惯用法，这只是自动资源管理的一个例子。你会找到许多类似的 `with-*` 函数用于打开和关闭文件句柄、网络连接等。见 364 页“`with-open :finally` 的挽歌”一节，详细讨论了 `with-open`，这是 Clojure 里最常用的 `with-*` 函数。

```
;= {:last_insert_rowid() 1}
;= {:last_insert_rowid() 2}
;= {:last_insert_rowid() 3})
```

`with-query-results` 用来从数据库取回数据。这里使用的 `doall` 很重要；我们很快会在 495 页“处理惰性”一节里详细解释：

```
[494]  (jdbc/with-connection db-spec
      (jdbc/with-query-results res ["SELECT * FROM authors"]
        (doall res)))
;= ({:id 1, :first_name "Chas", :last_name "Emerick"}
;= {:id 2, :first_name "Christophe", :last_name "Grand"}
;= {:id 3, :first_name "Brian", :last_name "Carper"})
```

注意，在整个例子里，我们只是处理原生的 Clojure 数据类型。表名和栏名都是关键字，数据库句柄是映射；映射用于向数据库插入数据，而且查询数据库返回映射的序列。这与 Clojure 拥有少量通用的数据类型和大量操作这些类型的函数这一设计理念契合。^{注3} 访问结果集的每一行的字段就是在映射里做简单的关键字查询。例如可以生成表中每个作者的全名，只需一个简单的 `map`。

例14-1：用`map`来转换结果集

```
(jdbc/with-connection db-spec
  (jdbc/with-query-results res ["SELECT * FROM authors"]
    (doall (map #(str (:first_name %) " " (:last_name %)) res))))
;= ("Chas Emerick" "Christophe Grand" "Brian Carper")
```

你可以用 Clojure 里可用的任何设施来处理结果集。

with-query-results 解析

`with-query-results` 是从数据库里获取数据的首要方法。`(with-query-results res query & body)` 将在数据库里执行 `query`，然后对 `body` 求值，查询的结果集与局部量 `res` 绑定。结果集自身是 Clojure 映射的一个惰性序列。

`with-query-results` 支持参数化的查询，这是 SQL 函数库的共同特性，字符串的查询模板包含对应于单独提供的查询参数的占位符，然后由数据库来插入。参数化的查询促进了查询的重用，这可能提升运行多次的查询的效率，与等价的作法（通过字符串拼接建立查询，因而为 SQL 注入攻击打开了大门）相比，这对安全也有好处。

查询应该是一个向量，第一个项目应该是一个 SQL 字符串，其后的值对应于字符串查询里的每个参数占位符。例如：

```
(jdbc/with-connection db-spec
```

注3： Perlis 值得参考，见 84 页“抽象优于实现”一节。

```
(jdbc/with-query-results res ["SELECT * FROM authors WHERE id = ?" 2] ❶
  (doall res))
:= ({:id 2, :first_name "Christophe", :last_name "Grand"})
```

❶ 查询里的 ? 代表一个参数，而 2 是绑定到这个参数的值。

注意，你提供的用于绑定到参数的值的类型并非至关重要。Clojure 作为一门动态语言，◀ 495 你提供的值会被插入到合适的基本 SQL 语句里。如果传递的值是不可接受的类型（即你的 JDBC 驱动无法把它强制转换成合适的类型），JDBC 将会抛出一个异常。

处理惰性。你可能已经注意到了 doall 和 with-query-results 一起使用。with-query-results 返回的结果集是一个惰性序列，^{注4} 其中的每个记录只在需要时才实现。这意味着我们可以处理查询返回的巨大数据集，不必遇到资源限制，但我们需要谨慎行事，因为数据的来源是暂时的数据库连接。考虑这个看起来很简单的例子：

```
(jdbc/with-connection db-spec
  (jdbc/with-query-results res ["SELECT * FROM authors"]
    res))
:= ({:id 1, :first_name "Chas", :last_name "Emerick"})
```

哇，这个查询只返回了一行，但我们预期会有三行。问题出在 with-query-results 返回的 res，我们的惰性结果集，没有对它求值。REPL 最终试图打印这个惰性序列，这最终强制对惰性序列求值。但为时已晚：数据库句柄已经关闭（因为我们已经离开了 with-connection 的作用范围），结果是看到不完整的（或者完全没有）结果。^{注5}

解决的办法是在数据库连接还没断开时从结果集获取我们所有需要的东西。某些操作，像 reduce，将会隐式地强制结果集被实现；使用产生惰性序列集的其他操作来转换结果集就需要在一个 doall 的调用里包装起来，如我们在例 14-1 所做的那样。

如果我们想做的只是进行一个查询并完整地获取结果，可以很容易设置一个工具函数来做这件事：

```
(defn fetch-results [db-spec query]
  (jdbc/with-connection db-spec
    (jdbc/with-query-results res query
      (doall res))))
:= #'user/fetch-results
(fetch-results db-spec ["SELECT * FROM authors"])
:= ({:id 1, :first_name "Chas", :last_name "Emerick"}
 := {:id 2, :first_name "Christophe", :last_name "Grand"}
 := {:id 3, :first_name "Brian", :last_name "Carper"})
```

注 4：见 93 页“惰性序列”一节。

注 5：其他 JDBC 驱动会在你试图从关闭的连接读取数据时抛出异常；SQLite 驱动似乎提前获取了第一行数据，因而有这个单行结果。

数据库操作要作为单个事务来执行，只需简单地在一个 `transaction` 形式里包装一下。`transaction` 接受任何数量的形式，然后作为单个事务逐一执行每个形式。如果抛出一个异常，或者执行的操作会违反数据库模式里确立的约束，这个事务会被回滚。如果 `transaction` 的主体执行完成没有出错，这个事务就被提交了。

```
(jdbc/with-connection db-spec
  (jdbc/transaction
    (jdbc/delete-rows :authors ["id = ?" 1])
    (throw (Exception. "Abort transaction!")))) ❶
  ;;= ; Exception Abort transaction!
  (fetch-results ["SELECT * FROM authors where id = ?" 1])
  ;;= ({:id 1, :first_name "Chas", :last_name "Emerick"}) ❷
```

❶ 这里抛出了一个异常，迫使事务中止。

❷ 数据库里的数据没有改变。

`transaction` 是一个宏，处理启动一个事务、在出现异常的情况下确保事务回滚的细节。它也处理在必要时禁用 JDBC 连接的自动提交功能及在 `transaction` 形式求值结束后恢复它的原值等琐事。

假设我们想将连接的事务分隔水平设置成 `TRANSACTION_SERIALIZABLE`。`closure.java.jdbc` 并不直接支持这点，但幸亏有 Clojure 的动态性质、良好的 Java 互操作支持以及避免数据隐藏的理念，我们仍然可以做到这点。

`TRANSACTION_SERIALIZABLE` 本身是 `java.sql.Connection` 类的一个静态成员，因而在 Clojure 里可以用 `java.sql.Connection/TRANSACTION_SERIALIZABLE` 指称。我们可以在 `with-connection` 的范围里用 `closure.java.jdbc` 的 `connection` 函数来访问动态绑定的当前连接。知道了这一点后，可以这样来设置事务分隔水平：

```
(jdbc/with-connection db-spec
  (.setTransactionIsolation (jdbc/connection)
    java.sql.Connection/TRANSACTION_SERIALIZABLE)
  (jdbc/transaction
    (jdbc/delete-rows :authors ["id = ?" 2])))
```

连接池

`with-connection` 易于使用，不过默认情况下每次它被调用都会打开和关闭一个新的数据库连接。这样虽然简单直接，不过可能成为大的瓶颈。

连接池是一种创建数据库连接缓存的办法，以便连接可以反复重用。许多应用服务器提

供基于 `DataSource` 的连接池，经常可以通过 JNDI 来寻址。如果你没有使用一个应用服务器，可能想用 c3p0 (<http://www.mchange.com/projects/c3p0/>)，一种流行的轻量级连接池函数库，可以在任何环境下部署：

```
[c3p0/c3p0 "0.9.1.2"]
```

把 c3p0 加入到项目的依赖后，就可以轻易地设置一个简单的函数，它接受一个映射风格的数据库规格说明，返回一个 c3p0 连接池支持的 `DataSource`：

```
(import 'com.mchange.v2.c3p0.ComboPooledDataSource)
; Feb 05, 2011 2:26:40 AM com.mchange.v2.log.MLog <clinit>
; INFO: MLog clients using java 1.4+ standard logging.
;= com.mchange.v2.c3p0.ComboPooledDataSource

(defn pooled-spec
  [{:keys [classname subprotocol subname username password] :as other-spec}]
  (let [cpds (doto (ComboPooledDataSource.)
    (.setDriverClass classname)
    (.setJdbcUrl (str "jdbc:" subprotocol ":" subname))
    (.setUser username)
    (.setPassword password))])
  {:datasource cpds}))
```

- ① c3p0 的 `ComboPooledDataSource` 是一个 `DataSource` (Java 对于任何数据库连接源的标准接口)，因而它可以不需修改直接用于 `with-connection`。

连接在首次使用时会被初始化，然后会（根据连接池的配置）保留下来供 `with-connection` 后续调用使用。

```
(def pooled-db (pooled-spec db-spec))
; Dec 27, 2011 8:49:28 AM com.mchange.v2.c3p0.C3P0Registry banner
; INFO: Initializing c3p0-0.9.1.2 [built 21-May-2007 15:04:56; debug? true; trace: 10]
;= #'user/pooled-db

(fetch-results pooled-db ["SELECT * FROM authors"])
; Dec 27, 2011 8:56:40 AM com.mchange.v2.c3p0.impl.AbstractPoolBackedDataSource
  getPoolManager
; INFO: Initializing c3p0 pool... com.mchange.v2.c3p0.ComboPooledDataSource
; [ acquireIncrement -> 3, acquireRetryAttempts -> 30, acquireRetryDelay -> 1000, ...
;= ({:id 1, :first_name "Chas", :last_name "Emerick"}
;= {:id 2, :first_name "Christophe", :last_name "Grand"}
;= {:id 3, :first_name "Brian", :last_name "Carper"})]

(fetch-results pooled-db ["SELECT * FROM authors"])
;= ({:id 1, :first_name "Chas", :last_name "Emerick"}
;= {:id 2, :first_name "Christophe", :last_name "Grand"}
;= {:id 3, :first_name "Brian", :last_name "Carper"})
```

第二个查询会重用第一个查询的同一个连接（第二个 `fetch-results` 调用缺少初始化日志记录暗示了这一点）。c3p0 的默认配置适合许多应用程序，不过我们推荐在你本地实现的 `pooled-spec` 里充分利用它所提供的众多配置选项，以便使吞吐量最大化。

498 Korma

Korma (<http://sqlkorma.com>) 是在 Clojure 里使用关系数据库的一门很有前途的领域专用语言。它的目标是提供“装备齐全的”、Clojure 原生的数据库交互体验；为此目的，它精心地为许多不同的流行数据库生成 SQL，并处理如像通过 c3p0 管理连接池的管理任务。对于那些熟悉 Ruby 的 ActiveRecord 或类似的对象-关系映射器的人来说，Korma 应该看起来相当眼熟，虽然它肯定不是对象-关系映射框架。

要使用 Korma，需要首先把它加到项目的依赖中。

```
[korma "0.3.0"]
```

前奏

我们先用 `closure.java.jdbc` 建几个表，插入一些数据供使用。

```
(require '[closure.java.jdbc :as jdbc])

(def db-spec {:classname "org.sqlite.JDBC"
              :subprotocol "sqlite"
              :subname "test.db"})

(defn setup []
  []
  (jdbc/with-connection db-spec
    (jdbc/create-table :country
      [[:id "integer primary key"]
       [:country "varchar"]])
    (jdbc/create-table :author
      [[:id "integer primary key"]
       [:country_id "integer constraint fk_country_id
                     references country (id)"]
       [:first_name "varchar"]
       [:last_name "varchar"]])
    (jdbc/insert-records :country
      {[:id 1 :country "USA"]}
      {[:id 2 :country "Canada"]}
      {[:id 3 :country "France"]})
    (jdbc/insert-records :author
      {[:first_name "Chas" :last_name "Emerick" :country_id 1]}
      {[:first_name "Christophe" :last_name "Grand" :country_id 3]}))
```

```
{:first_name "Brian" :last_name "Carper" :country_id 2}
{:first_name "Mark" :last_name "Twain" :country_id 1)}))

(setup)
:= ({:id 1, :country_id 1, :first_name "Chas", :last_name "Emerick"}
:= {:id 2, :country_id 3, :first_name "Christophe", :last_name "Grand"}
:= {:id 3, :country_id 2, :first_name "Brian", :last_name "Carper"}
:= {:id 4, :country_id 1, :first_name "Mark", :last_name "Twain"})
```

我们的表定义为作者与国家间的多对一关系。设置 Korma 可以很容易使用我们的数据库： [499](#)

```
(use '[korma db core])
(defdb korma-db db-spec)
```

`defdb` 定义一个 Korma 能用的连接，这个命令接受与传给 `clojure.java.jdbc` 的连接映射相同的参数，因而这里重用 `db` 映射。

最近求值的 `defdb` 形式被设为“默认”连接，会用于所有的查询。如果只有一个数据库要连接，这是很方便的，可以覆盖绝大多数用例。`defdb` 也非常有帮助地为数据库连接设立一个连接池，让你不用手工做这件事。⁶

设置 Korma 的下一步是定义实体，这是告诉 Korma 你的数据库表的各种属性的规格说明。实体与 Ruby 的 ActiveRecord 里的“模型”相似。我们的实体可以像这样：

```
(declare author)

(defentity country
  (pk :id)
  (has-many author))

(defentity author
  (pk :id)
  (table :author)
  (belongs-to country))
```

`defentity` 还定义了数据库里表之间的关系。

查询

表的关系定义好后，查询就简单了：

```
(select author
  (with country)
  (where {:first_name "Chas"}))
```

注 6： 你可以用 Korma 的 `get-connection` 函数来获得它所设立的连接池里的一个连接，例如 `(get-connection korma-db)`。如果正好需要做某些需要使用 `clojure.java.jdbc` 的事，这让你重用 Korma 的连接池。

```
;= [{:id 1, :country_id 1, :first_name "Chas",
     :last_name "Emerick", :id_2 1, :country "USA"}]
```

Korma 的 `select` 宏是执行 SQL SELECT 查询的 DSL。`select` 接受多种可以用于构建查询的函数为参数。例如这里用的 `with` 函数告诉 Korma 包含一个关系，前面是用 `defentity` 定义的。注意结果里包含的 `country` 键 - 值对。

500 > 再举一个更复杂的例子：

```
(select author
  (with country)
  (where (like :first_name "Ch%"))
  (order :last_name :asc)
  (limit 1)
  (offset 1))
;= [{:id 2, :country_id 3, :first_name "Christophe",
     :last_name "Grand", :id_2 3, :country "France"}]
```

`order`、`limit` 和 `offset` 是对应的 SQL 语句的简单明了的表示。更有意思的是，`where` 函数，它本身是构建 SQL 的 WHERE 语句的微型 DSL。`where` 能够处理相当复杂的情况：

```
(select author
  (fields :first_name :last_name)
  (where (or (like :last_name "C%")
             (= :first_name "Mark"))))
;= [{:first_name "Brian", :last_name "Carper"}
;= {:first_name "Mark", :last_name "Twain"}]
```

如果想看看后台 Korma 所生成的原始 SQL 语句，可以用 `sql-only` 函数：

```
(println (sql-only (select author
  (with country)
  (where (like :first_name "Ch%"))
  (order :last_name :asc)
  (limit 1)
  (offset 1))))
;= ; SELECT "author".* FROM "author" LEFT JOIN "country"
;= ; ON "country"."id" = "author"."country_id"
;= ; WHERE "author"."first_name" LIKE ?
;= ; ORDER BY "author"."last_name" ASC LIMIT 1 OFFSET 1
```

为什么不嫌麻烦地用 DSL？

当然你可以在 Clojure 里用字符串的原始 SQL 语句来运行 SQL 查询。事实上，那正是 `clojure.java.jdbc` 要求你做的。不过 Korma 的办法是有优势的。

SQL 字符串是没有结构的，操纵这样的字符串很难。给定 `SELECT * FROM foo ORDER BY bar`，你如何改变这个查询来选择 * 之外的东西呢？如何加一个 WHERE 语句呢？我们很可

能需要一个完整的 SQL 解析器。

Korma 把查询表示为简单的 Clojure 映射，而不是无结构的字符串。事实上，我们可以自己一点一点地组装起这些查询，使用 `select*` 而不是 `select`：

```
(def query (-> (select* author)
                  (fields :last_name :first_name)
                  (limit 5)))
:= #'user/query
```

我们来看看这个 `query` 映射像什么样子：

501

```
{:group [],
 :from
 [{:table "author",
   :name "author",
   :pk :id,
   :db nil,
   :transforms (),
   :prepares (),
   :fields [],
   :rel
   {"country"
    #<Delay@54f690e4:
     {:table "country",
      :alias nil,
      :rel-type :belongs-to,
      :pk {:korma.sql.utils/generated "\"country\".\"id\""},  
:fk
      {:korma.sql.utils/generated "\"author\".\"country_id\""}>}}],
:joins [],
:where [],
:ent
{:table "author",
 :name "author",
 :pk :id,
 :db nil,
 :transforms (),
 :prepares (),
 :fields [],
 :rel
 {"country"
  #<Delay@54f690e4:
   {:table "country",
    :alias nil,
    :rel-type :belongs-to,
    :pk {:korma.sql.utils/generated "\"country\".\"id\""},  
:fk {:korma.sql.utils/generated "\"author\".\"country_id\""}>}},
:limit 5,
```

```
:type :select,
:alias nil,
:options nil,
:fields {:last_name :first_name},
:results :results,
:table "author",
:order [],
:modifiers [],
:db nil,
:aliases #{}}
```

修改一个查询现在简单了，就是操纵 `select*` 生成的 Clojure 基准映射，这正像 `order`、`limit` 和 `offset` 等函数所做的事。现在不是总在从头到尾定义这些函数，我们可以增量地构建查询，然后在高兴的时候用 `exec` 函数执行这些查询；这让我们重用查询的某些部分，把查询变换封装在可重用的函数里，这些都是减少代码重复的办法。

Ruby on Rails 受人推崇的 ActiveRecord 最新版本（本书写作时为 3.0 版）已经转而使用一个非常相似的方法，对查询对象用方法调用形成 SQL 查询。在 Ruby on Rails 里，你可能写成像这样的东西：

```
employees = Person.where(:type => "employee")
# ... later ...
managers = employees.where(:role => "manager").order(:last_name)
managers.all.each do |e|
  ...
end
```

用 Korma 做这件事非常相似：

```
(def employees (where (select* employees) {:type "employee"}))

;; ... later ...
(let [managers (-> employees
                    (where {:role "manager"})
                    (order :last_name))]
  (doseq [e (exec managers)]
    ; ... process results ...
  ))
```

惰性。这种方式创建的查询是“惰性的”，因为直到使用 Korma 的 `select` 函数显式地执行查询，数据才会从数据库里获取的。这与 Clojure 的惰性数据结构相比是不同类型惰性，可能更准确地应该叫做“按需查询”。

假设我们有一个表，包含所有曾经生活过的人。可以指定这个表的记录总是按照出生日期排序。然后在获取数据前，用谓词把它的范围缩小，应用 LIMIT 和 OFFSET 来分页查询。

```

(def humans (-> (select* humans)
                  (order :date_of_birth))) ❶

(let [kings-of-germany (-> humans
                                (where {:country "Germany" :profession "King"}))] ❷
  (doseq [start (range 0 100 10)
          k (select kings-of-germany
                     (offset start)
                     (limit 10))]

  ...))

```

- ❶ 如果按照现状执行查询，结果集将会有数十亿个记录那么大。不过 `humans` 在整个使用中都捕获了我们想要的排序。
- ❷ 在特定环境下，可以根据需要改进查询的参数……
- ❸ 并用这个改进作为更进一步的细粒度查询的基础——如分页，不需重新表述此前在查询映射里累积的标准。 <503>

Hibernate

如果你已经在用 Java 或另一种 JVM 语言做 RDBMS 工作，那么很可能你在用 Hibernate (<http://www.hibernate.org>)，它多半是最流行的 Java 中的对象 - 关系映射函数库。Clojure 的一个优势之一是能够无缝地使用 Java 函数库和框架，Hibernate 也不例外。

Hibernate 和 Clojure 相比在理念上有很大的差别：它是通过创建对象、修改这些对象，然后把这些修改翻译成数据库查询来运作的。而 Clojure 非常灵活，让 Hibernate 几乎不费什么事就足以一起使用。

设置

让我们设置 Hibernate 来创建、访问和更新本章前面的 `authors` 表。首先需要把 Hibernate 加为项目的依赖：

```
[org.hibernate/hibernate-core "4.0.0.Final"]
```

在 Clojure 里使用 Hibernate 的多数情况可能会涉及已经在 Java 里存在的领域对象。^{注7} 这里是一个普通的 Java 类，表示作者名字信息，混合使用 Hibernate 和 JPA 注释来表示它代表一个数据库实体，并指明 `id` 字段通常的自动加一的行为：

```
package com.clojurebook.hibernate;
```

^{注7}： 你可以用 Clojure 写 Hibernate/JPA 的 Entity 类，相当容易。关于并用 Clojure 类型定义和 Java 框架注释的其他例子，参见 381 页“注解”一节。在用 Hibernate/JPA 时，你将需要用 `gen-class`，因为实体要求一个默认的 / 没有参数的构造函数在执行查询时创建实体的实例。

```
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Entity;
import org.hibernate.annotations.GenericGenerator;

@Entity
public class Author {
    private Long id;
    private String firstName;
    private String lastName;

    public Author () {}

    public Author (String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Id
    @GeneratedValue(generator="increment")
    @GenericGenerator(name="increment", strategy = "increment")
    public Long getId () {
        return this.id;
    }

    public String getFirstName () {
        return this.firstName;
    }

    public String getLastName () {
        return this.lastName;
    }

    public void setId (Long id) {
        this.id = id;
    }

    public void setFirstName (String firstName) {
        this.firstName = firstName;
    }

    public void setLastName (String lastName) {
        this.lastName = lastName;
    }
}
```

504

不像 clojure.java.jdbc 或 Korma 那样接受 spec 映射表，Hibernate 通常用一个 *hibernate.cfg.xml* 文件进行配置。这个文件指定应该把目标设为在内存里的 SQLite 数据库。

例14-2: rsrc/hibernate.cfg.xml

```
<!DOCTYPE hibernate-configuration SYSTEM  
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">  
<hibernate-configuration>  
  <session-factory>  
    <property name="hibernate.connection.driver_class">org.sqlite.JDBC</property>  
    <property name="hibernate.connection.url">jdbc:sqlite::memory:</property>  
    <property name="hibernate.dialect">org.hibernate.dialect.HSQLDialect</property>  
    <!-- Drop and re-create the database schema on startup -->  
    <property name="hbm2ddl.auto">create</property>  
    <mapping class="com.clojurebook.hibernate.Author"/>  
  </session-factory>  
</hibernate-configuration>
```

最后，如果用 Leiningen，只需要把一两个键加到 *project.clj*，一个用于我们想要编译的表示 Java 源文件根（例如 *Author.java*），另一个表示资源的根路径，我们把 *hibernate.cfg.xml* 文件保存在那儿：

```
:java-source-path "java"  
:resources-path "rsrc"
```

这使得我们的项目结构像下面这个样子：

```
|-- project.clj  
|-- rsrc  
|  '-- hibernate.cfg.xml  
|-- java  
|  '-- com  
|    '-- clojurebook  
|      '-- hibernate  
|        '-- Author.java
```

505

现在我们可以编译 Java 类，然后启动一个 REPL，开始看看是如何在 Clojure 里使用 Hibernate 的：

```
% lein javac  
...  
% lein repl
```

我们需要首先从 Hibernate 引进所需的 Java 类，还有新建的 Author 类：

```
(import 'org.hibernate.SessionFactory  
       'org.hibernate.cfg.Configuration  
       'com.clojurebook.hibernate.Author)
```

Hibernate 要求设置一个会话工厂对象来打开和关闭数据库会话、运行查询。重要的是，这个工厂对象只能初始化一次。在 Java 里，我们可能会创造一个工具类，它有一个 static final 成员代表这个会话工厂。工厂会在类被载入时实例化。这样的 Java 代码并

非罕见，实际上从 Hibernate 的文档里直接可见 (<http://docs.jboss.org/hibernate/core/3.3/reference/en/html/tutorial.html>) :

```
public class HibernateUtil {  
    private static final SessionFactory sessionFactory =  
        buildSessionFactory();  
  
    private static SessionFactory buildSessionFactory() {  
        try {  
            return new Configuration().configure().buildSessionFactory();  
        }  
        catch (Throwable ex) {  
            System.err.println("Initial SessionFactory creation failed." + ex);  
            throw new ExceptionInInitializerError(ex);  
        }  
    }  
  
    public static SessionFactory getSessionFactory() {  
        return sessionFactory;  
    }  
}
```

在 Clojure 里，有更为简单的解决方案，使用 `defonce` 和 `delay` :

```
(defonce session-factory  
  (delay (-> (Configuration.  
      .configure  
      .buildSessionFactory)))
```

- [506] 正如它的名称所暗示的，`defonce` 表现得和 `def` 一样，定义一个 var 有某个值，但让任何定义都保持不动，即使重新启动包含 `session-factory` 定义的命名空间。`delay` 确保 `->` 表达式在实际创造和配置这个会话工厂时没有被求值，直到它被 `deref` 解引用；这让我们载入或提前编译源文件，不会无意地试图连接到我们配置的数据库。

持续数据

`hibernate.cfg.xml` 设置为使用在内存的 SQLite 数据库，所以在 REPL 启动时它是空的。在 Java 里的典型做法可能是像这样的：

```
public static void saveAuthors (Author... authors) {  
    Session session = sessionFactory.openSession();  
    session.beginTransaction();  
    for (Author author : authors) {  
        session.save(author);  
    }  
    session.getTransaction().commit();  
    session.close();
```

```
}
```

```
saveAuthors(new Author("Christophe", "Grand"), new Author("Brian", "Carper"), ...);
```

把这个简单翻译成 Clojure 将会像这里的 add-authors :

例14-3: add-authors

```
(defn add-authors
  [& authors]
  (with-open [session (.openSession @session-factory)]
    (let [tx (.beginTransaction session)]
      (doseq [author authors]
        (.save session author))
      (.commit tx))))
```

```
(add-authors (Author. "Christophe" "Grand") (Author. "Brian" "Carper")
             (Author. "Chas" "Emerick"))
```

运行查询

现在有了一些持续的代码数据。我们从 authors 表中获取一个列的表单。在 Java 里，可能像下面这样做：

```
Session session = HibernateUtil.getSessionFactory().openSession();

try {
    return (List<Author>)newSession.createQuery("from Author").list();
} finally {
    session.close();
}
```

这个代码可以直截了当地翻译成 Clojure :

507

例14-4: get-authors

```
(defn get-authors
  []
  (with-open [session (.openSession @session-factory)]
    (-> session
          (.createQuery "from Author")
          .list)))
```

当然，因为我们在 Clojure 里，在操纵从 Hibernate 查询中获得数据上有更多的灵活性：

```
(for [{:keys [firstName lastName]} (map bean (get-authors))] ❶
  (str lastName ", " firstName))
  := ("Carper, Brian" "Emerick, Chas" "Grand, Christophe")
```

❶ 这里的 bean 把任何 JavaBean 风格的 Java 对象放入一个 Clojure 的哈希映射，每个读取函数对应于映射里的一个槽，它的关键字键与 JavaBean 里的读取函数同名。

对 Hibernate 查询对象使用 `.list` 方法返回一个 `java.util.ArrayList` 类型的结果。我们能够用 `doseq` 来循环它，是因为 Clojure 确保所有的 `java.util.List` 都可以序列化。

去掉样板

我们的 Clojure 代码能工作，但还可以改进。

注意两个函数里的重复现象。开启和关闭会话与开始和提交交易是两项不得不重复进行的事。

在 Java 里，你可能处于不断重敲代码不能自拔的情境；在 Clojure 中可不一样。好的开始是使用 Clojure 内置的 `with-open` 宏，它让我们自动打开一个连接或处理对象，运行某些代码，然后在使用结束后保证关闭处理对象或连接。因为 Hibernate 会话是用标准的 `.close` 方法关闭的，`with-open` 在 Hibernate 会话上也能用。

不过我们能做得更好。`with-open` 要求为本地 `session` 指定名称；在 `with-open` 之上使用一个宏，我们可以简单地说作为惯例，它总会在宏的主体内被具名为 `session`：

```
(defmacro with-session
  [session-factory & body]
  `(with-open [^'session (.openSession ~(vary-meta session-factory assoc
                                         :tag 'SessionFactory))] @@)
   @body))
```

❶ 第一个参数 `session-factory` 是一个会被用来获得打开的会话的形式。其他的参数是将会在这个打开的会话环境里执行的形式。

- 508 ➤ ❷ 不提示 `session-factory`，编译器不会知道 `(.openSession factory#)` 将会返回一个 Hibernate 的 `Session`，因而涉及 `session` 的所有调用都会是反射的。不过，我们不能只是提示未加引用的 `session-factory` 符号，因为那将应用于 `~session-factory` 形式，而不是用户提供的在 `session-factory` 宏里绑定的符号。修正办法是修改与 `session-factory` 绑定的符号的元数据，以便值会被合适地提示。注意，`:tag` 的值自身是一个符号，不是一个类。

不使用一个 `gensym` 来建立一个“安全的”本地绑定，不致在 `with-session` 的主体里遮蔽已有的绑定或“泄漏”到用户代码里，是因为我们想在那个范围内绑定 `session` 到 Hibernate 的 `Session`。为此，我们用 `~'session` 来强制发出一个无修饰的（没有命名空间的）符号。^{注8} 在宏里这个点上的光杆 `session` 符号会导致一个编译错误，因为宏扩展将会自动给这个符号加上当前的命名空间，即 `user/session`，这是没法在本地绑定中作为名称

注8： 通过确立对用户代码可见的隐式绑定，`with-session` 是一个“前指”的（有时称为“不健康的”）宏。关于前指宏的更多信息，参见 244 页“宏卫生”一节和 248 页“让宏的用户来选择名字”一节。

使用的。

这个宏到位后，我们可以把例 14-4 改写得更为简洁：

```
(defn get-authors
  []
  (with-session @session-factory
    (-> session
      (.createQuery "from Author")
      .list)))
```

就它本身而言，这算不上很大的好处，但在涉及打开的会话里乘以超过数十、数百甚至数千次 Hibernate 交互，这就算是不错的进展了。更有意义的是，在一个 Hibernate 事务的环境里执行操作的辅助宏：

```
(defmacro with-transaction
  [& body]
  `(let [^'tx (.beginTransaction `session)] ●
     @body
     (.commit ^'tx)))
```

- ① 因为 Hibernate 的 `Transaction` 对象提供了有用的方法（就像 `Session` 对象那样），我们可以隐式地把当前的 `Transaction` 与本地的 `tx` 绑定以便用户代码可以轻松访问它。这使得 `with-transaction` 成为一个前指的宏。

这里，`session` 是一个预期已经与当前打开的会话的值绑定的名称；因而 `with-transaction` 将无缝地与前指 `session` 名称在 `with-session` 里绑定。生成的代码将会开始一次事务，执行 `body` 里的其他形式，然后提交事务。这让我们生成一个比例 14-3 简单许多的实现：

```
(defn add-authors
  [& authors]
  (with-session @session-factory
    (with-transaction
      (doseq [author authors]
        (.save session author)))))
```

509

把像会话和事务的笔记中涉及的样板代码和语法复杂性从日常需要用到的代码基里剔除，这会使得代码更易读、使用起来更开心，也许最重要的是不容易出错。

最后的思考

Clojure 对于使用关系数据库有非常棒的支持。JVM 和 JDBC 为全面数据库支持提供了很好的基础，纯 Clojure 的选项提供了更多强大且可组合的分层功能，如果有需要，也很容易切换回成熟的 Java 框架，如 Hibernate。

使用非关系型数据库

对于需要持续化数据的应用程序开发者来说，关系型数据库成为功能上唯一可用的选择。许多年后，大量新类型的数据库逐渐被视为可以合法地替代无处不在的 RDBMS。这些新的数据库相互之间差别较大。尽管有这些差异，键 - 值存储和面向列和面向文档的数据库共同构成了长期居于主导地位的关系数据模型正统地位的替代方案；因而，它们经常一起被称为非关系型数据库。^{注1} 这些数据存储的独特能力和不断增长的流行程度使得它们成为新的 Clojure 应用程序的共同组件，所以值得看看这样的组合会是什么样子。

CouchDB 是一个面向文档的非关系型数据库，它定义了一个数据模型和架构，与 Clojure 的长处和世界观吻合。这个组合非常有力，让许多类型的应用程序，从典型的 Web 前端到非常容易扩展的消息系统，可以相对简单地实现。

作为开始，先描述一两个与 Clojure 特别相关的 CouchDB 的特性是有益的：

- 它的数据模型完全是由 JSON 文档组成的，这可以简单地从 Clojure 的数据结构转换来转换去。
- 它使用一种只能附加的基于 btree 的存储系统，默认保证数据的可持续性和操作的原子性。
- 它使用 MVCC 来处理来自多个客户端的并发修改，Clojure 的软件事务内存使用的是同样的乐观主义的、带版本标记的修改管理模型。
- 它允许以称为视图的数据转换方式定义不那么简单的查询，它可以用几乎任何语言来实现，包括 Clojure。

注1：或者，更为流行（而不幸）地称为“NoSQL”数据库。

512 ➤ Clojure 的持续数据结构和对良好定义的并发语义的关注使得它非常自然地与 CouchDB 配成一对，CouchDB 同样是以不变的文档定义的，在可持续性、原子性和对修改的冲突管理有明确的语义。

安装 CouchDB 和 Clutch

CouchDB 是一个非常成熟的数据库（尽管它的版本标签只是 1.2.0，这是我们在例子中使用的版本），关于它有大量的文档可以在 Apache 项目站点^{注2} 和 O'Reilly 出版的书中找到。^{注3} 想要完全理解它的人应该参考这些资源。

要开始在 Clojure 里使用 CouchDB，需要首先在本地运行 CouchDB。^{注4} 在所有的例子中，我们将使用 Clutch 提供的 API (<https://github.com/ashafa/clutch>)，这是一个 Clojure 函数库，为 CouchDB 的功能提供全面支持。要使用 Clutch，只需把它作为依赖加到你的项目里：

```
[com.ashafa/clutch "0.3.0"]
```

基本的 CRUD 操作

让我们用 REPL 来探索 CouchDB，从基本的创建、更新和删除文档开始。

例15-1：在REPL里与CouchDB的简单交互

```
(use '[com.ashafa.clutch :only (create-database with-db put-document
                                               get-document delete-document)
                                              :as clutch])

(def db (create-database "repl-crud"))         ❶

(put-document db {:_id "foo" :some-data "bar"})   ❷
;= {:_rev "1-2bd2719826", :some-data "bar", :_id "foo"}
(put-document db (assoc *1 :other-data "quux"))   ❸
;= {:_other-data "quux", :_rev "2-9f29b39770", :some-data "bar", :_id "foo"}
(get-document db "foo")                          ❹
;= {:_id "foo", :_rev "2-9f29b39770", :other-data "quux", :some-data "bar"}
(delete-document db *1)                        ❺
;= {:ok true, :id "foo", :rev "3-3e98dd1028"}
(get-document db "foo")                          ❻
;= nil
```

513 ➤ ❶ 首先，创建一个全新的数据库供 REPL 交互用。

注 2：<http://couchdb.apache.org>——这个维基是一个宝藏，有十分丰富的详细信息。

注 3：由 CouchDB 开发团队成员所写，可以从 <http://guide.couchdb.org> 免费获得，也可获得印刷版。

注 4：或者你可以使用 Cloudant 免费托管的 CouchDB 实例，获得除 Clojure 视图服务器例子外的一切：<https://cloudant.com>。

- ❶ 这里用一个 Clojure 映射创建一个文档。`put-document` 返回所创建的文档，这与提供的映射一样，除了这个额外的 `:_rev` 槽。注意，我们在这里定义了 `:_id`，它将作为文档的“主键”；如果没定义它，CouchDB 将会给文档指定一个 UUID 的 `:_id`。
- ❷ 一次更新操作。注意，`:_rev` 槽的值在返回值里已经被更新了，因为我们已经更新了这个文档。^{注5}
- ❸ 一次简单的读取操作，它总是返回所请求文档的更新修订；你可以选择请求以前的文档版本。
- ❹ 一次删除操作。虽然我们在这个案例里提供全部的文档映射，注意，我们其实只需要提供一个包括 `:_id` 和 `:_rev` 值的映射匹配已有文档的最新版本。
- ❺ 如果指定键没有文档，读取操作返回 `nil`。

相关数据表示间的平行关系使得 Clojure 和 CouchDB 之间的交互非常自然。CouchDB 在整个数据表示中使用 JSON（键 / 值的映射，以字符串为键，标量、数组或其他键 / 值的映射为值），这完美地映射到 Clojure 的映射、向量和标量。幸运的是，Clutch 使用的 JSON 解析器 (<https://github.com/clojure/data.json>) 把 JSON 映射的字符串类型的键转换成 Clojure 关键字；这便于键 / 值对的查找，便于使用像 `get-in` 这样的函数或 Clojure 的串行宏 `->`、`->>` 等遍历嵌套结构。

看看我们能如何简单地查看从 CouchDB 检索得到的文档，就可以说明这个的后果：

```
(clutch/create-document {:_id "foo"
                        :data ["bar" {:details ["bat" false 42]}]})
;= {:_id "foo", :data ["bar" {:details ["bat" false 42]}],
;=   :_rev "1-6d7460947434b90bf88f033785f81cdd"}
(->> (get-document db "foo"))
  :data
  second
  :details
  (filter number?))
;= (42)
```

结果是在 CouchDB 里储存和检索得到的文档差不多等于是“Clojure 原生的”数据结构，因而适宜 Clojure 查询和处理数据的所有惯用手段。这大大简化了应用程序的代码，使数据建模相对清晰、容易，而数据建模经常是数据库使用中最困难的方面之一。

514

^{注5}：记住，`*1` 是一个 REPL 绑定的 `var`，是最后求值的表达式的值，与 Ruby 中的 IRB 和 Python 解释器里的 `_` 相似。关于 REPL 绑定的 `var` 参见 399 页“REPL 绑定的 `var`”一节。

视图

CouchDB 并不提供 SQL 或类似的东西执行即席查询。文档自身只能用单一的“主键”字符串键来创建索引。

CouchDB 提供的替代方案称为“视图”。视图在概念上与一些关系型系统提供的实际的视图非常相似。

- 视图与“源”数据库分别存储、访问。
- 视图是程序定义的（几乎可以使用任何语言，包括我们很快要看到的 Clojure），并且是提前定义的。
- 视图在访问时反映对源数据库中文档的修改。

CouchDB 视图背后的主要见解是应用程序对数据的访问几乎都是可以枚举的，因而与 SQL 和其他即席查询机制相关的灵活性（及其伴随的运行时代价）一般是不恰当的。相反，CouchDB 的视图总是在访问前就在数据库里定义了，^{注6}好处是访问总是极为快速，不管生成视图的数据需要处理的量是多少。

要开始用视图，先来载入假想的日志信息数据集到一个新的数据库里，称为 *logging*。为此我们将用 Clutch 的 bulk-update 函数，它将钩入 CouchDB 的 _bulk_docsAPI；这将是载入大量数据到 CouchDB 的效率最高的途径：

```
(clutch/bulk-update (create-database "logging")
  [{:evt-type "auth/new-user" :username "Chas"}
   {:evt-type "auth/new-user" :username "Dave"}
   {:evt-type "sales/purchase" :username "Chas" :products ["widget1"]}
   {:evt-type "sales/purchase" :username "Robin" :products ["widget14"]}
   {:evt-type "sales/RFQ" :username "Robin" :budget 20000}])
```

一个简单的（JavaScript）视图

CouchDB 视图默认使用 JavaScript 来定义，我们将从这里开始熟悉如何在 Clojure 里访问视图，然后用 Clojure 自身来实现一两个视图。

无论你使用什么语言来实现视图，它都是由一个映射函数和一个可选的归一函数定义的。

515



这个“归一”的概念在 CouchDB 里的定义与在 Clojure 和其他函数式语言里有些不一样，也与 Google 普及的其他数据处理，系统如 Hadoop 和 MapReduce 模型不一样。归一（以及 CouchDB 特有的“重归一”的概念）的语义与 CouchDB 的 B-树数据索引策略紧密连在一起。更多信息参见 http://wiki.apache.org/couchdb/Introduction_to_CouchDB_views#Reduce_Functions 或相关图书。

注 6： 虽然你可以用专门方式创建临时视图，但那样的视图总是会远慢于提前配置和存储的“常规”视图。

我们的第一个视图将覆盖一个明显的用例：报告每一类型有多少日志信息被记录了。需要定义一个映射和一个归一函数来实现它：

```
function(doc) {  
    emit(doc["evt-type"], null);  
}  
  
function(keys, vals, rereduce) {  
    return rereduce ? sum(vals) : vals.length;  
}
```

你可以用 *Futon* 创建这个视图，这是由 CouchDB 提供的管理前端⁷ 或通过 Clutch 像这样来实现：

```
(clutch/save-view "logging" "jsviews" ❶  
(clutch/view-server-fns :javascript ❷  
  {:type-counts ❸  
   {:map "function(doc) {"  
         emit(doc['evt-type'], null);  
     }"  
   :reduce "function(keys, vals, rereduce) {"  
       return rereduce ? sum(vals) : vals.length;  
     "}}))
```

❶ `jsviews` 指定了设计文档的名称，我们的视图储存在这里。设计文档是 CouchDB 数据库里的特殊文档，专门用于保存驱动视图、过滤器和其他数据库内部功能的代码。

❷ 这个视图是用 JavaScript 实现的，我们必须指定它以便设计文档正确地配置。

❸ `:type-counts` 是视图名，可以一次性给 `view-server-fns` 提供许多视图，只用向提供的映射里加入另一条目即可。

现在已经准备好查询我们的视图：

◀ 516

```
(clutch/get-view "logging" "jsviews" :type-counts {:group true})  
:= ({:key "auth/new-user", :value 2}  
:= ({:key "sales/purchase", :value 2}  
:= ({:key "sales/RFQ", :value 1}))
```

这就是我们的记录计数，以日志信息的 `:evt-type` 槽为键。`get-view` 从指定的视图返回结果文档的一个惰性序列，这意味着数千甚至数百万个文档的视图结果可以在 Clojure 里消费、处理毫无困难。而且这些视图结果文档只不过是更多的 Clojure 映射和值，这使得我们很容易用 Clojure 的序列处理来产生略微有用一些的集合形式：

```
(->> (clutch/get-view "logging" "jsviews" :type-counts
```

注 7： 如果在本地运行 CouchDB，可以在 http://localhost:5984/_utils 访问 Futon。

```
(:group true)
  (map (juxt :key :value))
  (into {}))
:= {"auth/new-user" 2, "sales/purchase" 2, "sales/RFQ" 1}
```

正如你可能预期的，在新的日志信息流进入数据库时这些计数会按需更新。



上面使用的`:group`查询选项是把每个独特键的记录归一为视图的值的一个快捷方式。没有它，将得到单一的结果，值为5。视图有各种各样的查询选项，这里我们不会涉及；更多信息参见CouchDB文档(http://wiki.apache.org/couchdb/HTTP_view_API#Querying_Options)。

用Clojure写的视图

由于几乎每个人都知道JavaScript，用JavaScript写CouchDB视图是方便的，而CouchDB自带一个JavaScript视图服务器实现。



CouchDB视图是通过把定义一组视图的源代码与对应于“源”文档的JSON一起提供给称为“视图服务器”的单独进程。视图服务器只是从`stdin`读入数据，在`stdout`上输出应该作为视图结果而保存的东西。实现的简单意味着几乎用任何编程语言实现一个视图服务器都是很简单的。

几乎可以肯定，你永远不需要考虑写一个新的视图服务器：已经有数十种语言的实现，包括为了用Clojure编写视图的Clutch实现。

517> 不过有很好的理由使用非JavaScript的视图服务器，所有这些在Clutch中提供的Clojure视图服务器也是成立的：

- 你可以利用更丰富、更熟悉的语言。
- 与CouchDB预打包的JavaScript视图服务器所提供的相比，你所选的视图服务器的语言几乎肯定可以利用质量更高、更全面的函数库。
- 如果你的视图涉及任何种类的密集处理，你所选的视图服务器的语言很可能提供比JavaScript更高效的运行时。

把你本地的CouchDB实例配置成使用Clutch的视图服务器为Clojure提供视图，这通常要求在启用Clutch的REPL里使用`configure-view-server`函数：^{注8}

```
(use '[com.ashafa.clutch.view-server :only (view-server-exec-string)])
```

注8：ClojureScript(见584页“ClojureScript”一节)可与Clutch一起用于定义CouchDB里的Clojure视图，而不需要配置新的视图服务器——非常方便，特别是当你的CouchDB是托管在Cloudant或其他地方时：<https://github.com/clojure-clutch/clutch-clojurescript>。

```
(clutch/configure-view-server "http://localhost:5984" (view-server-exec-string))
```

`configure-view-server` 函数在你的 CouchDB 实例中创建一个 Clojure 视图服务器条目，带一个 shell 命令，CouchDB 将用它在需要时运行视图服务器。^{注9}

首先，我们先来看看把简单的 JavaScript 视图移植到一个 Clojure 视图的样子。这里，我们将使用 Clutch 把视图保存到一个不同的设计文档里 (`clj-views` 而不是 `jsviews`，在上面的 JavaScript 视图里使用了后者)：

```
(clutch/save-view "logging" "clj-views"
  (clutch/view-server-fns :clojure
    {:type-counts
      {:map (fn [doc]
        [[[{:evt-type doc} nil]]])
       :reduce (fn [keys vals rereduce]
        (if rereduce
          (reduce + vals)
          (count vals))))}))
```

CouchDB 的（因而也包括 Clutch 的）视图 API 是一样的，不论视图是用哪种语言实现的，因而除了指定不同的设计文档，访问 Clojure 视图的代码和以前一样，不用修改：

```
(-> (clutch/get-view "logging" "clj-views" :type-counts {:group true})
  (map (juxt :key :value))
  (into {}))
:= {"auth/new-user" 2, "sales/purchase" 2, "sales/RFQ" 1}
```

不过只是把圆括号替换成大括号不是这里的目标，用 Clojure 写视图，我们可以利用它所有的设施为视图产生数据，包括在需要时使用任何已有的 Clojure 和 Java 函数库。⁵¹⁸

考虑我们的数据集相关的问题域，这里看到的具体事件类型似乎存在某种层级关系（例如 `sales/purchase` 和 `sales/RFQ` 显然是相关联的）。把事件类型用斜杠字符分开，然后用 CouchDB 的视图对齐特性 (http://wiki.apache.org/couchdb/View_collation) 基于隐含的层级自然分组输出计数，它将是很简单的（无论用来写视图的是什么语言）。虽然这样是相当有用的，但是它迫使我们的事件类型使用严格的层级。通过给事件类型创造性地命名或使用某种多重事件分类（例如让 `:evt-type` 成为一个类型的数组）绕过这个问题也是可能的，但是不优雅、不灵活，也更复杂。

更好的方案是用 Clojure 的特别层级来定义事件类型，^{注10} 例如：

注9：这一安装调用只是为在 REPL 里使用方便，关于如何设置 CouchDB 以便能通用于 Clojure 视图服务器，请参考 Clutch 在 <http://github.com/ashafa/clutch> 的自述文件。

注10：关于 Clojure 里的层级与多重方法的更多内容见第 7 章。

例15-2：定义事件层级

```
(ns eventing.types)

(derive 'sales/purchase 'sales/all)
(derive 'sales/purchase 'finance/accounts-receivable)
(derive 'finance/accounts-receivable 'finance/all)
(derive 'finance/all 'events/all)
(derive 'sales/all 'events/all)
(derive 'sales/RFQ 'sales/lead-generation)
(derive 'sales/lead-generation 'sales/all)
(derive 'auth/new-user 'sales/lead-generation)
(derive 'auth/new-user 'security/all)
(derive 'security/all 'events/all)
```

然后可以在视图服务器里使用这种表示来扩展每种具体事件类型为它的所有父类型，有些可能永远不会作为日志消息的 :evt-type 实际出现，但可能表示所涉及的业务里非常重要的东西。选择这一途径让我们简明地提供能够产生多重影响的事件类型。这里是一个实现这一策略的 Clojure 视图：

例15-3：一个用层级扩展的Clojure视图

```
(clutch/save-view "logging" "clj-views"
  (clutch/view-server-fns :clojure
    {:type-counts
     {:map (do
             (require 'eventing.types)
             (fn [doc]
               (let [concrete-type (-> doc :evt-type symbol)]
                 (for [evtsym (cons concrete-type
                                      (ancestors concrete-type))]
                   [(str evtsym) nil])))))
      :reduce (fn [keys vals rereduce]
                (if rereduce
                    (reduce + vals)
                    (count vals)))))))

(->> (clutch/with-db "logging"
           (clutch/get-view "clj-views" :type-counts {:group true}))
      (map (juxt :key :value))
      (into {}))
    := {"events/all" 5,
    := "sales/all" 5,
    := "finance/all" 2,
    := "finance/accounts-receivable" 2,
    := "sales/lead-generation" 3,
    := "sales/purchase" 2,
    := "sales/RFQ" 1,
    := "security/all" 2,
    := "auth/new-user" 2})
```

- 首先，确保载入那个命名空间，其中定义了已知具体事件类型与“祖先”类型之间的关系。这只会发生一次，在视图映射在视图服务器里被实现时且在任何文档被处理之前。注意，所需的命名空间（这里是 `eventing.types`）必须在视图服务器的 classpath 上。
- 把每个 `:evt-type` 字符串转换成一个符号，以便……
- 以便可以获得那个符号在所定义层级的祖先。为每个符号发出单个视图结果，包括一个具体的 `:evt-type`。
- 归一函数仍然与前面那个简单计数视图的一样。

视图结果现在有意思多了。通过用 Clojure 定义事件类型层级，不仅能获得“分部门”的事件计数汇总，与按词典顺序命名方案所允许的相比，跨功能事件可以处理得更有用得多：

- `auth/new-user` 事件除了自然地在安全之下对齐外，还正确地与 lead generation 关联。
- `sales/purchase` 事件的范围已经被扩大到包括 `accounts-receivable` 和 `finance`。

更好的是，我们用 Clojure 写的事件类型层级可以根据需要增加或重组，完全独立于产生实际事件的不同模块或应用。例如，当新的审计需求强制要求用户注册应该被追踪并与大量其他商业数据一起保留，`security/all` 父节点可以被声明为 `audit/all` 新事件类型的一个子节点，因而把 `auth/new-users` 包括在审计范围内，关于如何发出 `auth/new-user` 事件，用户认证系统不需要改变任何东西。

视图函数必须是纯函数 ◀ 520

在写视图时必须记住的一件事是，你无法控制你的视图什么时候被调用或被调用多少次。这一 CouchDB 实现细节在使用 JavaScript 写视图时是无关紧要的，因为它没有提供设施进行 I/O 或其他有副作用的操作。不过在用 Clojure 写视图时（其实是任何非 JavaScript 语言），需要确保你写的函数是纯函数。¹¹ 例如，每次你的视图处理一个 `sales/purchase` 类型的事件时发出的通知邮件将会是一个灾难：每次当那个买卖事件文档变化或在数据库被压缩时（只是为了说明这一点）或 CouchDB 决定它需要使那个买卖事件文档的视图结果失效时，就会发出通知邮件。

_changes：把 CouchDB 滥用做消息队列

CouchDB 提供了一个变化通知 API（称为 `_changes`），让客户端有很大的灵活度定义如何对数据流进行反应。

注 11：关于引用透明与纯函数的讨论，见 76 页“纯函数”一节。

简单地说，`_changes` 是这样工作的：

1. 为感兴趣的 CouchDB 数据库打开一个 HTTP 连接指向 `_changes` URL。
2. 等待。在涉及的数据库里出现的任何变化（任何文档创建、删除或更新），一个 JSON 映射会发送给客户端，描述受影响的文档 ID 和修改。
3. 如果你选择不断接受变化通知，重复第 2 步。

通过 Clojure 使用 `_changes` 的最简单的一种可能是用 Clutch 的 `watch-changes` 函数向 `*out*` 回显所有的变化通知。这里为我们的 `_changes` 试验创建一个新数据库，为数据库设置一个 Clutch 监视函数，再加一些文档看看会发生什么：

```
(clutch/create-database "changes")
(clutch/watch-changes "changes" :echo (partial println "changes")) ❶

(clutch/bulk-update "changes" [{:_id "doc1"} {_id "doc2"}])
:= [{:id "doc1", :rev "5-f36e792166"}
:= {:id "doc2", :rev "3-5570e8bbb3"}]
; change: {:seq 7, :id doc1, :changes [{:rev 5-f36e792166}]}
; change: {:seq 8, :id doc2, :changes [{:rev 3-5570e8bbb3}]}
(clutch/delete-document "changes" (zipmap [:_id :_rev]
                                         ((juxt :id :rev) (first *1))))
:= {:ok true, :id "doc1", :rev "6-616e3df68"}
; change: {:seq 9, :id doc1, :changes [{:rev 6-616e3df68}], :deleted true}
(clutch/stop-changes "changes" :echo) ❷
:= nil
```

521 ➤

- ❶ 注册一个针对 `changes` 数据库的新监视函数，名为 `:echo`，它只是把 `_changes` 的通知回显给 `*out*`。
- ❷ 现在我们将接到对所监视的数据库做出的所有变化，这些通知对应于注册的回显函数的调用。
- ❸ `stop-changes` 会从数据库的 `_changes` 供应退订我们的监视函数。

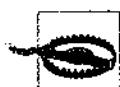
这个机制与可用于 Clojure 原生的引用类型如原子、`var`、`ref` 和代理的监视¹² 完全类似。从概念上说，每一个 CouchDB 数据库包含在单独的原子里；当对数据库做出修改时，那个变化被发送给与你的 `watch-changes` 调用相关联的代理。

此外，CouchDB 允许定义“过滤器”函数，让你可以决定加入程序化的文档过滤，这将会包含在 `_changes` 投递的变化供应里。这些过滤函数可以用任何语言定义——就像视图函数一样——而且在开始从 `_changes` 通过 Clutch 的 `watch-changes` 接受通知时它们的应用可以参数化，我们马上就会看到这一点。

注 12：关于监视的更多详情，参见 176 页“观察器”一节。

使用这些设施创建非常灵活的事件驱动应用程序的可能性是无限的。从系统的角度看，CouchDB 实际上既作为消息队列（或者也许更准确地说，提供事件队列功能的超集），又作为主要数据存储（或用数据仓库的术语“记录的系统”）的事实也有许多吸引人的特征，包括：

- 没有任何同步的代价或主数据库和单独专用消息队列（如 RabbitMQ、ActiveMQ 或各种 JMS 实现之一）之间延迟不匹配。
- 简化了操作上的考虑：如果其他条件不变的话，运行一个系统总是比运行两个系统要容易些。



只是因为你能够使用 CouchDB 作为一个过时的消息队列的基础并不意味着你应该那样做；能力越大，责任越大。现成的消息队列非常擅长于它们所做的，而且对许多常见用例处理得非常好。尽管如此，我们的观点是，一个有见识的实践者应该总在寻找能够优雅地解决独特的问题或者更简单地解决典型问题的替代方案。

522

可随意点选的消息队列

让我们组合已经学到的所有关于如何有效地在 Clojure 里使用 CouchDB 来实现一个基于像前面考虑的日志数据库发送事件的异步工作队列。首先，复习一下那样的数据看起来是什么样子的：

例15-4：样例事件数据

```
{:evt-type "auth/new-user" :username "Chas"}  
{:evt-type "auth/new-user" :username "Dave"}  
{:evt-type "sales/purchase" :username "Chas" :products ["widget1"]}   
{:evt-type "sales/purchase" :username "Robin" :products ["widget14"]}   
{:evt-type "sales/RFQ" :username "Robin" :budget 20000}
```

现在你可以在日志数据库里为数据库的变化加一个监视，但不是把这些事件回显到 *out*，而是用它们做一些有用的事；那样做是完全合理的，特别是如果你的需求相对简单时。不过大多数现实世界的系统需要一些额外的手段来控制事件处理负载和对模块化设计需求更合适的支持。

我们可以假设典型的网站或应用将会生成各种各样的事件；有些只需要保留一段时间，有些需要进入长期档案，其他的需要即刻响应。此前，我们在样例数据集里定义了具体事件类型的一个层级，现在来建一个平行的层级，定义不同类型的事件应该如何应对：

例15-5：定义一个事件处理的部分层级

```
(ns eventing.processing)
```

```
(derive 'sales/lead-generation 'processing/realtime)
(derive 'sales/purchase 'processing/realtime)

(derive 'security/all 'processing/archive)
(derive 'finance/all 'processing/archive)
```

523



注意这里的 4 个声明中有 3 个根本就没有提及具体的事件类型，但凭借我们前面定义的具体事件类型和更广泛的事件类型类别（如 '`security/all`'）之间的关系，我们在我们的处理“水平”上的这些层级完全适用于它们。在我们的例子里，这节约了相当一些键入，不过真正的收获在于在多个团队独立构建不同模块或应用而它们都要发出事件数据时：每个团队可以维护各自的事件类型层级（根据需要代表功能、业务或组织关系）而不必与写作这些模块或应用的人协调，这些模块或应用将会在这些事件上协调。

现在可以创建一个 `_changes` 过滤器，这个过滤器让我们选择事件类型 `isa?`^{注 13} 是另一个已经定义的类型的事件：

例15-6：创建一个 message-type _changes 过滤器

```
(clutch/save-filter "logging" "event-filters"
  (clutch/view-server-fns :clojure
    {:event-isa? (do
      (require '[eventing types processing])
      (fn [doc request]
        (let [req-type (-> request :query :type)
              evt-type (:evt-type doc)]
          (and req-type evt-type
            (isa? (symbol evt-type) (symbol req-type))))))))
```

这个过滤器是参数化的（注意，`request` 对象里持有的查询参数的用法），因而我们可以基于定义的层级选择什么事件应该包含在从 `_changes` 发出的流。在进入下一步前，我们来再次回显 `_changes`，但增加了一些东西：

```
(clutch/watch-changes "logging" :echo-leads (partial println "change:")
  :filter "event-filters/event-isa?"
  :type "sales/lead-generation"
  :include_docs true)

(clutch/put-document "logging"
  {:evt-type "sales/RFQ" :username "Lilly" :budget 20000})
;= {_id "8f264da359f887ec3e86c8d34801704b",
;= _rev "1-eb10044985c9dcc731bd5f31d0188c6",
;= :budget 20000, :evt-type "sales/RFQ", :username "Lilly"}
; change: {_seq 26, :id 8f264da359f887ec3e86c8d34801704b,
;           :changes [{:_rev 1-eb10044985c9dcc731bd5f31d0188c6}],
```

注 13：关于 `isa?` 的语义详情，参见第 7 章。

```
; :doc {:_id 8f264da359f887ec3e86c8d34801704b,
;     :_rev 1-eb10044985c9dccb731bd5f31d0188c6,
;     :budget 20000,
;     :evt-type sales/RFQ,
;     :username Lilly})
(clutch/stop-changes "logging" :echo-leads)
:= nil
```

524

- ① 同以前一样，我们将监视日志数据库的变化……
- ② 指定我们有层级知识的过滤器……
- ③ 参数化这个过滤器，使得只会发出对 lead-generation 事件的修改……
- ④ 并要求文档的全部内容都包括在变化通知对象里，而不是默认的仅包括它们的 :_id 和 :_rev 槽。
- ⑤ 当创建一个新的文档时，使用从指定的 lead-generation 类型派生的 :evt-type……
- ⑥ 我们的监视接收到预期的通知，以及相关事件的全部内容。

感谢我们定义的层级，这完全是由过滤器里 (`isa? 'sales/RFQ 'sales/lead-generation`) 返回真的事实驱动的。

可以根据需要设置多个数据库监视，与各种服务或事件优先级对应，或者只是要满足处理的需求（特别是如果涉及的处理是幂等的）。至少从我们应用程序的角度看，每个监视用做一个独立的队列。而且它处理的实现可以由任意数目的可能是领域特有的模块提供，这又是层级使之成为可能的。

让我们来实现一个这样的系统。首先，在某个中心位置定义一个多重方法^{注14}：

```
(ns eventing.processing)

(defmulti process-event :evt-type)
```

然后，根据需要增加 `process-event` 实现，与定义的层级对应。至此，多重方法的所有设施我们都可用。`process-event` 实现只是简单回显一些描述性文本到 `*out*`，当然，这些方法的实际实现会做一些更为实质性的事：对我们的情景而言，它们发出一个发票，把产品直运给客户，向 CRM 系统加一个线索，等等。

例15-7：实现与销售相关的“实时”事件的处理

```
(ns salesorg.event-handling
  (use [eventing.processing :only (process-event)]))
```

注 14：参见第 7 章。

```
(defmethod process-event 'sales/purchase
  [evt]
  (println (format "We made a sale of %s to %s!" (:products evt) (:username evt))))  
  
(defmethod process-event 'sales/lead-generation
  [evt]
  (println "Add prospect to CRM system: " evt))
```

525 ➤

最后，我们可以设置起监视，它会驱动对每个事件的处理，在取出引发这个事件的实际文档后，删除 CouchDB 特有的 :_id 和 :_rev 槽，并把具体的 :evt-type 字符串转变成一个符号以便在 process-event 多重方法里对它的转发是建立的层级驱动的，而不是由 CouchDB 返回给我们的字符串驱动的。这里将在 REPL 里来做这些，然后再次重新创建我们在查看的 5 个事件：

```
(require 'eventing.processing 'salesorg.event-handling)  
  
(clutch/watch-changes "logging" :process-events
  #(> %
    :doc
    (dissoc :_id :_rev)
    (update-in [:evt-type] symbol)
    eventing.processing/process-event)
  :filter "event-filters/event-isa?"
  :type "processing/realtme"
  :include_docs true)  
  
(clutch/bulk-update "logging"
  [{:evt-type "auth/new-user" :username "Chas"}
   {:evt-type "auth/new-user" :username "Dave"}
   {:evt-type "sales/purchase" :username "Chas" :products ["widget1"]}
   {:evt-type "sales/purchase" :username "Robin" :products ["widget14"]}
   {:evt-type "sales/RFQ" :username "Robin" :budget 20000}])
; Add prospect to CRM system: {:evt-type auth/new-user, :username Chas}
; Add prospect to CRM system: {:evt-type auth/new-user, :username Dave}
; We made a sale of ["widget1"] to Chas!
; We made a sale of ["widget14"] to Robin!
; Add prospect to CRM system: {:budget 20000, :evt-type sales/RFQ, :username Robin}
```

最后的思考

Clojure 和 CouchDB 都是非常适合处理异构的、松散结构的数据集——这是动态的、原型驱动的、先开火后瞄准式开发过程的本质特征，对于需要与不可修改的遗留系统很好集成的应用来说这经常也是非常需要的。它们在一起是强有力的组合，在最大化 CouchDB 的视图和过滤器的功用、提供大量设施使你能够充分利用 CouchDB 的特性、模型和可扩展性方面，Clojure 作出了很大的贡献。

Clojure 与 Web

Web 编程很容易被视为无处不在的领域：除了个别例外情况，如果你是今日的程序员，你不仅要知道如何构建 Web 应用程序，而且很可能经常构建 Web 程序、做 Web 编程的工作。既然这样，任何名义上的通用编程语言最好为构建 Web 应用提供不可抗拒的工作流和工具集。Clojure 轻易地满足了这一条件。

运行在 JVM 上并且拥有优秀的互操作特性，Clojure 不需要从基本套接字或 Apache 模块从头开始：经过实践考验的大量 Java Web 基础设施的所有好的部分已经等着被好好重用。同时，对于什么是好的 Web 应用架构，Clojure 生态系统已经演化出了自己的一套惯用语和原则，与典型的 Java 实践对比显著。

Clojure 栈

我们一直在重申什么是好的 Clojure 设计原则：相比具体类型和实现细节更强调共同的抽象，相比带可变状态的有副作用的方法更强调不可变数据的纯函数，把这些纯函数灵活组合、形成自身也是可靠的构建材料。因而，一点也不奇怪并没有权威的“Clojure 栈”，至少没有可比其他语言里经常构成 Web “栈”的单一框架。反之，Clojure 社区这些年里已经发展出许多模块化的函数库，汇总起来可以满足 Web 开发者的所有需求，但利用了 Clojure 的基本抽象和对函数式编程的强调。你和你的团队可以用这些部件来搭建一个适合你和你的应用、领域及个人风格和技能的栈。^{注1}

这一理念对许多有经验的 Rails、Django 或 Lift 开发者来说有些违反直觉。在这些“完整栈”里的特性之所以存在是有原因的：如果不存在，人们倾向于重新实现这些特性。不过，由于这些框架所运行的环境——满是带有显式控制器和视图的面向对象模型——少有共同的抽象，因而有效、高效地组合小而精的模块经常几乎是不可能的。但这在 Clojure

注 1：或者用最近出现的“配备齐全”的新 Web 框架之一开始，我们在本章末尾会提到这些框架。

里是那么容易，堆栈式的框架在 Clojure 中就不像在其他语言中那样节省工作量。简而言之，如果你习惯于更完整的栈，尝试保持开放的心态，直到你完成了一些应用；我们假设你不会需要太多。

我们将分三部分讨论 Web 应用程序，^{注2} 每部分由不同的函数库处理（经常从多个选项选择）：

- 请求和应答处理部分是运行一个 HTTP 服务器或把你的应用程序与某个 HTTP 服务器钩上的部分，构建与传入的请求对应的请求对象，然后生成所需的 HTTP 应答。
- 路由把请求传到你指定的处理代码。
- 模板化是你的处理函数产生的应答被序列化为 HTML（或你要求的任何输出媒体类型）。

我们想要展示的是一组特别受欢迎的函数库组合：

- Ring 用于基础的请求和应答处理。
- Compojure 用于路由。
- Enlive 用于模板化。

其他人会乐意混合使用其他选项来适应特别需求或个人喜好。Moustache^{注3} 是另一个用于路由的好选择。对于模板化，Hiccup^{注4} 也是一个受欢迎的选择，clostache^{注5} 借用了在许多其他语言框架里有的 Mustache^{注6} 模板化风格，甚至直接使用像 JSP、Velocity、stringtemplate 或者来自纯 Java Web 领域的其他函数库。

如果在尝试了我们这里演示的组合后你还想探索某些上面列举的其他选项，比较这些选项的一个不错起点是 http://brehaut.net/blog/2011/ring_introduction。

529

基石：Ring

你在本书处处都可看到，Clojure 擅长数据转换。不过，实际上不需要进行转换时我们最幸运；即如果你的域里存在合适的格式，不要费劲去发明一个新的。秉承这一精神，在 Python 的 WSGI 和 Ruby 的 Rack 启发下，Ring^{注7} 规范用 Clojure 数据结构定义了一个

注 2： 这些或多或少依赖你的需求，处理认证、表单验证、内容协商等。对于所有这些需求都有高质量的 Clojure 函数库，不过要覆盖 Web 编程的方方面面得用单独的一本书。目前我们只是想让你走上一条大道。

注 3： 参见 <https://github.com/cgrand/moustache>。

注 4： 可从 <https://github.com/weavejester/hiccup> 得到。要马上对 Hiccup 有所体味，再看看 481 页“HTML DSL 的成长”一节，作为测试练习我们重新实现了它的一个简单的子集。

注 5： 参见 <https://github.com/fhd/clostache>。

注 6： 参见 <http://mustache.github.com/>。

注 7： 参见 <https://github.com/mmcgrana/ring>。

标准的数据模式来表示 Web 请求和应答，以及基于函数组合的一两个关键架构概念：适配函数、处理函数和中间件。

理解 Ring 的规范^{注8} 对于用 Clojure 有效地构建 Web 应用至关重要。我们要在这里探索它的各个方面，包括在合适时逐字逐句地探讨它的某些部分。^{注9} 我们鼓励你完整地阅读 Ring 的规范至少一次，并且在学习如何使用它所定义的数据和抽象时能随时查阅它。

请求与应答

许多其他框架定义了固定的 API 来访问 Web 请求数据——如像所请求的 URI、请求的头部、查询和提交参数、主体内容等——还定义了其他 API 来发送 Web 应答，但 Ring 把请求和应答都表示为常规的 Clojure 映射。在这两种情况下，这些映射必须包含某些槽，可以包含其他槽，并且可以用来保存处理过程中所需的任何其他数据。

Ring 的请求映射（参见表 16-1）包含这些键（可选的槽用斜体表示）。

表16-1：Ring的请求映射

键	值的描述
<code>:server-port</code>	处理请求的端口
<code>:server-name</code>	解析后服务器名或者是服务器 IP 地址的字符串
<code>:remote-addr</code>	客户端的 IP 地址或者发送请求的最后一个代理
<code>:uri</code>	请求的 URI，字符串类型。必须以 “/” 开头
<code>:scheme</code>	传输协议，必须是 <code>:http</code> 或 <code>:https</code>
<code>:request-method</code>	HTTP 请求方法，必须是 <code>:get</code> 、 <code>:head</code> 、 <code>:options</code> 、 <code>:put</code> 、 <code>:post</code> 或 <code>:delete</code> 之一
<code>:headers</code>	一个从小写的头部名称字符串到对应的头部值字符串的 Clojure 映射
<code>:content-type</code>	请求主体的 MIME 类型，字符串类型的，如果已知
<code>:content-length</code>	请求主体的字节数，如果已知
<code>:character-encoding</code>	请求主体所用的字符编码名，字符串类型的，如果已知
<code>:query-string</code>	请求字符串，如果有
<code>:body</code>	请求主体的 <code>java.io.InputStream</code> ，如果有

◀ 530

这个模式封装了与单个 HTTP 请求相关的基本数据。例如，假设你试图访问这个 URL `https://company.com:8080/accounts?q=Acme`，对应的 Ring 请求映射将会看起来像这样：

```
{:remote-addr "127.0.0.1",
 :scheme :http,
 :request-method :get,
 :query-string "q=Acme",
```

注 8：发表在 <https://github.com/mmcgrana/ring/blob/master/SPEC>。

注 9：Ring 及其规范是以 MIT 方式许可的，Copyright © 2009–2010 Mark McGranahan。

```

:content-type nil,
:uri "/accounts",
:server-name "company.com",
:content-length nil,
:server-port 8080,
:body #<ByteArrayInputStream java.io.ByteArrayInputStream@604fd0e9>, ❶
:headers
{"user-agent" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6) Firefox/8.0.1",
 "accept-charset" "ISO-8859-1,utf-8;q=0.7,*;q=0.7",
 "accept" "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
 "accept-encoding" "gzip, deflate",
 "accept-language" "en-us,en;q=0.5",
 "connection" "keep-alive"}}

```

- ❶ 因为这是一个简单的 GET 请求，:body InputStream 会是空的，因而 :content-type 和 :content-length 都是 nil。

这是一个常规的 Clojure 映射，可以像你已经见过的其他映射那样处理和扩展。

同样，Ring 的应答也是映射（参见表 16-2），只要求有两个槽和一个可选的 :body。

表16-2：Ring的应答映射

键	值的描述
:status	HTTP 状态码，必须大于或等于 100
:headers	从 HTTP 头部名称到头部值的 Clojure 映射。这些值或者是字符串，这时一个名称 / 值的头部会在 HTTP 应答发送回去，或者是字符串的序列，这时每个这样的字符串值都会有一个名称 / 值的头部发送回去
:body	可选是一个字符串，一个 Clojure 序列的字符串，一个 <code>java.io.File</code> 或者一个 <code>java.io.InputStream</code>

也许现在你能够开始看到 Web 请求是如何用 Ring 处理的轮廓：你的代码会接受适合某个请求映射的行为，并返回合适的应答映射。例如前面考虑的 GET 请求可能合理地引发一个 Web 应用的某个 HTML 页面；这个应答看起来可能像这样：

```

{:status 200
:headers {"Content-Type" "text/html"}
:body "<html>...</html>"}

```

或者可以提供其他类型的应答 :body，如果应答对应于磁盘上的静态文件，可以直接提供文件：

```

{:status 200
:headers {"Content-Type" "image/png"}
:body (java.io.File. "/path/to/file.png")}

```

最后，正如我们后面将要看到的，Ring 完全能够使用不要求主体的 HTTP API 调用。如果收到上传文件的 HTTP PUT 请求，一个合适的应答可能是返回一个 201 HTTP 状态码表示请求被接受了、对应的服务器端资源已经创建：

```
{:status 201 :headers {}}
```

这里，你可能会想，如何得到这些表示请求的映射、如何让服务器把 Ring 应答映射到合适的 HTTP 应答。提供这一特别胶合功能的是适配函数。

适配函数

一个 Ring 适配函数在 Ring 应用和 HTTP 协议和 / 或服务器的本地具体实现之间起到桥梁作用。简而言之，当接收到一个 HTTP 请求时，一个适配函数把它解构成一个请求映射，然后把它传给 Ring 应用程序等待处理。这个调用必须返回一个应答映射，适配函数用它向客户端传回一个 HTTP 应答。

你很可能永远不需要写自己的适配函数，不过重要的是要知道适配函数是如何融入整个 Ring 架构的。存在许多适配函数，让 Ring 应用程序可以在不同的 HTTP 服务器和 HTTP API 后面运行：

Servlet. Ring 自身包含一个适配函数，让 Ring 应用程序可以作为 Java servlet 来用，适合于部署到任何 Java Web 应用服务器。这在 560 页“Web 应用程序打包”一节里有更多讨论。

ring-jetty-adapter. 包括在 Ring 里的还有 `ring-jetty-adapter`，这个适配函数使用一个嵌入的 Jetty (<http://jetty.codehaus.org/jetty/>) HTTP 服务器来处理请求。这是最常用的运行 Ring 应用程序的方法，我们很快就会看到它的实际运行。

ring-httpcore-adapter. 这个适配函数 (<https://github.com/mmcgrana/ring-httpcore-adapter>) 和 `ring-jetty-adapter` 非常相似，但是使用一个嵌入的 Apache HTTPCore 服务器而不是 Jetty。

Aleph. 如果你的印象里 Ring 是真的轻量级的，那你是对的。Ring 的核心贡献不是它对什么的具体实现（那是真的很轻），而是它对请求 / 应答数据模式和适配函数、中间件和处理函数关键概念的深思熟虑的定义使得它非常重要。的确，幸亏有这些抽象，Ring——它的标准实现本质上基本是同步的以适应多数 Web 应用程序的同步性——自身可以被其他与 Ring 兼容的实现所替换。一个引人注目的例子是 Aleph,^{注 10} 它提供与 Ring 兼容的适配函数，使用 Netty^{注 11} 对客户端提供异步应答服务，不需要对你的 Ring

< 532

注 10：参见 <https://github.com/ztellman/aleph>。

注 11：参见 <http://www.jboss.org/netty>。

应用程序做任何改变。

也有人写了其他的适配函数作为 Ring 应用程序和诸如 Mongrel 和 FastCGI 功能的服务器间的桥梁。

现在我们已经准备好看一个 Ring 应用程序里“真正”做事的部分——处理函数，看如何把它与我们所选的适配函数弄到一起来搭建 Web 应用程序。

处理函数

一个 Ring 处理函数不过是接受一个请求映射为参数并返回一个应答映射的函数。所有 Ring 应用程序由一组处理函数组成，这些函数被链接起来、组合起来并根据需要被委托以支持所需的行为和功能。

让我们以一个简单的回声服务器开始吧。首先给项目加上对 Ring 的依赖：^{注 12}

```
[ring "1.0.0"]
```

现在可以打开一个 REPL 写一个 Web 应用：

例16-1：从REPL里启动一个Ring应用程序

```
(use '[ring.adapter.jetty :only (run-jetty)])
;= nil
(defn app
  [{:keys [uri]}]
  {:body (format "You requested %s" uri)})
;= #'user/app
(def server (run-jetty #'app {:port 8080 :join? false}))
;= #'user/server
```

❶

❷

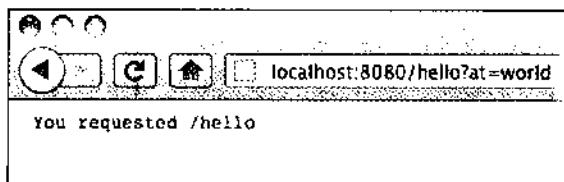
❸

533

- ❶ 我们的处理函数。所有处理函数接受一个 Ring 请求映射为唯一参数，并返回一个应答映射。作为开始，我们只是简单地以纯文本方式回显请求的 URI。
- ❷ 我们使用 Jetty 适配函数。所有适配函数都是实现为带两个参数的函数：服务请求的 Ring 处理函数和为适配函数提供选项的映射。这里我们请求 Jetty 运行在 8080 端口上，而且不应“并入”Jetty 将使用的线程；如果不指定这个将会导致我们的 REPL 因等待 Jetty 服务器关闭而阻塞。
- ❸ 我们决定在 `server var` 里保留指向 Jetty 服务器的一个引用。这让我们有可能（如果那样做的话）通过调用 (`.stop server`) 来停止 Jetty 服务器。

注 12：请不要害怕这里很多 “1.0.0” 版本号。提到的所有项目已经广为使用多年，“1.0.0” 版本是最近一下子指定了一批，部分是为了认可这些项目的稳定性。

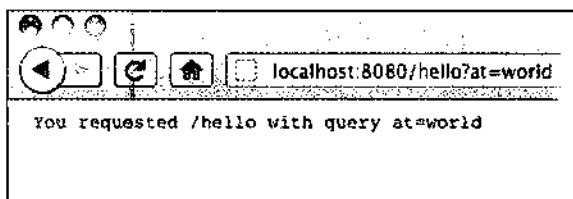
现在我们可以访问正在运行的 Ring Web 应用了：



这很好，不过注意，URI 里没有包含我们的查询参数 `?at=world`。因为我们将处理函数 `var`（即 `#'app`）启动 Jetty 适配函数的，而不是传递处理函数自身，可以轻易地重定义这个处理函数，不需重启 Jetty：

```
(defn app
  [{:keys [uri query-string]}]
  {:body (format "You requested %s with query %s" uri query-string)})
:= #'user/app
```

结果立即可用：



当然可以做得比这更好。我们不得不从一个字符串选取出查询参数是相当糟糕的。不过 Ring 关于请求映射的规范没有提到查询和表单参数可以有其他形式出现。

幸好我们还远没有陷入困境。有一个特别的 Ring 中间件可以处理这一常见需求。

中间件

534

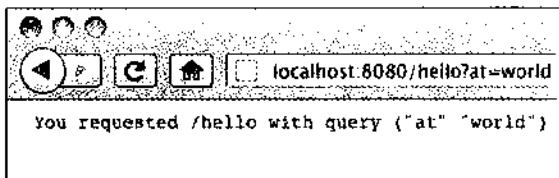
中间件是对处理函数的效果的任何增强或修改。记住，因为 Ring 的请求和应答都是 Clojure 映射，可以轻易地转换，而且由于处理函数只是函数，产生不同函数的组合以获得总体行为也是毫不费事的。这典型地表现在中间件作为一个高阶函数接受一个或多个处理函数（也许要一些配置），返回一个新的带有所需组合功能的处理函数。

说得再具体一点。如我们刚才所见，Ring 的请求默认不包含查询参数的任何结构化表示。通过加入一点中间件到我们的应用程序可以改变这一点；正是中间件会装饰从 Jetty 适配函数接收到的最初的请求映射，因而把我们的查询参数被有用地分解到一个映射里：

```
(use '[ring.middleware.params :only (wrap-params)])
;; nil
(defn app*
  [{:keys [uri params]}]
  {:body (format "You requested %s with query %s" uri params)})
:= #'user/app*
(def app (wrap-params app*))           ;;
:= #'user/app
```

- ❶ 现在我们的处理函数定义为 `app*`，以便用中间件增强的处理函数可以放在 `app var` 里，这是 Jetty 适配函数保留的 `var`。
- ❷ 不是在请求映射里查找 `:query-string`，我们的处理函数现在预期有一个 `:params` 的值。
- ❸ 把我们的处理函数和中间件组合起来是由一个简单的高阶函数调用完成的。一个新的处理函数由 `wrap-params` 返回，这形成了应用程序最外面的一层——在完成了解析请求的查询字符串里的所有参数和 POST 主体（如果有的话）之后，将会调用我们与请求映射一起提供的处理函数，新加入了 `:params` 槽里的所有参数。

我们来看一下：



非常棒，我们现在有了参数，应用程序行为可以基于这些参数。

Ring 包含了许多不同的中间件，你可以根据需要在自己的 Web 应用中混合使用，从对 535 请求头部解析 cookie 和会话数据，到对静态文件请求走捷径，到支持多部分表单提交和文件上传，等等。^{注13} 虽然 Ring 默认并不对处理函数应用任何一个中间件，但其他一些基于 Ring 的 Web 框架却是这样做的。

最后，由于中间件真的只是一种函数组合的方式，因此创建新的中间件异乎寻常的容易。一个简单的中间件的例子包含在 465 页“Ring”一节里。由于实现的容易与灵活的程度，你会发现许多对 Ring 的扩展是以中间件的方式实现的。

^{注 13} 关于所包含的中间件完整列表，见 Ring 的 GitHub 主页：<https://github.com/mmcgrana/ring>。



从概念上说，Ring 的中间件与 Java 的 servlet 过滤器相似：两种方法都允许事后对 Web 请求和应答进行修改。另一方面，中间件的实现和使用十分容易——就是定义然后调用一个高阶函数——相比之下，实现与 servlet 过滤器相关的各种接口并根据需要在部署时配置这些过滤器如同迷宫历险。更重要的是，servlet 过滤器经常不能有效地组合，因为自定义的 `ServletRequest` 和 `ServletResponse` 类型存在冲突，而且如果一个 servlet 碰巧命令式地在 `ServletResponse` 上发送出内容，一个过滤器完全无力阻止。这正是中间件的亮点：受益于 Ring 的模型，请求和应答是不可变的集合类，总是适合于同一个共同的抽象，中间件总是可以有效地组合起来……毕竟它们都只是函数！

用 Compojure 路由请求

到目前为止，我们只定义了单个函数来处理所有请求，除了最简单的应用程序，这样做是不行的。^{注14} 我们想能够自然地组织应用程序，把逻辑上不同的功能分割成可能在不同命名空间的不同的 Ring 处理函数，用正确的组合方式把它们放到一起。可以试着把请求 URI 字符串分别开来以便把请求处理委托给其他函数；不过，正如我们用某些中间件与 Ring 处理函数组合起来以扩展应用程序，有更好的办法实现我们的目的。

非常简单，路由（routing，动词）是选择一个应该用于应答一个 Web 请求的处理函数，而路由（routes，名词）是传入的请求属性的模式，用于驱动这个选择过程。抽象地说，可以容易想象用一个在各种命名空间里定义的具体处理函数对应的路由表来定义 Web 应用程序（见图 16-1）。

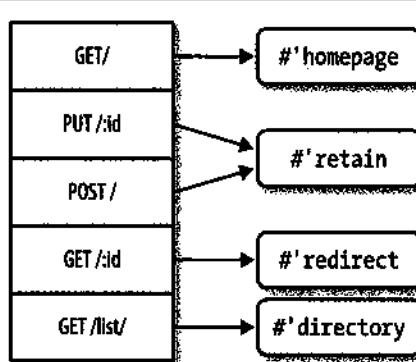


图16-1：处理函数与路由配对

注 14：当然，如果你写的是一个非常小的、有限制的 HTTP 服务，在 Jetty 里运行的单个 Ring 处理函数不需要进一步的装饰可能是完全合理的。

536 当接收到对应用程序的根(/)的一个 GET 请求时，这个请求会被路由到 `homepage` 处理函数。当接收一个对任何带单个节段（在图中用 `:id` 表示）的 URI 的 PUT 请求时，这个请求会被路由到 `retain` 处理函数；对应用程序根的任何 POST 请求也是同样处理的，等等。

我们来建一个路由与图 16-1 中的图形完全符合的 Web 应用程序。结果会是简单的简缩 URL 的 HTTP 服务，^{注 15} 与 bit.ly、tinyurl.com 等类似，这样的服务让用户指定并使用会重定向到标准 URL 的简缩的 URL。要做到这一点，我们将使用 Compojure (<https://github.com/weavejester/compojure>)，为 Ring 应用程序定义路由的最流行的函数库，模仿了你可能已经熟悉的许多框架里的 URL 路由特性，包括 Ruby on Rails 和 Django。

让我们启动一个新 REPL 会话，Compojure 包括在项目的依赖里，因而现在项目依赖包括 Compojure 和 Ring：

```
[compojure "1.0.1"]
[ring "1.0.1"]
```

我们应首先考虑储存 URL 和它们的简缩标识应该使用什么模型。我们想集中创建 Web 方面的东西，因而就把这个状态用引用类型保存在内存的一个映射里。^{注 16}

537 我们应该用哪种引用类型呢？我们会允许用户请求指派简缩的标识，因而需要防止某个标识已经被用时出现冲突，还需要协调对映射的修改，这样并发的请求不会用相同的键添加不同的条目。这要求把主映射保存在一个 ref 里：对映射的每次修改将会按事务处理，因而我们可以安全地避免影响已经注册的标识，并发的冲突请求（例如，如果两个客户端同时想用相同的简缩标识注册 URL）在我们的模型里永远会产生不一致的情形。

另一方面，在用户不提供他们自己的标识时，我们不需要为所使用的自动生成的标识使用那样的严格保证。我们可以用某种形式的哈希或随机标识，但最容易的办法是使用增量计数器。原子是维护这样一个计数器的完美候选方案。^{注 17}

这样我们有了自己的驻留在内存的模型、在原子里的计数器和在 ref 里的映射：

```
(def ^:private counter (atom 0))

(def ^:private mappings (ref {}))
```

我们应该有一两个专用函数来处理状态。它们不仅使得在 Web 环境之外测试核心功能轻

注 15： 你可能称它为一个 REST 服务，不过 REST 的语义很难弄正确，正如 Roy Fielding (REST 这个术语的首创者) 在 <http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven> 所指出的。我们可以欣慰地说，这个应用是一个表现很好的 HTTP 服务。

注 16： 如果要正确地把这个服务构建成真实使用的服务，可以轻易地使用在第 14 章或第 15 章讨论的数据之一来取代只在内存的状态。

注 17： 因为修改 ref 的映射的状态可能由于请求竞争导致重试，从计数器取得的某些自动生成的标识会丢失。这看来是合理的，但知道有这么回事是值得的。

而易举，如果要转而使用一个数据库后台，唯一的工作就是重新实现这两个函数：

```
(defn url-for
  [id]
  (@mappings id))

(defn shorten!
  "Stores the given URL under a new unique identifier, or the given identifier
  if provided. Returns the identifier as a string.
  Modifies the global mapping accordingly."
  ([url]
   (let [id (swap! counter inc)          ❶
         id (Long/toString id 36)]        ❷
     (or (shorten! url id)              ❸
         (recur url))))
  ([url id]
   (dosync
     (when-not (@mappings id)          ❹
       (alter mappings assoc id url)
       id))))
```

- ❶ swap! 会返回在 counter 原子里的新值，保证在 shorten! 调用里是唯一的。
- ❷ 这个 Long/toString 调用把数值 ID 转换成基为 36 的字符串，这比基为 10 的 ID str 要紧凑得多。538
- ❸ 我们利用接受两个参数的 shorten! 来尝试储存自动生成的 ID \Rightarrow URL 的映射。如果失败（由于 ID 已经存在于 mappings 里），就简单地 recur 来重试。
- ❹ 如果要加入的 ID 还不存在，就把它与映射里的标准 URL assoc 起来，并返回简缩的标识，否则返回 nil。

我们来看看这个模型工作得怎样：

```
(shorten! "http://closurebook.com")
;= "1"
(shorten! "http://closure.org" "clj")
;= "clj"
(shorten! "http://id-already-exists.com" "clj")
;= nil
@mappings
;= {"clj" "http://closure.org", "1" "http://closurebook.com"}
```

目前看还不错。现在我们来创建一两个函数，处理预期的 HTTP 请求。retain 是 shorten! 的 Web “前端”，它把与我们的映射的实际交互委托给 shorten!，并基于一个 URL（及其可选提供的 ID）是否可以保存来返回合适的应答映射：

```
(defn retain
  [& [url id :as args]]
  (if-let [id (apply shorten! args)]
      {:status 201
       :headers {"Location" id}
       :body (list "URL " url " assigned the short identifier " id)}
      {:status 409 :body (format "Short URL %s is already taken" id)}))
```

- ❶ 如果用某个 ID 成功地保存一个 URL，那么 `retain` 会返回一个 Ring 应答，`:status` 设为 201（这个 HTTP 应答代码表示资源已经成功创建），而且 `Location` 头部包含储存的 ID（这对非浏览器的 HTTP 客户端可能有用），而文本的 `:body` 适合人来阅读。
- ❷ 记得在表 16-2 里，Ring 的应答主体可以是一些不同的类型，包括字符串序列。
- ❸ 不能用简缩标识储存一个 URL 的唯一原因是用户请求的 ID 已经被用过了。这种情况下，我们返回一个 409 的 `:status`（这个 HTTP 应答码表示“冲突”阻止了所请求的行为发生）。

`redirect` 这个函数将查询所请求的 ID，如果这个 ID 已知并对应于一个 URL 就以一个浏览器重定向回应：

```
(require 'ring.util.response)

[539] (defn redirect
  [id]
  (if-let [url (url-for id)]
    (ring.util.response/redirect url)
    {:status 404 :body (str "No such short URL: " id)}))
```

如果给出的 `id` 在我们的 `mappings` 里找到，那么 `redirect` 使用 Ring 的 `ring.util.response` 命名空间^{注 18}里的一个工具函数来发送一个合适的重定向应答（对应于 HTTP 的 `:status` 值为 302）。

现在，`retain` 和 `redirect` 在技术上还不是 Ring 处理函数：它们返回 Ring 应答，因而是有用的辅助，但它们不接受请求映射。我们已经限制了这些函数的范围，虽然这是不需要的，但可以使它们更简单、更简洁、更容易测试。

最后可以按图 16-1 所描绘的顺序定义我们的 Compojure 路由：^{注 19}

注 18： `ring.util.response` 有许多非常有用的工具用于创建和修改应答映射。

注 19： 尽管这是不合逻辑的推论，如果不建议你深入理解 Tim Berners-Lee 发的牢骚“真正酷的 URI 不会变”则是我们的失职，请参见 <http://www.w3.org/Provider/Style/URI.html>。

例16-2：一个URL简缩服务的Compojure路由

```
(use '[compojure.core :only (GET PUT POST defroutes)])
(require 'compojure.route)

(defroutes app*
  (GET "/" request "Welcome!")
  (PUT "/:id" [id url] (retain url id))
  (POST "/" [url] (retain url))
  (GET "/:id" [id] (redirect id))
  (GET "/list/" [] (interpose "\n" (keys @mappings)))
  (compojure.route/not-found "Sorry, there's nothing here."))
```

`defroutes` 定义一个含有单个 Ring 处理函数的 var (这里是 `app*`)，试图把传入的请求按顺序分发给函数体里提供的每个 Ring 处理函数。第一个处理函数返回一个非 `nil` 的值，提前结束分发过程，那个返回值用做 `defroutes` 定义的“顶层” Ring 处理函数的返回值。

Compojure 的路由定义一个 Ring 处理函数和用于匹配这个处理函数的请求的模式。每个路由包括：

- 处理函数应该调用的 HTTP 方法。这些方法是由 Compojure 提供的宏，与常见的 HTTP 动词像 `GET` 和 `POST` 对应。^{注 20}
- 一个 URI 模式，描述这个处理函数应该调用的 `:uri`。
- 对请求和 / 或参数以及 URI 的匹配部分的绑定形式；这个形式里的名称会成为处理函数体的局部量。
- 处理函数体，可以包含任何 Clojure 代码，应该返回 `nil`（表示请求未被处理、应该继续分发给后面的路由）或可以在 Ring 应答映射里利用的某个其他值。

540

我们来看看其中一个路由：

```
(PUT "/:id" [id url] (retain url id))
```

这个路由将只与 `:request-method` 为 `:put`（与 HTTP 协议中的 `PUT` 对应）的请求匹配，所请求的 URI 为单个节段；例如 `"/some-id"` 能够匹配，但 `"/path/to/some-id"` 不能。URI 匹配串的 `:id` 部分将使 URI 的那一部分与局部量 `id` 绑定，这是由 `[id url]` 绑定向量确定的。局部量 `url` 在匹配字符串中没有提及，它与 `:url` 参数绑定（如果请求中没有提供这样的参数则为 `nil`）。绑定形式后面其余的 Clojure 代码定义了行为和返回值。

注 20：Compojure 自带的其他请求方法宏包括 `PUT`、`DELETE`、`HEAD` 和一个特别的宏 `ANY`，它可以用做通配方法。如有必要，路由宏根据需要可以定义为“非标准的”HTTP 动词，如 `COPY`。

因为 Compojure 的路由求值为 Ring 处理函数，我们在 REPL 里可以轻易感受到路由是如何工作的，试验 Compojure 的路由提供的所有选项。^{注21} 例如这是我们在 URL 简缩程序里所用的同样的路由（虽然这里所用的处理函数体不同）；我们可以清楚地看到参数和具名 URI 路径节段是与绑定向量里同名的符号绑定的：

```
((PUT "/:id"
      [id url]
      (list "You requested that " url " be assigned id " id))
  {:uri "/some-id" :params {:url "http://closurebook.com"} :request-method :put})
:= {:status 200, :headers {"Content-Type" "text/html"},  
:= :body ("You requested that " "http://closurebook.com" " be assigned id "  
"some-id"))}
```



注意，Compojure 路由产生的处理函数很有帮助地为应答映射的 :status 和 :headers 槽提供了典型的默认值。如果你知道生成的是没有错误的 HTML 页面，这让你可以简单地返回应答的主体。

路径节段由 :keyword 标识出来，会与除 /、..、.、; 和 ? 之外的任何字符匹配；一个星号的路径节段会匹配到下一个前斜杠的所有东西。你可以在 URI 模式中提供任意多的路径节段，把它们的任意集合绑定为你的处理函数体里的局部量。^{注22} 如果你的多个路径节段名称相同，每个匹配节段的向量将与具名的局部量绑定：

```
((PUT ["//*/:id/:id"]
      [* id]
      (str * id))
  {:uri "/abc/xyz/foo/bar" :request-method :put})
:= {:status 200, :headers {"Content-Type" "text/html"},  
:= :body ["\"abc\" \"xyz\""]["\"foo\" \"bar\"]")}
```

一个非常有用的特性是可以提供正则表达式来定义路径节段的性质和范围。这与 Rails 里基于正则表达式的 :constraints 非常相似，通过把 URI 模式包装在一个向量里，提供路径节段名和正则表达式的键 / 值来实现。例如，我们可以用 #"\d+" 正则表达式指定 :id 路径节段只含有数字：

```
((PUT ["/:id" :id #"\d+"]
      [id url]
      (list "You requested that " url " be assigned id " id))
  {:uri "/some-id" :params {:url "http://closurebook.com"} :request-method :put}) ❶
:= nil
((PUT ["/:id" :id #"\d+"]
```

注 21：Ring 请求相当容易建构，我们这里是直接做的。如果你发现自己经常“模拟”Ring 的请求（为了测试、试验或其他目的），ring-mock 可能会有所帮助：<https://github.com/weavejester/ring-mock>。

注 22：或者你可以在一个 :route-params 映射里访问路径节段，那是 Compojure 把它加到请求里去的。

```
[id url]
  (list "You requested that \" url \" be assigned id \" id\"")
  {:uri "/590" :params {:url "http://closurebook.com"} :request-method :put} ) *
:= {:status 200, :headers {"Content-Type" "text/html"}, 
:= :body "You requested that http://closurebook.com be assigned id 590"}
```

- ❶ 请求与定义的路径不匹配，因为 :uri 路径节段 `some-id` 不是数字的，因而路由返回 `nil`（因而表明请求转发应该继续到下一个路由和处理函数）。
- ❷ :uri 路径节段与数字开始的请求能够匹配，因而处理函数体被调用，产生一个非 `nil` 的应答。

最后，你总可以提供一个符号或映射解构形式作为路由里的绑定形式；那将把整个请求映射与处理函数体绑定：

```
((PUT "/:id" req (str "You requested: " (:uri req)))
  {:uri "/foo" :request-method :put})
:= {:status 200, :headers {"Content-Type" "text/html"}, :body "You requested: /foo"})
((PUT "/:id" {:keys [uri]} (str "You requested: " uri))
  {:uri "/foo" :request-method :put})
:= {:status 200, :headers {"Content-Type" "text/html"}, :body "You requested: /foo"}
```

例 16-2 中的最后路由可以匹配任何东西，让我们控制如果在 `defroutes` 形式里定义的路由没有一个可以匹配请求时该怎么做并返回非 `nil` 的应答。这会返回一个 404 的 HTTP 状态码，主体由我们供给 `compojure.route/not-found` 辅助函数的处理函数体定义，在这里的情形，一个简单的字符串就行了。

542

我们的应用程序已经完成了，只不过还没有运行！不过那只是简单的簿记之事。首先，因为我们依赖请求里的参数被解析并加到 Ring 请求里，需要加入某些中间件。`Compojure` 在 `compojure.handler` 命名空间里方便地提供了两个做这事的辅助函数：`:api` 用 Ring 提供的一两个中间件包装一个 Ring 处理函数，负责处理可能需要的所有参数处理，最适合用于 HTTP 服务；`:site` 提供同样的功能，但加上其他中间件来支持一个面向用户 / 浏览器的网站所要有的功能，像 cookie 处理、会话、多部分表单提交与上传，等等。我们选择前者：

```
(require 'compojure.handler)

(def app (compojure.handler/api app*))
```

现在只需用 Jetty 适配器启动应用程序：

```
(use '[ring.adapter.jetty :only (run-jetty)])
:= nil
(def server (run-jetty #'app {:port 8080 :join? false}))
:= #'user/server
```

所有这些将可以用任何浏览器轻易访问，但形式足够好因而我们可以用任何 HTTP 函数库通过编程使用这个服务。为了正确地测试这个 URL 简缩服务，需要向这个服务发送 PUT 或 POST 请求，表单参数设置正确。这可能有点难办，因为浏览器需要一个表单才能发送这两种请求。不过 curl 是一个广泛使用的、容易获得的命令行工具，它使得相对容易端对端地测试我们的服务。curl 让你用 -X 参数来设置 HTTP 方法，如果用 -i 参数来执行会输出来自服务器的全部应答。

我们先用 PUT 来确立一两个简缩 URL ID：

```
% curl -X PUT 'http://localhost:8080/sicp?url=http://mitpress.mit.edu/sicp/'  
URL http://mitpress.mit.edu/sicp/ assigned the short identifier sicp  
  
% curl -X PUT 'http://localhost:8080/clj?url=http://clojure.org'  
URL http://clojure.org assigned the short identifier clj
```

curl 会输出它从我们的服务接收到的头部，如果包含了 -i 选项；如果包含这个选项同时向这个服务 POST 以获得一个自动生成的 ID，就可以看到服务正确地返回了 201 Created HTTP 状态码以及 Location 头部：

```
% curl -i -X POST 'http://localhost:8080/?url=http://clojurebook.com'  
HTTP/1.1 201 Created  
Date: Sun, 18 Dec 2011 20:58:09 GMT  
Location: 1  
Content-Length: 58  
Server: Jetty(6.1.25)
```

543 ➤ URL http://clojurebook.com assigned the short identifier 1

现在如果尝试注册一个已经被使用了的简缩 URL ID，服务会正确地以失败来回应，一个代理程序可以用这个 409 Conflict HTTP 状态为用户提供有帮助的信息：

```
% curl -i -X PUT 'http://localhost:8080/1?url=http://apple.com'  
HTTP/1.1 409 Conflict  
Date: Sun, 18 Dec 2011 20:58:40 GMT  
Content-Length: 28  
Server: Jetty(6.1.25)  
  
Short URL 1 is already taken
```

注册了一些 URL 之后，我们可以使用 /list/ 路由来获得已知简缩标识的列表：

```
% curl http://localhost:8080/list/  
1  
clj  
sicp
```

请求任何没有对应的路由（或一个只有单节段的 URI 但不与任何简缩 URL ID 对应）的 URI 正确地产生一个 404 Not Found 应答和信息：

```
% curl -i http://localhost:8080/foo
HTTP/1.1 404 Not Found
Date: Sun, 18 Dec 2011 21:21:39 GMT
Content-Length: 22
Server: Jetty(6.1.25)
```

No such short URL: foo

而请求其他任何非简缩 URL ID 路径会返回通用的 404 应答：

```
% curl -i http://localhost:8080/some/other/url
HTTP/1.1 404 Not Found
Date: Sun, 18 Dec 2011 21:21:53 GMT
Content-Length: 28
Server: Jetty(6.1.25)
```

Sorry, there's nothing here.

最后，请求一个与某个简缩 URL ID 对应的 URI，可以看到重定向工作正确：

```
% curl -i http://localhost:8080/sicp
HTTP/1.1 302 Found
Date: Sun, 18 Dec 2011 20:59:12 GMT
Location: http://mitpress.mit.edu/sicp/
Content-Length: 0
Server: Jetty(6.1.25)
```

```
% curl -L http://localhost:8080/sicp
<HTML><HEAD><TITLE>Welcome to the SICP Web Site</TITLE></HEAD>
```

544

① -L 选项告诉 curl 跟随 302 Found 状态码指示的 HTTP 重定向和 Location 头部。

这个 URL 简缩服务的最终代码在下面重列出来，如同出现在一个源文件里。

例16-3：函数式的URL缩减程序

```
(ns com.clojurebook.url-shortener
  (:use [compojure.core :only (GET PUT POST defroutes)])
  (:require [compojure.handler route]
            [ring.util.response :as response]))

(def ^:private counter (atom 0))

(def ^:private mappings (ref {}))
```

```

(defn url-for
  [id]
  (@mappings id))

(defn shorten!
  "Stores the given URL under a new unique identifier, or the given identifier
  if provided. Returns the identifier as a string.
  Modifies the global mapping accordingly."
  ([url]
   (let [id (swap! counter inc)
         id (Long/toString id 36)]
     (or (shorten! url id)
         (recur url))))
  ([url id]
   (dosync
     (when-not (@mappings id)
       (alter mappings assoc id url)
       id)))))

(defn retain
  [& [url id :as args]]
  (if-let [id (apply shorten! args)]
    {:status 201
     :headers {"Location" id}
     :body (list "URL " url " assigned the short identifier " id)}
    {:status 409 :body (format "Short URL %s is already taken" id)}))

(defn redirect
  [id]
  (if-let [url (url-for id)]
    (response/redirect url)
    {:status 404 :body (str "No such short URL: " id)}))

(defroutes app*
  (GET "/" request "Welcome!")
  (PUT "/:id" [id url] (retain url id))
  (POST "/" [url] (if (empty? url)
                      {:status 400 :body "No `url` parameter provided"}
                      (retain url)))
  (GET "/:id" [id] (redirect id))
  (GET "/list/" [] (interpose "\n" (keys @mappings)))
  (compojure.route/not-found "Sorry, there's nothing here."))

(def app (compojure.handler/api app*))

;; ; To run locally:
;; (use '[ring.adapter.jetty :only (run-jetty)])
;; (def server (run-jetty #'app {:port 8080 :join? false}))
```

组合路由。因为 `defroutes` 分发给任何 Ring 处理函数，而 `defroutes` 自身也创建一个处理函数，你可以不费力地组合出分层级的路由。如果想要给这个 URL 简缩服务加一个管理控制台，可以毫不费力地混合进已有的 `app*` 路由：

```
(defroutes app+admin
  (GET "/admin/" request ...)
  (POST "/admin/some-admin-action" request ...)
  app*)
```

唯一需要记住的是，在把路由组合成更高层次的分组时不能包括“匹配任何东西”的路由（即由像 `compojure.route/not-found` 产生的那些路由），否则在上面的 `app*` 之后出现的路由将永远不被转发到，它们的处理函数也不会被调用。

使用模板

目前我们已经涉及一个 Web 框架应该做的所有事情，只差实际生成复杂 HTML 了。虽然 HTTP 服务和文件服务器是非常有用的，但是大多数人在听到“Web 应用”时想到的是“HTML”。大多数 Web 框架使用模板系统，HTML 与可执行代码或未求值的“指令”（引用在页面生成的范围内定义的约束变量）混合在一起，结果是填充了内容的 HTML 页面。

这种方法的一个例子是 Ruby 的 ERB 模板语言：

例16-4：一个ERB HTML模板的例子

```
<h1>Hello, <%= @user.name %></h1>

<p>These are your friends:</p>
<ul>
<% @user.friends.each do |friend| %>
  <li><%= friend.name %></li>
<% end %>
</ul>
```

那些不熟悉 ERB 语法的人可能会看不明白，不过这与 Django 或 JSP 或上百个你可能熟悉的模板系统中的模板并没有多大不同。所有这些系统基本上就是通过字符串替换工作的，就 ERB 而言，执行 `<%` 和 `%>` 之间的 Ruby 代码并对 `<%=` 和 `%>` 之间的表达式求值以获得应该包含在输出中的字符串。ERB 模板在一个环境里运行，因而像 `@user.name` 这样的表达式指向一个在这个环境里的局部量，获得它的 `name` 属性。

546

这一方法已经被无数语言和成百上千万的程序员使用了多年，这甚至是 PHP 的首要操作模式。不过并非所有方面都是好的。第一个明显的问题是这些模板里的 HTML 难以调试和优化；在一个可用的 Web 栈之前，我们不能确定模板 HTML 能否工作，也不能调整它的样式表。代码也有同样的问题，因为不模仿出一个含有合适的域对象等的并不简

单的环境是难以单独测试代码的。更准确地说，模型和视图之间的耦合过于紧密。最后，模板的理想开发者得既精通 HTML 又精通 Ruby(或 Python 或 PHP)，由软件栈驱动模板。

尽管模板方法大受欢迎，但既是专家级 Web 设计师又是专家级软件工程师的人极为稀少。经常委托一个设计师来创建 HTML 模板，然后由开发者手工修改模板，添加开发者所首选的模板语言的标记。如果有什么设计变化，这些变化经常是对原 HTML 文件的更新，开发者必须重新检查一遍，小心地合并进他们的模板版本。显然这不是最佳情景。我们能做得更好吗？

Enlive：基于选择器的 HTML 转换

Enlive (<http://github.com/cgrand/enlive>) 提出一个激进的办法来解耦代码和模板：不是定义一个特别的局部语法来把值插入模板，Enlive 模板是普通的 HTML 文件，没有特别标签，没有特别属性，没有特别类，也没有特别语法。反而是内容被注入到模板里，使用选择器（深受 CSS 的选择器的启发）的 Clojure 代码指定要修改什么，Clojure 函数定义要应用什么转换。^{注 23}

严格地关注分离——设计与转换它的代码保持完全分离——使得程序员和设计师间的反复合作特别容易。而且因为用于标识要被修改的模板部分的选择器很可能与用于对生成的内容提供样式风格的某些 CSS 选择器是一样的，设计师（或者参与设计的程序员）容易提前识别出模板中哪些改变可能对转换这一设计的代码产生影响。

547

试探

使用 Enlive 的第一步通常是写出 HTML 文件——称之为模板源文件，不过我们先在 REPL 里摆弄摆弄感受一下 Enlive 是如何工作的。把对它的依赖加到项目里：

```
[enlive/enlive "1.0.0"]
```

并启动一个 REPL。

```
(require '[net.cgrand.enlive-html :as h])
 ;;= nil
 (h/snippet "<h1>Lorem Ipsum</h1>")
 ;;= "<h1>Lorem Ipsum</h1>"
```

snippet 是 Enlive 提供的一个工具，可以使在 REPL 里试验和转换 HTML 片段更简单。这里没有指定一个转换，因而我们的输入没有改变就返回了。

注 23： 模板只是转换 HTML 或 XML 数据的特殊情况：Enlive 可以使用用于模板的同样选择器来从 HTML 和 XML 文档中提取内容。不过我们在这里只讨论 Enlive 用于模板的情况。

```
(h/snippet "⟨h1>Lorem Ipsum</h1⟩"
  [:h1] (h/content "Hello Reader!"))
:= "<h1>Hello Reader!</h1>"
```

`[:h1]` 是一个选择器——对应于 CSS 的选择器 `h1`——与被转换的 HTML 里的所有 `h1` 元素匹配。`content` 是一个高阶函数，返回一个函数把匹配的元素体设置成提供给 `content` 的值。Enlive 提供了大量现成的转换器高阶函数，可以满足在一个典型的 Web 应用里所有的常见模板需求，它的选择器提供了 CSS 选择器的超集，用于指定这些转换应该用在哪儿。

让我们稍微揭开盖子看看，以便理解更复杂的转换是怎么形成的。Enlive 有一个 `html-snippet` 函数，用于解析任何 HTML 内容，返回映射的序列，每个映射代表一个元素及其属性和子内容：

```
(h/html-snippet "<p>x, <a id=\"home\" href=\"/\">y</a>, <a href=\"..\">z</a></p>")
:= ({:tag :p,
:=   :attrs nil,
:=   :content
:=   ("x, "
:=   {:tag :a, :attrs {:href "/", :id "home"}, :content ("y")})
:=   ","
:=   {:tag :a, :attrs {:href ".."}, :content ("z")}}))
```

对 HTML/XML 的这一表示与 `clojure.xml` 命名空间里的函数产生和接受的表示法是匹配的——这是选择一个共同的抽象使得数据交换更简单、功能易于组合的又一例子。

知道了这一点，就不难想象 Enlive 是如何实现转换的：

548

1. 选择器遍历所涉及的 HTML 的树形表示以找到匹配的元素。
2. 这些元素被传递给转换函数，并配上每个要应用的选择器。函数的结果替代所选择的元素。

这些操作是在 Clojure 的持续数据结构上用纯函数完成的，因而可以根据需求任意链接起来，组合起来，重复利用：

例16-5：一个不太简单的HTML转换

```
(h/snippet "<p>x, <a id=\"home\" href=\"/\">y</a>, <a href=\"..\">z</a></p>"
  [:a#home] (h/set-attr :href "http://closurebook.com")
  [[{:a (h/attr= :href "..")}] (h/content "go up"))
:= "<p>x, <a href=\"http://closurebook.com\" id=\"home\">y</a>, <a href=\"..\">go up</a>
  </p>"
```

要充分利用 Enlive，要求理解如何构造合适的选择器和转换函数。

选择器

Enlive 选择器的语法初看起来有些吓人，不过如果你知道一些 CSS 其实是非常容易学会的。多数时候，在每一步前加上冒号，把所有东西包装到一个向量里就足够了。例如 CSS 选择器 `div span.phone` 变成了 `[:div :span.phone]`，`#summary .kw` 变成了 `[:#summary :.kw]`，等等。

Enlive 初学者经常为选择器里嵌套的向量含义而挣扎。规则很容易：最外层向量表示链接，所有其他向量表示合取。因而 `[:div [:span :.phone]]` 与上面的 `[:div :span.phone]` 等价。合取的向量可以即兴嵌套：`[:div [:span [:phone :.mobile]]]` 与 `[:div :span.phone.mobile]` 是一样的。

注意，最外层的向量不是可选的，即使选择器只有一步：`:h1` 不是一个有效的选择器，`[:h1]` 才是。

Enlive 也支持析取。与 CSS 选择器 `div#info span.phone`、`div#info span.email` 等值的是 `#{:div#info :span.phone} [:div#info :span.email]`。不过与 CSS 不同的是，析取不限于顶层：`[:div#info #{:span.phone :span.email}]` 甚至是 `[:div#info [:span #{:phone :.email}]]` 都是同一个选择器的不同表示法。



总之，集合表示析取，内部向量表示合取，最外层向量表示层次链接。

- 549 ➤ 所有其他测试都是用谓词执行的，而且可以随意扩展。这就是 `attr?` 的情况：CSS 选择器 `a[class]` 成为 `[[:a (attr? :class)]]`。注意嵌套向量：单个向量选择器——即 `[:a (attr? :class)]`——与 CSS 中的 `a *[class]` 等价。差异是显著的：

```
(h/snippet "p class=\"><a href=\"\" class=\"></a></p>"
  [[:p (h/attr? :class)]] (h/content "XXX"))
:= "<p class=\">XXX</p>

(h/snippet "p class=\"><a href=\"\" class=\"></a></p>"
  [[:p (h/attr? :class)]] (h/content "XXX"))
:= "<p class=\"><a class=\"> href=\"\" href=\"\">XXX</a></p>"
```

大多数 CSS 选择器的直接结果在 Enlive 里已经提供（包括所有的 `:nth-*` 伪类）。在那个初始集之外，你可以定义自己的选择器，这些只不过是函数。这可以用 Enlive 提供的 `pred` 或 `zip-pred` 高阶函数来完成，它们分别接受对元素的谓词和对 zipper^{注24} 的谓词为参数并产生一个 Enlive 可以用做选择器步骤的函数。

注 24： 在 151 页“遍历、更新以及 Zipper”一节里讨论了 zipper 数据结构。

在 attr= 里, Enlive 已经提供了一个选择器来匹配特定属性的值与给定值匹配的元素。^{注25}
我们来定义一个新的选择器步骤函数, 它匹配任何属性的值与给定值匹配的元素。

```
(defn some-attr=
  "Selector step, matches elements where at least one attribute
  has the specified value."
  [value]
  (h/pred (fn [node]
            (some #{value} (vals (:attrs node))))))
```

我们看看它是如何工作的 :

```
(h/snippet "<ul><li id=\"foo\">A<li>B<li name=\"foo\">C</li></ul>"
  [(some-attr= "foo")] (h/set-attr :found "yes"))
:= "<ul>
:=   <li found=\"yes\" id=\"foo\">A</li>
:=   <li>B</li>
:=   <li found=\"yes\" name=\"foo\">C</li>
:= </ul>"
```

Enlive 在控制转换的效果上已经给了我们很大的灵活度。如你所见, 对你想做的事来说灵活性不够时, 你可以定义自己的选择器, 使用你想要的任何标准。

迭代与分支

目前我们已经看到了如何识别节点及如何转换它们 (如像 content 或 set-attr), 不过还没有涉及模板的两个支柱: 条件和迭代。

Enlive 里迭代和分支的关键是理解转换可以是下列情况之一:

550

- 参数为一个元素并返回一个元素的函数。
- 参数为一个元素并返回一组元素的函数。
- nil, 与 (fn [_] nil) 等值。

由此得出, 举例来说, 显示一个可选的信息就像使用 when 一样简单, 它的条件在逻辑上为假时求值得到 nil :

```
(defn display
  [msg]
  (h/snippet "<div><span class=\"msg\"></span></div>"
  [:msg] (when msg (h/content msg)))
:= #'user/display
(display "Welcome back!")
:= "<div><span class=\"msg\">Welcome back!</span></div>"
```

注 25: attr= 用于例 16-5。

```
(display nil)
;= "<div></div>"
```

当存在信息时，when 形式求值为 (h/content msg) 转换函数，它会把匹配的 span 元素的内容设置为提供的信息。另一方面，如果没有信息要显示，when 形式会求值为 nil，信息的占位内容会被移除。

或者，你可能需要保留空的 span，因为客户端代码需要它；在那种情况下，就用 if（或 cond 或任何你偏爱的其他条件形式）而不是 when 形式：

```
(defn display
[msg]
(h/snippet "<div><span class=\"msg\"></span></div>"
[:.msg] (if msg
           (h/content msg)
           (h/add-class "hidden"))))

;= #'user/display
(display nil)
;= "<div><span class=\"msg hidden\"></span></div>"
```

在 Enlive 里迭代是用 clone-for 完成的，它看起来和表现上都很像 for：

```
(defn countdown
[n]
(h/snippet "<ul><li></li></ul>"
[:li] (h/clone-for [i (range n 0 -1)]
                    (h/content (str i)))))

;= #'user/countdown
(countdown 0)
;= "<ul></ul>"
(countdown 3)
;= "<ul><li>3</li><li>2</li><li>1</li></ul>"
```

在底层，一个 for 扩展产生一个序列的转换函数（在上面这个例子里，n 个 (h/content (str i)) 实例），每一个用于基于这个选择器选择的单个节点生成元素。生成的元素替代最初的节点。

关于迭代，一个常见的需求是移除原先用于选择节点的某些属性——例如一个 ID。这是用 do-> 函数完成的，它会组合转换，按顺序应用这些转换：

```
(defn countdown
[n]
(h/snippet "<ul><li id=\"foo\"></li></ul>"
[:#foo] (h/do->
          (h/remove-attr :id)
          (h/clone-for [i (range n 0 -1)]
                      (h/content (str i))))))
```

```
r= #'user/countdown
(countdown 3)
r= "<ul><li>3</li><li>2</li><li>1</li></ul>"
```

当然，`do->` 可以用于一个转换可以出现的任何地方，因为它求值为其自身，因而对如何组合转换函数没有限制。

把所有东西拼到一起

`sniptest` 是一个非常棒的探索式辅助工具，让我们可以演示 Enlive 的所有基本知识，不过在真实的应用里它就没什么用处了。我们需要从磁盘载入 HTML——或者更具体地说，是从应用的 classpath 载入。这是 `deftemplate` 和 `defsnippet` 的工作。

`defsnippet` 定义了一个函数从 classpath 上的文件载入 HTML，可以像 `sniptest` 那样任意转换。这些函数的目的是从其他片段或 `deftemplate` 函数里调用，作为组合一个个单元的内容的简便办法。例如，假设在 classpath 根目录里有一个文件 `footer.html`：

例16-6： footer.html

```
<div class="footer"/>
```

可以定义一个可以利用的 `footer` 片段：

```
(h/defsnippet footer "footer.html" [:footer]
  [message]
  [:footer] (h/content message))
```

❶ `footer` 是我们所定义的 var 和函数的名字。`"footer.html"` 是要载入内容的 HTML 文件的路径，可以是一个字符串或 `java.io.File`、`java.net.URL` 或 `java.net.URI` 的一个实例。`defsnippet` 的第三个参数是一个选择器，指定在载入的 HTML 文件里要应用转换的根元素。这里的 `[:footer]` 确保我们会扔掉 `<html>` 和 `<body>` 元素，这都是 Enlive 在载入片段和模板时隐式添加的。一个单个 HTML 文件可以包含好几个片段，每个 `defsnippet` 只选择相关的节点——在同一个文件里有许多可利用的组件，光用一个 Web 浏览器就可以预览这些组件是很方便的。

◀ 552

❷ 参数向量；这个片段函数接受单个参数，`message`。

❸ `defsnippet` 形式其余部分由一对对的选择器和转换器组成，这我们已经看到了。



如果使用 Leiningen，放 HTML 模板的最佳地方是在 `resources` 目录里（这是 `project.clj` 里：`:resources-path` 选项的默认值）。在 Maven 里，HTML 模板的根目录典型地是 `src/main/resources`。

与 `sniptest` 不同的是，调用 `defsnippet` 函数产生表示 HTML 元素的映射序列：

```
(footer "hello")
:= ({:tag :div, :attrs {:class "footer"}, :content ("hello")})
```

`deftemplate` 的工作方式几乎一样，不过你不能为应用的转换定义一个根，`deftemplate` 函数返回包含 HTML 片段的字符串的惰性序列，可以方便地用做一个 Ring 应答映射的 `:body`，而不是返回表示 HTML 的映射序列。

知道了这些，我们可以轻易地重新生成原先在例 16-4 里考虑的 ERB 模板的结果。首先定义模板文件：

例 16-7: friends.html

```
<h1>Hello, <span class="username"/></h1>
<p>These are your friends:</p>
<ul class="friends"><li/></ul>
```

然后，或者在应用程序里，或者在一个 REPL 里定义一个 Enlive 模板函数，生成一些完整的 HTML 文档：

```
(h/deftemplate friends-list "friends.html"
  [username friends]
  [:username] (h/content username)
  [:ul.friends :li] (h/clone-for [f friends]
    (h/content f)))

(friends-list "Chas" ["Christophe" "Brian"])
:= ("<html>" "<body>" "<h1>" "Hello, " "<span class=\"username\">""
:= "Chas" "</span>" "</h1>" "\n" "<p>These are your friends:</p>"
:= "\n" "<ul class=\"friends\">" "<li>" "Christophe" "</li>" "<li>" "Brian"
:= "</li>" "</ul>" "\n" "</body>" "</html>")
```

① 记得 `deftemplate` 返回字符串的序列。尽管在通过 Ring 向一个客户端返回内容时这是不需要的，如果你碰巧需要一个 Enlive 操作结果是拼接的字符串，只要调用 `(apply str (friends-list ...))` 就可以得到。

现在我们已经重新生成了原先考虑的 ERB 模板，但似乎并没有多领先。那个 ERB 模板虽然是一堆问题的根源，但表面看来有简洁的优势。即使考虑到 ERB 代码是嵌入在 HTML 内容里这一点，我们的 Enlive 结果看来仍然要求写更多的代码。由于在其他方面的收获，那可能是可接受的牺牲。毕竟没有什么办法是完美的。

在我们的模板和功能需求超越简单的范畴时，才能体会到 Enlive 的真正好处。因为 Enlive 的操作都实现为对一套标准的数据结构操作的函数，可以轻易地组合。这与大多数模板系统形成鲜明对照，那些是在字符串和字符串拼接的层次上操作的。

演示一下，把一个新类加到例子中产生的每个列表元素里，用 do-> 来组合两个转换器：

```
(h/deftemplate friends-list "friends.html"
  [username friends friend-class]
  [:username] (h/content username)
  [:ul.friends :li] (h/clone-for [f friends]
    (h/do-> (h/content f)
      (h/add-class friend-class)))))

(friends-list "Chas" ["Christophe" "Brian"] "programmer")
:= ("<html>" "<body>" "<h1>" "Hello, " "<span class=\"username\">" "Chas"
:= "</span>" "</h1>" "\n" "<p>These are your friends:</p>" "\n"
:= "<ul class=\"friends\">" "<" "li" " " "class" "=\"" "programmer" "\"
:= ">" "Christophe" "</" "li" ">" "<" "li" " " "class" "=\"" "programmer"
:= "\'" ">" "Brian" "</" "li" ">" "</ul>" "\n" "</body>" "</html>")
```

几乎没有增加代码的密度，涉及的修改是明显的，在代码中可以分离出来；同时我们的 HTML 模板文件没有变化。作为比较，传统的 ERB 方法开始演变成噪声：

```
<h1>Hello, <@user.name %></h1>

<p>These are your friends:</p>
<ul>
<% @user.friends.each do |friend| %>
  <li class=<% @friendclass %>><@= friend.name
%></li>
<% end %>
</ul>
```

最后，把页脚加入页面：

```
(h/deftemplate friends-list "friends.html"
  [username friends friend-class]
  [:username] (h/content username)
  [:ul.friends :li] (h/clone-for [f friends]
    (h/do-> (h/content f)
      (h/add-class friend-class))))
  [:body] (h/append (footer (str "Goodbye, " "username")))) ◉
```

554

```
(friends-list "Chas" ["Christophe" "Brian"] "programmer")
:= ("<html>" "<body>" "<h1>" "Hello, " "<span class=\"username\">" "Chas"
:= "</span>" "</h1>" "\n" "<p>These are your friends:</p>" "\n"
:= "<ul class=\"friends\">" "<" "li" " " "class" "=\"" "programmer" "\"
:= ">" "Christophe" "</" "li" ">" "<" "li" " " "class" "=\"" "programmer"
:= "\'" ">" "Brian" "</" "li" ">" "</ul>" "\n" "<div class=\"footer\">"
:= "Goodbye, Chas" "</div>" "</body>" "</html>")
```

❶ content 取代所选元素的子内容，append 附加到所选元素。如果在这里用 content，页脚将会是生成的 HTML 的 <body> 里的唯一内容。

可以在你的模板函数里直接调用 `defsnippet` 函数，像我们上面做的那样；或者可以把片段函数或调用片段函数的结果作为模板函数的参数传入；或者你甚至可以做一些像根据模板文件 HTML 里的类名查找并调用片段函数这样的事。因为建立 Enlive 模板涉及的所有实体都是通用的——都是消耗和产生坚持共同抽象的集合类的函数——如何组合模板和片段以组成完整的页面完全取决于你。

最后的思考

在本章里，我们看到了函数式软件设计方法是如何产生小巧但非常强大的抽象，让我们可以快速做出服务器端应用程序。在短短 26 行代码里做出了一个 HTTP 服务。我们显示了通过 Enlive 可以采用一种非常不同的 Web 内容生成方法，在全新的维度上把模板从服务器代码里分割出去。我们看到了组成一个中等规模的 Web 框架所需的一切。如果加上合适的持续存储，几乎没有 Web 特性你不能用这些组件解决的了。

不过比我们已经展示的更为有意思的是我们还没有展示的：大量的框架和生成器。这可以视为对 Clojure 的 Web 函数库的一个抱怨。那些来自 Python 或 Ruby（或者甚至是 Java）的程序员可能已经习惯于生成器、fixtures、控制器和视图。我们的例子中没有这些，有经验的 Web 开发者可能会因此感到不安。

经验丰富的函数式程序员可能注意到这种状态实际上对大多数函数式编程语言来说相当普遍。由于函数式编程的特性，非常容易把非常简单、基本的函数编织成复杂的系统。例如可以通过加入一个简单的中间件来把授权添加到我们的书签示例程序里。涉及的函数库和抽象的轻量级特性使得这成为可能且容易实现，这与 Django 或 Spring 增强的 Java 等完整的 Web 栈里的极为重量级的授权钩子形成鲜明对照。

实际上，已经对这种轻量级函数式编程方法产生了这样的印象，它开始反过来对传统 Web 编程社区产生影响。小的“超轻量级”Web 框架正在各个编程语言里冒出来，只关注基本的路由、渲染和简单的建模。

这并不是说 Clojure Web 开发就是由 Ring、Compojure 和 Enlive 说了算。多种“装备齐全”的框架²⁶ 和各种更大规模的框架和应用服务器已经开始出现。²⁷ 不过大多数这些进展非常乐意建立在我们上面探索的这些基础构件之上，这很说明问题。

注 26：例如 Noir (<http://www.webnoir.org>) 和 Ringfinger (<https://github.com/myfreeweb/ringfinger>)。

注 27：最值得注意的是 Immutant，这是建立在 JBoss 之上的一个 Clojure 应用服务器：<http://immutant.org>。

部署Clojure Web应用程序

一旦你的 Clojure 能力超过了某一点，开发的应用程序临近完成，不可避免地将需要向你的用户和客户发布。现代的分发规范倾向于部署在服务器端（经常“在云里”），客户通过 Web 服务和接口与它交互。我们在本章将探索打包和部署 Clojure Web 应用程序的各种办法，充分利用 JVM 和 Java 生态为此提供的成熟设施。^{注1}

Java 与 Clojure Web 架构

Clojure Web 应用程序打包、部署为 servlet，几乎没有例外，与用 Java 写成的 Web 应用程序所用的基本架构一样。Web servlet 不过是 Java 类，扩展了 `javax.servlet.http.HttpServlet`，这个基类为处理 HTTP 请求定义了一个可编程的接口。每个 HTTP 请求方法（GET 和 POST 等）都定义有方法，每个方法都接受请求对象和应答对象；每个 HTTP 请求方法实现查看传入的请求，协调写出应答的内容。遵循 servlet 规范（这归根结底就是实现单个 Java 接口、遵循某些打包约定）的应用程序可以在数十个应用服务器中的任何一个上部署为 Web 应用，而许多应用服务器在基准 Java servlet 支持基础上还提供各种专有功能（像数据库连接池、消息队列实现、管理与监控特性等）。

除了个别例外，应用服务器提供多重租用，可以在同一个应用服务器上部署多个应用程序，每个应用程序可能包含多个 servlet（参见图 17-1）。几乎所有的应用服务器也都是 Web 服务器（经常有非常成熟、高效的 HTTP/HTTPS 实现），不过你也可以选择部署到

<558

注1： 我们在这里描述的实践和基础架构（经过小修小补）能有效地重用于部署和管理不提供 Web 服务的服务器程序。更广泛地说，客户端 Clojure 应用程序（用于桌面或移动环境）可以采取 Clojure Web 应用程序所用的一般部署路子：看看对应的 Java 程序是如何部署的，重用同样的基础架构和过程。

一个应用服务器，由专用的 Web 服务器（如像 Apache httpd、lighttpd 或 IIS）作代理。^{注2}

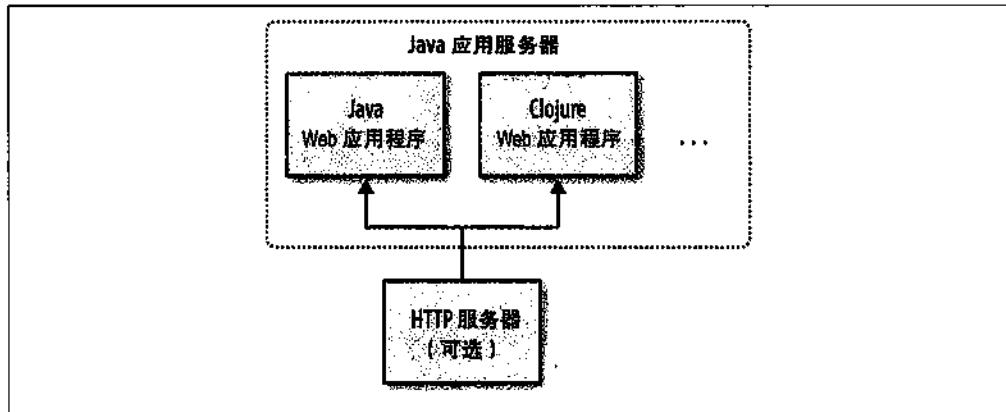


图17-1：Java Web架构

通过借用这一架构和生态，Clojure Web 应用程序享受到与 Java Web 应用相同的多样性部署目标和运营能力。



Java 服务器规范包罗万象，servlet 只是其中（无可否认基础性的）一部分。其他规范包括消息队列支持、目录访问、数据库持续方式等标准，分别由 JMS、JNDI 和 JPA 定义。如果你的组织已经在这些规范上投资，那么你会高兴地获悉 Clojure 能够用于并使用这些设施，就像它能够轻松地利用 servlet 规范一样。另一方面，如果你从来没有听说过这些 JEE (Java Enterprise Edition, 最初叫 J2EE) 标准，你同样高兴地了解到不用知道关于这些的任何东西也可以建构和部署功能齐全的 Clojure Web 应用。

虽然你可以用 Clojure 直接实现 servlet（毕竟这只是从 HttpServlet 基类派生的一个子类而已），但更为可取的是使用 Clojure Web 框架（像第 16 章仔细研究过的 Ring）从 servlet API 的编程（而且明显地命令式）本质抽象出来。就 Ring 而言，每个处理函数是接受一个参数的函数，这是一个从 servlet 的 HTTP 请求对象里的数据翻译成的地地道 Clojure 映射，返回值是一个 Ring 适配函数输出到 servlet 的 HTTP 应答对象。这些处理函数然后与路由合并——HTTP 请求方法（像 GET）和 URL 匹配模式的配对组合——产生单个函数封装你的应用程序在所有各类受支持的 HTTP 动词和 URL 上的合并功能。Ring 提供一两个适配函数把一个具体 servlet 的处理函数方法委托给那个函数。

注 2：一些应用服务器提供除 HTTP 代理之外的其他方法用于在应用服务器实例和专用 Web 服务器之间通信；对于高负载的情景，这些办法可能比 HTTP 代理效率更高。例如，Tomcat 应用服务器提供一组 Apache 和 ISAPI 模块实现一种紧凑的二进制通信协议：<http://tomcat.apache.org/connectors-doc/>。

如果“拉近镜头”看一下在图 17-1 里的 Clojure Web 应用程序，可以把 servlet 和 Ring 之间的关系可视化为：

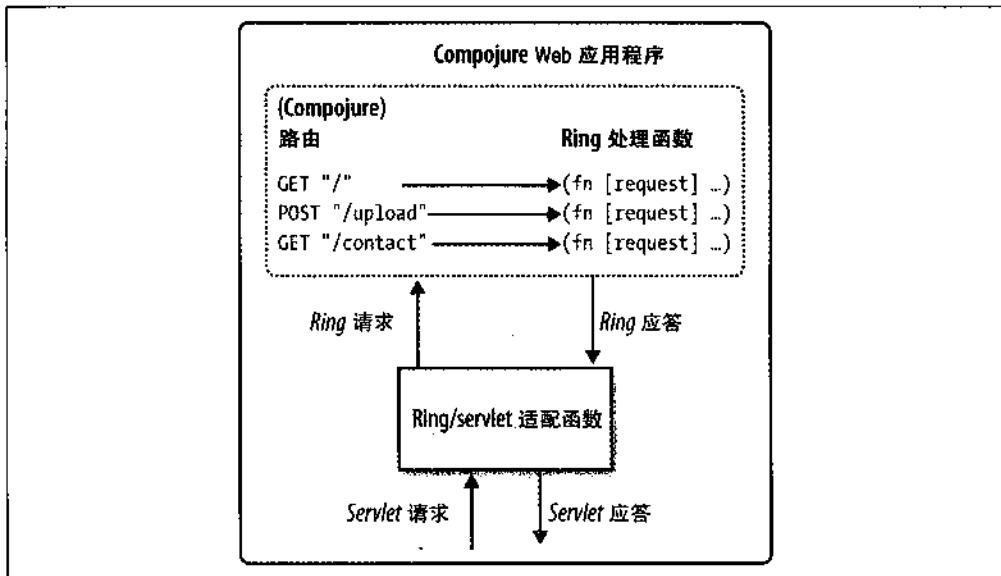


图17-2：Clojure/Ring Web应用程序架构

把 servlet 架构和 Ring 的模型连接起来有几种方法，取决于你的部署策略：

1. 如果要使用嵌入式应用服务器（Jetty 和 Glassfish 是最流行的），你可以在运行时生成一个 servlet 包装，简单地把它交给在同一个 JVM 里运行的应用服务器。所有一切都在运行时发生，不需要特别的打包或构建过程，这对于开发和本地测试情景十分合适。
2. 部署到一个独立的应用服务器^{注3}一般要求把 Web 应用程序打包成 .war 文件。打包这一步对已经在用的构建过程是微不足道的，会让你在任何符合规范的应用服务器（包括托管的平台如 Google App Engine^{注4} 和 Amazon 的 Elastic Beanstalk^{注5}）上部署你的 Clojure Web 应用程序。^{注6}

< 560

两种方法各有得失。部署到独立的应用服务器的确需要一定的构建、打包配置和过程。

注 3： 有数十个成熟的、有良好支持的应用服务器，每个在基础 servlet 和其他 JEE 规范之上都提供了一组自己的额外特性。

注 4： 参见 <https://github.com/gcvc/appengine-magic>。

注 5： 参见 <http://aws.amazon.com/elasticbeanstalk>。

注 6： 参见注意最近最为流行的 Clojure 部署目标之一，Heroku，在应用程序部署上有显著的不同；见 587 页“Heroku 上的 Clojure”一节。

另一方面，在嵌入式应用服务器上运作产品（有时称之为无容器的方法）迫使你创建、维护你需要的初始引导、部署和管理过程，不能简单地重用一个独立的应用服务器所提供的（可能测试得更多的）设施。我们推荐你开始时使用最习惯的做法，在可能的时候两种方法都试试，以便最后选择最适合你的项目和团队的那种。

我们已经在第 16 章演示了在嵌入的 Jetty 运行时里从 REPL 如何运行应用程序；现在我们来解决 .war 打包的问题，这会让我们将程序灵活地部署到产品级应用服务器和托管平台上。

Web 应用程序打包

一个 Java Web 应用程序被打包成一个 .war 文件，这是在 340 页“构件与坐标”一节里讨论的 .jar 文件打包的一种扩展。^{注7} 典型的 .war 文件布局包括如下内容。

- 像 HTML 文件和图形等资源，这些是在 .war 文件的顶层静态提供的。
- 以 WEB-INF 目录为根的各种数据，其中你能找到：
 - 一个 web.xml 文件，描述 .war 文件应该如何部署到一个 Web 应用服务器。
 - 一个 lib 目录，可以包括任何数目的嵌套的 .jar 文件。这通常是放置一个 Web 应用程序所有的传递性依赖的地方，并且使得 .war 文件成为自足的可部署的单元（与之相反的是，其他服务器应用架构要求大量的服务器配置，以保证所部署的应用的依赖都可用）。
 - 一个 classes 目录，包括 Clojure 源文件、JVM class 文件（包括从 Clojure 源文件提前编译产生的 class 文件）及其他资产。这与典型的 .jar 文件的内容对应，是 Web 应用程序的“顶层”代码放置的地方（这与它的依赖不同）。

561 →

.war 打包约定的关键是 web.xml 文件。就是用它来定义一个应用程序应该如何部署的；它在某种程度上可以：

- “载入”某个路径上的具体 servlet，让单个 .war 文件包含多个根在不同路径的独立应用程序。
- 指定哪一类型的静态资源（如像 CSS、JavaScript 和图形文件）应该由应用服务器直接提供，不需要由装载的 servlet 处理。
- 配置用户会话行为，包括会话超时和应该使用哪种存储机制来保持会话数据。
- 配置应用服务器特定的服务和特性。

^{注7}： 这里只是走马观花地介绍了你能够如何控制 Web 应用程序是如何管理和部署的，特别是通过设置包含在 .war 里的 web.xml 文件里的各种参数。关于 .war 文件和 web.xml 选项的更多信息请参见 http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/WCC3.html。

这里是一个非常简单的 `web.xml` 样例文件，定义了单个 servlet (`com.closurebook.hello_world`)，装载在应用服务器环境的根 (/) 上，并且要求“默认”servlet（每个应用服务器都有一个默认 servlet 用于提供静态资源）应该按照文件扩展名的定义负责提供各种静态文件：

例17-1：简单的web.xml文件

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

    <servlet>
        <servlet-name>app</servlet-name>
        <servlet-class>com.closurebook.hello_world</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>app</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>default</servlet-name>
        <url-pattern>*.css</url-pattern>
        <url-pattern>*.js</url-pattern>
        <url-pattern>*.png</url-pattern>
        <url-pattern>*.jpg</url-pattern>
        <url-pattern>*.gif</url-pattern>
        <url-pattern>*.ico</url-pattern>
        <url-pattern>*.swf</url-pattern>
    </servlet-mapping>
</web-app>
```

562

这是在 Maven 构建的 Clojure Web 应用程序里我们将要使用的 `web.xml` 文件，Leiningen 采用了一种不同的方法，它的 `web.xml` 文件是从 `project.clj` 文件里的配置生成的。正如我们将在本章其余部分所看到的，对构建工具的选择的确影响到了 Web 应用程序打包和组织的某些次要的细节。

用 Maven 构建 .war 文件

正如我们在 345 页“Maven”一节所见，Maven 项目默认生成 `.jar` 文件，至少当你的 `pom.xml` 把 `<packaging>` 定义为 `closure`（或者默认值 `jar`）时。把 `<packaging>` 改为 `war` 会导致 `mvn package`（或任何依赖 `package` 的阶段，如 `install` 或 `deploy`）被调用的任何时候项目会被打包成 `.war` 文件。因为不再使用 `closure` 打包方式，我们也需要把 `closure-maven-plugin` 的 `compile` 目标加到 Maven 的 `compile` 阶段：

例17-2：适合简单Clojure Web应用项目的pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/maven-v4_0_0.xsd">
<modelVersion>4.0.0</modelVersion>

<groupId>com.clojurebook</groupId>
<artifactId>sample-maven-web-project</artifactId>
<version>1.0.0</version>
<packaging>war</packaging>

<dependencies>
    <dependency>
        <groupId>org.clojure</groupId>
        <artifactId>clojure</artifactId>
        <version>1.3.0</version>
    </dependency>
    <dependency>
        <groupId>compojure</groupId>
        <artifactId>compojure</artifactId>
        <version>1.0.1</version>
    </dependency>
    <dependency>
        <groupId>ring</groupId>
        <artifactId>ring-servlet</artifactId>
        <version>1.0.1</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>com.theoryinpractise</groupId>
            <artifactId>clojure-maven-plugin</artifactId>
            <version>1.3.8</version>
            <extensions>true</extensions>
            <configuration>
                <warnOnReflection>true</warnOnReflection>
                <temporaryOutputDirectory>false</temporaryOutputDirectory>
            </configuration>
            <executions>
                <execution>
                    <id>compile-clojure</id>
                    <phase>compile</phase>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

563

```

        </execution>
        </executions>
    </plugin>
    <plugin>
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>maven-jetty-plugin</artifactId>
        <version>6.1.15</version>
        <configuration>
            <webAppConfig>
                <extraClasspath>
                    src/main/webapp,src/main/resources,src/main/clojure
                </extraClasspath>
            </webAppConfig>
            <reload>manual</reload>
        </configuration>
    </plugin>
</plugins>
</build>
</project>

```

这个样例 *pom.xml* 也包括了使用 Compojure 和 Ring 所需的基准依赖以及 maven-jetty-plugin 所需的一些配置，后者让我们使用嵌入的 Jetty 应用服务器以在本地运行 Web 应用程序，这对开发和本地测试非常理想。

这个简单的 Web 应用由一个简单的处理函数构成，因为在用 Maven 构建它时可以包含自己的 *web.xml* 文件，可以依赖应用服务器来处理对静态文件的请求，因而图片等不需要额外的路由。

例17-3：com.clojurebook.hello-world

```

(ns com.clojurebook.hello-world
  (:use
   [ring.util.servlet :only (defservice)]
   [compojure.core :only (GET)])
  (:gen-class
   :extends javax.servlet.http.HttpServlet))

(defservice
  (GET "*" {:keys [uri]}
    (format "<html>
              URL requested: %s
              <p>
                <a href=\"/wright_pond.jpg\">
                  Image served by app server via web.xml <servlet-mapping>
                </a>
              </p>
              </html>" uri)))

```

564

用 Leiningen 构建 .war 文件

`lein-ring`^{注8} 是一个 Leiningen 插件，它为使用 Leiningen 开发 Ring Web 应用程序提供了许多有用的设施。其中之一是生成 `.war` 文件，基于你的 `project.clj` 文件里的配置生成 `web.xml` 文件。这一方法与 Maven 方法差异很大（而且短小、简单得多！），用 Maven 方法你得自己写这个 `web.xml` 文件。

例17-4：适合简单Clojure Web应用项目的`project.clj`

```
(defproject com.clojurebook/sample-lein-web-project "1.0.0"
  :dependencies [[org.clojure/clojure "1.4.0"]
    [compojure/compojure "1.0.1"]
    [ring/ring-servlet "1.0.1"]]
  :plugins [[lein-ring "0.6.2"]]
  :ring {:handler com.clojurebook.hello-world/routes})
```

`lein-ring` 要求的唯一配置是这个 `:ring :handler` 槽，你在这里指定储存应用程序顶层请求处理函数的 `var` 全名（含命名空间）。`lein-ring` 将从这个名称生成一个 `servlet` 类（这里叫 `com.clojurebook.servlet`），它会把请求委托给我们的处理函数，还会生成一个定义这个 `servlet` 的 `web.xml` 文件并在根 URL 路径 (/) 装载这个 `servlet`。这等同于在 Maven 项目里指定我们自己的 `web.xml` 并使用 `gen-class` 和 `defservice` 来生成这个 `servlet` 类和请求委托。

有了这个 `project.clj`，运行 `lein ring uberwar` 会生成一个我们能用的 `.war` 文件以部署到任何应用服务器。

因为 `lein-ring` 生成的 `web.xml` 文件，没有指定应用服务器的默认 `servlet` 应该处理静态资源的请求，我们需要在主处理函数中包含一个路由用于从我们的 `classpath` 提供这些资产：

565 ➤ 例17-5：`com/clojurebook/hello_world.clj`

```
(ns com.clojurebook.hello-world
  (:use
    [compojure.core :only (GET defroutes)]
    [compojure.route :only (resources)]))

(defroutes routes
  (resources "/")
  (GET "*" {:keys [uri]}
    (format "<html>
      URL requested: %s
      <p>
        <a href=\"/wright_pond.jpg\">
          Image served by compojure.route/resources
      </p>
    </html>")
    ))
```

注 8： 关于完整文档和详情请参见 <https://github.com/weavejester/lein-ring>。

```
</a>
</p>
</html>"  
uri)))
```



`lein-ring` 只让你定义 `web.xml` 文件里可用配置选项的子集，而且目前不让你提供自己的 `web.xml` 文件。因而如果想使用 `servlet` 规范的某些特性（如 `servlet` 过滤器、环境参数和默认 `servlet` 映射——在 Maven 样例中我们用这些特性让应用服务器处理静态资源的请求），你需要：

- 用 `leinring-war` 插件⁹（它只构建 `.war` 文件，不提供 `lein-ring` 所提供的本地 Jetty 部署的特性），或……
- 在使用 `lein-ring` 的基础上增加一些附加的脚本功能，以把定制的 `web.xml` 文件换进它所生成的 `.war` 文件，或……
- 用 Maven。

在本地运行 Web 应用

Jetty 用于为开发和测试目的在本地运行的 Web 应用，无论你是偏爱 Maven 或是 Leiningen。这让你可以用快速原型法试验新特性和排错方案，不需要经历产品或常见远程测试环境，“真正”部署所要求的完整的打包和部署循环。

Maven。正如在 562 页“用 Maven 构建 `.war` 文件”一节所说明的，Web 应用可用的样例 `pom.xml` 包含对 `maven-jetty-plugin` 的配置。在这样配置的项目里运行 `mvn jetty:run`，Jetty 会在 `localhost` 8080 端口启动，运行项目的 Web 应用。在修改 Clojure 代码、静态资产或 `web.xml` 文件时，你会希望不用完全停止再重启 Jetty，这些修改就可以为 Jetty 里正在运行的应用程序所用。为此，在你启动 Maven 的控制台按回车键，这会重新载入你的应用程序正在运行的 Jetty Web 应用环境，这比杀掉 Maven/Jetty 进程再从头开始要快得多。

566

Leiningen。Leiningen 对在本地运行 Web 应用的支持基本上与 Maven 一样。`lein ring server` 会启动一个 Jetty 服务器，所有请求都会路由到 `project.clj` 文件的 `:ring :handler` 槽所提供的根处理函数。`lein-ring` 采用了略有不同的办法把有修改的文件载入到正在运行的 Web 应用：不是等待你请求重新载入应用程序的环境，`lein-ring` 每次接收到一个使用 Clojure 的 `require` 函数的请求时它都会重新载入项目里所有的 Clojure 源文件。

通过“远程”REPL 载入更新的代码。比 `maven-jetty-plugin` 或 `lein-ring` 提供的代码

注 9：参见 <https://github.com/alienscience/leinring-war>。

载入解决方案更灵活的是把一个 REPL 服务器与你的 Web 应用程序绑定到一起，并把它配置为在部署时启动。这会让你从开发环境（如像 Eclipse + Counterclockwise 或 Emacs + SLIME）连接到“远程”REPL 服务器，并在你想要的时候载入 Clojure 代码，而不是依赖于超时、环境切换到 Jetty 启动时的终端，或潜在可能延迟把修改保存到磁盘，直到准备好把这些修改载入到正在运行的应用里。

这个办法是通用的，不限于 Web 应用程序，也不限于本地部署；详情见第 10 章。

Web 应用程序部署

能够把 Clojure Web 应用程序打包成标准的 `.war` 文件给了我们很大的部署灵活度：可用于 Java Web 应用程序的所有部署实践和设施因而都可用于 Clojure Web 应用。一般来说，部署这个动作有以下要求：

1. 设立、配置一个应用服务器。
2. 把构建进程产生的 `.war` 文件复制到应用服务器上。
3. 如有需要，重启应用服务器。
4. 如有需要，把应用程序的 `.war` 文件恢复到之前的一个版本（例如万一最后部署的版本含有一个回归错误）。

这些事你肯定可以手工或以定制的方式做——程序员和系统管理员“亲手”操作应用程序部署已经有很长时间了。而且如果你的机构已经部署了 Java Web 应用程序，你几乎可以肯定能把新开发的 Clojure Web 应用纳入同一过程。

不过如果你愿意尝试，有一些对 Clojure 友好的工具链可以使应用程序部署比多数其他选项简单许多、容易许多，更容易自动化。我们来看看其中一个，Amazon 的 Elastic Beanstalk 服务，可以广泛用于 Clojure Web 应用程序，自动化服务器的提供、配置和把应用程序部署到这些服务器上。

567 在 Amazon 的 Elastic Beanstalk 上部署 Clojure 应用

Amazon 的 Elastic Beanstalk (EB) 是一个用做服务的平台，它基于 Amazon Web 服务 (AWS) 更低层的 EC2 计算与负载均衡服务提供一薄层的自动化和部署管理工具。EB 让你可用程序提供和控制环境（一个或多个应用服务器，前端使用一个负载均衡器），你可以向环境部署不同版本的应用程序。

EB 使用的负载均衡器与这一预备机制集成，以便在应用程序经历较高负载（基于你定义的指标，如每分钟请求数或带宽使用总计）时，对应的 EB 环境扩展为包含更多应用

服务器来服务负载。特别是如果你的应用程序使用 AWS 的数据库设施^{注 10} 或可以在 AWS 里共处的其他的托管数据库^{注 11}，EB 可能是非常引人注目的部署选择，可以处理应用程序整个开发、部署和维护生命周期。

AWS 为它的服务提供了全面的 Java API，包括 EB，因而在 Clojure 里与它交互很简单。方便的是，有一个 Leiningen 插件用于与 EB 交互，lein-beanstalk。^{注 12}

基本设置与部署。需要对适合 Leiningen 的 Web 应用程序项目做一两处修改才能使它适用于 lein-beanstalk。首先把 lein-beanstalk 插件加到你的 *project.clj* 的 plugins 向量中。

```
[lein-beanstalk "0.2.2"]
```

其次，需要加一个 Compojure 路由来支持 Elastic Beanstalk 的“心跳”。定期用一个对根 (/) URI 的 HEAD 请求轮询你部署的应用程序；如果它没有成功应答，Elastic Beanstalk 就会假设整个应用程序死了，并重新部署、重新启动它。在这个例子里，我们重用在 535 页“用 Compojure 路由请求”一节里建的 URL 简缩服务；将在它自己的命名空间设置 Beanstalk 心跳路由，并且——作为对 Compojure 路由和一般的 Ring 处理函数可组合性的进一步证实——直接加入实际 URL 简缩服务的顶层路由：

例17-6: com.clojurebook.url-shortener.beanstalk

◀568

```
(ns com.clojurebook.url-shortener.beanstalk
  (:use [compojure.core :only (HEAD defroutes)])
  (:require [com.clojurebook.url-shortener :as shortener]
            [compojure.core :as compojure]))


(compojure/defroutes app
  ; This HEAD route is here because Amazon's Elastic Beanstalk determines if
  ; your application is up by whether it responds successfully to a
  ; HEAD request at /
  (compojure/HEAD "/" [] ""))
shortener/app)
```

需要对项目做的最后修改是加上适当的 ring :handler 配置，因为 lein-beanstalk 重用 lein-ring 来为部署生成 .war 文件。下面是这个 URL 简缩程序的 *project.clj* 文件的样子，包括 lein-beanstalk 插件和对 :ring :handler 的合适配置：

注 10： AWS 提供三个“受管理的”数据库服务，与 Elastic Beanstalk 支持的最小运行代价匹配得非常好： DynamoDB 和 SimpleDB，两者本质上是非关系型数据库（后者是前者的前身），和 AmazonRDS，提供关系型 Oracle 和 MySQL 环境。我们有一人维护着 SimpleDB 的一个 Clojure API，可在 <https://github.com/cemerick/rummage> 得到。

注 11： 像用于托管的 CouchDB 集群的 Cloudant：<https://cloudant.com>。

注 12： 参见 <https://github.com/weavejester/lein-beanstalk>。

例17-7: project.clj

```
(defproject com.clojurebook/url-shortener "1.0.0"
  :description "A toy URL shortener HTTP service written using Ring and Compojure."
  :dependencies [[org.clojure/clojure "1.3.0"]
    [compojure "1.0.1"]
    [ring "1.0.1"]]
  :plugins [[lein-beanstalk "0.2.2"]]
  :ring {:handler com.clojurebook.url-shortener.beanstalk/app})
```

最后，需要把你的 AWS 资格证明加入到 `~/.lein/init.clj` 文件的一个定义里^{注13}，这些将被用于认证你所有的 `lein-beanstalk` 活动：

例17-8: ~/.lein/init.clj

```
(def lein-beanstalk-credentials
  {:access-key "XXXXXXXXXXXXXX"
   :secret-key "YYYYYYYYYYYYYY"})
```

一旦做了这些修改，就可以通过 Elastic Beanstalk 部署我们的应用了。一个命令行的调用就可以做到这一点：

```
lein beanstalk deploy development
```

这个命令使得 `lein-beanstalk`：

1. 使用 `lein-ring` 为你的 Web 应用产生一个 `.war` 文件。
2. 把那个文件复制到 Amazon 的 S3 服务里。
3. 创建一个与你的项目名称相同的 Elastic Beanstalk 应用程序，如果这个程序还不存在的话。^{注14}
4. 为这个 Elastic Beanstalk 应用程序创建一个环境，称之为 `development`。
5. 请求把已经复制到 S3 的 `.war` 文件部署到 `development` 环境。

当 `deploy` 命令结束时（可能会花几分钟，特别是第一次部署某个应用或为特别环境部署时），`lein-beanstalk` 会表明你的应用已经启用，可以被访问。^{注15}

应用程序版本管理。 Elastic Beanstalk 在 S3 里保留你的应用程序所有之前的版本，因而你可以在任何时候通过 AWS 控制台回滚一个部署。`lein-beanstalk` 用你的 `project.clj`

^{注13}：像这样的安全资格证明永远不应该放在你的 `project.clj` 里，或可能提交到源文件版本控制的任何项目文件里。在内部那样做的后果已经足够糟，如果碰巧把那样的内容提交到公开可访问的托管服务，如 GitHub 或 BitBucket 简直就是灾难性的了。

^{注14}：例如，如果你使用 `lein-beanstalk` 样例项目——它的坐标是 `com.clojurebook/url-shortener`，那么应用程序的名称将会是 `url-shortener`。Elastic Beanstalk 应用程序名称必须是全局唯一的，因而如果你使用（别人的）样例项目，一定要把构件 ID 修改成你肯定是最唯一的东西。

^{注15}：URL 的形式是 `http://your-project-name.elasticbeanstalk.com`。举例来说，可以在你的域的 DNS 配置中用一个 CNAME 记录把 `www.yourdomain.com` 指向这个 `elasticbeanstalk.com` 域，透明地从 Elastic Beanstalk 提供有更好品牌的站点。

文件里声明的版本号来创建那些 EB 版本号，因而很容易理解你所部署的是什么版本以及它与你的应用程序之前的版本的关系。通过运行 `lein beanstalk info` 命令可以查看上传到 Elastic Beanstalk 的最近的那些版本：

```
% lein beanstalk info
Application Name: url-shortener
Last 5 Versions: 1.0.0-SNAPSHOT-20111219051007
                  1.0.0-SNAPSHOT-20111219045316
Created On:       Mon Dec 19 04:53:53 EST 2011
Updated On:       Mon Dec 19 04:53:53 EST 2011
Deployed Envs:   development (Ready)
```

在肯定你所部署的令人满意后，可以用 `lein beanstalk clean` 从 S3 消除不用的版本和对应的 `.war` 文件。

环境。在给定的 Elastic Beanstalk 应用程序里可以有任意多个环境，并可用你喜欢的名字。`lein-beanstalk` 默认定义三个环境：`development`、`staging` 和 `production`，其中第一个在上面的例子中用到。可以在你的 `project.clj` 里修改或增加这些默认环境；关于详情可参考 `lein-beanstalk` 的文档。

运行 `lein beanstalk info <environment-name>` 命令可以获得关于特定环境的各种运行细节：

```
% lein beanstalk info development
Environment ID: e-cnjm4hrqki
Application Name: url-shortener
Environment Name: development
Description:
URL:           url-shortener-dev.elasticbeanstalk.com
Load Balancer URL: awseb-development-1574221210.us-east-1.elb.amazonaws.com
Status:         Ready
Health:         Green
Current Version: 1.0.0-SNAPSHOT-20111219051007
Solution Stack: 32bit Amazon Linux running Tomcat 6
Created On:     Mon Dec 19 05:10:44 EST 2011
Updated On:     Mon Dec 19 05:13:11 EST 2011
```

570

超越简单 Web 应用程序部署

当然，取决于你的需求和环境，应用程序部署还有很多很多我们这里没有覆盖到的。许多项目（甚至非常大的、访问量巨大的站点）可以在 Elastic Beanstalk 等服务上毫无问题地度过整个存续期；不过其他项目会需要自动化更多过程，而不仅是简单地把 `.war` 文件在一个运行的容器里设置起来。Web 前端和负载均衡器的提供和配置的精细控制、对数据库设施的完全控制、定制的网络路由和监控、也许在云服务和内部基础架构的异质

混和环境里……任何这些问题都可能使本来可能是简单地使用一个易用的云服务变样，要求在处理通常用例之外的某些工具。Pallet 是一个 Clojure 工具链，用于处理像这样的挑战，参见 586 页“Pallet”一节。

无论你用什么工具或有什么需求，不要忘记可应用于 Java 应用程序的解决方案都可用于 Clojure 应用程序。同样的，你总是可以使用非 Clojure 的工具（不论是 Chef、Puppet 或其他）来部署和管理 Clojure 应用程序，直接使用这些工具为 Java 和一般的为 JVM 应用提供的技巧和习惯用法。

杂项

明智地选择 Clojure 类型定义形式

Clojure 提供了许多不同的形式用于定义类型：

- `deftype`、`defrecord` 和 `reify` 是 Clojure 的主要数据类型抽象，在 270 页“定义你自己的类型”一节探讨过。
- 所有种类的映射，对最灵活的临时类型特别有用，在第 3 章中讨论过。
- `proxy` 和 `gen-class`，关注于提供全部的 Java 和 JVM 交互性，在 371 页“定义类、实现接口”一节涉及过。

这些形式分别代表不同的利弊权衡。特别在你新学 Clojure 时，可能难于决定什么时候选一种类型定义形式而不选另一种，什么时候该用 `deftype` 而不是 `defrecord`，该用 `gen-class` 而不是 `deftype`，或者用 `proxy` 而不用 `reify`？

我们在上面提到的章节里已经努力探索了这些形式的所有细微差别。不过有时候这些东西有一个直观的参考更有帮助些，即使只是总结性质的。记住这一点，我们希望你会觉得图 18-1 所示的流程图有用。以你想要用 Clojure 定义一个类型为前提，这个流程图会引导你辨别这个语言的类型定义形式之间最重要的区别要点，以便能找到对特定情况最合适的形式：^{注1}

注1：这个流程图最新的标准版在 <https://github.com/cemerick/clojure-type-selection-flowchart> 维护，包括对它的翻译，目前有荷兰语、德语、日语、葡萄牙语和西班牙语。

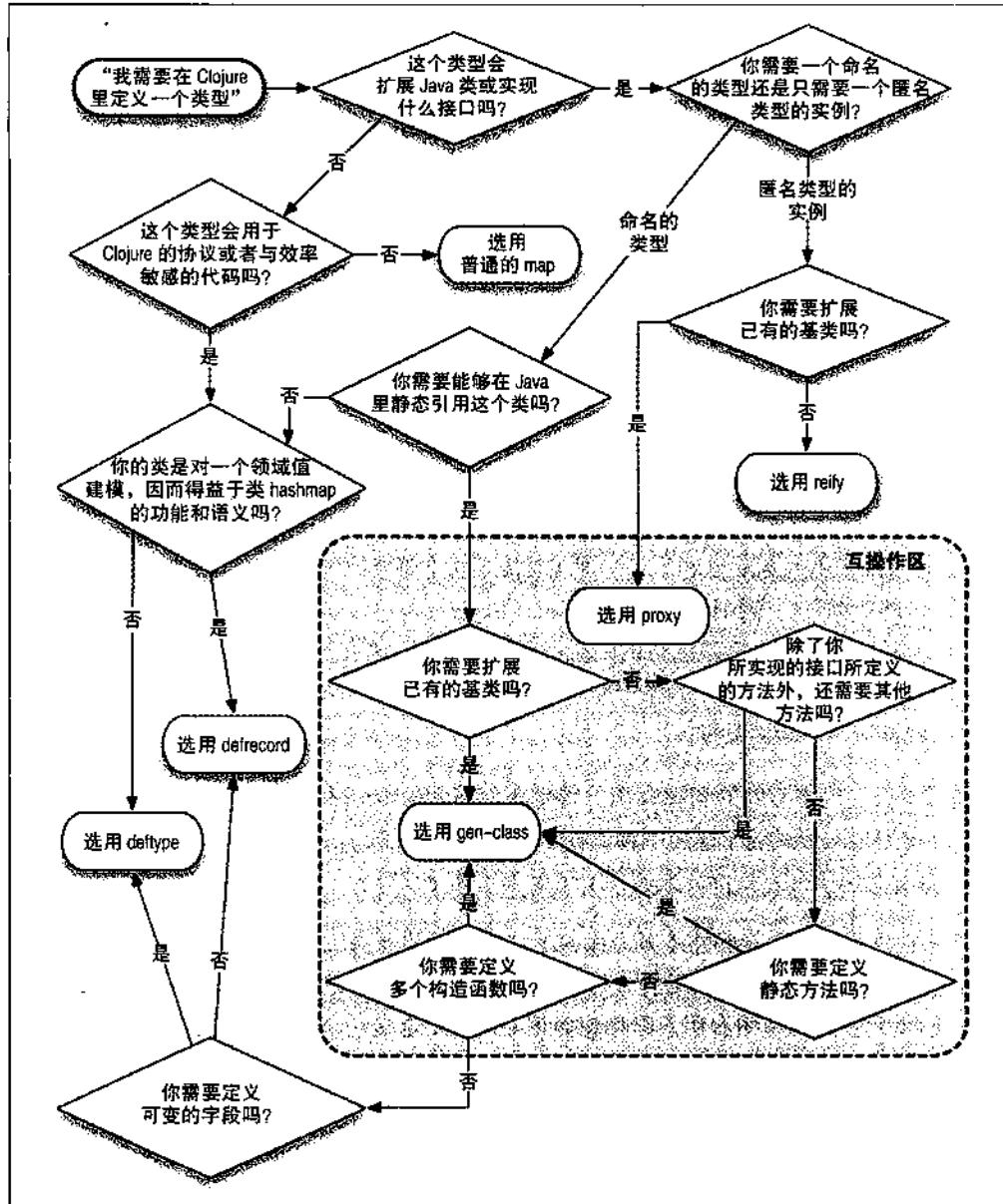


图18-1：明智地选择Clojure类型定义形式

“互操作区”划定了只属于 Clojure 与 JVM 互操作支持方面的用例（例如需要定义多个构造函数）和形式（proxy 和 gen-class）的界线。使用这些形式是可以的，不过需要明白的是，这样做你就踏出了 Clojure 的“原生”抽象的范围。除非你要定义一个类型的

目的就是满足互操作的要求，Clojure 中某种更简单的类型定义形式可能更能满足你的需求。

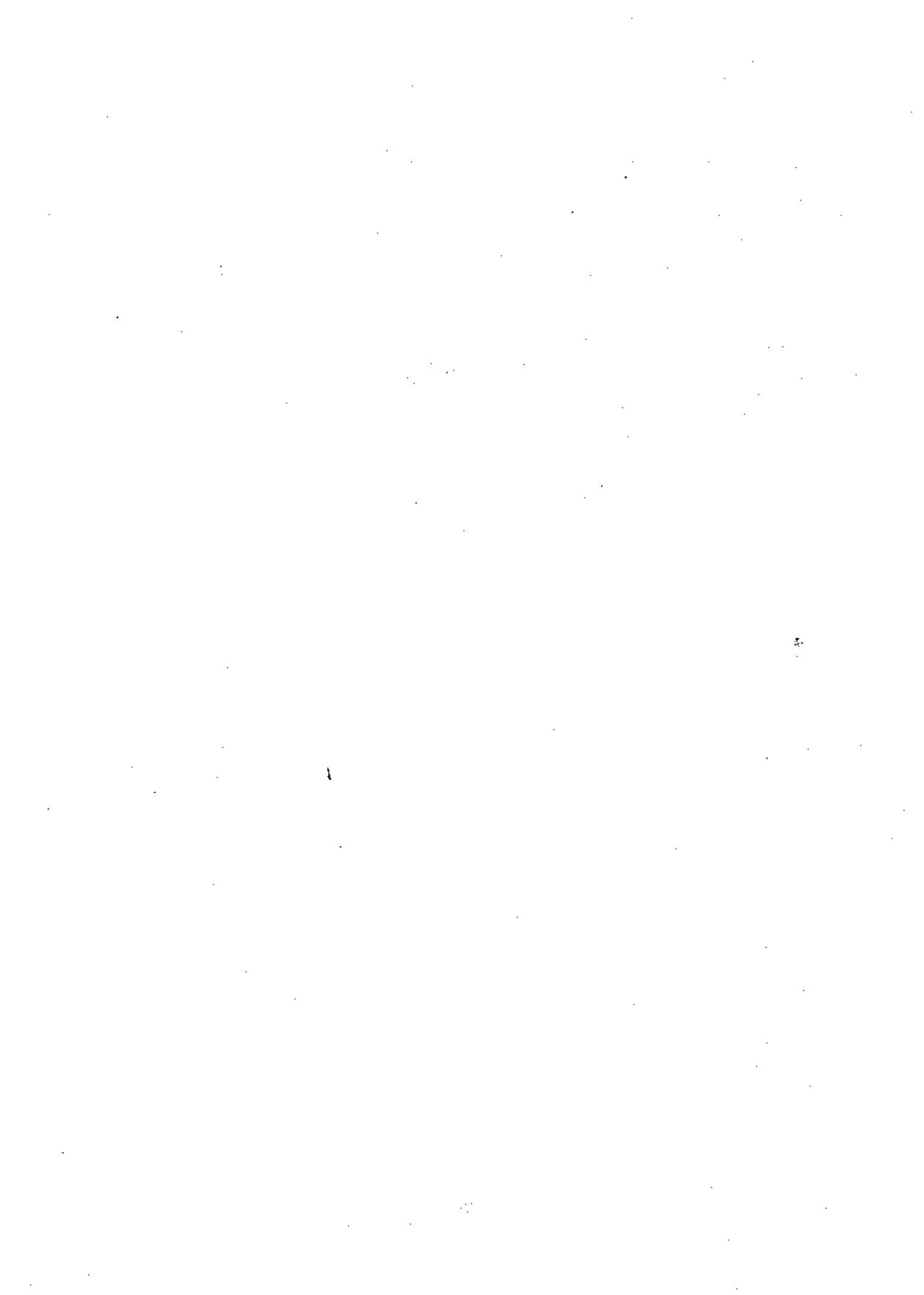
◀ 575

“效率敏感代码”的一个例子是在热点循环里访问某些槽。在这种情况下：

- 常规的映射可能很要命。
- 把 :keyword 访问函数用于一个记录或 deftype 实例会实现接近 Java 的效率。
- 使用直接字段访问（即 (.field val)）用于记录或 deftype 实例将会与 Java 定义的同样快。

这并不意味着你应该总是使用字段访问方式，甚至不是大多数时候。那是优化手段，只在绝对需要的时候才应该用，特别是考虑到相关的代价：高效的字段访问把使用它的代码与特定类型绑定在一起，这经常会使通用功能的实现复杂化并限制可组合性。^{注2}

注 2：记得“过早的优化是所有罪恶的根源”这句名言吧！谢谢您，高德纳教授！



在工作场所引进Clojure

(或者绕过老板偷偷用 Clojure^{#1})

一个令人悲伤的事实是，许多程序员，每天使用的语言和工具是他们恼恨的。或者由于历史的偶然、组织的惰性或者业务上铁的事实，经常发现自己陷入困境，希望能够使用某种技术甚至是任何其他技术来完成自己的工作。

如果你理解和欣赏 Clojure 已经到了想在每天的工作中或下一个咨询业务中使用它的程度，这样的现状可能特别令人沮丧。我们这里只是为你提供一个简短的指南、一个速查表、一组要点和策略，以帮助你在工作场所成功引进 Clojure。通过这样做，你将有望最终有更多高效率的日子、更少沮丧的夜晚和一个利润可观的生意。

事实是……

Clojure 是具有创新性的新语言。在 Clojure 相对很短的历史里，它已经颠覆了编程语言许多传统智慧。五年前难以想象一门鼓励函数式编程、默认使用持续的不可变的数据结构、提供易驾驭的并发与并行原语、提供广泛的元编程设施、运行在 JVM 之上但效率上与 Java 相差无几的语言能够流行起来。

这个故事远不止于此，即使它已经提供的继续成熟并变得广为人知时，也没有理由认为 Clojure 会不继续拓展边界、覆盖新的领域。

Clojure 有经过尝试、测试的可靠基础。就编程语言而言，Clojure 自身相对还是很新的，但它植根很深。Rich Hickey (Clojure 的创始人) 在 Clojure 之前创造了不下三个项目着

注 1：当然我们在开玩笑，半认真的。

着眼于 Lisp 环境和 JVM 和 / 或 .NET 之间的互操作。^{注2} 这些早期的试验为 Clojure 的出现铺平了道路，它实现了在 Lisp 基础与所依托的虚拟机环境之间寻求平衡的办法并证明了这个具体方法是有效的。

当然，Clojure 在大大小小方面深受之前的 Lisp 语言和环境的启迪，就像多数现代语言（包括 Java、Python、Ruby 和 JavaScript，只举少数几个例子）那样。这个血统最吸引人的部分是 Clojure 全身心地拥抱函数式编程和宏，宏是一种元编程方法，尽管在 Clojure 到来前的数十年已经得到验证，在 Lisp 语言家族之外仍然没有可以匹敌的。

Java 和 JVM 最初的创造者借用了使得早期 Lisp 系统成为令人愉快的、富有生产效率的编程环境的许多东西。^{注3} 从许多方面来说，Clojure 目标针对 JVM 使这个过程形成了完整的循环。

Clojure 让你利用 JVM 的所有益处。许多程序员，通常受到解释性语言（如 Ruby 和 Python）的表达力和相对灵活性的诱惑，长期以来渴望有更好的运行时。特别是对安于动态类型的团队来说，Clojure 提供了两个世界各自的最好方面：一流表达力架在 JVM 之上，在将近 20 年的历史里 JVM 已经被打造成极为先进的通用计算平台。

Clojure 运行在 JVM 之上让你受益于 JVM 广泛使用的所有成果：可靠的运营和效率特征，高质量的第三方（既有开源也有商业的）函数库组成的庞大生态系统，成熟的服务器和工具支持，没有争议的依赖管理，等等。

Clojure 让你重用在 Java 上的投资。如果你今天使用 Java，或者在 JVM 上已经有投资，Clojure 帮你利用它。Clojure Web 应用程序将与你的其他 Java Web 应用程序并肩运行，毫无问题。你可以在用 Clojure 写的代码里调用已有的 Java 函数库，你也可以在用 Java（或者运行在 JVM 上的任何其他语言，包括 JRuby、Jython、JavaScript（通过 Rhino）、Scala 和 Groovy 等）写的代码里调用 Clojure 函数、创建 Clojure 类型的实例。你已经学习的有关构建、打包、持续集成、JVM 运维和优化等所有东西都可应用于将用 Clojure 完成的工作。

579 Clojure 是对你已有的 JVM 投资一次递进式增加，不是一次激进的偏离。

注 2：按时间顺序是：DotLisp，工作在 .NET (<http://dotlisp.sourceforge.net>) 上；jFLI (<http://jffi.sourceforge.net>)，目标是 JVM 以及 FOIL (<http://foil.sourceforge.net>)，与这两种虚拟机都可以工作。

注 3：Guy L. Steele 是最初的 Scheme 编程语言的创造者之一，也是 *Java Language Specification* 和 *Common Lisp, the Language* (ANSI Common Lisp 标准的基础) 的作者，他被广泛认为说了这样的话：“我们成功地把许多人（C++ 程序员）拖到了走向 Lisp 之路的途中。”

“Clojure 只是另一个 *.jar* 文件”。Clojure 让你重用在 Java 上的投资的一个直接结果是 Clojure 真的“只是另一个 *.jar* 文件”。这意味着你可以把它打包成发布的应用程序的众多依赖之一，包含 Clojure 源代码（或从这些源代码提前编译成的 class 文件），你的顾客和客户不会察觉有什么不同。

我们不是建议你推翻管理或欺骗顾客。不过，如果你已经拥有合法的地位和权威选择用什么语言为客户提供和建构软件，那么与选择用 ANTLR 来建立语法、有充分理由地把 Lucene 索引滥用做通用的数据存储或转向 Visual Studio 来为基于 Java 的 Windows 桌面程序构建一个正常的启动程序相比，使用 Clojure 作为一个完整方案的一部分就没什么差别。

强调生产效率

少写代码，多做事。非常有可能的是，Clojure 对给定问题的解决方案没有用另一种语言对同一个问题的解决方案那么累赘、那么复杂，而是更易理解。Clojure 需要应付的可变状态远少于许多其他语言。并发与并行在消除了锁和死锁之后有了一致的、简化的语义。类和类型是可选的，存取方法是不必要的。合理使用宏，语法模式可以被抽象出去。这个列表还可以继续下去。

利用 Clojure 将很可能对你的工作效率产生如同 Java 对曾经使用 C/C++ 的人产生的影响那样、或 Ruby 或 Python 对曾经使用 Java 的人产生的影响那样。这些好处将会放大到你的整个团队，因为其他语言一些特别强加的概念和设计上的代价被移除了。

程序员从一开始就渴望更高级的语言，Clojure 不过是这一进展的又一步。

Clojure 使得创新性地解决复杂问题成为可能。我们这里谈论的许多东西都与 Clojure 能够适应已有的投资、语言偏爱等有关。不过最能感受到的 Clojure 的某些好处与它如何能帮助你脱颖而出有关。

多数编程语言预定了某个架构方面的视角，你必须让你的问题域适应这一视角。Clojure 与这一方法形成强烈对照，它鼓励使用函数式编程、以专门建造的（例如由协议、记录和多重方法等使之成为可能的）抽象来建模解决方案。通过这样做，Clojure 迫使你只为你所使用的那部分语言“付出代价”——所涉及的货币不是现金，而是建模的复杂度和心理代价。◀ 580

所有这些意味着在你用 Clojure 设计一个 API、一个函数库或整个系统时，这个设计不会被 Clojure 为你做出的决定拖累。这可以使得对小问题的解决方案异常简单，并且使得非常复杂的问题能有易于处理的实现。

Clojure 可以给你带来优势。一个工具能给你或你的组织带来相对于竞争对手不平等的优势，你会使用吗？^{注4}

回看几年，如果你能在 1999 年使用 Ruby 和 Rails 做 Web 开发将会怎样？或者在 1990 年，如果你能用 2000 年前后的 Java 建构大规模系统将会怎样？我们猜你会迫不及待地抓住那样的机会。

现在，如果我们瞥一眼，世界正运行于 Java、PHP、C/C++ 和少许“边缘”语言如 Ruby、Python 和 Scala 之上。我们可以有把握地假设 Clojure 不会像 Java、PHP 或 C++ 那样使用广泛，不过我们可以肯定地说它的某些关键特征——例如它的软件事务内存架构和持续的函数式数据结构实现——正被采纳、改编，有时甚至是被其他非常有力的编程语言直接使用。不过正如 Java 不会因为 JVM 的设计者从 Lisp 系统借用一大堆技巧和特性就是一门 Lisp 语言，今天从 Clojure 借用功能和特性也不等同于 Clojure。

如果你现在开始使用 Clojure，可能给自己带来在未来数年拥有不对等优势的机会。

强调社区

Clojure 是开放源代码的，欢迎大家贡献。Clojure 源代码是开放的，以自由使用的许可协议授权，^{注5}使得它非常适合包含在商业产品中和在商业机构内部使用（当然包括非商业的、慈善的或个人使用）。

尽管 Clojure 自身是一个人的愿景的体现，^{注6}围绕在 Clojure 周围的项目努力是非常有包容性的：数百个个体（其中许多代表公司）登记为贡献者，^{注7}他们帮助改进这个项目的各个方面。在本书写作时，Clojure 语言自身就已经有来自超过 70 个人的贡献历史。

结果是：

1. 你在使用 Clojure 上永远不会受限于法律方面的外部因素。
2. 如果在 Clojure 的实现里发现了一个错误，你对它会被解决充满信心——或者由已有的贡献者对错误报告做出反应，或者经你自己之手（如果你选择自己成为一名贡献者的话）。

Clojure 有一个巨大的、成长中的友好社区。Clojure 社区很大，并且在快速地成长：

注 4： Paul Graham 的 *Beating the Averages* 的文章与这一点非常有关联，具体请参见 <http://www.paulgraham.com/avg.html>，如果我们不引用它，那将是我们的失职。

注 5： Eclipse 公共许可协议，这允许自由的商业使用和重新分发：<http://www.eclipse.org/legal/epl-v10.html>。

注 6： 就 Clojure 来说，是 Rich Hickey，他的作用类似于 Python 的 Guido Van Rossum、Ruby 的松本行弘、Perl 的 Larry Wall 和 C++ 的 Bjarne Stroustrup。

注 7： 所有被承认的贡献者列于 <http://clojure.org/contributing>。

- 主要的邮递列表 (<http://groups.google.com/group/clojure>) 有超过 6000 名订阅者，并且一直很活跃。
- 主要的 IRC 频道^{注8} 经常有超过 400 人在里面，并且一直很活跃。
- 主要的 Clojure 函数库和工具都有各自的邮递列表（其中多数通过 Google Groups 托管），由其作者和主开发者监控。
- 到你阅读本书时，至少将会有 6 本关于 Clojure 的书已经出版，全部由主要的出版商出版。

不过规模不是全部。虽然有些编程语言和框架不知怎么形成了敌对的、有害的、自尊心驱动的社区，但 Clojure 的社区正建立起以对新手友好和欢迎的姿态闻名的良好声誉。实际上在 2011 年，Clojure 现状调查^{注9} 问到哪些因素对他们使用 Clojure 是最让人泄气的时，只有 2% 的人回答（这个问题最小的人群）说“不愉快的社区交互”是一个问题。

所有这些意味着在你想要和有经验的、有见识的 Clojure 程序员谈话时——不论是提供帮助或是求助——你总有地方可去。

Clojure 使用广泛。Clojure 的核心团队维护着一个维基网页，使用 Clojure 的机构可以广而告之：

<http://dev.clojure.org/display/community/Clojure+Success+Stories>

在这个页面里，你会发现：全球的知名公司像 Citicorp 和 Akamai 使用 Clojure；新创公司像 Backtype（现在是 Twitter）、The Climate Corporation（曾用名 Weatherbill）和 Woven 使用 Clojure；咨询公司像 Relevance 使用 Clojure；正牌的研究机构像马克斯·普朗克分子生物医学研究所使用 Clojure。

如果你的机构开始用 Clojure，会有很好的同伴。

562

审慎

不要把方枘放到圆凿里。像任何工具一样，Clojure 提供了许多东西，在把它用于能够让这些东西提供最大优势时，Clojure 的亮点才能尽显。尽管你可能非常喜欢 Clojure，想要经常用它，在游说之前还是应该确保它适合于你的机构和目标。

因而，如果你的公司（举例来说）用 C++ 来建嵌入系统，尝试说服你的战友用 Clojure

注 8：<irc://irc.freenode.net/clojure> 或在浏览器里访问 <http://webchat.freenode.net/?channels=%23clojure>。

注 9：本书作者之一曾在过去连续两年组织社区范围的调查，以衡量 Clojure 的来源、心态和优先事项；这个调查的最后一次全部结果可以在 <http://cemerick.com/2011/07/11/results-of-the-2011-state-of-clojure-survey/> 得到。

来替代原有的代码可能是不现实的。^{注10}

小步开始，缓步推进。有时候渐进手段是把新东西引入到已有系统的最好方法。因此，也许开始用 Clojure 写项目的全部测试代码或者用 Clojure 建次要或内部工具会容易些。在同事内部建立起使用 Clojure 成功的记录经常会比任何宣传或演示更能说服人们相信 Clojure 的能力和好处。

注 10：至少不是一夜之间的事，现在也不是时候。不过有人正在尝试把 Clojure 编译成更低层的语言适合作为“原生”程序发布。例如参见 <http://nakkaya.com/2011/06/29/ferret-an-experimental-clojure-compiler/>。

下一步？

如果你已经从本书学到了所有你能学到的东西——或者想看看一旦牢固掌握 Clojure 基础之后还有什么——你可能想看看这里列出的项目和资源。这是我们所知在 Clojure 世界里最酷的一些东西。有些会帮你更好地学习 Clojure，有些可能对你的“真实”项目有用，其他的太有趣了，至少得详细了解一次。

最后，因为 Clojure 是一门相对较新的语言，社区像疯了似的在成长、建设，新的、令人惊叹的东西不时冒出。如果你来到 <http://clojurebook.com>，我们将尽最大努力紧跟发展，给你优中选优。

(dissoc Clojure 'JVM)

另外有两个有良好支持的 Clojure 实现，目标是其他执行环境。如果你喜欢这里看到的一切，但需要部署到 JVM 之外，那么 ClojureCLR 或 ClojureScript 可能正是你所需要的。

ClojureCLR

ClojureCLR^{注1} 把 Clojure 移植到 .NET CLR 上。它不是幼稚的交叉编译，而是单独维护的，目的是提供与 CLR 寄主相同程度的紧密交互能力，就像 Clojure 为 JVM 所做的一样。由于 JVM 和 CLR 之间的各种差异，这意味着 ClojureCLR 提供了一些 Clojure 没有的功能。

ClojureCLR 还没有 Clojure 自身使用广泛，但它是成熟的、充分测试过的，对需要在 CLR 上部署应用程序又对 Clojure 的舒适恋恋不舍的人来说，它代表一个很棒的选项。

注 1： <https://github.com/clojure/clojure-clr>，维护者博客在 <http://clojureclr.blogspot.com>。

584 ClojureScript

ClojureScript (<https://github.com/clojure/clojurescript>) 则完全是另外一回事。它的目标是 JavaScript (准确地说，是 ECMAScript 3)，因而产生的代码可以在所有现代浏览器里运行，以及其他一些 JavaScript 执行环境，如 Node.js 和 CouchDB^{注2} 等，与 CoffeeScript 或 Dart 的方式差不多。这意味着你可以写 ClojureScript——使用宏、类型、协议、多重方法等——并把它部署到任何 JavaScript 平台上。

有两个警告我们需要指出来：

1. ClojureScript 还非常新。这个项目首次公开是在 2011 年，在本书印刷时不断有重大的修改。尽管如此，它基本上是相对稳定的，使用 ClojureScript 的项目和网站投入正式使用已经有好几个月了。
2. Clojure 和 ClojureScript 之间有一些重要的差异。这部分归因于 JavaScript 平台和 JVM 之间的巨大差异，也是因为 ClojureScript 是一个源到源的编译器：不是直接生成载入平台的字节码，ClojureScript 编译器必须生成 JavaScript 源代码。Clojure 和 ClojureScript 之间的差异总览可在 <https://github.com/clojure/clojurescript/wiki/Differences-from-Clojure> 找到。

ClojureScript 最初是由 Rich Hickey 在 2011 年夏天的一次 ClojureNYC 月度会上介绍的；^{注3} 那次介绍的视频在 <http://blip.tv/clojure/rich-hickey-unveils-clojurescript-5399498> 可以看到。

4Clojure

4Clojure (<http://4clojure.com>) 提供交互式 Clojure 编程的小挑战，从 Clojure 的基础知识到需要绞尽脑汁才能解决的问题都有。如果你完成了一个问题，可以看看其他玩家的解决办法。这已经获得了相当的支持，有数千人参与，数十万次问题被解决。

在看其他人的解答时，要记住“代码高尔夫”（译者注：一种娱乐性编程比赛，代码要尽量短）(https://en.wikipedia.org/wiki/Code_golf) 在 4Clojure 上是常见的活动。你在那儿看到的技巧和思路是很有价值的，只是不要把 4Clojure 的解当做权威的风格指南。

585 Overtone

Overtone^{注4} 是用 Clojure 写的一个开源音频环境，通过 SuperCollider^{注5} 合成服务器产生

注 2：ClojureScript 可以与一起用 Clutch 一起用，在第 15 章里使用得很多，以定义 CouchDB 里的视图：
<https://github.com/clojure-clutch/clutch-clojurescript>。

注 3：参见 <http://www.meetup.com/Clojure-NYC/>。

注 4：参见 <http://overtone.github.com>。

注 5：参见 <http://supercollider.sourceforge.net>。

声音和音乐。这让你用 Clojure 以非常高层的抽象来写乐器和合成器，然后把它们混合在一起（或者载入来自互联网的样例）成为随心所欲的完整作品。除了编程制作声音和音乐外，Overtone 还提供了对 TouchOSC、Monome 及其他硬件输入的支持，从而可以用手势和触摸进一步控制你用代码生成的声音，就像使用其他乐器一样。

Clojure Conj 2011 (<http://clojure-conj.org>) 有一个关于 Overtone 的演示，是由项目主程序员做的，我们极力推荐你看看，感受一下 Overtone 的历史和总目标：<http://blip.tv/clojure/sam-aaron-programming-music-with-overtone-5970273>。

core.logic

在 136 页“用 Clojure 的集合来小试牛刀”一节我们讨论了面向关系的编程。`core.logic` (<https://github.com/clojure/core.logic>) 启用一种形式的逻辑编程，称为关系式编程，从声明性来看，这将是下一步。关系式编程的其他例子包括 SQL 和 Prolog。`core.logic` 更倾向于后者，因为它实现了一种形式的 miniKanren，这是一种关系式编程语言，非常适合约束逻辑编程。

随着额外的声明性而来的是很棒的特性：一个 `core.logic` 目标（函数的关系式对应物）不区分参数与返回值，因而使得“逆向”运行成为可能，从而对一个已知的“返回”值可以得到一个（或几个）可能的参数。^{注6}

展示这种威力的最简单的例子是 `conso`，这是 `cons` 在关系式的对等物。因为目标（关系式函数）不区分参数与返回值，`conso` 把它的“返回值”作为额外的第三个即最后一个参数。我们来看看它的应用：

```
(use '[clojure.core.logic])
(run* [x] (conso 1 [2 3] x))          ●
:= ((1 2 3))
(run* [x]
  (fresh [_]
    (conso x _ [1 2 3])))           ●
:= (1)
(run* [x]
  (fresh [_]
    (conso _ 3 [1 2 3])))           ●
:= ((2 3))
(run* [q]
  (fresh [x y]
    (conso x y [1 2 3])))          ●
```

586

注 6：“返回”被引号括起来是为了提醒你在关系式编程里没有在面向对象或函数式编程里熟悉的返回值回事。

```
(== q [x y]))  
:= ((1 (2 3)))
```

- 先以“显而易见的”方式调用它，返回值是结果的列表，与求解一个等式的答案集或执行 SQL 查询的结果集相似。这个程序的唯一解答是 (1 2 3)。
- 当调用的前两个参数的值未知 (x 和 _) 时，而且只是询问第一个参数可能是什么值时，唯一的答案是 1。这样 `cons0` 可以像 `first` 那样来用。
- 这次只是返回第二个参数，唯一的答案是 (2 3)，因而 `cons0` 也可以用做 `rest`。
- 也可以同时返回前两个参数，有点像解构这个列表，如在 `[x & [y z]]` 里那样。

甚至像 `cons`（或者说 `cons0`）这样一个简单的函数都变得更有表达力、更强大，它打开了优雅地表达难题求解的新思路。

Pallet

在第 17 章里从头到尾过了一遍 Clojure Web 应用程序的基本部署过程，不过你的需求可能超越在那儿所展示的。Pallet (<https://github.com/pallet/pallet>) 是一个 Clojure 工具链，它以自动化的方式处理应用程序部署和计算基础设施管理中涉及的许多复杂挑战。虽然像 Chef 和 Puppet 这样的工具越来越常用于自动化系统配置，Pallet 的目标是提供它们的功能的超集——包括跨多个云服务商自动提供运行实例。^{注7}当然，作为一个 Clojure 工具，你使用 Clojure 代码通过 REPL 来配置、扩展 Pallet（及组成你的计算基础设施的硬件、服务器和进程）并与之交互，因而可以在服务提供、系统管理和配置管理等领域里利用这个语言的所有优势。

587 Avout

在第 4 章里所有关于并行的讨论都是集中在进程内的并行，在单个 Clojure 运行时里协调交替和平行的进程。Avout (<http://avout.io>) 是为分布式、多进程应用对 Clojure 的原子和 ref（因而包括后者的软件事务内存）的重新实现。它是建立在 Apache Zookeeper (<http://zookeeper.apache.org>) 的基础之上的，但是为可扩展性设计的，因而可以加入新的后台。如果 Clojure 的 STM 能吸引你，但你需要在不同应用间有事务式语义（或同一个应用程序运行于不同的进程里），那么 Avout 可能是天生的选择。

注 7：很大程度上多亏 jcclouds 函数库在数十种不同云服务之上提供了统一的抽象：<http://code.google.com/p/jclouds>。

Heroku 上的 Clojure

第 17 章描述了使用容器（如在 Amazon 的 Elastic Beanstalk 上的 Tomcat）部署 Web 应用程序的一个非常典型的方法，也还有一些方法不需要容器就可以运行 Clojure Web 应用，因而也不需要容器所暗示的打包工作。虽然这种“无容器的”部署选项在 JVM 领域是相对较新的方法，但已经越来越常见。目前最流行的一种是由 Heroku (<http://heroku.com>) 提供的，这是一种可缩放的应用程序部署平台，自身托管在 Amazon Web 服务上，现在直接支持用 Leiningen 部署的基于 Ring 的 Web 应用程序^{注 8}，不需要任何单独的编译或打包步骤。

Heroku 还有一个附加的好处是提供应用程序“插件”——管理数据库集群、消息队列、Web 服务等——你可以在自己的 Clojure 项目中配置、使用，不需要自己去安装和管理。这是一个专有平台，不过它可能为所有 Clojure 应用程序如何部署在无容器的未来铺平道路。

注 8：完整的过程和文档可参见 <http://devcenter.heroku.com/articles/clojure>。

索引[†]

Symbols

! (exclamation mark), 133
& (ampersand), 31
&env, 252–253
&form, 254–258
 arguments, 253
 macro error messages, 254
 user-provided type hints, 256–258
' (quote), 19, 24
* (asterisks), 202
"math-context", 432
"unchecked-math", 431
"warn-on-reflection", 440, 449
+ function, 322
, (commas), 19
-> and ->> in macros, 71, 259–262
. special form, 44
.jar file, 340, 579
.tar.gz file, 340
.war files
 about, 560
 Leiningen, 564
 Maven, 562
.zip, 340
::collection, 307
::as, 34
::body, 530
::character-encoding, 530
::content-length, 530
::content-type, 529
::error-handler, 214
::headers, 529, 530
::or, 34
::query-string, 530

::reload, 397
::remote-addr, 529
::request-method, 529
::scheme, 529
::server-name, 529
::server-port, 529
::status, 530
::uri, 529
= (equality), 434
@ (at sign), 160
^::const, 200
_ (underscore), 28, 329
_changes, CouchDB, 520
{...} (braces), 6
~@, 242
404 Not Found, 543
409 Conflict, 543
4Clojure, 584

A

abstract syntax tree, 9
abstractions, 84–114
 associative, 99–103
 collection abstractions, 292
 collections, 87

[†]: 索引所列页码为本书英文版页码, 请参照用“□”表示的原书页码。

stacks, 104
access, collections, 111–114
adapters, Ring, 531
add-watch, 177
advice, 467
agents, 209–224
 errors in agent actions, 212
 I/O, transactions and nested sends, 214–
 224
aget, 445, 446
Aleph, 532
algorithms, defined, 15
aliases, namespaces, 22
alter, 183, 184
alter-var-root, 207
amap, 446
Amazon, EB, 567–569
ampersand (&), 31
annotations, 381–385
 JAX-RS, 383
 JUnit tests, 382
anonymous classes, proxy, 372
anonymous functions versus function literals,
 64
anonymous types, reify, 284
AOP (aspect-oriented programming), 466–
 470
AOT compilation
 about, 337
 configuration, 349
 need for, 390
APIs
 dynamic scope, 202
 manipulating code, 232
 size of, 85
appendTo, 365
applications, versioning, 569
arbitrary implementation, 291
arbitrary precision versus bounded, 428–430
arbitrary-precision decimals ops, scale and
 rounding modes, 432
arbitrary-precision numbers, 17
areduce, 446
argument vectors, 369
arguments
 &form and &env, 253
 destructuring function arguments, 38–40
 keywords, 39
 mutable arguments and functions, 56
performance, 438
primitive types, 440
test arguments, 38
arithmetic functions, 78
arities
 functions, 37, 41
 Multimethods, 302
 performance, 67
ArrayList, 360
arrays
 classes, 444
 hints, 445
 Java, 370
 operations, 370
 primitive arrays, 442–449
 about, 443
 type hinting of multidimensional array
operations, 447
ArraySet, 295
artifactId, 340
artifacts, Maven dependency management
 model, 340
aset, 445, 446
aspect-oriented programming (AOP), 466–
 470
AspectJ, 467
assertions, testing, 486
assoc, 100, 127
associative abstraction
 data structures, 99–103
 records, 273
asterisks (*), 202
at sign (@), 160
atoms
 concurrency and parallelism, 174
 operations on, 192
attrs, 483
auto-gensym, 246
autoboxing, 437, 441
autopromoting operators, 429
auxiliary constructor, 275
averaging numbers, 5
Avout, 587
await, 211
await-for, 212

B

back references, 138
bare version numbers, 344

barging, 194
BigDecimal, 423, 432
BigInt, 423, 428
BigInteger, 423, 428
binding
 local bindings: let, 27
 multiple resources, 366
 value to their keys' names, 34
vectors, 250
with-redefs, 472
binding conveyance, 206
blocks, Ruby, 40
boilerplate, Hibernate, 507
booleans, 13
bounded versus arbitrary precision, 428–430
boxed decimals, 422
boxed integers, 422
braces {...}, 6
branching, Enlive, 549
browsing namespaces, 404
build solutions, 336–353
 compilation, 337
 dependency management, 339–344
 artifacts and coordinates, 340
 dependencies, 342
 repositories, 341
 tools and configuration patterns, 344–353
 AOT compilation configuration, 349
 Leiningen, 347
 Maven, 345
 mixed source projects, 351
built-in operators, 233

C

catch, 363
chain of responsibility, 463–466
character literals, 13
checked exceptions, 364
chunking, 168
class, 360
classes, 371–385
 annotations, 381–385
 JAX-RS, 383
 JUnit tests, 382
 arrays, 444
 deftype and dfrecord classes, 388–390
 hierarchies, 290
 inline and protocol implementations, 415
 Java, 357–360, 371

multimethods, 314
named classes, 374–381
proxy, 372
classpath
 JVM, 4
 limitations, 415
 namespaces, 331
Clojars.org, 341
Clojure
 collections and data structures, 83–157
 concurrency and parallelism, 159–226
 datatypes and protocols, 263–299
 design patterns, 457–470
 FP, 51–82
 future, 583–587
 general information about, 1–48
 introducing to your workplace, 577–582
 Java, 355–392
 macros, 229–262
 multimethods, 301–317
 nonrelational databases, 511–525
 numerics and mathematics, 421–455
 projects, 321–353
 relational databases, 491–509
 REPL (Read, Evaluate, Print, Loop), 393–417
 testing, 471–489
 type definition forms, 573
 web applications, 557–570
 web development, 527–555
clojure-maven-plugin, 347, 352
clojure-mode, 406
clojure.core, 323
clojure.java.jdbc, 491–497
 connection pooling, 496
 transactions, 496
 with-query-results, 494
clojure.lang.IEditableCollection, 132
clojure.repl, 400
clojure.set, 325
clojure.string, 325
clojure.test, 473–481
 defining tests, 474–476
 fixtures, 479
 test suites, 477
ClojureCLR, 583
ClojureScript, 584
clone-for, 550
Clutch, 512

code blocks: do, 25
code generation versus macros, 232
code-as-data, 9
codebases, functional organization of, 334
coercion functions, 441
collection abstractions, 292
collection literals, 19
collection types, destructuring, 28
collections and data structures, 83–157
 abstractions, 84–114
 associative, 99–103
 collections, 87
 indexed, 103
 sequences, 89–99
 set abstractions, 105
 sorted, 106–111
 stacks, 104
 collection access, 111–114
 idiomatic usage, 112
 keys and higher-order functions, 113
Conway’s Game of Life, 139–145
data structure types, 114–122
 lists, 114
 maps, 117
 sets, 117
 vectors, 115
identifiers and cycles, 137
immutability and persistence, 122–134
 structural sharing, 123–130
 transients, 130–134
macros, 235
maze generation, 145–151
metadata, 135
navigation, update and zippers, 151–156
performance, 437
command-line utility, 376
commas (,), 19
comments, 18
community, 580
commute, 185–189, 193
comp, 69
comparators, ordering, 107–111
compare, 107
compare-and-set, 174
compilation
 AOT, 337, 349
 macros, 230
 REPL, 4
Compojure
about, 536
routes, 539
compositionality, 68–76
 building a primitive logging system, 72–76
 higher-order functions, 71
concat, 242
concurrency and parallelism, 159–226
 about, 2
 agents, 209–224
 errors in agent actions, 212
 I/O, transactions and nested sends, 214–224
 agents and workloads, 217
 atoms, 174
 compared, 166
 coordination, 172
 delays, 160
 futures, 162
 Java’s concurrency primitives, 224
 notifications and constraints, 176–179
 validators, 178
 watches, 176–178
 parallelism on the cheap, 166
 promises, 163
 reference types, 129, 170
 refs, 180–198
 mechanics of ref change, 181–191
 STM, 180
 transactional memory, 191–198
 state and identity, 168
 synchronization, 172
 vars, 198–208
 defining, 198–201
 dynamic scope, 201–206
 forward declarations, 208
 versus variables, 206
concurrency forms, dynamic scope, 206
concurrency primitives, Java, 224
.concurrency-ref-mechanics, 215
cond, 42
conditionals: if, 42
configuration patterns, 344–353
configuration, AOT compilation configuration, 349
configure-view-server, 517
conj, 84, 87, 104, 114, 124
connection pooling, 496
cons, 93
constants, 200

constraints (see notifications and constraints)
constructors, factory functions, 275
constructs, redefining, 415
containerless approach, 560
contains?, 101
continue, 213
Conway’s Game of Life, 139–145
coordinates, Maven dependency management model, 340
coordination, concurrency and parallelism, 172
core.logic, 585
core.memoize, 80
CouchDB, 511–525
 about, 512
 CRUD, 512
 message queues, 522–525
 views, 514–520
 Clojure, 516–520
 JavaScript, 514–516
 _changes, 520
count, 88, 92
Counterclockwise, 403
CRUD, nonrelational databases, 512
CSS selectors, 549
curl, 542
cycles
 about, 138
 collections and data structures, 137
cyclic dependencies, 165

D

data
 Clojure programs represented as, 11
 functions as, 59
 persisting data in Hibernate, 506
data structures (see collections and data structures)
databases (see nonrelational databases; relational databases)
dataflow variables, 164
DataSource-based connection pooling, 496
datatype and protocols, 263–299
 about protocols, 264
 classes, 415
 collection abstractions, 292
 defining your own types, 270–280
 deftype, 277–280
 records, 272–277
extending existing types, 266–270
implementing protocols, 280–288
 inline, 281–285
 reusing implementations, 285–288
protocol dispatch edge cases, 290
protocol inspection, 289
 using Clojure from Java, 390
debugging
 macros, 237
 REPL, 411–414
 SLIME, 410
dec, 424
declarations
 about, 367
 forward declarations, 208
declarative concurrency, 164
declare functions, 438–442
declare macro, 208
def, 26, 416
defdb, 499
defining vars: def, 26
defmethod, 301
defmulti, 302, 415
defn-, 199
defonce, 396, 415, 506
defproject, 348
defrecord, 270, 272, 282, 374, 415
defroutes, 545
defsnippet, 551, 554
deftemplate, 552
deftype, 270, 277, 282, 283, 371, 374, 415
deftype class, 388–390
delays, 160
dependencies
 clojure.java.jdbc, 491
 cyclic dependencies, 165
 cyclic namespaces, 329
 interleaving source dependencies, 353
dependency injection, 459–462
dependency management, 339–344
 artifacts and coordinates, 340
 dependencies, 342
 repositories, 341
deployment, 341
deref, 160, 162, 171, 194
derive, 308
deserialization, 12
design patterns, 457–470
 AOP, 466–470

- chain of responsibility, 463–466
dependency injection, 459–462
strategy pattern, 462
destructuring, 28–36
 collection types, 28
 function arguments, 38–40
 map destructuring, 32–36
 sequential, 30–32
dfrecord classes, 388–390
Digital Subscriber Line (DSL) versus Korma, 500
disjunctions, 548
dispatch function
 about, 302
 multimethods, 305, 316
 multiple, 311
dissoc, 100, 274
do expressions, implicit, 26
do forms, implicit, 40
do->, 551
do: code blocks, 25
doall, 97, 168, 495
doc, 400
docstrings, 199
dorun, 97, 167
doseq, 91
dosync, 182, 192
doto, 360, 361
double, 427
Double box class, 423
double evaluation, macros, 249
downloading Clojure, 3
DSL (Digital Subscriber Line) versus Korma, 500
dynamic expression problem, 263
dynamic redefinition, 397
dynamic scope, 201–206
 concurrency forms, 206
 visualizing, 203
- E**
- each, 479
earmuffs, 202
EB (Elastic Beanstalk), 567–569
Eclipse, 403–405
editing
 Clojure editing support, 403
 source code, 402
efficiency (see performance)
- Emacs, 405–411
 clojure-mode and paredit, 406
 inferior-lisp, 406
 SLIME, 408
empty, 87
Enlive, 546–554
 about, 547
 iterating and branching, 549
 selectors, 548
 using, 551
ensure, 198
environments, 569
equality and equivalence, 433–436
 numeric equivalence, 435
 object identity, 433
 reference equality, 434
equals, 283
ERB templates
 templating language, 545
 using, 552
error handling
 agents, 212, 213
 Java, 362–366
 macros, 234, 254
escape hatch, 360
escaping checked exceptions, 364
eval
 about, 46
 Ruby eval versus Clojure macros, 234
Evaluate, 20
evaluating
 suppressing: quote, 24
 symbols, 23
event types, hierarchies, 518
exceptions
 agents, 212
 Java, 362–366
 throwing, 363
 try and throw, 45
types
 custom, 378
 reusing, 363
exclamation mark (!), 133
expanding macros, 237
expression problem, 263
expressions
 about, 7
 code blocks, 25
 regular expressions, 17

values, 54
extend, 285
extend-protocol, 266
extend-type, 266, 282
extenders, 289
extends?, 289
extensibility, Emacs, 408
extra-positional sequential values, 31

F

factory functions
 constructors, 275
 protocols and vectors, 268
failed agents, 212
fallbacks, 194
false, 114
false values, 103
false?, 42
fields
 immutable fields, 278
 Java, 357–360
 mutable fields, 278
 object fields, 359
files, namespaces, 328–331
fill-dispatch, 308
filter, 113
filter functions, 521
finally, 363, 364
find, 102
find-doc, 400
fire-and-forget persistence mechanism, 217
first-class functions, 59–68
fixnums, 433
fixtures, 479
flow, 398
fn, 36–41, 487
for, 88
form-level comments, 18
forms, 23
 (see also special forms)
 comment forms, 18
 concurrency forms, 206
 expanding nested forms, 239
 println forms, 18
 type definition forms, 573
forward declarations, 208
FP (functional programming), 51–82
 about, 52
 compositionality, 68–76

building a primitive logging system, 72–76
higher-order functions, 71
first-class and higher-order functions, 59–68
pure functions, 76
values, 52–59
 about, 53
 comparing to mutable objects, 54–58
 unfettered object state, 58

frequencies, 442
function application, 65–68
function composition, 69
function literals
 versus anonymous functions, 64
 versus partial literals, 67

functions, 51
 (see also FP)
 anonymous functions versus function literals, 64
arithmetic functions, 78
arity, 41
coercion functions, 441
collections and, 111
collections keys, 112
constructors and factory functions, 275
creating functions: fn, 36–41
declare functions, 438–442
dynamic scope, 204
factory functions, 268
filter functions, 521
first class and higher order functions, 59–68
first class values, 61
indexed-step functions, 139
keys and collections, 113
multiple arguments, 36
multiple arities, 37
mutable arguments, 56
mutually recursive, 37
nesting literals, 41
primitive types, 440
protocols, 264
pure functions, 78
rest arguments, 31
sequential collections, 29
side-effecting functions, 80
single-arity functions, 38
symbols, 23

testing, 471
variadic functions, 38, 90
versus macros, 232
versus records, 277

Futon, 515
future, 583–587
 4Clojure, 584
 about, 162
 Avout, 587
 ClojureCLR, 583
 ClojureScript, 584
 core.logic, 585
 Heroku, 587
 Overtone, 585
 Pallet, 586

G

Game of Life, 143
games
 concurrency-ref-mechanics, 215
 ref change, 181
gen-class, 371, 375–381, 415
gensyms, macros, 246
get, 101, 102, 103
GET, 536
group-by, 119
groupId, 340

H

handlers, Ring, 532
handling exceptions: try and throw, 45
hash-map, 118
hash-set, 117
hashmaps, 14
head retention, 98
Heroku, 587
heterogeneous arguments, 426
hexadecimal notation, 16
Hibernate, 503–509
 boilerplate, 507
 persisting data, 506
 queries, 506
 setup, 503–506
Hiccup, 481
hierarchies, 304–311
 classes, 290
 event types, 518
higher-order functions, 59–68

hints

arrays, 445
type hints
 macros and &form, 256–258
 multidimensional array operations, 447
history, 195
homogeneously typed arguments, 426
homoiconicity, 7, 9–12, 230
HTML DSL, 481–485
HTML templates, Leiningen, 552
html-snippet, 547
hygienic macros, 244

I

I/O, agents, 214–224
identifiers, collections and data structures, 137
identities
 about, 138
 concurrency and parallelism, 168
 object identity, 433
identity, 2
IDEs (Integrated Development Environments), 398
if, 42
if-let, 42
IFn, 438
immutability, 122–134
immutable fields, 278
immutable functions, testing, 471
immutable objects, 52
immutable values, 52
import, 326
in-ns, 322
indexed abstractions, 103
indexed-step functions, 139
indices
 destructuring, 33
 issues with, 138
 vectors, 29
inferior-lisp, 406
infix operators, 9
inheritance
 limitations, 458
 multimethods, 313
inline implementation of protocols, 281–285
 example, 280
 Java interfaces, 282
 reify, 284

inline interfaces, classes, 415
inner classes, 327
inner map destructuring, 33
insert-records, 493
inspecting protocols, 289
inspector, 409
instance field access, 45
instance method
 calls, 9
 invocation, 45
instance?, 360
integers, 54
Integrated Development Environments (IDEs),
 398
interaction styles, REPL, 6
interactive development, 393–398
interfaces
 inline implementation of Java interfaces,
 282
 Java classes, 371–385
 using Clojure from Java, 390
 versus protocols, 264
interleaving source dependencies, 353
interop forms, 357
interop utilities
 Java, 360
Interop Zone, 574
interoperability
 Java, 44
 Java and JVM, 355
 numeric primitives, 16
into, 131
into-array, 444
introducing Clojure to your workplace, 577–
 582
 community, 580
 facts, 577
 productivity, 579
 prudence, 582
introspecting
 multimethods, 314
 namespaces, 401
invariants, 59
invoke, 438
invokePrim, 439
is, 473
isa?, 307
isolated mutation of local arrays, 442
isolation, 181

iterating, Enlive, 549
iterators
 about, 458
 versus sequences, 91

J

Java, 355–392, 557–565
 abstraction, 85
 arrays, 370
 classes and interfaces, 371–385
 annotations, 381–385
 named classes, 374–381
 proxy, 372
 classes, methods and fields, 357–360
 Clojure’s foundation, 356
 concurrency primitives, 224
 dependency injection, 459
 exceptions and error handling, 362–366
 escaping checked exceptions, 364
 with-open and finally, 364
 inline implementation of protocols with
 Java interfaces, 282
 interfaces, 282
 interop forms equivalents, 357
 interop utilities, 360
 interoperability, 44
 maps, 85
 mutability, 360
 servlet filters, 535
 type hinting, 366–370
 using Clojure from Java, 385–392
 deftype and defrecord classes, 388–390
 protocol interfaces, 390
web application packaging, 560–565
 .war files with Leiningen, 564
 .war files with Maven, 562
 web architecture, 558
 wildcard import, 327
java.io.Serializable, 291
java.lang, 327
java.lang.Integer, 55
java.lang.Runnable, 225
java.util.ArrayList, 115
java.util.Collection, 290
java.util.concurrent, 224, 225
java.util.concurrent.Callable, 225
java.util.List, 29, 290
java.util.Map, 291
JavaScript, views, 514–516

- JAX-RS annotations, 384
JAX-RS web service endpoints, 383
JMX (Java Management Extensions), 414
JSON, 513
JUnit tests, 382
JVM (Java Virtual Machine)
 Clojure hosted on, 2
 reusing investment in, 578
- K**
- keys, 118
 - binding values, 34
 - collections, 112
 - destructuring, 33
 - keywords, 14
 - arguments, 39
 - associative collections, 273
 - as functions, 112
 - hierarchy, 306
 - Korma, 498–503
 - queries, 499
 - using, 498
 - versus DSL, 500
- L**
- lambdas, Python, 40
 - lazy seqs, 93–98
 - lazy-seq, 90, 93
 - lein compile, 353
 - lein-ring, 564
 - Leiningen, 347
 - AOT compilation, 350
 - compilation, 352
 - HTML templates, 552
 - .war files, 564
 - web apps, 566
 - let, 27
 - letfn, 37
 - lexical scope, 201
 - libraries, dynamic scope, 202
 - Library Coding Standards style guide, 120
 - libspect, 324
 - LIFO (last-in, first-out), stacks, 104
 - LinkedHashMap, 372
 - Lisp
 - and Clojure, 2
 - special forms, 24
 - list function, 239, 241
- list*, 93
lists
 - about, 8
 - data structure type, 114
 - quote ('), 19
 - quoting, 25
 - structural sharing, 124
 - versus sequences, 92
- literals (see collection literals; scalar literals)
- live lock, 194
- local arrays, isolated mutation of, 442
- local bindings
 - destructured value, 32
 - let, 27
- local consistency, validators, 189
- locals, destructuring, 30
- locking
 - concurrency primitives, 225
 - primitives: monitor-enter and monitor-exit, 45
- logging
 - building a primitive logging system, 72–76
 - databases, 522
 - states, 216
 - write-behind log, 215–217
- Long box class, 423
- loop special form, 28
- loops
 - loop and recur, 43
 - replacing, 140
- LRU cache, 372
- lucene-core, 340
- lucene-queryparser, 340
- M**
- macroexpand-1, 237
 - macroexpand-all, 239
 - macroexpansion, 231
 - macros, 229–262
 - > and ->>, 259–262
 - about, 229–235
 - versus functions, 232
 - versus Ruby eval, 234
 - what macros are not, 231
 - comment macros, 18
 - debugging, 237
 - getting started, 235
 - hygiene, 244–250
 - double evaluation, 249

gensyms, 246
names, 248
idiom and patterns, 250
implicit arguments, 251–259
 &env, 252–253
 &form, 254–258
 testing contextual macros, 258
redefining, 415
syntax, 239–242
 quote versus syntax-quote, 240
 unquote and unquote-splicing, 241
when to use, 243
make-array, 444
make-hierarchy, 308
Mandelbrot Set, 449–455
many-to-one relationships, 499
map function, 62, 446
maps
 collections, 28
 data structure type, 117
 destructuring, 32–36
 Java, 85
 metadata maps and vars, 199
 nested maps, 121
 structural sharing, 125
 transient variants, 132
 when to use, 277
“math-context”, 432
mathematics (see numerics and mathematics)
matrices, map destructuring, 33
Maven
 AOT compilation, 350
 clojure-maven-plugin, 347, 352
 layout conventions, 332
 repositories, 341
 version range formats, 343
 .war files, 562
 web apps, 565
Maven dependency management model, 339–
 344
about, 345
artifacts and coordinates, 340
dependencies, 342
 repositories, 341
maze generation, 145–151
memoization, 79
memory (see STM: transactional memory)
message queues, 522–525
metadata
&form, 254
about, 20
agents, 222
annotations, 381
associative collections, 274
collections and data structures, 135
constants, 200
docstrings, 200
macros, 256
maps, 199
multimethods, 315
realized?, 161
types, 278
var names, 370
metaprogramming, 577
methodName, 265
methods
 functions, 264
 Java, 357–360
 protocols, 266
middleware
 Ring, 534
 using, 465
Midje, 473
mixins, 286
mocking objects, 472
modes
 agent error handlers, 213
 scale and rounding modes for arbitrary-
 precision decimals ops, 432
monitor-enter, 45
monitor-exit, 45
monitoring REPL, 411–414
monkey-patching, 263
multidimensional arrays
 performance, 446
 type hinting, 447
multimethods, 291, 301–317
 about, 301–303
 hierarchies, 304–311
 inheritance, 313
 introspecting, 314
 multiple dispatch, 311
 range of dispatch functions, 316
 redefining, 415
 type versus class, 314
multiplayer games, 181
multiple arities, 37
multitenancy, 558

mutable fields, 278
mutable objects
 comparing to values, 54–58
 unfettered object state, 58
mutable state, 52
mutations
 primitive arrays, 445
 reference types, 28
mutually recursive functions, 37

N

named classes, 374–381
names, macros, 248
namespace-global identity, 198
namespaced keywords, 14
namespaced symbols, 15
namespaces, 322–332
 about, 20
 browsing, 404
 codebases, 334
 files, 328–331
 hierarchies and multimethods, 306
 introspecting, 401
 macros, 255
 projects, 322–328
 protocols, 265, 268
 types, 271
 vars, 198
natural keys versus synthetic keys, 137
nested collections, accessing values in, 29
nested forms, expanding, 239
nested maps, reduce-by, 121
nested send, 215
nested vectors, destructuring, 31
nesting, function literals, 41
networks, security, 414
new special form, 44
next, 90
nil, 13, 102, 114
nonrelational databases, 511–525
 CouchDB and Clutch, 512
 CRUD, 512
 message queues, 522–525
 views, 514–520
 Clojure, 516–520
 JavaScript, 514–516
 _changes, 520
notifications and constraints, 176–179
 validators, 178

watches, 176–178
nREPL, 404
ns, 327
ns-aliases, 401
ns-imports, 401
ns-interns, 401
ns-map, 401
ns-publics, 401
ns-refers, 401
ns-unalias, 401
ns-unmap, 401, 476
nth, 104, 115
number literals, 15
numbering states, 137
numeric literals, 16
numeric primitives, interoperability, 16
numerics and mathematics, 421–455
 equality and equivalence, 433–436
 numeric equivalence, 435
 object identity, 433
 reference equality, 434
 Mandelbrot Set, 449–455
 mathematics, 427–432
 bounded versus arbitrary precision, 428–430
 scale and rounding modes, 432
 unchecked ops, 430
 numerics, 421–427
 mixed numerics model, 422
 numeric contagion, 425
 rationals, 424
 representations, 422
 optimizing numeric performance, 436–449
 declare functions, 438–442
 primitive arrays, 442–449

0

objects
 fields, 359
 identity, 433
 instantiation, 45
 mocking, 472
 mutable objects
 comparing to values, 54–58
 unfettered object state, 58
 types, 424
 unfettered object state, 58
octal notation, 16
once, 479

operations, arrays, 370
operators, 7
opt-in computation, 161
or, 256
ordering, comparators and predicates, 107–111
outer map destructuring, 33
OutputStream, 214
overloading, 301
overriding local binding, 28
Overtone, 585

P

packaging, 340
Pallet, 570, 586
parallelism (see concurrency and parallelism)
parameterized queries, 494
paredit, 402, 403, 406
parentheses, 6
partial literals versus function literals, 67
partition, 141
patching, 413
path segments, 540
peek, 104
performance, 436–449
 declare functions, 438–442
 higher order functions and arities, 67
immutable data structures, 123
persistent data structures, 130
pmap, 167
primitive arrays, 442–449
 about, 443
 type hinting of multidimensional array operations, 447
 type hinting, 366–370
persistence, 122–134
 data in Hibernate, 506
 reference states, 215–217
 structural sharing, 123–130
 benefits, 129
 lists, 124
 maps, vectors and sets, 125
 transients, 130–134
plug-ins
 about, 3
 Leiningen, 347
 Maven, 347
pmap, 167, 206
pointcuts, 467

polymorphism, 84, 277
pop, 104, 114
POST, 542
postconditions, assertions, 487
postwalk, 236
pre- and postconditions, 40
precedence, 7
preconditions, assertions, 487
predicates
 ordering, 107–111
 testing, 549
 turning into comparators, 107
prefer-method, 313
preferences, multiple inheritance, 313
prime operators, 430
primitives
 64-bit integers, 422
 arrays, 442–449
 about, 443
 type hinting of multidimensional array operations, 447
 concurrency primitives, 224
 declare functions, 438–442
 locking: monitor-enter and monitor-exit, 45
 numeric primitives interoperability, 16
 numerics, 422
 performance, 437
 types, 424
println forms, 18
private vars, 198
productivity, 579
program state, 52
project.clj, 348
projects, 321–353
 build solutions, 336–353
 compilation, 337
 dependency management, 339–344
 tools and configuration patterns, 344–353
codebases, 334
layout conventions, 332
namespaces, 322–332
 classpath, 331
 files, 328–331
promises, 163
protocols (see datatypes and protocols)
proxy, 371
pure functions, 76

PUT, 536, 542
Python
 destructuring and unpacking, 30
 lambdas, 40
 numeric types, 422
PYTHONPATH, 331

Q

queries
 Hibernate, 506
 Korma, 499
queues, message queues, 522–525
quote (`), 19, 24
quote versus syntax-quote, 240

R

random numbers, 77
ranges of versions, 343
rasterization, 453
rational numbers, 17
rationals, 422, 424
Read, 20
reader, 12–20
 collection literals, 19
 comments, 18
 scalar literals
 booleans, 13
 characters, 13
 keywords, 14
 nil, 13
 numbers, 15
 regular expressions, 17
 strings, 13
 symbols, 15
 syntax, 20
 transactional memory, 194
 whitespace and commas, 19
realized?, 161
records, 272–277–280
 constructors and factory functions, 275
 when to use, 277
recur special form, 28, 43
recursive functions, letfn, 37
redefining
 constructs, 415
 macros, 415
 multimethods, 415
redirect, 539

reduce
 about, 63
 Conway's Game of Life, 140
 CouchDB, 515
 over arrays, 446
reduce-by, 121
ref-history-count, 195
ref-max-history, 195
ref-min-history, 195
ref-set, 189
refactoring, 276, 329
refer, 323
reference equality, 434
reference states, 215–217
reference types
 about, 2, 170
 coordinated and asynchronous semantics, 173
 mutation semantics, 28
 using, 537
referential transparency, 79
refs, 180–198
 mechanics of ref change, 181–191
 alter, 183
 commute, 185–189
 ref-set, 189
 validators, 189
STM, 180
transactional memory, 191–198
 readers may retry, 194
 side-effecting functions, 192
 transaction scope, 193
 write skew, 196
regular expressions, 17
reify
 anonymous types, 284
 Java classes, 371
relational databases, 491–509
 clojure.java.jdbc, 491–497
 connection pooling, 496
 transactions, 496
 with-query-results, 494
Hibernate, 503–509
 boilerplate, 507
 persisting data, 506
 queries, 506
 setup, 503–506
Korma, 498–503
 queries, 499

using, 498
versus DSL, 500

releases, 342
remove, 114
remove-ns, 402
render-image, 453
render-text, 451
REPL (Read, Evaluate, Print, Loop), 3–6, 393–417
agents, 211
classpath, 332
debugging, monitoring, and patching production, 411–414
interactive development, 393–398
monitoring, 412
multimethods, 307
namespaces, 322
redefining constructs, 415
tooling, 398–411
bare REPL, 399–402
Eclipse, 403–405
Emacs, 405–411
transactions, 194
uploading via remote REPLs, 566
web applications, 532

REPL-bound vars, 399

repositories, Maven dependency management model, 341

representations, numerics, 422

require, 324

requirements, 3

responsibility, chain of responsibility, 463–466

rest, 31, 35, 38, 90, 114, 125

restarting agents, 213

result set, 495

retain, 539

return values, performance, 438

reusing exception types, 363

Ring, 529–535
adapters, 531
handlers, 532
middleware, 534
requests and responses, 529
web app architecture, 559

ring-httpcore-adapter, 531

ring-jetty-adapter, 531

Robert Hooke library, 468

root bindings, 201, 207

rounding modes, arbitrary-precision decimals ops, 432

routing requests, 535–545

rseq, 106

rsubseq, 106, 110

Ruby
blocks, 40
ERB templating language, 545
eval versus Clojure macros, 234
lists and hashes, 85
numeric types, 422
strings, 56

runtime
analysis, 412
compilation, 337
dynamic redefinition, 397

S

satisfies?, 289

scalar literals, 13–17
booleans, 13
characters, 13
keywords, 14
nil, 13
numbers, 15
regular expressions, 17
strings, 13
symbols, 15

scale, arbitrary-precision decimals ops, 432

scope
dynamic scope, 201–206
concurrency forms, 206
visualizing, 203
transactions, 193

security, networks, 414

select, 499

selectors, Enlive, 548

semantics
reference types, 173
reference types and mutation semantics, 28
value semantics, 272

send, 209

send-off, 209

sends, agents and nested sends, 214–224

sequences, 89–99
creating seqs, 92
head retention, 98
lazy seqs, 93–98
lists, 115

performance, 437
seq, 84
transients, 133
versus iterators, 91
versus lists, 92
sequential collections, 28
sequential destructuring, 30–32, 97
serializable snapshot isolation, 185
serialization, 13
servlets
 about, 557
 filters, 535
 Ring, 531
session factories, 505
set abstractions, 105
set function, 117
set!, 45, 206, 359
sets
 data structure type, 117
 structural sharing, 125
sharing (see structural sharing)
shorten!, 538
side effects
 defined, 76
 lazy sequences, 98
side-effecting functions, 80, 192
single-arity functions, 38
single-segment namespaces, 330
SLIME, 407, 410
slots, 118
snapshots, 342
sniptest, 547, 551
software transactional memory (see STM)
sorted abstractions, 106–111
sorted-map, 108
sorted-map-by, 108
sorted-set, 108
sorted-set-by, 108
source code
 printing, 400
 structural editing, 402
SPEC, 529
special forms, 23–45
 code blocks: do, 25
 conditionals: if, 42
 creating functions: fn, 36–41
 destructuring function arguments, 38–40
 defining vars: def, 26
destructuring, 28–36
 map destructuring, 32–36
 sequential, 30–32
exception handling: try and throw, 45
Java interoperability: . and new special forms, 44
local bindings: let, 27
looping: loop and recur, 43
primitive locking: monitor-enter and monitor-exit, 45
referring to vars: var, 44
suppressing evaluation: quote, 24
versus macros, 233
splicing, unquote and unquote-splicing, 241
split-with, 99
SQLite, 492
stacks
 abstractions, 104
 Clojure stack, 527
 space, 43
states
 concurrency and parallelism, 2, 168
 logging, 216
 numbering, 137
 reference states, 215–217
 vars, 198
static field access, 45
static methods
 calls, 9
 gen-class, 376
 invocation, 45
stepper, 145
STM (software transactional memory)
 about, 180
 agents, 214
strategy pattern, 462
strings
 about, 13
 Ruby, 56
struct map, 272
structs, maps as ad-hoc structs, 118
structural editing, source code, 402
structural sharing, 123–130
 benefits, 129
 lists, 124
 maps, vectors and sets, 125
subseq, 106, 110
suppressing evaluations: quote, 24
swap!, 174

symbols
about, 15
evaluation of, 8, 23
functions, 23, 112
hierarchy, 306
macros, 248

synchronization
agents, 214
concurrency and parallelism, 172

syntax
about, 7
for destructuring, 29
macros, 239–242
reader, 20

syntax-quote versus quote, 240

syntax-quoted lists, 242

syntax-quoting, 240

synthetic keys versus natural keys, 137

T

templating, 545–554
Enlive, 546–554
about, 547
iterating and branching, 549
selectors, 548
using, 551

temporaryOutputDirectory, 350

testing, 471–489
&env, 253
assertions, 486
clojure.test, 473–481
defining tests, 474–476
fixtures, 479
test suites, 477

contextual macros, 258

HTML DSL, 481–485

immutable values and pure functions, 471

JUnit tests, 382

mixed floating point equality tests, 436

pure functions, 79

thread pools
agents, 210
defined, 225

threading macros, 259

threads
agents, 210
dynamic scope, 206

throw, 45

throwing exceptions, 363

time-it, 468

transactional memory, 191–198
readers may retry, 194
side-effecting functions, 192
transaction scope, 193
write skew, 196

transactions
agents, 214–224
clojure.java.jdbc, 496
commute, 185–189
conflicts, 185
modifications, 182
scope, 193

TRANSACTION_SERIALIZABLE, 496

transients, immutability and persistence, 130–134

transitive dependencies, 342

trap door, 360

trees, data structures and persistent semantics, 128

true?, 42

try, 45, 363

try-with-resources, 365

tuples, as vectors, 116

type definition forms, 573

type hints
macros and &form, 256–258
multidimensional array operations, 447
performance, 366–370

types, 263
(see also datatypes and protocols; reference types)
about, 2
errors, 440
multimethods, 314
numerics, 422

U

unary operators, 9

unchecked ops, 430

unchecked-*, 431

unchecked-math, 431

uncoordinated operations, 172

underscore (_), 28, 329

unfettered object state, 58

Unix, classpath, 332

unpacking, 30

unquote and unquote-splicing, 241

unquote-splicing operator, 242

uploading via remote REPLs, 566
URL shortener, 544
use, 324

V

validators

enforcing local consistency, 189
notifications and constraints, 178

vals, 118

value semantics, 272

values, 52–59

about, 53, 138

comparing to mutable objects, 54–58

program state, 52

unfettered object state, 58

versus vars, 416

variables

dataflow variables, 164

versus vars, 21, 206

variadic functions, 38, 90

vars, 198–208

defined, 21

defining, 26, 198–201, 251

constants, 200

docstrings, 199

private vars, 198

defonce, 396

dynamic scope, 201–206

concurrency forms, 206

visualizing, 203

forward declarations, 208

REPL-bound vars, 399

symbols, 20, 44

versus values, 416

versus variables, 206

vary-meta, 135

vector?, 116

vectors

argument vectors, 369

bindings, 250

data structure type, 115

defined, 115

HTML example, 482

indices, 29, 100

nested, 31

nth, 104

structural sharing, 125

transient variants, 132

versioning

about, 129

applications, 569

snapshots and release versions, 342

version string, 341

views, 514–520

Clojure, 516–520

JavaScript, 514–516

W

warn-on-reflection, 440, 449

warnings, primitives, 440

warnOnReflection, 350

watches, notifications and constraints, 176–178

web, 527–555, 557–570

beyond simple web application deployment, 570

Clojure stack, 527

Java, 557–565

web application packaging, 560–565

Ring, 529–535

adapters, 531

handlers, 532

middleware, 534

requests and responses, 529

routing requests, 535–545

templating, 545–554

Enlive, 546–554

web application deployment, 566–569

EB, 567–569

web apps, 565

web app, 559

web crawlers, 218

when-let, 42

where, 500

white space, 19

why Clojure, 1

wildcard import, 327

Wilson's algorithm, 149

with, 499

with-connection, 496

with-meta, 135

with-open, 364, 507

with-precision, 432

with-query-results, 493, 494

with-redefs, 208, 472

workflow, Emacs, 407

workloads, agents, 217–224

write skew, 196

write-behind log, 215–217

Z

- zero-arg arity, 38
- zippers, 151–156
 - Ariadne's, 154
 - custom, 153
 - manipulating, 152

关于作者

Chas Emerick 从 2008 年年初开始一直活跃于 Clojure 社区。他曾为 Clojure 核心语言作出贡献，参与了数十个 Clojure 开源项目，经常作关于 Clojure 和软件开发方面的写作和演讲。

Chas 维护着 Clojure Atlas (<http://clojureatlas.com>)，这是 Clojure 语言和标准函数库的一个交互式可视化和辅助学习工具。

作为 Snowtide (<http://snowtide.com>，在马萨诸塞州西部的一个小软件公司) 的创始人，Chas 的主要领域是无结构数据提取，特别专长于 PDF 文档相关方面。Chas 在 <http://cemerick.com> 上的写作涉及 Clojure、软件开发、企业活动及其他爱好。

Brian Carper 是从 Ruby 程序员转变成一名 Clojure 爱好者的。他从 2008 年开始用 Clojure 编程，在家里和工作上使用 Clojure 编写从 Web 开发到数据分析，到 GUI 应用程序的一切东西。

Brian 是 Gaka (<https://github.com/briancarper/gaka>，一种 Clojure 到 CSS 的编译器) 和 Oyako (<http://github.com/briancarper/oyako>，一种对象 - 关系映射函数库) 的作者。他在 <http://briancarper.net> 的写作涉及 Clojure 和其他的话题。

Christophe Grand 是函数式编程的长期爱好者，失落在 Java 王国，在 2008 年年初接触到 Clojure 时真是一见钟情！他创作了 Enlive (<https://github.com/cgrand/enlive>，一种 HTML/XML 转换、提取、模板函数库)、Parsley (<https://github.com/cgrand/parsley>，一种渐进式解析器生成器) 和 Moustache (<https://github.com/cgrand/moustache>，一种用于 Ring 的路由和中间件应用程序 DSL)。

作为独立咨询师，他开发 Clojure、辅导并提供 Clojure 方面的培训。他也在 <http://clj-me.cgrand.net> 写作一些关于 Clojure 的文章。

封面介绍

本书的封面动物是彩鹬。彩鹬（彩鹬科）有三种：大彩鹬、澳洲彩鹬和南美洲彩鹬。

这些沼泽鸟类与真正的鹬科不同，如它们的名称所暗示的，彩鹬科的毛色更为鲜亮。它们与水雉或矶鹬亲缘关系更近。彩鹬生活在沼泽和各种湿地，取食植物种子、大米、小米、昆虫、蜗牛和甲壳动物。它们除了繁殖季节，终日独居，躲藏起来，所以难得一见。

大彩鹬生活在非洲、印度和东南亚。澳洲彩鹬长期被认为是一个亚种，发现于澳大利亚，被认定为濒危物种。这两种彩鹬表现出不寻常的性二形性，雌鸟比雄鸟更大、毛色更鲜亮，而雄鸟负责孵化并养育幼鸟。

南美洲彩鹬分布在南美洲南部。它们与其他彩鹬的区别是趾上有蹼。南美洲彩鹬只与一只配偶交配，没有表现出大彩鹬和澳洲彩鹬那样的性二形性。在智利和阿根廷，人们捕猎它们为食。

封面图片来自 *Riverside Natural History* 一书。