The Pragmatic Programmers

新·锐·编·程·语·言·集·萃·

Programming Clojure

Second Edition

Clojure程序设计

【美】Stuart Halloway
Aaron Bedra 著

温瑞云 译

- Clojure编程经典之作
- Clojure之父作序推荐

人民邮电出版社
POSTS & TELECOM PRESS

The Pragmatic Programmers

·新·锐·编·程·语·言·集·萃·

Programming Clojure

Second Edition

Clojure程序设计

【美】Stuart Halloway
Aaron Bedra 著

温瑞云 译

人民邮电出版社

北京

图书在版编目（CIP）数据

Clojure程序设计 / (美) 哈罗威 (Halloway,S.), (美) 拜卓 (Bedra,A.) 著 ; 温瑞云译. -- 北京 : 人民邮电出版社, 2013.4
ISBN 978-7-115-30847-4

Ⅰ. ①C… Ⅱ. ①哈… ②拜… ③温… Ⅲ. ①程序设计 Ⅳ. ①TP311.1

中国版本图书馆CIP数据核字(2013)第010005号

版 权 声 明

Clojure 程序设计

- 著　［美］Stuart Halloway　Aaron Bedra
  译　温瑞云
  责任编辑　陈冀康
- 人民邮电出版社出版发行　北京市崇文区夕照寺街 14 号
  邮编　100061　电子邮件　315@ptpress.com.cn
  网址　http://www.ptpress.com.cn
  北京天宇星印刷厂印刷
- 开本：800× 1000　1/16
  印张：17.5
  字数：330 千字　2013 年 4 月第 1 版
  印数：1– 3 000 册　2013 年 4 月北京第 1 次印刷

著作权合同登记号　图字：01-2012-4614 号

ISBN 978-7-115-30847-4

定价：49.00 元

读者服务热线：(010)67132692　印装质量热线：(010)67129223
反盗版热线：(010)67171154
广告经营许可证：京崇工商广字第 0021 号

内容提要

Clojure 是一种基于 Java 虚拟机的动态编程语言。它功能强大，支持函数式编程，简化了并发编程，并且能调用 Java 程序。正是这些优点，使其赢得了众多程序员的青睐。

本书是介绍 Clojure 编程语言和程序设计的经典之作。第 2 版针对 Clojure 1.3 进行了全面的更新。全书共包括 10 章，分别介绍了 Clojure 概览和基本特性、序列和函数式编程、并发编程模型、协议和数据类型、宏、多重方法，以及 Clojure 对 Java 的调用。最后提供了一个完整了解 Clojure 应用开发全过程的实例。

本书适合有不同语言背景而想要学习和了解 Clojure 编程的程序员阅读。函数式程序员、Java 和 C#程序员、Lisp 程序员，以及 Perl、Python 和 Ruby 程序员，都能够通过阅读本书得到不同程度的收获。

献给 Craig Bedra，我的父亲和导师。

是你让我懂得了通过探索进行学习的价值，

并且让我明白诸如魔法一类的事情是不存在的。

——Aaron

致谢

有许多人为本书那些精彩的部分做出了贡献。残留的问题和错误完全是由我们造成的。

感谢 Relevance 和 Clojure/core 这两个了不起的团队，是你们创造性的氛围，让好点子得以茁壮成长。

感谢 Clojure 邮件列表[①]里友善的人们，是你们给予了我们莫大的帮助和鼓励。

感谢 Pragmatic Bookshelf 的所有人。尤其是我们的编辑 Michael Swaine，为我们有些狂妄的交付时间表提供了很好的建议。感谢 Dave Thomas 和 Andy Hunt，你们创建了一个有趣的技术书籍编写平台，它令书籍作者热情满怀。

感谢所有向本书勘误表页面[②]投递建议的人们。

感谢技术评审们的意见和建议，你们是 Kevin Beam、Ola Bini、Sean Corfield、Fred Daoud、Steven Huwig、Tibor Simic、David Sletten、Venkat Subramaniam 和 Stefan Turalski。

特别要感谢 David Liebke，是你编写了第 6 章“协议和数据类型”的最初版本。如果不是因为你用全新的思路提供了一份奇妙的指南，这本书不会成为现在的样子。

感谢 Rich Hickey，是你创造了 Clojure 这种杰出的编程语言，并培育出围绕着它的社区。

感谢我的妻子 Joey，我的女儿 Hattie、Harper 和 Mabel Faire。是你们让我每天都如沐阳光。

——Stuart

感谢我的妻子，Erin，是你给予了我无尽的爱与鼓励。

——Aaron

① http://groups.google.com/group/clojure。

② http://www.pragprog.com/titles/shcloj2/errata。

第 1 版序

我们正在被复杂性淹没。其中绝大部分是偶然复杂性——并非源自问题本身，而是源自于我们为了解决问题而采用的方法。面向对象编程看起来容易，但生产出来的程序，往往会成为一张由可变对象交织而成的复杂巨网。单单是因为调用了一个对象上的方法，就可能会引起遍及整个对象关系图的一连串变化。在这种情况下，想要理解何时将会发生何事，事物是如何进入某种状态的，或是为修正某个 bug 而试图让对象回到特定状态，都变得极为复杂。而且一旦混入了并发问题，事情很快就会变得无法收拾。我们对程序使用了仿制对象（mock objects）和测试套件（test suites），但依然收效甚微。这不得不让我们对手头的工具和编程模型产生质疑。

函数式编程提供了别的选择。通过强调纯函数——传入和返回的都是不可变值（immutable value），副作用的存在成为了特例，而不再是常态。在多核架构中，我们面临的并发问题日益增加。因此，这种特性只会变得更加重要。Clojure 的设计目标，就是要让函数式编程更加平易近人，且兼备商业软件开发者所需的实用性。首先要能运行在像 Java 虚拟机这样可靠的基础设施之上，还要能支持现有客户在 Java 框架及库方面的投资。Clojure 非常清楚满足上述两点的必要性，以及这么做能带来的巨大实用性。

这是一种面向专业开发者（Stuart 自己就是）的编程语言。Stuart 的这本书之所以让人如此激动，源于他对 Clojure 的把握。很明显，他对 Clojure 瞄准的痛处有着充分的体验，这也增强了本书务实的态度。阅读本书，就像是一次充满激情的旅行，以实际应用为基础，逐步介绍 Clojure 的关键特性——有可能是一些全新的概念。我希望它能鼓舞你运用 Clojure 去编写软件，然后，回过头来你会说：“我不仅仅完成了工作，而且没想到采用的方法竟然如此健壮和简单，最棒的是，编写 Clojure 代码实在是太有乐趣了！”。

——Rich Hickey

Clojure 之父

第 2 版序

自本书第 1 版发行之后，很多东西发生了改变。是的，Clojure 语言本身得以增强，例如协议（protocol）和记录（record）的引入。然而，最有意义的是，Clojure 已应用于各种不同的领域。人们正在使用 Clojure 建立新系统，分析大规模数据集，以及处理通信、金融、互联网和数据库方面的工作。同时，一个大型的、乐于助人的社区已经围绕着 Clojure 成长起来。随之而来，涌现出了大量的程序库。这些库的出现格外令人激动，不仅仅在于它们提供的功能设施；还在于它们当中最优秀的那些库，欣然接受了 Clojure 倡导的方法和机制，从而在简洁性和互操作性方面达到了全新的高度。

Stuart 和 Aaron 确保在第 2 版中覆盖了语言的新增功能，并尝试着借助一些来自社区的库，来演示这些功能是如何运作的。本书仍然提供了令人振奋的 Clojure 简介，我希望它能激发你加入我们的社区，并最终对 Clojure 生态系统做出贡献。

——Rich Hickey

Clojure 之父

前　言

Clojure 是一种基于 Java 虚拟机（JVM，Java Virtual Machine）的动态编程语言（dynamic programming language）。它具有以下引人注目的特性。

- Clojure 非常优雅。
  摒弃了杂乱累赘的语法束缚，Clojure 干净、仔细的设计使你在编写代码时总能立刻切入问题的本质。

- Clojure 是 Lisp 的再度崛起。
  Clojure 从 Lisp 继承了强大的力量，却未受到 Lisp 历史的束缚。

- Clojure 是一种函数式语言（functional language）。
  作为一门函数式语言，Clojure 的数据结构具有不可变性（immutable），且大多数函数没有副作用（side effect）[①]。因此，编写正确的程序更加容易，也能更轻松地将小程序组合成一个大家伙。

- Clojure 简化了并发编程。
  很多其他语言围绕同步锁（locking）建立的并发模型，难以驾驭。为此，Clojure 提供了数个锁机制的替代方案：软事务内存（Software Transactional Memory，STM）、代理（agent）[②]、原子（atom）和动态变量（dynamic variable）。

- Clojure 与 Java 彼此亲密无间。
  在 Clojure 中调用 Java 代码，无需任何中间的转换层，直接而且快速。

- 不同于许多其他流行的动态语言，Clojure 运行飞快。
  Clojure 的实现利用了现代 Java 虚拟机上的众多优化技术。

尽管许多其他语言也包含了上述诸多特性中的一部分，但与它们相比，Clojure 仍

① 译注：初次接触函数式编程的读者可能对何为“副作用”略感疑惑。所谓副作用，就是指某个函数执行期间导致系统状态发生了变化。我们常见的大多数语言都有副作用。而典型的函数式语言没有赋值语句，也就不会对包括全局变量、函数参数或局部变量在内的各种系统状态造成影响。

② 译注：本书中后续章节中还将出现另外一个含义的代理（Proxy），但绝大多数情况下，根据上下文即可分辨文中出现的“代理”应该是 Agent 还是 Proxy，在可能会引起误解的地方，译者会加以注明。

显得魅力非凡。上述任何一个特性，都极为强大和有趣。Clojure 的迷人之处在于，将这些特性以非常干净的方式融合在了一起，且做到彼此协作无间。本书的第 1 章“启航”，将介绍以上这些特性及更多的内容。

谁应该阅读本书

Clojure 是一种强大的通用型（general-purpose）编程语言。如果你是一名经验老到的程序员，具备类似 C#、Java、Python 或者 Ruby 这样的现代编程语言的开发经验，并正在寻找更为强大、更加优雅的编程语言，那么本书是为你量身定做的。

Clojure 构建于 Java 虚拟机之上，并且运行飞快。如果你是一名对表现力丰富的动态语言馋涎已久，但却因为对性能问题的担忧而裹足不前的 Java 程序员，那么本书将引起你特别的兴趣。

Clojure 有助于重新定义，一种通用型编程语言应该包含哪些特性。如果你使用 Lisp，或使用一种诸如 Haskell 这样的函数式语言，又或者正编写明显存在并发的程序，那你一定会享受 Clojure 的一切。Clojure 融合了来自 Lisp、函数式编程和并发编程领域的理念，使得初次接触这些概念的程序员，一切触手可及。

Clojure 是本轮编程语言形态大规模演化现象的一部分。诸如 Erlang、F#、Haskell 和 Scala 这样的语言，由于它们支持函数式编程，或是由于它们的并发模型，最近都得到了格外的关注。作为这些语言的忠实信徒，你也一定会从 Clojure 当中找到众多共通之处。

本书主要内容

第 1 章，启航。本章将展示作为一门通用型语言，Clojure 的优雅特质及其函数式风格，以及独特的并发模型如何令其独一无二。阅读完本章，你还将能够轻松完成 Clojure 的安装，并学会如何使用 REPL 进行交互式开发。

第 2 章，探索 Clojure。在这里，我们将对 Clojure 的核心构造进行一次广度优先的概览。完成本章的阅读后，你将能顺畅地阅读大多数常规的 Clojure 代码。

接下来的两章将讨论函数式编程。第 3 章，一切皆序列，将展示 Clojure 如何使用强大的序列隐喻，统一了所有的数据形态。

第 4 章，函数式编程。本章将向你展示如何编写与序列库代码风格相同的函数式程序。

第 5 章，状态。本章我们将深入 Clojure 的并发编程模型。探讨 Clojure 中用于处理并发问题的 4 种强大模型。此外还有来自 Java 并发库中的精华内容一并奉上。

第 6 章，协议和数据类型。本章将逐个介绍在 Clojure 中的记录（record）、类型（type）和协议（protocol）。这些概念自 Clojure 1.2.0 版本首次引入后，在 Clojure 1.3.0 版本中得到了进一步增强。

第 7 章，宏。本章将不加掩饰地炫耀这一来自 Lisp 中的标志性特性——宏（Macros）。你将看到它如何利用“Clojure 代码本身就是数据”这一特质，提供了在其他非 Lisp 语系中极难甚至无法实现的非凡的元编程能力。

第 8 章，多重方法。本章将讨论 Clojure 解决多态问题的众多方法中的一种。多态，通常意味着“获取第一个参数的类型，并据此调度到相应的方法”。Clojure 的多重方法，使你可以更进一步，选择适用于所有参数的任意函数来进行调度。

第 9 章，极尽 Java 之所能。在本章中，你将看到如何从 Clojure 中调用 Java，以及从 Java 中调用 Clojure。你还将看到如何让 Clojure 疯狂运转，获得原生 Java 级别的性能。

最后，第 10 章，搭建应用。本章提供了一个可以让你完整了解 Clojure 应用开发全过程的视角。在这里，你将从头开始创建一个应用，并深入解决问题的方方面面，同时，还会考虑关于简单和质量的话题。你将借助一组有用的 Clojure 库，生产并发布一个 Web 应用。

附录，编辑器。这里列出了可供你选择的 Clojure 代码编辑器列表，并分别提供链接指向它们各自的安装说明。

如何阅读本书

所有读者都应该按顺序阅读最初的两章。请特别关注 1.1 节，这里提供了 Clojure 具备哪些优势的概述，里面的内容你一定会感兴趣。

持续的试验。Clojure 提供了一个可以让你立即获取反馈的交互式环境。请阅读 1.2.1 小节，以获得更多的信息。

读完最初的两章，你就可以随意翻阅了。但如果你打算开始阅读第 5 章，那么，确保你已经读过了第 3 章。顺序阅读这几章，将引导你从理解 Clojure 的不可变数据结构开始，一直到能够利用 Clojure 强大的并发模型，编写正确的并发程序。

当你开始接触后面各章中那些较长的代码示例时，请确保你使用的编辑器能为你提供 Clojure 代码自动缩进功能。附录“编辑器”列举了编写 Clojure 代码的通常选择。如果可能，请尝试使用支持括号匹配功能的编辑器，例如 Emacs 的 paredit 模式或者安装了 CounterClockWise 插件的 Eclipse。这些编辑功能将为你顺利学习 Clojure 编程提供巨大的帮助。

致函数式程序员

- Clojure 的函数式编程之道，在于将理论的纯粹之美，与 Clojure 需要运行在当前 Java 虚拟机之上的现实做出了完美的平衡。倘若仔细地阅读了第 4 章“函数式编程”，你将了解到 Clojure 与诸如 Haskell 这样的学院派语言之间存在的风格差异。
- Clojure 的并发模型（第 5 章），提供了数个直截了当的途径，用于处理并发世界中副作用和状态的问题。这也使得广大读者可以深入地体验函数式编程之魅力。

致 Java 和 C#程序员

- 请认真阅读第 2 章，Clojure 只有很少的语法规则（相比 Java 或 C#而言），所以我们能很快地熟悉它们。
- 请特别留意第 7 章，Java 或 C#背景的程序员将会发现，这部分是 Clojure 与他们所熟悉的语言之间的最大不同。

致 Lisp 程序员

- 第 2 章中的一些内容你可能已经很熟悉了，但无论如何，还是应该读一下这一章。Clojure 从 Lisp 中承袭了众多关键特性，但它也在一些地方打破了 Lisp 传统，这里将会讨论这些内容。
- 请密切关注第 4 章中的惰性序列。
- 为你的 Emacs 装备一个“clojure-mode”吧，这将为你享受后面章节中的代码

示例提供很大便利。

致 Perl、Python 和 Ruby 程序员

- 仔细阅读第 5 章，在 Clojure 中，进程内并行计算是一个非常重要的话题。
- 拥抱宏吧（第 7 章）。但请不要寄予太大的期望，能将你所用语言中的元编程风格轻松套用到 Clojure 宏中。请牢记，Clojure 的宏更为强大，并且，它是在代码读取期间被执行的，而非在运行期执行。

编写体例

以下编写体例将从始至终地贯穿于本书之中。

代码示例采用以下字体。

```
(+ 2 2)
```

为区别代码示例及其执行结果，我们会在执行结果前放置一个箭头（->）。

```
(+ 2 2)
-> 4
```

同样，控制台的输出也不容易与示例代码和结果区别开来，因此，我们会在控制台的输出前放置一个管道（|）符。

```
(println "hello")
| hello
-> nil
```

当首次引入某个 Clojure 形式（form），我们需要说明其语法时，将采用下述表示法。

```
(example-fn required-arg)
(example-fn optional-arg?)
(example-fn zero-or-more-arg*)
(example-fn one-or-more-arg+)
(example-fn & collection-of-variable-args)
```

这是一种非正式的语法，采用?、*、+和&符号，用于说明不同的参数传递模式。

Clojure 的代码是以程序库的形式进行组织的。如果本书某段示例代码所依赖

的库没有包含在 Clojure 语言核心中，我们将用 Clojure 的 use 或 require 对此加以说明。

```
(use '[lib-name :only (var-names+)])
(require '[lib-name :as alias])
```

此处使用 use 引入仅出现在列表 var-names 中的名称。使用 require 则创建一个库别名，使得每个引入函数的来源更加明晰。例如，来自于 clojure.java.io 库中的常用函数 file。

```
(use '[clojure.java.io :only (file)])
(file "hello.txt")
-> #<File hello.txt>
```

或使用基于 require 的版本。

```
(require '[clojure.java.io :as io])
(io/file "hello.txt")
-> #<File hello.txt>
```

事实上，如果成功调用了 use，Clojure 会返回 nil。但为使本书更加简洁，这个输出在示例清单中省略了。

在阅读本书期间，你将在名为 REPL 的 Clojure 交互式环境中输入代码。REPL 的控制台提示符形如下。

```
user=>
```

提示符中的 user 表明了你当前所在的 Clojure 名字空间。在本书大多数的例子中，当前位于哪个名字空间无足轻重。在这种情况下，我们将其省略，采用下述更简洁的语法表示在 REPL 中发生的一切。

```
(+ 2 2)       ; 没有命名空间提示的输入行
-> 4          ; 返回值
```

少数情况下，当位于哪个名字空间非常重要时，我们将采用如下语法。

```
user=> (+ 2 2)    ; 有命名空间提示的输入行
-> 4              ; 返回值
```

Web 资源及反馈

本书的英文官方主页[①]位于 Pragmatic Bookshelf 站点。在这里你可以订购本书的纸质版或是电子版，并且下载本书的示例代码。同样，你也可以将你的反馈提交至勘误表[②]或是直接发表至本书论坛[③]。

下载示例源码

你可以在下列任意位置找到本书的示例源码。

- 本书主页[④]上有链接指向官方发布的源码。同时，每次本书发布新版时，源码也将得到更新。
- 处于实时更新的本书 git 源码仓库[⑤]。这里有最新、最棒，且有时甚至强过书中所示的源码。

除非另行说明，示例文件都分别放在 examples 目录中。

贯穿于本书，示例源码的文件名列于源码清单起始位置，并采用灰色背景加以区别。例如，下面的源码清单来自于 src/examples/preface.clj。

```
src/examples/preface.clj
(println "hello")
```

如果你正在阅读的是本书的 PDF 版本，你可以直接点击文件名下载对应的源码清单文件。

有示例源码在手，你就可以准备启航了。首先，我们将领略究竟是怎样的特性组合，使得 Clojure 如此的独一无二。

① http://www.pragprog.com/titles/shcloj2/programming-clojure。
② http://www.pragprog.com/titles/shcloj2/errata。
③ http://forums.pragprog.com/forums/207。
④ http://www.pragprog.com/titles/shcloj2。
⑤ http://github.com/stuarthalloway/programming-clojure。

目 录

第 1 章

启航

有许多因素共同促成了 Clojure 的快速崛起。不妨在网络上进行一次快速搜索，会看到人们是这样描述 Clojure 的。

- 是一种函数式语言。
- 是一种为 Java 虚拟机设计的 Lisp。
- 具备处理并发问题的特殊能力。

所有这些特征都很重要，但却不是 Clojure 思想的核心。在我们看来，是这两个至为关键的概念驱动着 Clojure 的一切：简单、强大。

简单，在软件开发中被赋予了许多含义。但这里我们想要表达的，是它最初，也是最佳的定义：简单，就是不复杂。采用简单的组件，能让系统的设计者专注于他们手边的任务，从而免受各种无关杂事的打扰。以我们的经验来看，无关复杂性很快就会演变为危险复杂性。

强大，同样也具有很多含义。此处我们尤其关心，它是否足以完成我们将要承担的任务。作为一名程序员而言，强大，意味着你需要在一个功能丰富，且得到广泛部署的基础（例如 Java 虚拟机）之上构建应用。然后，你使用的工具必须能让你完全不受限制地访问这个平台。对于那些必须充分发挥平台能力的项目而言，这通常是决定性的需求。

作为程序员，我们年复一年地忍受着那些巴洛克式的复杂工具，仅仅因为这是我们获取必需的力量的唯一途径。如若不然，我们就不得不接受一个固然能使头脑保持清醒，但却弱小得多的简化编程模型。诚然，有一些取舍是无法回避的，但强大和简

单绝不属于此列。Clojure 向我们表明，强大与简单完全可以齐头并进。

1.1 为什么是 Clojure

所有 Clojure 的特色功能，要么简单，要么强大，或两者兼而有之。下面举几个例子。

- 函数式编程很简单，原因是它将计算的过程与状态及标识隔离开来。优点：函数式程序更容易理解、编写、测试、调优和并行化。
- Clojure 与 Java 的互操作极为强大，允许你直接访问 Java 平台的语义。优点：你能拥有与 Java 等同的性能和语义。最重要的是，你不必为了获得这点额外的能力而“下降”到一门低级别的语言。
- Lisp 的简单在于两个关键方面：它将代码的读取与求值分开了，并且语法仅由少数几个正交的部分构成。优点：能用语法抽象来捕获设计模式；此外，当需要的时候，S 表达式（S-expressions）能成为 XML、JSON 或是 SQL。
- Lisp 也很强大，它在运行期提供了一个编译器和宏（macro）系统。优点：Lisp 具有晚绑定的决策能力，并且很容易定制领域特定语言（DSL，Domain Specific Language）。
- Clojure 的时间模型很简单，将值、标识、状态和时间相互分离。优点：程序可以放心地感知并记住信息，完全不必担心在这段时间里，有人正打算对其乱涂乱画一番。
- 协议（Protocols）很简单，将多态性（polymorphism）和派生（derivation）分离。优点：不必纠结于设计模式，或是依赖于脆弱的猴子补丁（monkey patching），你就能得到既安全又极富扩展性的类型与抽象。

这个功能列表可以作为本书剩余部分的路线图，所以，即便此刻你尚无法充分理解每个小细节，也不必太过忧虑。上面的每个特性，都分别用了整整一章来加以详述。

让我们构建一个小型的应用，看看其中一些特性是如何运作的。沿途你将学会如何加载并执行那些较大的示例，本书的后半部分会用到它们。

1.1.1 Clojure 非常优雅

Clojure 高信号，低噪音。因此，Clojure 程序都非常简短。短小的程序，无论是构建、部署，还是维护，都要便宜得多[①]。尤其当程序是简明的（concise）而不仅仅是简

①《软件估算：黑匣子揭秘》 [McC06]这是一部重要的著作，里面有越小越便宜的实例。

短（terse）的时候就更是如此了。举个例子，考虑下面这段来自于 Apache Commons 的 Java 代码。

data/snippets/isBlank.java

```
public class StringUtils {
    public static boolean isBlank(String str) {
        int strLen;
        if (str == null || (strLen = str.length()) == 0) {
            return true;
        }
        for (int i = 0; i < strLen; i++) {
            if ((Character.isWhitespace(str.charAt(i)) == false)) {
                return false;
            }
        }
        return true;
    }
}
```

isBlank()方法用于检查目标字符串是否是*空白的*：没有任何字符，或者只包含空格符。这里是 Clojure 的类似实现。

src/examples/introduction.clj

```
(defn blank? [str]
  (every? #(Character/isWhitespace %) str))
```

Clojure 版本要短得多。但更重要的是，它更加简单：没有变量，没有可变状态，也没有分支结构。这可能要归功于高阶函数（higherorder functions）。高阶函数本身也是一个函数，它接受其他函数作为参数；也可以把函数作为返回值。every?函数接受一个函数 f 和一个容器（collection）[①]c 作为它的参数，对于容器 c 中的每个元素，如果函数 f 都返回真的话，every?函数也就返回真。

由于 Clojure 的这个版本没有分支结构，所以无论是阅读还是测试都更容易。在大一些的程序中，这种优势还将会进一步扩大。而且，简洁的代码也更具可读性。事实上，Clojure 的这段程序读起来就像是一份关于何为空白的定义：如果一个字符串中的每个字符都是空格，那么这个字符串就是空白的。这要比一般的方法好太多了，在那些方法中，对空白的定义被隐藏在了由循环和分支语句组成的实现细节背后。

另外一个例子，考虑用 Java 定义一个微不足道的 Person 类。

① 译注：本书中将 collection 译作容器而不是集合，是为了与 set 类型加以区分。它们与 J2EE 中的 EJB 容器或 Servlet 容器没有任何关系。

data/snippets/Person.java

```
public class Person {
    private String firstName;
    private String lastName;
    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

在 Clojure 中，用一行代码就可以定义这个 Person。

```
(defrecord Person [first-name last-name])
```

然后像下面这样使用：

```
(def foo (->Person "Aaron" "Bedra"))
-> #'user/foo
foo
-> #:user.Person{:first-name "Aaron", :last-name "Bedra"}
```

在第 6.3 节“协议”中，包含了 defrecord 及其相关函数的介绍。

除代码短了一个数量级以外，Clojure 采用的方法还有一处不同：Clojure 版本的 Person 是不可变的。不可变数据结构生来就是线程安全的，Clojure 中可以通过使用引用、代理和原子来更新数据，这些内容将在第 5 章“状态”中详加讨论。正因为记录（record）是不可变的，Clojure 也就自动提供了正确的 hashCode()和 equals()实现。

Clojure 内建了大量优雅的特性，但倘若你发现还是遗漏了某样东西的话，你可以自己添上，这完全要归功于 Lisp 的强大。

1.1.2　Clojure 是 Lisp 的再度崛起

Clojure 是一种 Lisp 方言。数十年来，拥护者们指出了 Lisp 与其他语言相比的诸多优点。但同时，Lisp 一统天下的计划看起来却遥遥无期。

如同其他所有的 Lisp 一样，Clojure 也面临着两个挑战。

- Clojure 必须成功地说服 Lisp 程序员，作为一种 Lisp 方言，Clojure 包含了 Lisp 的关键部分。
- 同时，Clojure 还需要成功地赢得广泛的程序员社区支持，而这正是过去那些 Lisp 的失败之处。

为了应对这些挑战，Clojure 提供了 Lisp 元编程能力，与此同时还包含了一整套增强的语法，使得 Clojure 对于非 Lisp 程序员而言显得更为友好。

1.1.3　为什么是 Lisp

Lisp 的语言核心非常小，几乎没有什么语法，但却提供了一个强大的宏设施。借助这些特性，你可以根据你的设计需要对 Lisp 随意地直接定制。这样就不必使用其他那些绕来绕去的方式了。考虑以下 Java 代码片段。

```
public class Person {
  private String firstName;
  public String getFirstName() {
  // 以下省略…
```

在这段代码中，getFirstName()是一个方法（method）。方法具有多态性，可以根据你的需要加以调整。但对于 Java 而言，示例中其他单词的语义，其解释都是固定的。然而，有时你确实需要改变这些词语的含义。举例来说，你可能会像下面这么做。

- 重新定义 private：对于产品代码保持私有，但允许来自序列化（serialization）和单元测试代码的访问。
- 重新定义 class：自动为每个私有字段都生成 getters 和 setters，除非另有指示。
- 创建 class 的一个子类，提供面向生命周期事件的回调钩子。例如，对于“可感知生命周期”的类而言，只要创建了这个类的一个实例，就会激发相应的事件。

我们一定见过需要上述特性的程序。由于缺乏这些特性，程序员们不得不去借助一些重复性的、容易出错的变通方法。结果是，人们在这上面白白浪费了数百万行的

代码，而罪魁祸首就是编程语言中类似特性的缺失。

对大多数编程语言而言，你只能祈求语言的实现者们尽快增加上面提到的这类特性。但在 Clojure 中，你能凭借宏来自行添加属于你自己的语言特性（第 7 章“宏”）。事实上，Clojure 本身就是用宏来进行扩建的，比如 defrecord。

```
(defrecord name [arg1 arg2 arg3])
```

如果你需要的语义与此不同，写一个你自己的宏就行。比如你想得到记录的一个变种，它具备强类型并具有可选的空字段校验能力，你可以创建自己的 defrecord 宏。这个新的 defrecord 用法如下。

```
(defrecord name [Type :arg1 Type :arg2 Type :arg3]
  :allow-nulls false)
```

这种对语言进行再编程，从而改变语言自身的能力，是 Lisp 的独门优势。下面用不同的方式来描述这一思想。

- Lisp 具有同像性（homoiconic）[①]。也就是说，Lisp 代码其实就是 Lisp 数据。这样就很容易让程序自己去编写其他的程序。
- 这就是语言的全部，且始终如此。保罗·格雷厄姆在其散文《书呆子的复仇》[②]中，解释了为什么这会如此的强大。

Lisp 语法也废除了运算符优先级和结合性的规则。翻遍本书的任何一个角落，你都不会看到用来说明运算符优先级或结合性的表格。凭借完全的括号表示法，就能避免产生任何这方面的歧义。

简单、整齐的 Lisp 语法也存在负面因素，至少对于初学者而言，成堆的括号，以及将列表作为核心数据类型都会成为一种障碍。为此，Clojure 提供了有趣的功能组合，对于非 Lisp 程序员而言，这个 Lisp 显得要亲切得多。

1.1.4 它是 Lisp，但括号少了

对于来自其他 Lisp 方言的程序员来说，Clojure 的优势显而易见。

- Clojure 泛化了 Lisp 的物理列表，将其抽象为序列（sequence）。这样既保留了列表的强大能力，同时还将这种能力扩展到了其他各种类型的数据结构。

① http://en.wikipedia.org/wiki/Homoiconicity。
② http://www.paulgraham.com/icad.html。

- 依托于 Java 虚拟机，Clojure 提供了一个范围广泛的标准库及部署平台。
- Clojure 提供的符号解析和语法引述（syntax quoting）方式，使得编写许多普通宏的时候更加容易了。

许多 Clojure 程序员可能会是 Lisp 的新手，他们也许听说过诸多关于 Lisp 括号的可怕传言。是的，Clojure 保留了括号表示法（当然也保留了 Lisp 的强大!），但在以下方面对传统 Lisp 语法进行了改进。

- 在 Clojure 中，除列表之外，还提供了更为便利的正则表达式、映射表、集合，向量和元数据等多种数据结构的字面表示语法。这些特性使得 Clojure 代码相比其他多数 Lisp 语言而言，过度列表化（listy）的症状要轻很多。例如，Clojure 函数的参数是在一个向量（[]）中指定的，而不是使用列表（()）。

src/examples/introduction.clj

```
(defn hello-world [username]
  (println (format "Hello, %s" username)))
```

向量令参数列表变得非常醒目，也使得 Clojure 的函数定义更易于阅读。

- 与大多数 Lisp 语言不同，在 Clojure 中，逗号就是空格。

```
; 这让向量看起来就像是其他语言中的数组一样。
[1, 2, 3, 4]
-> [1 2 3 4]
```

- 地道的 Clojure 不会内联不必要括号。考虑一下在 Common Lisp 和 Clojure 中都有的 cond 宏。cond 对一组成对的“测试/结果”逐个求值，当遇到第一个求值结果为真的测试时，返回其对应的结果。Common Lisp 中，每一对“测试/结果”都得像下面这样，用括号进行分组。

```
; Common Lisp cond
(cond ((= x 10) "equal")
  ((> x 10) "more"))
```

而在 Clojure 中则避免了额外的括号。

```
; Clojure cond
(cond (= x 10) "equal"
  (> x 10) "more")
```

这是一种审美决定，且双方都各有其支持者。但重点在于，Clojure 获得了在不减损 Lisp 威力的前提下，尽可能减少过度列表化的机会。

Clojure 是一种卓越的 Lisp 方言，无论对于 Lisp 专家，还是 Lisp 新手，皆是如此。

1.1.5 Clojure 是函数式语言

Clojure 虽然是一种函数式语言，但不像 Haskell 那样纯粹。函数式编程语言具有下列属性。

- 函数是一等公民。换言之，函数能在运行期间被创建，被当做参数传递，被用作返回值，并且能像其他数据类型那样，被用于各种用途。
- 数据是不可变的。
- 函数是纯粹的，也就是说，它们不会造成任何副作用。

对许多任务而言，函数式程序更容易理解，不容易出错，且更利于重用。例如，下面这个小程序从乐曲数据库中，查询有哪些作曲家创作了《Requiem（安魂曲）》。

```
(for [c compositions :when (= "Requiem" (:name c))] (:composer c))
-> ("W. A. Mozart" "Giuseppe Verdi")
```

这里的 for，并不意味着引入了循环，而是进行了一次列表解析（list comprehension）。所以，这段代码应该这么读：“对于乐曲库中的每支乐曲 c，当 c 的名称是《Requiem》时，则获取 c 的作曲家信息”。本书第 3.2.4 小节“序列转换”中有关于列表解析的完整讨论。

这个例子的可取之处有以下 4 方面：

- 非常简单，没有任何循环结构、变量或是可变的状态；
- 线程安全，不需要锁机制即可得到保证；
- 可并行化，无需修改代码，你就可以将单独的步骤转移至多个线程；
- 非常通用，乐曲库可以是一个普通集合、XML 或是一个数据库结果集。

这里，函数式程序与命令式程序形成鲜明对比，在命令式程序中，是用显式的语句来改变程序状态的。大多数面向对象程序都是采用命令式风格写就的，在前面列出的这几方面，它们劣势尽显（关于函数式和命令式风格的逐项对比，请阅读 2.7 节）。

如今人们已经知道了函数式语言的优势。然而，像 Haskell 那样的纯函数式语言却没能接管世界，这是因为开发者们发现，纯粹的函数式观点无法轻易地解决所有问题。

与过去的那些函数式语言相比，有 4 个原因使得 Clojure 能够吸引更多的注意。

- 对函数式编程的需要，比以往任何时候都显得更加迫切。规模庞大的多核硬件

已指日可待，函数式语言提供了一种清晰的方式对其加以利用。本书第 4 章“函数式编程”详细讨论了这个话题。

- 当确实需要对状态进行修改时，纯粹的函数式编程语言就显得颇为尴尬了。Clojure 则通过软事务内存（STM，software transactional memory）及引用、代理、原子和动态绑定，提供了结构良好的机制用于处理可变状态。
- 许多函数式语言都是基于静态类型的。而 Clojure 的动态类型系统，使得程序员学习函数式编程更加容易。
- Clojure 的 Java 调用方式是非函数式的。当你调用 Java 程序时，你会进入那个熟悉的，可变的世界。这为函数式编程的初学者提供了一个舒适的港湾，此外当你需要时，这也是能够提供函数式风格替代品的务实之选。第 9 章“极尽 Java 之所能”详细讨论了关于 Java 调用方面的内容。

Clojure 中不必显式锁定，就允许并发地更改状态。这种方式是 Clojure 函数式核心的有力补充。

1.1.6　Clojure 简化了并发编程

Clojure 支持函数式编程，使得编写线程安全的代码非常容易。由于不可变数据结构在任何时候都不会被修改，因此避免了数据会被另外一个线程破坏的危险。

然而，仅仅是函数式编程，还不足以体现 Clojure 对并发程序支持之卓越。当你需要引用可变数据时，Clojure 会通过软事务内存对其加以保护。在线程安全方面，相比 Java 提供的锁定机制，软事务内存是一种更高级的方法。你可以借助事务来保护共享状态，而不是去琢磨那些既脆弱，又易于出错的锁定策略。源于数据库方面的经验，很多程序员对何为事务早就了然于胸，所以这也是一种更富成效的做法。

例如，下面的代码创建了一个线程安全的内存数据库用于存放账号。

```
(def accounts (ref #{}))
(defrecord Account [id balance])
```

ref 函数创建了一个引用，代表数据库的当前状态，这个引用会得到事务的保护。更新操作实在是微不足道。下列代码向数据库中添加一个新的账号。

```
(dosync
  (alter accounts conj (->Account "CLJ" 1000.00)))
```

dosync 开启了一个事务，允许对 accounts 进行更新。这样既确保了线程安全，同时

也比锁机制更容易使用。得益于事务，你不必再操心应该锁定哪些对象，以及应该以什么顺序来锁定等等问题。在一些常见的使用场景中，因为读取操作不会被阻塞，所以事务机制能够非常高效地运转。

虽然这是个微不足道的例子，但其展现的技术是通用的，完全可用于解决现实世界中的问题。请参阅第 5 章“状态”，那里有更多关于 Clojure 中并发及软事务内存方面的讨论。

1.1.7 Clojure 与 Java 虚拟机彼此亲密无间

从 Clojure 访问 Java，清晰、简单、直接。你能直接调用任何 JavaAPI。

```
(System/getProperties)
-> {java.runtime.name=Java(TM) SE Runtime Environment
... many more ...
```

Clojure 为调用 Java 提供了很多语法糖。我们不需要在这里深入过多细节（参阅第 2.5 节“调用 Java”），但请注意，下面的代码中，Clojure 的那个版本无论是点号（.），还是括号（()），数量都比 Java 版本要少。

```
// Java
"hello".getClass().getProtectionDomain()
; Clojure
(.. "hello" getClass getProtectionDomain)
```

Clojure 提供了简单的函数用于实现 Java 接口，以及从 Java 基类派生。此外，Clojure 的所有函数都实现了 Callable 和 Runnable 接口。这使得采用下面所示的匿名函数来构建 Java 线程竟然如此轻松。

```
(.start (new Thread (fn [] (println "Hello" (Thread/currentThread)))))
-> Hello #<Thread Thread[Thread-0,5,main]>
```

这里有个有趣之处，就是 Clojure 打印 Java 对象实例的方式。Thread 是这个实例的类名，然后 Thread[Thread-0,5,main]是这个实例的 toString 方法返回值。

注意，前例中的这个新线程会持续运行直至完成，但其输出可能会以某种奇怪的方式，同 REPL 的提示符产生交错现象。但这并非 Clojure 的问题，只不过是有多个线程同时向输出流进行写入数据的结果罢了。

由于在 Clojure 中调用 Java 程序的语法干净而且简单，作为 Clojure 的惯例，会更加倾向于直接对 Java 进行调用，而不是把 Java 隐藏到一层 Lisp 化的封装背后。

好了，现在你已经看到一些为什么要使用 Clojure 了，是时候开始编写一些代码了。

1.2　Clojure 编程快速入门

要运行 Clojure 及本书的示例代码，你需要两件东西。

- Java 运行时。请下载[①]并安装 Java 5 或是更高版本。Java 6 具有显著的性能提升和更好的异常报告，如果可能就尽量选它吧。
- Leiningen[②]。Leiningen 是一个用于管理依赖项的工具，并且可以基于你的代码启动各种任务。在 Clojure 世界中，它是处理这项工作最常用的工具了。

你将会使用 Leiningen 来安装 Clojure 和本书所有示例代码的依赖项。如果你已经安装了 Leiningen，那就应该已经熟悉相关的基础知识了。否则，你应该快速的浏览一下 Leiningen 的 GitHub 主页[③]，在那儿你能找到如何安装，以及其基本用法的说明。现在还不用急着去学习所有这一切，因为本书会指引你轻松地掌握那些必须的命令。

在阅读本书期间，请使用与本书示例代码匹配的 Clojure 版本。读完本书后，你就可以按照“自行构建 Clojure”中的说明，构建一个最新鲜的 Clojure 版本。

自行构建 Clojure

你可能希望从源码构建 Clojure，以获得最新的特性和 bug 修复。这里是具体做法。

```
git clone git://github.com/clojure/clojure.git
cd clojure
mvn package
```

本书的示例代码会经常更新，与匹配 Clojure 当前最新的开发版本。请检查示例代码目录中的 README 文件，里面有最近一次测试通过时，对应的 Clojure 修订版本号。

请参阅前言中“下载示例代码”中说明，下载本书的示例代码。当你下载了示例代码后，你还需要使用 Leiningen 获取它们的依赖项。请在示例代码的根目录下执行。

```
lein deps
```

那些依赖项将被下载到本地并放置在适当的位置。为了测试安装是否正确，你可

① http://www.oracle.com/technetwork/java/javase/downloads/index.html。
② http://github.com/technomancy/leiningen。
③ http://github.com/technomancy/leiningen。

以进入到放置示例代码的目录，并启动一个 Clojure 的 REPL（读取-求值-打印循环）。Leiningen 包含了一个启动 REPL 的脚本，它可以连同依赖项一起加载 Clojure，本书后面部分会用到那些依赖项。

```
lein repl
```

当你成功的启动了 REPL，它将会显示“user=>”对你进行提示。

```
Clojure
user=>
```

现在，你已经为“Hello World.”做好了准备。

1.2.1 使用 REPL

来看看如何使用 REPL，让我们创建几个“Hello World”的变体。首先，在 REPL 提示符下键入(println "hello world")。

```
user=> (println "hello world")
->hello world
```

第二行的“hello world”，就是 REPL 针对你提交的请求，产生的控制台输出。

接下来，将你的“Hello World”封装成一个函数，让它可以通过名字向人问好。

```
(defn hello [name] (str "Hello, " name))
-> #'user/hello
```

我们来分析一下。

- defn 定义了一个函数。
- hello 是这个函数的名称。
- hello 函数接受一个参数 name。
- str 是一个函数调用，把由任意参数组成的列表连接为一个字符串。
- defn、hello、name 和 str 都是符号（symbols），代表了它们各自涉及事物的名称。在第 2.1.2 小节“符号”中有关于合法符号的定义。

再看看这行代码的返回值：#'user/hello。前缀#'表示这个函数是用一个 Clojure 变量（var）来保存的，其中 user 是这个函数所在的命名空间（namespace）（就像 Java 的默认包一样，user 是 REPL 的默认命名空间）。你现在还不必为变量和名字空间担忧，第 2.4 节“变量、绑定和命名空间”里有关于它们的讨论。

你现在可以调用 hello 函数，并传入你的名字了。

```
user=> (hello "Stu")
-> "Hello, Stu"
```

如果你发现 REPL 的状态令你倍感困惑，最简单的解决办法就是直接关闭这个 REPL（Windows 下使用 CTRL+C，*nix 下则是 CTRL+D），然后再另外启动一个。

1.2.2　特殊变量

REPL 包括几个有用的特殊变量。当你使用 REPL 时，最近三次求值结果的描述被分别存储在特殊变量*1、*2 和*3 中。这使得进行迭代变的非常容易。下面，让我们向几个不同的名字问声好。

```
user=> (hello "Stu")
-> "Hello, Stu"
user=> (hello "Clojure")
-> "Hello, Clojure"
```

现在，你可以使用那几个特殊变量，把最近的几个工作成果组合起来。

```
(str *1 " and " *2)
-> "Hello, Clojure and Hello, Stu"
```

如果你在使用 REPL 的过程中犯了错，你会看到一个 Java 异常。出于简洁方面的考虑，细节往往被省略了。例如，除以零是不允许的。

```
user=> (/ 1 0)
-> ArithmeticException Divide by zero clojure.lang.Numbers.divide
```

这是个显而易见的问题，但有些时候问题会更加微妙，这时你就需要获得更详细的堆栈跟踪（stack trace）信息了。最后一个异常被保存在特殊变量*e 中。由于 Clojure 异常就是 Java 异常，所以你能使用 pst 函数[①]（print stacktrace）得到堆栈跟踪信息。

```
user=> (pst)
-> ArithmeticException Divide by zero
| clojure.lang.Numbers.divide
| sun.reflect.NativeMethodAccessorImpl.invoke0
| sun.reflect.NativeMethodAccessorImpl.invoke
| sun.reflect.DelegatingMethodAccessorImpl.invoke
| java.lang.reflect.Method.invoke
| clojure.lang.Reflector.invokeMatchingMethod
```

① pst 函数仅适用于 Clojure1.3.0 及更高版本。

```
| clojure.lang.Reflector.invokeStaticMethod
| user/eval1677
| clojure.lang.Compiler.eval
| clojure.lang.Compiler.eval
| clojure.core/eval
```

更多与 Java 互操作方面的内容，参见第 9 章“极尽 Java 之所能”。如果你的代码块实在太大，不便于在 REPL 中逐行敲入，不妨将代码保存到一个文件中，然后通过 REPL，使用绝对路径或是相对路径（相对于启动 REPL 的路径）来加载这个文件。

```
; 保存一些东西到 temp.clj 中，然后执行...
user=> (load-file "temp.clj")
```

REPL 是一个美妙的场所，在这里你可以尝试各种想法并立即获得反馈。为达到最佳效果，阅读本书时，请务必保持随时都开启着 REPL。

1.2.3 添加共享状态

上一节中的 hello 函数是“纯粹的”，也就是说，它不会产生任何副作用。纯函数易于开发、测试，并易于理解，你应该优先选择它们来处理任务。

可是，大多数程序拥有共享状态，并且需要使用非纯粹的函数来管理这些共享状态。让我们对 hello 函数进行扩展，使其能够追踪过往访客的足迹。首先，你需要一种数据结构来追踪访客。集合就非常合适。

```
#{}
-> #{}
```

#{}是空集合的字面表示法。接下来，你需要 conj 函数。

```
(conj coll item)
```

conj 是 conjoin（连接）的缩写，它会新建一个含有新增项的集合。将元素连接到集合，就好像是创建了一个新的集合。

```
(conj #{} "Stu")
-> #{"Stu"}
```

现在你可以创建新的集合了，但你还需要某种方法来对当前访客的集合保持跟踪。为此，Clojure 提供了几种引用类型。最基本的引用类型是原子。

```
(atom initial-state)
```

你可以使用 def 来为你的原子命名。

```
(def symbol initial-value?)
```

def 有点像 defn，但更为通用。Def 既能定义函数，又能定义数据。下面使用 atom 创建一个原子，并用 def 将这个原子绑定到名称 visitors 上。

```
(def visitors (atom #{}))
-> #'user/visitors
```

要更新一个引用，你需要使用诸如 swap!这样的函数。

```
(swap! r update-fn & args)
```

swap!会对拿引用 r 去调用 update-fn，并根据需要传递其他可选的参数。下面试一下用 conj 作为更新函数，把一个访客 swap!进入到访客集合中。

```
(swap! visitors conj "Stu")
-> #{"Stu"}
```

原子只是 Clojure 的几种引用类型之一。选择恰当的引用类型时，需要格外小心仔细（相关讨论参见第 5 章“状态”）。

你可以在任何时候使用 deref 或者它的缩写@号来提取引用内部的值。

```
(deref visitors)
-> #{"Stu"}
@visitors
-> #{"Stu"}
```

现在，是时候创建这个更加复杂的新版 hello 了。

src/examples/introduction.clj

```
(defn hello
  "Writes hello message to *out*. Calls you by username.
   Knows if you have been here before."
  [username]
  (swap! visitors conj username)
  (str "Hello, " username))
```

下一步，检查一下看能否在内存中正确地追踪了。

```
(hello "Rich")
-> "Hello, Rich"
@visitors
-> #{"Aaron" "Stu" "Rich"}
```

你的访客列表十有八九与此处显示的不同。这就是状态捣的乱！结果是否会有差别，取决于事情何时发生。你还可以据此推论出一个函数是否管理着本地信息。对状态进行推断，需要对其演变历史有着充分的认识。

只要可能，就应该极力避免状态。但是当你确实需要它的时候，通过使用诸如原子这样的引用类型，就能让状态保持完整以及可控。原子（和所有其他的 Clojure 引用类型）对于多个线程和多个处理器都是安全的。更棒的是，获得这种安全性无需借助声名狼藉的锁定机制，那实在是太让人抓狂了。

至此，你应该已经能很舒畅的在 REPL 中录入那些较短的代码了。其实那些较长的代码也没有太多不同，你同样可以在 REPL 中加载并运行大量有成百上千行代码的 Clojure 库。下面让我们来探索一下吧。

1.3 探索 Clojure 的程序库

Clojure 代码通常都被打包在程序库中。每个 Clojure 库都属于某个命名空间，这与 Java 的包非常类似。你可以通过 require 来加载一个 Clojure 库。

```
(require quoted-namespace-symbol)
```

当你使用 require 加载了一个名为 clojure.java.io 的库时，Clojure 会在 CLASSPATH 中查找名为 clojure/java/io.clj 的文件。试试看。

```
user=> (require 'clojure.java.io)
-> nil
```

起头的单引号（'）是必不可少的。它表示对库名的引用（关于引用的内容参见 2.2 节“读取器宏”）。返回 nil 表示库加载成功。如果你想测试一下，可以加载本章的示例代码 examples.introduction。

```
user=> (require 'examples.introduction)
-> nil
```

examples.introduction 库中包含了一个 Fibonacci 数列的实现，这是函数式编程语言传统的“Hello World”程序。在 4.2 节“怎样偷个懒”中，我们会探索关于 Fibonacci 数列的更多细节。现在，你只要确保能够执行这个示例函数 fibs 即可。你可以在 REPL 中输入下面的代码行以获取前 10 个 Fibonacci 数。

```
(take 10 examples.introduction/fibs)
-> (0 1 1 2 3 5 8 13 21 34)
```

如果你看到的前 10 个 Fibonacci 数与此处列出的相同，说明你已经成功安装了本书的示例。

本书的所有示例都进行了单元测试，测试代码位于 examples/test 目录。本书并未明确包含这些示例的测试本身，但你会发现它们可以作为非常有用的参考。你也可以自己执行 lein test 命令来运行这些单元测试。

1.3.1　require 和 use

当你用 require 加载了一个 Clojure 程序库，就需要使用命名空间限定的名称来引用这个库里面的内容。你必须这么写：examples.introduction/fibs，而不是简单地写下 fibs 了事。下面，确保你新启动了一个 REPL[①]，然后试着输入。

```
(require 'examples.introduction)
-> nil
(take 10 examples.introduction/fibs)
-> (0 1 1 2 3 5 8 13 21 34)
```

总要使用完全限定名实在是太啰嗦了。你可以对命名空间使用 refer，将其下的所有名称映射到你的当前命名空间中。

```
(refer quoted-namespace-symbol)
```

对 examples.introduction 调用 refer，然后确认你能否直接调用 fibs 了。

```
(refer 'examples.introduction)
-> nil
(take 10 fibs)
-> (0 1 1 2 3 5 8 13 21 34)
```

为方便起见，Clojure 的 use 函数把 require 和 refer 合并成了一个步骤。

```
(use quoted-namespace-symbol)
```

在新启动的 REPL 中，你应该能够顺利执行下列代码。

```
(use 'examples.introduction)
-> nil
(take 10 fibs)
-> (0 1 1 2 3 5 8 13 21 34)
```

① 新开启一个 REPL，能防止你之前输入的代码，与本书示例代码中的同名函数之间产生名字冲突。如同你将在“命名空间”中看到的那样，在实际开发中这不会成为问题。

在使用本书的示例代码期间，你可以在调用 require 或 use 时，使用:reload 标记用来强制重新加载一个程序库。

```
(use :reload 'examples.introduction)
-> nil
```

如果你进行了一些修改，想看看结果如何，但又不想重新启动 REPL，那么:reload 标记就能帮上大忙。

1.3.2 查找文档

通常情况下，你都可以在 REPL 中找到你需要的文档。最基本的辅助函数[①]是 doc。

```
(doc name)
```

试试使用 doc 来打印 str 函数的文档。

```
user=> (doc str)
-------------------------
clojure.core/str
([] [x] [x & ys])
With no args, returns the empty string. With one arg x, returns
x.toString(). (str nil) returns the empty string. With more than
one arg, returns the concatenation of the str values of the args.
```

doc 输出的第一行包含了目标函数的全限定名称。接下来的一行包含了可能的参数列表，这是直接从代码中生成的。“参数命名惯例”中有一些常用的参数名称，及它们的用途解释。最后，剩下的那几行是这个函数的文档字符串（doc string），当然，如果在函数定义中包含了它们的话。

你可以为自己的函数添加文档字符串，只要将它放置在紧接着函数名的位置即可。

src/examples/introduction.clj

```
(defn hello
  "Writes hello message to *out*. Calls you by username"
  [username]
  (println (str "Hello, " username)))
```

有时你想要查阅文档时，不清楚目标的确切名称。find-doc 会用你传入的正则表达式或是字符串，找出所有那些调用 doc 函数得到的输出能与之匹配的东西。

```
(find-doc s)
```

① doc 实际上是一个 Clojure 的宏。

试试用 find-doc 检索一下 Clojure 是如何进行 reduce 的。

```
user=> (find-doc "reduce")
-------------------------
clojure/areduce
([a idx ret init expr])
Macro
... details elided ...
-------------------------
clojure/reduce
([f coll] [f val coll])
... details elided ...
```

reduce 用于对 Clojure 容器进行归纳，第 3.2.4 小节“序列转换”中讨论了该话题。areduce 则作用于 Java 数组，第 9.4.2 小节“使用 Java 容器”中有关于它的讨论。

参数命名惯例

reduce 和 areduce 的文档字符串展示了几个简练的参数名称。表 1-1 列出了一些约定的参数名，以及通常情况下应该如何使用它们。

表 1-1　参数名

参数	用途
a	Java 数组
agt	代办
coll	容器
expr	表达式
f	函数
idx	索引
r	引用
v	向量
val	值

这些名称看起来似乎过于简练了，但采用它们有一个很好的理由：“好名称”往往已经被 Clojure 函数给占用了！尽管从语法角度，参数可以和其他函数重名，但这是一种糟糕的风格：参数会遮蔽函数，当它们同处一室时，后者将会失效。所以，千万不要把你的引用叫做 ref、把代理叫做 agent，或者把数量叫做 count。这些名称代表的是函数。

Clojure 自己的源码中，很多都是用 Clojure 本身写成的，阅读这些源码极具启发

性。你可以使用 repl 库的 source 函数来查阅某个 Clojure 函数的源码。

```
(clojure.repl/source a-symbol)
```

查阅一下简单的 identity 函数源码看看。

```
(use 'clojure.repl)
(source identity)
-> (defn identity
     "Returns its argument."
     {:added "1.0"
      :static true}
     [x] x)
```

当然，你也可以使用 Java 的反射 API。用诸如 class、ancestors 和 instance?这样的方法，来反射其底层的 Java 对象模型。例如，Clojure 的容器同样也是 Java 容器。

```
(ancestors (class [1 2 3]))
-> #{clojure.lang.ILookup clojure.lang.Sequential
    java.lang.Object clojure.lang.Indexed
    java.lang.Iterable clojure.lang.IObj
    clojure.lang.IPersistentCollection
    clojure.lang.IPersistentVector clojure.lang.AFn
    java.lang.Comparable java.util.RandomAccess
    clojure.lang.Associative
    clojure.lang.APersistentVector clojure.lang.Counted
    clojure.lang.Reversible clojure.lang.IPersistentStack
    java.util.List clojure.lang.IEditableCollection
    clojure.lang.IFn clojure.lang.Seqable
    java.util.Collection java.util.concurrent.Callable
    clojure.lang.IMeta java.io.Serializable java.lang.Runnable}
```

http://clojure.github.com/clojure 有完整的 Clojure API 在线文档。右侧栏按照名称对所有的函数和宏建立了链接，左侧栏则链接至一系列描述 Clojure 的特性的文章。

1.4 小结

你刚才完成了一次旋风般的 Clojure 之旅。你已经看到了 Clojure 极富表现力的语法，了解了 Clojure 的 Lisp 之道，还见到了从 Clojure 中调用 Java 代码有多么容易。

你已经有了一份运行在你自己环境中的 Clojure，此外你还在 REPL 中编写了几款短小的程序，用来演示函数式编程和解决状态问题的引用模型。现在是时候去探索一下整个语言了。

第 2 章

探索 Clojure

函数式风格、对并发的强力支持以及干净的 Java 互操作，给 Clojure 提供了极其强大的力量。但是，在领会所有这些特性之前，你必须先从语言的基础部分开始学习。在本章中，你将对 Clojure 语言本身进行一次快速浏览，包括以下内容。

- 形式（Forms）
- 读取器宏
- 函数
- 绑定和命名空间
- 流程控制
- 元数据

如果你的主要背景是命令式语言，你会发现本章似乎漏掉了诸如变量和 for 循环这样的关键语言结构。第 2.7 节“我的 for 循环哪儿去了？”将会告诉你，远离 for 循环和变量以后，生活将会变得何等美好。

Clojure 语言极富表现力，本章相当快速地覆盖了许多概念。如果你未能理解每一个细节，别担心，在随后的章节中，还会详细地重新审视这些主题。可能的话，不妨启动 REPL，紧随你沿路遇到的例子一同前进。

2.1 形式

Clojure 具有同像性[①]，也就是说 Clojure 代码本身，是由 Clojure 数据构成的。当

① http://en.wikipedia.org/wiki/Homoiconicity。

你运行一段 Clojure 程序，作为 Clojure 组成部分的读取器（reader），读入那些被称为“形式”的程序文本块，然后将它们翻译为 Clojure 的数据结构。接下来，Clojure 编译并执行这些数据结构。

表 2-1 对 Clojure 中的形式进行了总结。为了理解形式是如何运作的，让我们从一些支持数值类型的简单形式开始。

表 2-1 Clojure 的形式

Form	示例	主要章节
Boolean	`true, false`	布尔值与 nil
Character	`\a`	字符串与字符
Keyword	`:tag, :doc`	映射表、关键字和记录
List	`(1 2 3), (println "foo")`	第 3 章“一切皆序列”
Map	`{:name "Bill", :age 42}`	映射表、关键字和记录
Nil	`nil`	布尔值与 nil
Number	`1, 4.2`	使用数值类型
Set	`#{:snap :crackle :pop}`	第 3 章“一切皆序列”
String	`"hello"`	字符串与字符
Symbol	`user/foo, java.lang.String`	符号
Vector	`[1 2 3]`	第 3 章“一切皆序列”

2.1.1 使用数值类型

数值字面量本身就是形式。数字会简单的对其自身进行求值。如果在 REPL 中输入一个数字，他会原样返回给你。

```
42
-> 42
```

由数字组成的向量是另外一种形式。下面创建了一个包含数字 1、2 和 3 的向量。

```
[1 2 3]
-> [1 2 3]
```

列表也是一种形式。一个列表可以“仅仅只是数据”，但也可以用于调用函数。下

面创建一个列表，它的第 1 项是一个 Clojure 函数，例如+号。

```
(+ 1 2)
-> 3
```

正如你看到的，Clojure 对这个列表进行求值的时候，是把它当作了一次调用。与更常见的中缀表示法（infix notation，例如：1+2=3）相对，这种将函数放到最前面的风格被称为前缀表示法（prefix notation）[①]。当然，当函数名是一个单词时，前缀表示法是相当常见的。例如，concat 作为函数名会出现在表达式的起始位置，这就符合大多数程序员的预期。

```
(concat [1 2] [3 4])
-> (1 2 3 4)
```

Clojure 只不过是像对待所有其他函数一样，把数学运算符也简单地放到起始位置罢了。

采用前缀表示法还有一个实际好处，你是可以很容易将其扩展为任意数量的参数。

```
(+ 1 2 3)
-> 6
```

甚至在没有参数的这种退化情况（degenerate case）下，它仍然能按照你期望的那样，返回一个零。这非常有利于消除为处理边界条件而产生的特例逻辑，此类逻辑往往非常脆弱。

```
(+)
-> 0
```

如你所料，Clojure 的许多数学和比较运算符，都具有与其他语言相同的名称和语义。加法、减法、乘法、比较和等于操作符都会按照你期望的方式来工作。

```
(- 10 5)
-> 5
(* 3 10 10)
-> 300
(> 5 2)
-> true
(>= 5 5)
-> true
(< 5 2)
-> false
(= 5 2)
-> false
```

① 更具体的说，它被称为剑桥波兰表示法（Cambridge Polish Notation）。

然而除法可能会令你大吃一惊。

```
(/ 22 7)
-> 22/7
```

如你所见，Clojure 内建了一个比例（Ratio）类型。

```
(class (/ 22 7))
-> clojure.lang.Ratio
```

如果你想要的是十进制除法，就得用浮点数的字面量来作为被除数。

```
(/ 22.0 7)
-> 3.142857142857143
```

如果只需要整数结果，那你可以使用 quot 和 rem 函数来获取整型的商和余数。

```
(quot 22 7)
-> 3
(rem 22 7)
-> 1
```

当你需要任意精度的浮点运算时，在数字后面追加一个大写的 M，就可以创建一个 BigDecimal 类型的字面量。

```
(+ 1 (/ 0.00001 1000000000000000000))
-> 1.0
(+ 1 (/ 0.00001M 1000000000000000000))
-> 1.000000000000000000000001M
```

为了得到任意精度的整数，可以通过在数字后面追加一个大写的 N，来创建一个 BigInt 类型的字面量。

```
(* 1000N 1000 1000 1000 1000 1000 1000)
-> 1000000000000000000000N
```

注意，算式中只需要一个 BigInt 字面量就够了，因为它会传染到整个计算过程当中。

2.1.2 符号

诸如+、concat 和 java.lang.String 这样的形式都被称为符号，用来为事物命名。例如，用+命名了这样一个函数，它能把一些东西加到一块儿。Clojure 中，符号用来对各式各样的东西命名。

- 函数，例如 str 和 concat。

- 操作符，例如+和-，它们终究不过是函数罢了。
- Java 类，例如 java.lang.String 和 java.util.Random。
- 命名空间和 Java 包，例如 clojure.core 和 java.lang。
- 数据结构和引用类型。

符号不能以数字开头，但可以包含字母、数字、加号（+）、减号（-）、乘号（*）、除号（/）、感叹号（!）、问好（?）、英文句号（.）和下划线（_）。此处列出的是 Clojure 承诺支持的合法符号的最小字符集。你应该在自己的代码中坚持只使用这些字符，但千万不要假设其他的 Clojure 代码也会如此。Clojure 会采用一些未文档化的其他字符作为其内部符号，另外将来也可能会为符号增加更多的合法字符。请查阅 Clojure 在线文档[①]，以获取合法符号字符列表的更新。

/和.会被 Clojure 加以特殊对待，用于支持命名空间，详情参见第 2.4.3 小节“命名空间”。

2.1.3　字符串与字符

字符串是另外一种读取器形式。Clojure 字符串就是 Java 字符串。它们使用双引号来划定界限，并且可以跨越多行。

```
"This is a\nmultiline string"
-> "This is a\nmultiline string"
"This is also
a multiline string"
-> "This is also\na multiline string"
```

如你所见，REPL 回显字符串的字面量时，总是包括了换行转义符。如果你确实“打印”了一个多行字符串，那它就会以多行的方式输出。

```
(println "another\nmultiline\nstring")
| another
| multiline
| string
-> nil
```

Clojure 并未封装大多数的 Java 字符串功能。作为替代，你可以使用 Clojure 的 Java 互操作形式来直接调用它们。

```
(.toUpperCase "hello")
-> "HELLO"
```

① http://clojure.org/reader。

toUpperCase 前面的句点告知 Clojure，应该将其视为一个 Java 方法，而非 Clojure 函数。

被 Clojure 封装了的字符串功能之一是 toString。你无需直接调用 toString，而是应该使用 Clojure 的 str 函数。

```
(str& args)
```

str 函数与 toString 有两点不同。一是它能接受多个参数，二是它会跳过 nil 而不引发错误。

```
(str 1 2 nil 3)
-> "123"
```

Clojure 字符同样也是 Java 字符。其字面语法是\{letter}，letter 可以是一个字母，或者下列这些字符的名称：backspace、formfeed、newline、return、space 和 tab。

```
(str \h \e \y \space \y \o \u)
-> "hey you"
```

和字符串一样，Clojure 并未封装 Java 的字符处理功能。因此，你同样可以使用 Java 互操作，比如 Character/toUpperCase。

```
(Character/toUpperCase \s)
-> \S
```

与 Java 互操作相关的形式，详见 2.5 节“调用 Java”。关于 Java 字符类 Character 的更多内容，请参见 API 文档 http://tinyurl.com/java-character。

字符串是由字符组成的序列。当你对字符串调用 Clojure 的序列处理函数时，你会得到由这些函数返回的一个字符序列。假设你希望通过交错插值的方式，用一段无关的消息来隐藏机密消息。你可以使用 interleave 函数来混合这两段消息以达成目标。

```
(interleave "Attack at midnight" "The purple elephant chortled")
-> (\A \T \t \h \t \e \a \space \c \p \k \u \space \r
\a \p \t \l \space \e \m \space \i \e \d \l \n \e
\i \p \g \h \h \a \t \n)
```

看起来不错，但你可能更希望得到的结果是字符串，而非序列。直接使用 str 函数把这些字符打包进一个字符串的想法极具诱惑力，但很可惜这是行不通的。

```
(str (interleave "Attack at midnight" "The purple elephant chortled"))
-> "clojure.lang.LazySeq@d4ea9f36"
```

最主要的问题是，str 函数接受的是数量可变的参数，但你传给它的参数只有一个，一个包含了参数列表的序列。解决方案是 apply 函数。

```
(apply f args* argseq)
```

apply 函数接受一个函数 f、一些可选的 args 和一个序列 argseq 作为参数。然后，他会调用 f，并将 args 和 argseq 解开为一个参数列表传给 f。下面使用(apply str ...)由字符序列来创建一个字符串。

```
(apply str (interleave "Attack at midnight" "The purple elephant chortled"))
-> "ATthtea cpku raptl em iedlneipghhatn"
```

你还可以再次使用(apply str ...)，显示那条机密消息。

```
(apply str (take-nth 2 "ATthtea cpku raptl em iedlneipghhatn"))
-> "Attack at midnight"
```

调用(take-nth 2 ...)，会从序列中依次剔除每第 2 个元素，这样就还原了被混淆的消息。

2.1.4　布尔值与 nil

Clojure 的布尔值规则很容易理解。

- true 为真，false 为假。
- 除了 false，在进行布尔求值的上下文中，nil 也为假。
- 除了 false 和 nil，其他任何东西在布尔求值的上下文中都为真。

Lisp 程序员们请注意：在 Clojure 中，空列表不为假。

```
;         (if 部分)                (else 部分)
(if () "We are in Clojure!" "We are in Common Lisp!")
-> "We are in Clojure!"
```

下面这个警告是针对 C 程序员的：在 Clojure 中，零也不为假。

```
;      (if 部分)       (else 部分)
(if 0 "Zero is true" "Zero is false")
-> "Zero is true"
```

谓词（predicate）是一种返回 true 或 false 的函数。作为 Clojure 惯例，给谓词命名时皆以问号结尾，例如 true?、false?、nil?和 zero?。

```
(true? expr)
(false? expr)
```

```
(nil? expr)
(zero? expr)
```

true?用于测试值是否确实为 true，而不仅仅是在布尔求值的上下文中为真。唯一能通过 true?测试的只有 true 本身。

```
(true? true)
-> true
(true? "foo")
-> false
```

nil?和 false?也是一样。只有 nil 能通过 nil?的测试，也只有 false 能通过 false?的测试。zero?可以对任何一种数值类型使用，如果值为零则返回 true。

```
(zero? 0.0)
-> true
(zero? (/ 22 7))
-> false
```

更多 Clojure 中的谓词。可在 REPL 中输入(find-doc #"\?$")来查看它们。

2.1.5 映射表、关键字和记录

Clojure 中，映射表是一种由键值对组成的容器，使用一对花括号作为其字面表示法。你可以使用映射表字面量来创建一个由编程语言发明人组成的查询表。

```
(def inventors {"Lisp" "McCarthy" "Clojure" "Hickey"})
-> #'user/inventors
```

值“McCarthy”被关联至键“Lisp”，同时值“Hickey”被关联至键“Clojure”。

如果想增强它的可读性，你可以使用逗号来分割每组键值对。Clojure 对此毫不在意，它会把逗号视为空格符。

```
(def inventors {"Lisp" "McCarthy", "Clojure" "Hickey"})
-> #'user/inventors
```

映射表同时也是函数。如果你把键作为参数传给映射表，那么它会返回与这个键对应的值，或是当键不存在时直接返回 nil。

```
(inventors "Lisp")
-> "McCarthy"
(inventors "Foo")
-> nil
```

你也可以使用更繁琐一些的 get 函数。

```
(get the-map key not-found-val?)
```

get 函数允许你为缺失的键指定一个默认返回值。

```
(get inventors "Lisp" "I dunno!")
-> "McCarthy"
(get inventors "Foo" "I dunno!")
-> "I dunno!"
```

由于 Clojure 的数据结构是不可变的，并且都正确的实现了 hashCode，所以任意一种 Clojure 数据结构都可以用做映射表的键。其中，Clojure 关键字是一种相当常见的键类型。

除了是以冒号（:）起头之外，关键字的其他部分与符号非常相似。看下面，关键字会解析为它们自身。

```
:foo
-> :foo
```

这是它们与符号之间最大的不同，符号总是会引用某种东西。

```
foo
-> CompilerException java.lang.RuntimeException:
Unable to resolve symbol: foo in this context
```

关键字会被解析为他们自身的这个特点，非常有利于令其成为映射表的键。你可以使用关键字作为键，来重新定义前面的那个发明人映射表。

```
(def inventors {:Lisp "McCarthy" :Clojure "Hickey"})
-> #'user/inventors
```

关键字同样也是函数。它们接受一个映射表作为参数，并在该映射表中查找其自身。只要把关键字和 inventors 的位置相互调换，你就可以通过调用映射表函数或关键字函数来查找语言的发明人了。

```
(inventors :Clojure)
-> "Hickey"
(:Clojure inventors)
-> "Hickey"
```

在调用诸如引用（第 5 章）和代理 API 那样的高阶函数时，这种灵活性就能大派用场。

如果有好几个映射表都拥有相同的键，那么你可以用 defrecord 创建一种记录类

型，对这一事实加以描述（同时也是强化）。

```
(defrecord name [arguments])
```

此处的参数名称会转换为键。这样当创建一条记录时，就需要传入相对应的值。下面使用 defrecord 创建了一种记录类型 Book。

```
(defrecord Book [title author])
-> user.Book
```

接下来，你就可以用 user.Book 来实例化一条记录了。

```
(->Book "title" "author")
```

一旦你实例化了一个 Book，你会发现它的行为与其他映射表简直像极了。

```
(def b (->Book "Anathem" "Neal Stephenson"))
-> #'user/b
b
-> #:user.Book{:title "Anathem", :author "Neal Stephenson"}
(:title b)
-> "Anathem"
```

记录也可以使用别的方法来实例化。最初的语法你应该已经见过了。

```
(Book. "Anathem" "Neal Stephenson")
-> #user.Book{:title "Anathem", :author "Neal Stephenson"}
```

你也可以采用字面语法来实例化一条记录。要这么做，只要照原样输入你在 REPL 中看到的返回内容即可。你会发现与前面唯一的区别是，记录的字面量必须使用全限定名。

```
#user.Book{:title "Infinite Jest", :author "David Foster Wallace"}
-> #user.Book{:title "Infinite Jest", :author "David Foster Wallace"}
```

到目前为止，你已经见过了数值字面量、列表、向量、符号、字符串、字符、布尔值、记录和 nil。其余的那些 Clojure 形式，会在需要时，在本书的后续章节中加以讨论。作为参考，你可以查阅表 2-1“Clojure 的形式”，那儿列出了所有本书中会用到的 Clojure 形式，包括简短的例子和完整讨论的章节指引。

2.2 读取器宏

读取器负责读入 Clojure 形式，并将其从文本转换为 Clojure 数据结构。除了那些

基本的形式，Clojure 读取器还能识别一组专门的读取器宏[①]。所谓读取器宏，是指由宏字符前缀触发的特殊读取器行为。

最常见的读取器宏是注释。触发注释的宏字符是分号（;），其对应的特殊读取器行为是“忽略后面的所有内容直至本行结束”。

相较于那些冗长的列表形式，读取器就是它们的缩写，这有利于减少不必要的视觉干扰。你已经见过其中的一个了。引号（'）可以阻止求值。

```
'(1 2)
-> (1 2)
```

'(1 2)与较长的(quote (1 2))等价。

```
(quote (1 2))
-> (1 2)
```

本书稍后会讨论其他的读取器宏。从表 2-2 中，你能找到一份语法快速概览，还能找到更多的相关讨论。

表 2-2　读取器宏

读取器宏	示例	主要章节
匿名函数	`#(.toUpperCase %)`	第 2.3 节“函数”
注释	`; single-line comment`	第 2.2 节“函数”
解引用（Deref）	`@form => (deref form)`	第 5 章“状态”
元数据	`^metadata form`	第 2.8 节“元数据”
引述（Quote）	`'form=> (quote form)`	第 2.1 节“形式”
正则表达式	`#"foo" => a java.util.regex. Pattern`	第 3.4.2 小节“序化正则表达式”
语法引述（Syntax-quote）	`'x`	第 7.3 节“让宏更加简单”
解引述（Unquote）	～	第 7.3 节“让宏更加简单”
解引述拼接（Unquote-splicing）	～@	第 7.3 节“让宏更加简单”
变量引述（Var-quote）	`#'x => (var x)`	第 5 章“状态”

Clojure 不允许程序定义新的读取器宏，其原因在 Clojure 邮件列表中[②]进行了阐述（还有为些展开的激烈的辩论）。如果你是从 Lisp 那也过来的，这可能会有点令人沮丧。

① 读取器宏与宏是完全不同的，在第 7 章中有详细讨论。
② http://tinyurl.com/clojure-reader-macros。

我们能体会你的痛苦。但在灵活性上做出的这个妥协，能给予 Clojure 一个更加稳定的内核。如果允许自定义读取器宏，很可能会降低 Clojure 程序的互操作性，并且难以阅读。

2.3 函数

所谓函数调用，在 Clojure 中，只不过是一个列表的起始元素可以被解析成函数而已。例如，下面调用了 str 函数，并将它的参数连接为一个字符串。

```
(str "hello" " " "world")
-> "hello world"
```

函数名通常表明了其单复数，例如 clear-agent-errors。如果函数是一个谓词，那么按照惯例，它的名称应该以一个问号结束。如下所示，这些以问好结尾的谓词，会对它们的参数进行类型测试。

```
(string? "hello")
-> true
(keyword? :hello)
-> true
(symbol? 'hello)
-> true
```

可以使用 defn 来定义你自己的函数。

```
(defn name doc-string? attr-map? [params*] body)
```

attr-map 表示关联到函数对象上的元数据。详情请参阅第 2.8 节“元数据”。为了说明函数定义中的其他部分，下面创建一个 greeting 函数，它接受一个名称，然后返回以“Hello”开头的问候语。

src/examples/exploring.clj

```
(defn greeting
  "Returns a greeting of the form 'Hello, username.'"
 [username]
  (str "Hello, " username))
```

你可以这样来调用 greeting。

```
(greeting "world")
-> "Hello, world"
```

你也可以这样来查阅 greeting 的文档。

```
user=> (doc greeting)
-------------------------
exploring/greeting
([username])
  Returns a greeting of the form‘Hello, username.’
```

如果 greeting 的调用者遗漏了 username 参数会如何？

```
(greeting)
-> ArityException Wrong number of args (0) passed to: user$greeting
   clojure.lang.AFn.throwArity (AFn.java:437)
```

Clojure 函数强调元数（arity），也就是它们期望获得的参数数量。如果你调用函数时传入的参数数目不正确，Clojure 会抛出一个 ArityException。如果你希望当调用者遗漏了 username 参数时，greeting 函数也能表达通用的问候，那么你可以使用 defn 的另外一种形式，它允许函数接受多组参数列表和函数主体。

```
(defn name doc-string? attr-map?
  ([params*] body)+)
```

同一个函数的不同元数之间能够彼此相互调用，所以，你就可以很容易地创建一个没有参数的 greeting，然后将功能委托给那个单参数的 greeting，并传入一个默认的 username。

src/examples/exploring.clj

```
(defn greeting
  "Returns a greeting of the form‘Hello, username.’
 Default username is‘world’."
  ([] (greeting "world"))
  ([username] (str "Hello, " username)))
```

检验一下这个新的 greeting 是否符合预期。

```
(greeting)
-> "Hello, world"
```

在参数列表中包含一个&号，你就能创建一个具有可变元数的函数。Clojure 会把所有剩余的参数都放进一个序列中，并绑定到&号后面的那个名称上。

下面的函数允许两个人约会时有可变数量的监护人相随。

src/examples/exploring.clj

```
(defn date [person-1 person-2 & chaperones]
  (println person-1 "and" person-2
    "went out with" (count chaperones) "chaperones."))
(date "Romeo" "Juliet" "Friar Lawrence" "Nurse")
| Romeo and Juliet went out with 2 chaperones.
```

在递归定义中，变参非常有用。具体示例请参阅第 4 章“函数式编程”。

为不同的元数编写函数实现是非常有用的。但如果你有面向对象编程的背景，那么你一定还会联想到“多态”（polymorphism），也就是能够根据类型的不同，来选取相应的实现。Clojure 能够做到的远不止于此。请参阅第 8 章“多重方法”和第 6 章“协议和数据类型”以获取更多详情。

defn 意在命名空间的顶层定义函数。但如果你希望能够通过函数来创建函数，那就应该使用匿名函数加以替代。

2.3.1 匿名函数

除了能用 defn 来创建具名函数以外，你还能用 fn 创建匿名函数。采用匿名函数至少有以下 3 个原因。

- 这是一个很简短且不言自明的函数，如果给它取名字的话，不会令可读性增强，反而使得代码更难以阅读。
- 这是一个仅在别的函数内部使用的函数，需要的是局部名称，而非顶级绑定。
- 这个函数是在别的函数中被创建的，其目的是为了隐藏某些数据。

用作过滤器的函数总是简短且不言自明的。例如，假设你要为一个由单词组成的序列创建索引，同时你并不关心那些字符数小于 3 的单词。那么，你可以编写一个这样的 indexable-word?函数。

src/examples/exploring.clj

```
(defn indexable-word? [word]
  (> (count word) 2))
```

接下来，你可以使用 indexable-word?从句子中提取那些可索引的单词。

```
(require '[clojure.string :as str])
(filter indexable-word? (str/split "A fine day it is" #"\W+"))
-> ("fine" "day")
```

上例中通过调用 split，将句子分解为单词，然后 filter 对每个单词调用 indexable-word?，并返回那些被 indexable-word?判定为 true 的单词。

匿名函数能让你仅用一行代码就做到相同的事情。下面是最简单的匿名函数形式 fn。

```
(fn [params*] body)
```

通过这种形式，你能在调用 filter 时直接插入 indexable-word?的实现。

```
(filter (fn [w] (> (count w) 2)) (str/split "A fine day" #"\W+"))
-> ("fine" "day")
```

还有一种采用隐式参数名称，也更加简短的匿名函数语法。其参数被命名为%1、%2，以此类推。对于第一个参数，你也可以使用%表示。该语法形如下。

```
#(body)
```

你可以使用这种更简短的匿名形式来重新调用 filter。

```
(filter #(> (count %) 2) (str/split "A fine day it is" #"\W+"))
-> ("fine" "day")
```

使用匿名函数的第二个动机是，确定想要一个具名函数，但该函数仅在其他函数的作用域内使用。继续这个 indexable-word?的例子，你可以像下面这样写。

src/examples/exploring.clj

```
(defn indexable-words [text]
  (let [indexable-word? (fn [w] (> (count w) 2))]
    (filter indexable-word? (str/split text #"\W+"))))
```

let 将你刚才写的那个匿名函数与名称 indexable-word?绑定在了一起，但这次它被限定在了 indexable-words 的词法作用域内。关于 let 的更多细节，请参见第 2.4 节“变量、绑定和命名空间”。验证一下这个 indexable-words，看其是否符合预期。

```
(indexable-words "a fine day it is")
-> ("fine" "day")
```

采用这种 let 和匿名函数的组合，相当于你对代码的读者说：“函数 indexable-word?有足够的理由拥有一个名称，但仅限于在 indexable-words 中。”

使用匿名函数的第三个原因是，有时你需要在运行期动态创建一个函数。此前，你已经实现了一个简单的问候函数 greeting。拓展一下思路，你还可以创建一个用来创建 greeting 函数的 make-greeter 函数。make-greeter 函数接受一个 greeting-prefix 参数，并返回一个新函数，这个新函数会将 greeting-prefix 和一个姓名组合起来，成为问候语。

src/examples/exploring.clj

```
(defn make-greeter [greeting-prefix]
  (fn [username] (str greeting-prefix ", " username)))
```

一般来说，为一个通过 fn 得到的函数命名是没有意义的，因为每次调用

make-greeter 都会创建一个不同的函数。然而，在某次调用了 make-greeter 之后，你也许会想为那个特殊的具体结果命名。如果真是这样，你可以使用 def 对 make-greeter 创建的函数进行命名。

```
(def hello-greeting (make-greeter "Hello"))
-> #'user/hello-greeting
(def aloha-greeting (make-greeter "Aloha"))
-> #'user/aloha-greeting
```

现在，你就可以像调用其它函数那样调用这些函数了。

```
(hello-greeting "world")
-> "Hello, world"
(aloha-greeting "world")
-> "Aloha, world"
```

此外，没有必要为每个问候函数都命名。你可以简单地创建一个问候函数，并将它放置到列表形式的第一个（函数）槽（slot）中。

```
((make-greeter "Howdy") "pardner")
-> "Howdy, pardner"
```

如你所见，不同的问候函数会记住创建它们时的那个 greeting-prefix 值。用更为正式的说法，这些问候函数对 greeting-prefix 的值构成了闭包（closures）。

2.3.2 何时使用匿名函数

匿名函数那极度简洁的语法并不总是恰当的。也许你实际上更偏向于明确化，喜欢创建诸如 indexable-word?这样的命名函数。这完全没有问题，并且如果 indexable-word?需要在多处调用时，这是理所当然的明智之选。

匿名函数只是一种选择，而非必须。只有当你发现它们可以令你的代码更具可读性时，才应该使用这样的匿名形式。如果开始越来越频繁地使用它们，不必惊讶，这只不过是你开始有一点习惯它们了。

2.4 变量、绑定和命名空间

当你使用 def 或 defn 定义了一个对象时，这个对象会被存储在一个 Clojure 变量（var）中。例如，下面的 def 创建了一个名为 user/foo 的变量。

```
(def foo 10)
-> #'user/foo
```

符号 user/foo 指向一个变量，该变量绑定了 10 这个值。如果你要求 Clojure 对符号 foo 进行求值，那么，它会返回与其关联的那个变量所绑定的值。

```
foo
-> 10
```

一个变量的初始值被称为它的根绑定（root binding）。有时候为一个变量提供线程内绑定（thread-local bindings）是非常有用的，该话题参见第 5.5 节“用变量管理线程内状态”。

你可以直接引用一个变量。特殊形式（special form）var 能返回变量自身，而不是变量的值。

```
(var a-symbol)
```

你可以使用 var 来获取绑定到 user/foo 上的那个变量。

```
(var foo)
-> #'user/foo
```

在 Clojure 代码中，你几乎找不到直接用 var 的地方。相反，你会见到与其等价的读取器宏#'，它同样会返回与符号绑定的那个变量。

```
#'foo
-> #'user/foo
```

那什么时候你会想要直接去引用一个变量呢？大多数时候这是不需要的，你总是可以简单的忽略符号与变量之间的差异。

但请务必留意，除了用来保存值以外，变量还有许多其他能力。

- 同一个变量，可以在多个命名空间中具有别名（参见第 2.4.3 小节“命名空间”）。这样你就可以使用便利的短名称了。
- 变量可以有元数据（第 2.8 节“元数据”）。元数据包括文档（第 1.3.2 小节“查找文档”）、用于优化的类型提示，还有单元测试。
- 变量可基于每个线程进行动态重绑定（第 5.5 节“用变量管理线程内状态”）。

2.4.1　绑定

除了变量与名称之间的绑定之外，也有针对其他类型的绑定。例如，在函数调用

中，参数值与参数名称之间的绑定。看下面这次调用，在 triple 函数内部，10 和名称 number 绑定了。

```
(defn triple [number] (* 3 number))
-> #'user/triple
(triple 10)
-> 30
```

函数的参数绑定具有词法范围：它们仅在函数主体代码的内部可见。函数并不是创建词法绑定的唯一方式。作为另外一个特殊形式，let 的作用就是来建立一组词法绑定。

```
(let [bindings*] exprs*)
```

这里的 bindings 会在随后的 exprs 中生效，此外，exprs 中最后一个表达式的值，就会成为 let 的返回值。

试想你要根据给定的 bottom、left 和 size，为一个正方形的四个角建立坐标。你可以基于这些给出的值，使用 let 来绑定 top 和 right 坐标。

src/examples/exploring.clj

```
(defn square-corners [bottom left size]
  (let [top (+ bottom size)
        right (+ left size)]
    [[bottom left] [top left] [top right] [bottom right]]))
```

let 对 top 和 right 进行了绑定。这省却了你要来来回回计算 top 和 right 的麻烦。两者都需要计算两次。然后 let 返回了其最后一个形式，在本例中这也成为了 square-corners 函数的返回值。

2.4.2 解构

在许多编程语言中，即使你需要访问的只是某个容器中的一部分元素，你也不得不把整个容器都绑定到一个变量上。

假想你正在使用一个保存了图书作者的数据库。你同时保存了姓和名字，但有的函数只需要名字就足够了。

src/examples/exploring.clj

```
(defn greet-author-1 [author]
  (println "Hello," (:first-name author)))
```

greet-author-1 函数一切正常。

```
(greet-author-1 {:last-name "Vinge" :first-name "Vernor"})
| Hello, Vernor
```

可是你不得不绑定整个 author。这一点实在无法令人满意。事实上你并不需要整个 author，你需要的只是 first-name 罢了。Clojure 中通过解构来解决这一问题。在任意一个需要绑定名称的位置，你都可以在绑定式中嵌入一个向量或是映射表，藉此深入容器内部，绑定你真正需要的那个部分。下面是一个 greet-author 的变形，它仅绑定了名字。

src/examples/exploring.clj

```
(defn greet-author-2 [{fname :first-name}]
  (println "Hello," fname))
```

{fname :first-name}告诉 clojure，应该把参数 fname 绑定至:first-name。greet-author-2 具有和 greet-author-1 相同的行为。

```
(greet-author-2 {:last-name "Vinge" :first-name "Vernor"})
| Hello, Vernor
```

正如使用映射表可以解构任何关联性容器，你也能用向量来解构任何顺序性容器。例如，你可以仅绑定三维坐标空间中的前两个坐标。

```
(let [[x y] [1 2 3]]
  [x y])
-> [1 2]
```

表达式[x y]对向量[1 2 3]进行解构，将 x 和 y 绑定到了 1 和 2 上。由于最后一个元素 3 没有符号与之排列对应，所以它也就不会与任何东西绑定。

有时候你会想要跳过容器的几个起始元素。此处展示了你要怎样做，才能只绑定 z 坐标。

```
(let [[_ _ z] [1 2 3]]
  z)
-> 3
```

下划线（_）是一个合法的符号，同时作为惯用法，它还用来表示：“我对这个绑定毫不关心”。由于绑定是从左向右进行的，所以“_”实际上被绑定了两次。

```
; 不符合惯例!
(let [[_ _ z] [1 2 3]]
  _)
-> 2
```

另外它也可以同时绑定整个容器与容器内的元素。在解构表达式内部，:as 字句允许你绑定整个闭合结构。例如，你可以单独绑定 *x* 和 *y* 坐标，并把整个容器绑定至 coords，以报告维度的总数。

```
(let [[x y :as coords] [1 2 3 4 5 6]]
  (str "x: " x ", y: " y ", total dimensions " (count coords)))
-> "x: 1, y: 2, total dimensions 6"
```

下面尝试使用解构来创建一个 ellipsize 函数。ellipsize 接受一个字符串，并返回该字符串的前三个单词，并在末尾加上省略号。

src/examples/exploring.clj

```
(require '[clojure.string :as str])
(defn ellipsize [words]
  (let [[w1 w2 w3] (str/split words #"\s+")]
    (str/join " " [w1 w2 w3 "..."])))
(ellipsize "The quick brown fox jumps over the lazy dog.")
-> "The quick brown ..."
```

split 基于空格来对字符串进行切分，然后使用解构形式来[w1 w2 w3]捕获其前三个单词。正如我们所期望的，解构忽略了其他内容。最后，通过 join 将这三个单词重组，并在末尾追加省略号。

解构本身就是一门小型的语言，还有其他几个特性未能在此处展示。第 5.6 节“Clojure 贪吃蛇”中的贪吃蛇游戏，大量的使用了解构。完整的解构选项列表，请参见 let 的在线文档[①]。

2.4.3 命名空间

根绑定存在于命名空间中。当你启动 REPL，并创建了一个绑定时，就能证实这一点。

```
user=> (def foo 10)
-> #'user/foo
```

提示符 user=>说明你当前正工作在 user 命名空间下。[②]你可以将 user 视为一个用于探索性开发的临时命名空间。

当 Clojure 解析名称 foo 时，它会用当前命名空间 user 对 foo 进行命名空间限定

① http://clojure.org/special_forms。

② 为保持简洁，本书中列出的大多数的 REPL 会话都省却了 REPL 提示符。但在本节中，如果当前命名空间显得非常重要，那么就会包含 REPL 提示符。

（namespace-qualifies）。你可以通过调用 resolve 来加以验证。

```
(resolve sym)
```

resolve 会返回在当前命名空间中，解析符号得到的变量或是类。下面使用 resolve 对符号 foo 进行显式解析。

```
(resolve 'foo)
-> #'user/foo
```

你可以使用 in-ns 来切换命名空间，必要时 Clojure 还会新建一个新的。

```
(in-ns name)
```

试试看创建一个 myapp 命名空间。

```
user=> (in-ns 'myapp)
-> #<Namespace myapp>
myapp=>
```

你现在已经位于 myapp 命名空间中了，这时候你 def 或 defn 的任何东西都将属于 myapp。

当你使用 in-ns 新建了一个命名空间时，Clojure 会自行导入 java.lang 包。

```
myapp=> String
-> java.lang.String
```

在学习 Clojure 期间，每当你转移到一个新的命名空间时，你都应该立即使用 use 来导入 clojure.core 命名空间，这样 Clojure 的核心函数才能在这个新的命名空间中使用。

```
myapp=> (clojure.core/use 'clojure.core)
-> nil
```

默认情况下，java.lang 以外的其他类都必须使用全限定名。例如，你不能只写 File。

```
myapp=> File/separator
-> java.lang.Exception: No such namespace: File
```

相反，你必须指定全限定的 java.io.File。请注意，你的文件分割符可能会与此处显示的不同。

```
myapp=> java.io.File/separator
-> "/"
```

倘若不想使用全限定类名，你可以使用 import 把一个或者多个类名从 Java 包映射

到当前命名空间中。

```
(import '(package Class+))
```

一旦导入了一个类，你就可以使用其短名称了。

```
(import '(java.io InputStream File))
-> java.io.File
(.exists (File. "/tmp"))
-> true
```

import 仅用于 Java 类。你如果想使用另一个命名空间中的 Clojure 变量，同样也需要采用其全限定名，或者将其名称映射到当前空间中。例如，位于 clojure.string 的 Clojure 函数 split。

```
(require 'clojure.string)
(clojure.string/split "Something,separated,by,commas" #",")
-> ["Something" "separated" "by" "commas"]

(split "Something,separated,by,commas" #",")
-> Unable to resolve symbol: split in this context
```

为了在当前命名空间中引入 split 别名，可以包含对 split 的命名空间 clojure.string 调用 require，并用 str 用作其别名。

```
(require '[clojure.string :as str])
(str/split "Something,separated,by,commas" #",")
-> ["Something" "separated" "by" "commas"]
```

就像早些时候展示的那样，这种简单形式的 require 会把 clojure.string 中所有的公共变量引入到当前命名空间内，并且还可以通过别名 str 来访问它们。不过，这可能会令人感到有些困惑，因为引入了哪些名称其实并不明确。

作为惯例，在一个 Clojure 源文件的顶部，我们会使用 ns 宏来 import Java 类和 require 命名空间。

```
(ns name& references)
```

ns 宏将当前命名空间（可通过*ns*获取）设置为 name，必要时还会创建这个命名空间。references 部分则可以包含:import、:require 和:use。它们的工作方式与各自对应的同名函数类似。这样仅需要一个形式，就可以完成命名空间映射相关的所有设置。例如，在本章实例代码的顶部，就调用了 ns。

src/examples/exploring.clj

```
(ns examples.exploring
  (:require [clojure.string :as str])
  (:import (java.io File)))
```

Clojure 命名空间函数的功能，要比我在此处展示的多得多。

你可以遍历命名空间，并随时添加或者删除映射。欲了解更多信息，可在 REPL 中输入以下命令。由于我们刚才在 REPL 中执行过一些操作，所以还需要确保我们位于 user 命名空间中，这样 REPL 工具才能有效地为我们工作。

```
(in-ns 'user)
(find-doc "ns-")
```

或是浏览位于 http://clojure.org/namespaces 的文档。

2.5　调用 Java

Clojure 提供了简单、直接的语法用来调用 Java 代码，包括：创建对象、调用方法、访问静态方法和字段。此外，Clojure 提供的语法糖，使得用 Clojure 调用 Java 时，甚至比用 Java 自己来调用还要简洁！

并非所有的 Java 类型都是生来平等：基本类型与数组就大不一样。对于这些 Java 的特例，Clojure 同样允许你直接访问它们。最后，Clojure 还提供了一组便利的函数用来处理一些常见任务，不要小看了它们，对于 Java 而言，处理那些任务可是相当笨拙的。

2.5.1　访问构造函数、方法和字段

在许多同 Java 交互的场景中，第一步都要创建 Java 对象。为此，Clojure 的 new 应运而生。

```
(new classname)
```

来创建一个 Random 对象试试看。

```
(new java.util.Random)
-><Random java.util.Random@667cbde6>
```

REPL 在打印这个新的 Random 实例时，只是简单的调用了它的 toString()方法。为了随后能使用这个 Random 实例，你还需要把它保存到某个地方。现在，我们先简单的用 def 把它保存到一个 Clojure 变量中。

```
(def rnd (new java.util.Random))
-> #'user/rnd
```

现在，你可以使用 Clojure 的句点（.）这个特殊形式来调用 rnd 的方法。

```
(. class-or-instance member-symbol & args)
(. class-or-instance (member-symbol & args))
```

例如，下述代码调用了 nextInt()方法的无参数版本。

```
(. rnd nextInt)
-> -791474443
```

Random 还有另外一个 nextInt()，它接受一个参数。你只要把参数追加至列表中，就能调用这个单参数的版本了。

```
(. rnd nextInt 10)
-> 8
```

在之前的调用中，句点被用来访问实例方法。但实际上它对各种类成员都有效：无论是字段还是方法，静态的或是的实例。下面你会看到如何使用句点来获得 pi 的值。

```
(. Math PI)
-> 3.141592653589793
```

注意，Math 没有采用全限定名。因为没有必要那么做，Clojure 会自动导入 java.lang。但 java.util 就没有这么幸运了，为了避免到哪儿都要输入冗长的 java.util.Random，你可以用 import 明确的将其导入。

```
(import [& import-lists])
; import-list => (package-symbol & class-name-symbols)
```

import 接受数量可变的列表作为参数，每个列表的第一项是要导入的包名称，其余部分则是导入的项的名称。因为执行了下面这个 import，接下来就可以使用非全限定的名称来访问 Random、Locale 和 MessageFormat 了。

```
(import '(java.util Random Locale)
        '(java.text MessageFormat))
-> java.text.MessageFormat

Random
-> java.util.Random

Locale
-> java.util.Locale
```

```
MessageFormat
-> java.text.MessageFormat
```

至此，你几乎已经掌握了通过 Clojure 调用 Java 所需的一切知识。借助它们，你现在可以做到以下几点。

- 导入类名
- 创建实例
- 访问字段
- 调用方法

然而，这种程度的语法还不足以令人感到特别兴奋。充其量，这也就是“采用另外一种方式来打括号的 Java”罢了。好戏还在后头，第 9 章中大有乾坤。

2.5.2　Javadoc

尽管通过 Clojure 访问 Java 非常容易，但仅凭记忆来掌握 Java 的海量细节，无疑是一个巨大的挑战。因此，Clojure 提供了一个 javadoc 函数，它能使你的生活轻松许多。当你在 REPL 中探索时，尽情体验它为你带来的愉悦吧。

```
(javadoc java.net.URL)
->
```

2.6　流程控制

Clojure 中用于流程控制的形式数量极少。本节中，你会遇到 if、do 和 loop/recur。人们后来发现，有了它们，几乎就别无所求了。

2.6.1　分支结构与 if

Clojure 的 if 会对其第一个参数进行求值。倘若结果逻辑为真，就返回对第二个参数求值的结果。

src/examples/exploring.clj

```
(defn is-small? [number]
  (if (< number 100) "yes"))

(is-small? 50)
-> "yes"
```

如果传给 if 的第一个参数逻辑为假，is-small?会返回 nil。

```
(is-small? 50000)
-> nil
```

如果你希望定义“else”部分定义一个结果，将其作为 if 的第三个参数即可。

src/examples/exploring.clj

```
(defn is-small? [number]
  (if (< number 100) "yes" "no"))

(is-small? 50000)
-> "no"
```

流程控制宏 when 和 when-not 建立在 if 的基础之上，参见第 7.2.3 小节“when 与 when-not”。

2.6.2 用 do 引入副作用

Clojure 的 if，其每个分支只能有一个形式。如果你想在某个分支中多做几件事情，该如何是好？例如，你可能想要记录被选中的究竟是哪条分支。do 可以接受任意数量的形式，对这些形式逐个求值，并返回最后一个形式的求值结果。

在 if 中，你可以使用 do 来打印一条日志语句。

src/examples/exploring.clj

```
(defn is-small? [number]
  (if (< number 100)
    "yes"
    (do
      (println "Saw a big number" number)
      "no")))
```

结果如下。

```
(is-small? 200)
| Saw a big number 200
-> "no"
```

这是一个出现了副作用的例子。对计算 is-small?的返回值来说，println 没有任何贡献。相反，它触及了函数外部的世界，并确确实实“做了某些事情”。

为了能对纯函数和副作用进行融合，许多编程语言不惜采用极其古怪的方式。但 Clojure 不这样。在 Clojure 中，哪儿有副作用一眼就能看出来。do 就是用来申明“注

意，接下来会有副作用”的方式之一。因为除了最后一个形式，do 会把所有其他形式的返回值都给忽略掉，所以，这也就意味着对于那些被忽略掉的形式来说，必须具有某种副作用才有其存在的意义。

2.6.3　循环与 loop/recur

在 Clojure 中，loop 就是流程控制的瑞士军刀。

```
(loop [bindings *] exprs*)
```

loop 与 let 的工作方式颇为相似，首先建立绑定 bindings，然后对 expres 求值。不同的地方是，loop 设置了一个循环点（recursion point），随后这个循环点将成为特殊形式 recur 返回目标。

```
(recur exprs*)
```

早先由 Loop 建立的绑定，会被 recur 重新绑定为新的值，并且控制程序流程返回到 loop 的顶端。例如，下面的 loop/recur 会返回一组倒计数。

src/examples/exploring.clj

```
(loop [result [] x 5]
  (if (zero? x)
    result
    (recur (conj result x) (dec x))))
-> [5 4 3 2 1]
```

首次进入时，loop 把 result 和一个空向量进行了绑定，并把 x 绑定为 5。由于 x 不为零，recur 随后对名称 x 和 result 再次进行了绑定。

- result 被绑定至一个新的向量，该向量是通过连接早先的 result 和 x 得到的。
- x 则被绑定为前一个 x 的递减结果。

接下来程序流回到了 loop 的顶端。由于这一次 x 仍然不为零，循环继续，再次对 result 加以累积并递减 x。最终，x 递减为零，if 终结了循环并返回 result。

去掉 loop，你就能让 recur 递归至函数的起始位置。这就使得编写那种把整个主体都用作隐式循环的函数变得极为容易。

src/examples/exploring.clj

```
(defn countdown [result x]
  (if (zero? x)
```

```
    result
    (recur (conj result x) (dec x))))
(countdown [] 5)
-> [5 4 3 2 1]
```

尽管 recur 结构非常强大。但你用到它的机会并不多，因为许多常用的循环已经作为 Clojure 序列库的一部分直接提供了。

例如，倒计数也可以用下面的任意一种方式来表示。

```
(into [] (take 5 (iterate dec 5)))
-> [5 4 3 2 1]
(into [] (drop-last (reverse (range 6))))
-> [5 4 3 2 1]
(vec (reverse (rest (range 6))))
-> [5 4 3 2 1]
```

现在还不到深究它们的时候，目前只要知道，相比直接使用 recur，它们是更加常用的替代方式。第 3.2 节“使用序列库”中，有关于此处用到的这些序列库函数的讨论。另一方面，Clojure 无法进行自动尾部调用优化（tail-call optimization，TCO）。然而，对 recur 的调用则会得到优化。第 4 章“函数式编程”中定义了何为尾部调用优化，并且对递归和尾部调用优化的细节进行了探索。

至此，你已经领略了相当多的语言特性，但那种能变化的“变量”却始终没有出现。有些事物的确会发生变化，第 5 章“状态”将会向你展示 Clojure 如何处理可以改变的“引用”。但大多数传统语言中的变量，既无必要，同时还相当危险。让我们来看看 Clojure 是如何摆脱它们的。

2.7 我的 for 循环哪儿去了

Clojure 没有 for 循环，也没有可以直接改变的变量[①]。那么，应该如何编写你习惯于使用 for 来完成的那些循环呢？

这次我们不再凭空假想出一个示例了，我们决定在开源的 Java 源码中翻一翻，随便找出一个使用了 for 循环和变量的方法，然后将其移植到 Clojure。我们打开了应用广泛的 Apache Commons 项目，并选择 Commons Lang 中的 StringUtils 类：这是一个只需要少量领域知识即可理解的类。随后我们开始寻找那种在多处用到 for 循环和局部变量的方法，最后找到了 indexOfAny。

① Clojure 提供了间接可变的引用，但这需要在你的代码中显式地指明才行。详情参阅第 5 章“状态”。

data/snippets/StringUtils.java

```
// 源自 Apache 的 Commons Lang, 参见 http://commons.apache.org/lang/
public static int indexOfAny(String str, char[] searchChars) {
    if (isEmpty(str) || ArrayUtils.isEmpty(searchChars)) {
        return -1;
    }
    for (int i = 0; i < str.length(); i++) {
        char ch = str.charAt(i);
        for (int j = 0; j < searchChars.length; j++) {
            if (searchChars[j] == ch) {
                return i;
            }
        }
    }
    return -1;
}
```

indexOfAny 对 str 中的字符进行遍历，并报告第一处能与 searchChars 中任意一个字符相匹配的字符位置，如果未能成功匹配则返回-1。

下面的示例结果来自于 indexOfAny 文档。

```
StringUtils.indexOfAny(null, *)               = -1
StringUtils.indexOfAny("", *)                 = -1
StringUtils.indexOfAny(*, null)               = -1
StringUtils.indexOfAny(*, [])                 = -1
StringUtils.indexOfAny("zzabyycdxx",['z','a']) =  0
StringUtils.indexOfAny("zzabyycdxx",['b','y']) =  3
StringUtils.indexOfAny("aba", ['z'])          = -1
```

indexOfAny 中有两处 if、两处 for、三处可能的返回点和三个可变的局部变量，同时方法的长度为 14 行（用 David A. Wheeler 的 SLOCCount 工具[①]统计得出）。

现在让我们来一步步地创建 Clojure 版本的 index-of-any。倘若只想要找出匹配，使用 Clojure 的 filter 即可做到。但我们还希望能够找出匹配位置的索引。所以，首先让我们创建一个 indexed 函数，它接受一个容器作为参数，对该容器建立索引并返回结果。

src/examples/exploring.clj

```
(defn indexed [coll] (map-indexed vector coll))
```

indexed 返回一个序列，该序列由成对的[idx elt]组成。下面试试看对一个字符串建立索引。

① http://www.dwheeler.com/sloccount/。

```
(indexed "abcde")
-> ([0 \a] [1 \b] [2 \c] [3 \d] [4 \e])
```

下一步，我们需要找出字符串中所有与搜索集相匹配的字符索引。

让我们创建一个与 Clojure 的 filter 相似的 index-filter 函数，只不过它返回的是索引，而非匹配项本身。

src/examples/exploring.clj

```
(defn index-filter [pred coll]
  (when pred
    (for [[idx elt] (indexed coll) :when (pred elt)] idx)))
```

Clojure 的 for 用于序列解析（sequence comprehension，参见第 3.2.4 小节“序列转换”），而不是循环。仅当（pred elt）为真时，（indexed coll）中的索引/元素对才会与名称 idx 和 elt 绑定。最后，根据每一对匹配结果的 idx 值将出结果序列。

Clojure 的集合（set）本身也可以用做函数，判定目标是否存在。所以，你可以向 index-filter 传入一个字符集合和一个字符串，然后得到同属于它们两者的字符在字符串中的索引。快用几个字符串和字符集合尝试一下吧。

```
(index-filter #{\a \b} "abcdbbb")
-> (0 1 4 5 6)
(index-filter #{\a \b} "xyz")
-> ()
```

至此，我们甚至超出了既定目标。index-filter 返回了所有匹配位置的索引，然而我们仅需要第一处而已。因此，下面的 index-of-any 只是简单的摘取了 index-filter 结果中的第一项。

src/examples/exploring.clj

```
(defn index-of-any [pred coll]
  (first (index-filter pred coll)))
```

下面用了几个不同的输入来测试 index-of-any 是否正常。

```
(index-of-any #{\z \a} "zzabyycdxx")
-> 0
(index-of-any #{\b \y} "zzabyycdxx")
-> 3
```

相比 Java 那样的命令式版本，Clojure 的这个版本无论从哪个方面来度量，都要简单得多（参见表 2-2“命令式与函数式 indexOfAny 的复杂度对比”）。究竟是什么导致

了这样的差异呢？

- 命令式的 indexOfAny 必须处理一些特殊情况：字符串为 null，或者是个空字符串；搜索字符集为 null，或者是个空集合。为了处理这些特殊情况，流程分支和方法退出点不可避免的增多了。然而对于函数式而言，无须任何显式的编码，绝大多数此类特例都能得到正确的处理。
- 命令式的 indexOfAny 引入了局部变量用于遍历容器（字符串和字符集合）。通过使用 map 这样的高阶函数和 for 这样的序列解析式，函数式的 index-of-any 能够规避所有对变量的需求。

不必要的复杂性如同雪球一样越滚越大。例如，命令式的 indexOfAny 的特殊情况分支中，使用了魔数（magic number）-1 表示未匹配。那么，这个魔数是否应该定义为一个符号常量呢？无论你认为正确答案是什么，在函数式的版本中，这个问题本身就根本不存在。此外，除了更加短小并且更为简单，函数式的 index-of-any 也更加通用。

- indexOfAny 只能检索字符串，然而 index-of-any 可以检索任意序列。
- indexOfAny 只能依据一组字符进行匹配，而 index-of-any 可以依据任意谓词进行匹配。
- indexOfAny 只能返回第一处匹配，而 index-filter 能够返回所有的匹配，并且还能进一步与其他过滤器进行组合。

为了体验函数式的 index-of-any 能通用到什么程度，你可以参照我们刚刚编写的代码，从一系列抛硬币的结果中，找出第三次正面朝上的索引位置。

```
(nth (index-filter #{:h} [:t :t :h :t :h :t :t :t :h :h])
2)
-> 8
```

表 2-3　命令式与函数式 indexOfAny 的复杂度对比

指标	代码行数	分支数	方法退出点	变量数
命令式版本	14	4	3	3
函数式版本	6	1	1	0

表 2-3 表明了编写函数式风格的 index-of-any[①]，不需要循环或者变量。此外相比命令式的 indexOfAny 而言，函数式的 index-of-any 更简单、更不容易出错，也更加通用。在规模较大的代码中，这些优势会变得更有说服力。

① 值得一提的是，你也可以使用原生的 Java 编写一个函数式风格的 indexOfAny，尽管这与惯例不符。等 Java 语言提供了闭包功能之后，也许这么做会变得更顺畅一些。更多信息参见 http://functionaljava.org/。

2.8 元数据

维基百科是这么解释元数据的[①]：用来“描述数据的数据。”这样说没错，但还不够具体。在 Clojure 中，元数据是与一个对象逻辑上的值产生正交的那些数据。例如，一个 person 的名和姓是普通的旧式数据（plain old data）。但一个 person 对象可以被序列化为 XML 这件事情，其实与 person 对象本身毫不相干，这就是元数据。同样，person 对象当前正处于脏状态，需要被刷新到数据库中，这也是元数据。

读取器元数据

Clojure 语言自身在许多地方用到了元数据。例如，变量持有一个元数据映射表，包含了文档、类型信息和源码信息。此处显示了变量 str 的元数据。

```
(meta #'str)
-> {:ns #<Namespace clojure.core>,
    :name str,
    :file "core.clj",
    :line 313,
    :arglists ([] [x] [x & ys]),
    :tag java.lang.String,
    :doc "With no args, ... etc."}
```

表 2-4 展示了一些通用元数据的键以及它们的用途。

表 2-4 通用元数据的键

元数据键	用途
:arglists	被 doc 用到的参数信息
:doc	被 doc 用到的文档
:file	源文件名
:line	源代码行号
:macro	true 表示这是一个宏
:name	本地名称
:ns	命名空间
:tag	预期的参数或返回值类型

一个变量的大部分元数据都是由 Clojure 编译器自动添加的。为了给一个变量添加

① http://en.wikipedia.org/wiki/Metadata。

你自己的元数据键/值对，可以使用元数据读取器宏。

```
^metadata form
```

例如，你可以创建一个简单的 shout 函数，用来把字符串转换为大写，并用:tag 注明其参数和返回值都必须是字符串。

```
; 稍后会在后面看到它的元数据
(defn ^{:tag String} shout [^{:tag String} s] (.toUpperCase s))
-> #'user/shout
```

检查一下 shout 的元数据，看看 Clojure 是否添加了:tag。

```
(meta #'shout)
-> {:arglists ([s]),
    :ns #<Namespace user>,
    :name shout,
    :line 32,
    :file "NO_SOURCE_FILE",
    :tag java.lang.String}
```

你提供了:tag，Clojure 则提供了其它的一些键。其中:file 的值为 NO_SOURCE_FILE，表示这些代码是在 REPL 中录入的。

因为元数据:tag 实在是太常用了，你也可以使用其简化形式^Classname，它会被展开为^{:tag Classname}。使用这个简化形式，你可以这么来重写 shout。

```
(defn ^String shout [^String s] (.toUpperCase s))
-> #'user/shout
```

在你阅读函数定义时，如果发觉元数据会造成视觉混乱，也可以把它们放到最后。这得用到 defn 的一个变体，先是一或多个带括号的函数主体，随后紧接一个元数据映射表。

```
(defn shout
  ([s] (.toUpperCase s))
  {:tag String})
```

2.9　小结

这章可真够长的。不妨回顾一下你已经走过了多少地方：你学会了实例化基础字面类型；定义并调用函数；管理命名空间；读取和写入元数据。你还学会了编写纯粹的函数式代码，也能在需要时，轻松地引入副作用。此外，你已经遇见了包括读取器

宏、特殊形式和解构在内的一些 Lisp 概念。

对于大多数语言来说，往往需要好几百页才能覆盖上述内容，而我们只用了一章。真是因为 Clojure 的方式要简单得多吗？某种程度上，是的。本章能如此高效，有一半的荣誉应归于 Clojure。Clojure 优雅的设计和抽象决策，使得这门语言相较大多数其它语言，学起来要更加轻松。

乍一看，学习这门语言似乎并不十分轻松。那是得益于 Clojure 强大的力量，我们的前进速度大大超越了其他大多数的编程语言书籍罢了。

因此，本章的另外一半荣誉应属于你，亲爱的读者。Clojure 会对你的付出给予多得多的回报。要想充分领会本章的那些示例，并对得心应手地使用 REPL，你可能还需要一些时间。没关系，本书的剩余部分有足够多的机会能让你做到这一点。

第 3 章

一切皆序列

程序操作的对象是数据。在最底层，程序面对的是字符串、列表、向量、映射表、集合和树这样的数据结构。与此同时，在较高的层面上，同样的数据结构抽象也一再出现。例如以下几个方面证明了此点。

- XML 数据是一棵树。
- 数据库结果集可以被看作是列表或向量。
- 目录层次也是树。
- 文件通常被看作是一个大的字符串，或是由文本行组成的向量。

在 Clojure 中，所有这些数据结构都可以通过同一个抽象概念来访问：序列。

序列（seq，发音“seek”）是一种逻辑上的列表。说它是逻辑上的，是因为 Clojure 的序列，并没有被捆绑在某种列表的实现细节上（比如说 Lisp 的 cons cell，列表构造单元，cons 的历史参见“Cons 的起源”）。相反，序列是一种抽象，可以用于任何地方。

可被视为序列的容器，被称为可序化的（seq-able，发音“SEEKa-bull”）。本章中，你会遇到各种各样可序化的容器。

- 所有的 Clojure 容器
- 所有的 Java 容器
- Java 数组和字符串
- 正则表达式的匹配结果
- 目录结构
- 输入/输出流

- XML 树

你还会遇到序列库，这是一组函数，对任何可序化的东西都有效。由于有太多太多的东西都是序列，所以相较其他语言中的容器 API 而言，Clojure 序列库更强大，也更通用。序列库包含了用于创建、过滤和转换数据的函数。这些函数在 Clojure 中，不仅仅扮演着容器 API 的角色，还替代了许多你在命令式语言中需要手工编写的循环结构。

在本章，你将会成为一名 Clojure 序列的进阶用户。你将会看到如何使用一组极具表达力的常用函数，来处理范围广泛到令人难以置信的数据类型。随后，在下一章（第 4 章“函数式编程”），你将会学习函数式风格，本章介绍的序列库都是基于这种风格编写的。

3.1 一切皆序列

每一种聚合的（aggregate）数据结构，在 Clojure 中都能被视为序列。序列具有三大核心能力。

- 你能够得到序列的第一个元素。

```
(first aseq)
```

如果参数 aseq 是空的或者是 nil，则 first 返回 nil。

- 你能够获取第一个元素后面的一切东西，换句话说，就是序列的剩余（rest）部分。

```
(rest aseq)
```

如果没有更多的项，则 rest 返回一个空序列（而不是 nil）。

- 你可以通过向现有序列的前端添加元素，来创建一个新的序列。这就是所谓的 cons。

```
(cons elem aseq)
```

在 Clojure 内部，这三个功能是在 Java 接口 clojure.lang.ISeq 中声明的。在阅读 Clojure 代码时尤其要记住这一点，因为名称 ISeq 常被用来与 seq 进行互换。

seq 函数会返回一个序列，该序列源自任何一个可序化的其他容器。

```
(seq coll)
```

如果 coll 是空的或者是 nil，则 seq 返回 nil。next 函数也会返回一个序列，该序列由除第一个元素以外的其他所有元素组成。

```
(next aseq)
```

(next aseq)等价于 (seq (rest aseq))。表 3-1“澄清 rest 和 next”阐明了 rest 与 next 的行为方式。

表 3-1　澄清 rest 和 next

形式	结果
(rest ())	()
(next ())	nil
(seq (rest ()))	nil

如果你具有 Lisp 背景，想必已经料到这些序列函数能作用于列表。

```
(first '(1 2 3))
-> 1

(rest '(1 2 3))
-> (2 3)

(cons 0 '(1 2 3))
-> (0 1 2 3)
```

Clojure 中，还是这些函数，对其他数据结构也同样有效。你可以把向量作为序列。

```
(first [1 2 3])
-> 1

(rest [1 2 3])
-> (2 3)

(cons 0 [1 2 3])
-> (0 1 2 3)
```

当你对向量使用 rest 或 cons 时，得到的结果是一个序列，而非向量。就像你从前面的输出中看到的那样，在 REPL 中，序列被打印出来以后，就好象是个列表一样。你可以用 class 函数来获取它的类，检查其实际的返回类型。

```
(class (rest [1 2 3]))
-> clojure.lang.PersistentVector$ChunkedSeq
```

类名末尾的那个$ChunkedSeq 是 Java 为了对内联类进行名称改编，而采取的方式。你从某个特定容器类型产生出来的序列，总会被实现为 ChunkedSeq，并内联到原始容器类里（此例中是 PersistentVector）。

Cons 的起源

Clojure 序列是一种以 Lisp 实体列表为基础的抽象概念。在 Lisp 最初的实现中，有三个基本的列表操作分别名为：car、cdr 和 cons。car 和 cdr 是首字母缩写，涉及最初 IBM 704 平台上的 Lisp 实现细节。不过包括 Clojure 在内的许多 Lisp 方言，都把这两个玄奥的名称替换为更有意义的名称：first 和 rest。

这第三个函数 cons，是 construct 的简写。Lisp 程序员把 cons 同时用作名词、动词和形容词。你可以使用 cons 创建一种被称为 cons cell 的数据结构，或者就将其简称为 cons。

包括 Clojure 在内的大多数 Lisp，保留了 cons 这个最初的名称。这是因为“construct”对于表示 cons 的用途而言，是一个相当不错的助记符。这也有助于提醒你，序列是不可变的。方便起见，你也可以说 cons 给序列添加了一个元素，但更准确的说法还是 cons 构建了一个新的序列。这个新序列与原来的那个相似，只不过新增了一个元素。

尽管序列的泛化及其强大，但有时你也会想要直接处理某种特定的实现类型。相关内容参见第 3.5 节“调用特定于结构的函数”。

如果你认为键值对也可算作是序列的元素，那么映射表也可以作为序列。

```
(first {:fname "Aaron" :lname "Bedra"})
-> [:lname "Bedra"]

(rest {:fname "Aaron" :lname "Bedra"})
-> ([:fname "Aaron"])

(cons [:mname "James"] {:fname "Aaron" :lname "Bedra"})
-> ([:mname "James"] [:lname "Bedra"] [:fname "Aaron"])
```

你也可以把集合当作序列。

```
(first #{:the :quick :brown :fox})
-> :brown

(rest #{:the :quick :brown :fox})
-> (:quick :fox :the)

(cons :jumped #{:the :quick :brown :fox})
-> (:jumped :brown :quick :fox :the)
```

为什么执行函数时传入的是向量，却返回了列表？

当你在 REPL 中尝试执行示例时，rest 和 cons 的结果总是被显式为列表，甚至输入的是向量、映射表和集合时也是如此。这是否意味着 Clojure 会在内部把所有东西都转换成列表了呢？答案是，不！无论输入的是什么，序列函数返回的总是序列。你可以通过检查返回对象的 Java 类型来验证这一点。

```
(class '(1 2 3))
-> clojure.lang.PersistentList
(class (rest [1 2 3]))
-> clojure.lang.PersistentVector$ChunkedSeq
```

如你所见，(rest [1 2 3])的结果是某种类型的序列，而非列表。那么，为什么结果看起来会是个列表呢？

答案就在 REPL。当你要求 REPL 显示一个序列时，它仅知道那是一个序列。这个序列究竟是用哪种类型的容器构建的，REPL 一无所知。因此，它干脆就采用相同的方式来打印所有序列：遍历整个序列，并将其作为一个列表打印出来。

映射表和集合的遍历顺序是稳定的，但这个顺序取决于具体的实现细节，所以你不应该依赖它。比如，集合的元素不一定会依照你存放的顺序返回。

```
#{:the :quick :brown :fox}
-> #{:brown :quick :fox :the}
```

你如果想要可靠的顺序，可以用这个。

```
(sorted-set& elements)
```

sorted-set 会依据自然顺序对值进行排序。

```
(sorted-set :the :quick :brown :fox)
-> #{:brown :fox :quick :the}
```

同样，映射表键值对也不一定按照你存放的顺序返回。

```
{:a 1 :b 2 :c 3}
-> {:a 1, :c 3, :b 2}
```

你可以使用 sorted-map 来创建一个有序的映射表。

```
(sorted-map& elements)
```

sorted-map 也不会按照你存放的顺序返回，但它会根据键来进行排序。

```
(sorted-map :c 3 :b 2 :a 1)
-> {:a 1, :b 2, :c 3}
```

除了上述几个序列的核心函数，还有两个函数也值得马上介绍一下，它们是conj 和into。

```
 (conj coll element & elements)
(into to-coll from-coll)
```

conj 会向容器添加一个或是多个元素，into 则会把容器中的所有元素添加至另一个容器。添加数据时，conj 和 into 都会根据底层数据结构的特点选取最高效的插入点。对于列表而言，conj 和 into 会在其前端进行添加。

```
(conj '(1 2 3) :a)
-> (:a 1 2 3)

(into '(1 2 3) '(:a :b :c))
-> (:c :b :a 1 2 3)
```

而对于向量，conj 和 into 则会把元素添加至末尾。

```
(conj [1 2 3] :a)
-> [1 2 3 :a]

(into [1 2 3] [:a :b :c])
-> [1 2 3 :a :b :c]
```

因为 conj（及其相关函数）会针对底层数据结构高效的进行操作，所以你总是能编写既高效又与底层特定实现完全解耦的代码。

Clojure 序列库特别适合于那些庞大的（甚至是无限的）序列。绝大多数 Clojure 序列都是惰性的：只有当确实需要时，它们才真正的把元素生成出来。因此，Clojure 序列函数能够处理那些无法驻留在内存中的超大序列。

Clojure 序列是不可变的：它们永远都不会发生变化。所以我们可以很容易的就做出推断：Clojure 序列在并发访问时是安全的。的确如此。然而，对人类语言来说，这也惹来了一个小麻烦。在描述那些可变的事物时，语言会显得更加顺畅。不妨考虑一下对下面这个假想序列函数 triple 的两种描述。

- triple 会把序列中的每个元素分别乘与三。
- triple 接受一个序列，并返回一个新的序列，这个新序列的每个元素，都是原序列元素的三倍。

后一个版本具体并且准确。前者是更容易阅读一些，但可能会导致这个错误的印象：序列实际上改变了。莫要被愚弄了，序列永不改变。如果你看到这样的说法：“foo 改变了 x”，内心中应该这样来解读：“foo 返回了一个 x 的更改过的拷贝。”

3.2　使用序列库

Clojure 序列库提供了一组丰富的函数，可用于任意序列。如果你的背景是来自被名词统治着的面向对象世界，那么序列库就真的可以被称作是“动词的复仇”[①]了。这些函数提供了丰富的基础功能，任何遵循了基本 first/rest/cons 契约的数据结构都能从中获益。

接下来介绍的函数分为四大类。

- 创建序列的函数。
- 过滤序列的函数。
- 序列谓词。
- 序列转换函数。

这样来分类是有一些武断。由于序列是不可变的，所以大多数序列函数实际上都会创建新的序列。还有一些序列函数，同时兼具过滤和转换的功能。然而，这些分类毕竟提供了一份粗略的路线图，能在探索这个大型的函数库时提供帮助。

3.2.1　创建序列

除了序列字面量以外，Clojure 还提供了若干函数用于创建序列。ranage 会生成一个从 start 开始到 end 结束的序列，每次的增量为 step。

```
(range start? end step?)
```

范围包含了 start，但并不包含 end。如果你没有指定的话，start 默认为 0，step 默认为 1。试试看在 REPL 中创建几个“范围”。

```
(range 10)
-> (0 1 2 3 4 5 6 7 8 9)


(range 10 20)
-> (10 11 12 13 14 15 16 17 18 19)
(range 1 25 2)
-> (1 3 5 7 9 11 13 15 17 19 21 23)
```

repeat 函数会重复 n 次元素 x。

```
(repeat n x)
```

① Steve Yegge 的“名词王国的死刑”（http://tinyurl.com/the-kingdom-of-nouns）认为，面向对象编程将名词推到了不切实际的主导位置，而这正在随着时间发生改变。

用 REPL 中尝试一下 repeat。

```
(repeat 5 1)
-> (1 1 1 1 1)

(repeat 10 "x")
-> ("x" "x" "x" "x" "x" "x" "x" "x" "x" "x")
```

range 和 repeat 都表现出来了这样一种想法：它们可以无限延伸。你不妨把 iterate 就想成是 range 的无限扩展。

```
(iterate f x)
```

iterate 起始于值 x，并持续地对每个值应用函数 f，以计算下一个值，直至永远。

如果你是用 inc 来进行迭代，并且起始于 1，那么你就能生成所有的正整数。

```
(take 10 (iterate inc 1))
-> (1 2 3 4 5 6 7 8 9 10)
```

由于这是个无限序列，你需要另一个新函数，来帮助你在 REPL 中查看这个序列。

```
(take n sequence)
```

take 会返回一个包含了容器中前 n 项元素的惰性序列，这就提供了一种在无限序列上创建有限视图的途径。

能包含全部整数的序列相当有用，让我们将其定义为函数，供将来使用。

```
(defn whole-numbers [] (iterate inc 1))
-> #'user/whole-numbers
```

当只传入一个参数时，repeat 会返回一个惰性的无限序列。

```
(repeat x)
```

在 REPL 中尝试一下这种形式的 repeat 吧，不要忘了用 take 把结果包起来哦。

```
(take 20 (repeat 1))
-> (1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)
```

cycle 函数接受一个容器，并无限的对其进行循环。

```
(cycle coll)
```

在 REPL 中尝试对一些容器进行循环吧。

```
(take 10 (cycle (range 3)))
-> (0 1 2 0 1 2 0 1 2 0)
```

interleave 函数接受多个容器作为参数，并产生一个新的容器，这个新容器会从每个参数容器中交错地提取元素，直至其中某个容器元素被耗尽。

```
(interleave& colls)
```

当其中的某个容器被耗尽时，interleave 就会终止。所以，你可以把有限容器与无限容器混合到一块儿。

```
(interleave (whole-numbers) ["A" "B" "C" "D" "E"])
-> (1 "A" 2 "B" 3 "C" 4 "D" 5 "E")
```

与 interleave 密切相关的 interpose 函数，把输入序列中的每个元素用分隔符隔开，并作为新的序列返回。

```
(interpose separator coll)
```

你可以使用 interpose 来创建，分隔的字符串。

```
(interpose "," ["apples" "bananas" "grapes"])
-> ("apples" "," "bananas" "," "grapes")
```

interpose 和(apply str ...)的结合，正好可以用来产生输出字符串。

```
(apply str (interpose \, ["apples" "bananas" "grapes"]))
-> "apples,bananas,grapes"
```

作为一种惯用法，(apply str ...)是如此的常用，以至于 Clojure 甚至专门把它封装为 clojure.string/join。

```
(join separator sequence)
```

下面借助 clojure.string/join，用逗号来对一个单词列表进行分隔。

```
(use '[clojure.string :only (join)])
(join \, ["apples" "bananas" "grapes"])
-> "apples,bananas,grapes"
```

对应每种 Clojure 中的容器类型，都有一个可以接受任意数量参数的函数，用来创建该类型的容器。

```
(list_& elements)
(vector_& elements)
(hash-set_& elements)
(hash-map key-1 val-1 ...)
```

hash-set 的近亲 set 与其工作方式稍有不同：set 函数期望其第一个参数是个容器。

```
(set [1 2 3])
-> #{1 2 3}
```

而 hash-set 则接受可变的参数列表。

```
(hash-set 1 2 3)
-> #{1 2 3}
```

vector 也有一个近亲 vec，vec 接受容器作为参数，而非可变的参数列表。

```
 (vec (range 3))
-> [0 1 2]
```

现在，你已经掌握了创建序列的基础知识。接下来，你可以使用一些其他的 Clojure 函数，来对序列进行过滤和转换。

3.2.2 过滤序列

Clojure 提供了若干函数用来过滤序列，即由原序列得到一个子序列。这些函数中，最基本的是 filter。

```
(filter pred coll)
```

filter 接受一个谓词和一个容器作为参数，并返回一个序列，这个序列的所有元素都经谓词判定为真。你可以对上一节的 whole-numbers 进行过滤，得到奇数或是偶数。

```
(take 10 (filter even? (whole-numbers)))
-> (2 4 6 8 10 12 14 16 18 20)

(take 10 (filter odd? (whole-numbers)))
-> (1 3 5 7 9 11 13 15 17 19)
```

你还可以使用 take-while 从序列中截取开头的一段，其每个元素都被谓词判定为真。

```
 (take-while pred coll)
```

例如，为了从字符串中逐个获取第一个元音字符之前的所有非元音字符，可以如下这么做。

```
(take-while (complement #{\a\e\i\o\u}) "the-quick-brown-fox")
-> (\t \h)
```

在这儿发生了两件有趣的事情。

- 集合同时也可作为函数。所以你可以把#{\a\e\i\o\u}读作“元音集”，或是“用于检测参数是否为元音的函数。”
- complement 会反转另一个函数的行为。前例的那个反转函数用于检测参数是不是一个元音。

与 take-while 相对的是 drop-while 函数。

```
(drop-while pred coll)
```

drop-while 从序列的起始位置开始，逐个丢弃元素，直至谓词判定为真，然后返回序列剩余的部分。你可以使用 drop-while 来丢弃字符串中起头的那些非元音字符。

```
(drop-while (complement #{\a\e\i\o\u}) "the-quick-brown-fox")
-> (\e \- \q \u \i \c \k \- \b \r \o \w \n \- \f \o \x)
```

split-at 和 split-with 能把一个容器一分为二。

```
(split-at index coll)
(split-with pred coll)
```

split-at 接受一个索引作为参数，而 split-with 则接受一个谓词。

```
(split-at 5 (range 10))
->[(0 1 2 3 4) (5 6 7 8 9)]
(split-with #(<= % 10) (range 0 20 2))
->[(0 2 4 6 8 10) (12 14 16 18)]
```

当然，所有 take-、split-和 drop-打头的函数，返回的都是惰性序列。

3.2.3　序列谓词

过滤函数往往会接受谓词作为其参数，并返回新的序列。序列谓词与其密切相关。序列谓词会要求其他谓词应如何判定序列中的每一个元素。例如，every？要求其他谓词对序列中的每个元素都必须判定为真。

```
(every? pred coll)
(every? odd? [1 3 5])
-> true


(every? odd? [1 3 5 8])
-> false
```

some 设置的门槛相对较低。

```
(some pred coll)
```

只要有一个元素被谓词判定为非假，some 就会返回这个值，如果没有任何元素符合，则 some 返回 nil。

```
(some even? [1 2 3])
-> true
(some even? [1 3 5])
-> nil
```

注意，some 没有以问号结尾。尽管总被当作谓词使用，但 some 并非谓词。因为 some 返回的是第一个符合项的值，而非 true。由于 even?本身就是一个谓词，所以当 some 与 even?配对使用时，其间的差异并不明显。为了体验非 true 匹配，让我们试着用 some 和 identity 一起来找出序列中的第一个非 nil 值吧。

```
(some identity [nil false 1 nil 2])
-> 1
```

其他谓词从名称就能很明显的表现出其行为。

```
(not-every? pred coll)
(not-any? pred coll)
```

不是所有的正整数都是偶数。

```
(not-every? even? (whole-numbers))
-> true
```

但声称所有的正整数都是偶数则显然是一句谎言。

```
(not-any? even? (whole-numbers))
-> false
```

请注意，这里我们选择的问题都是事先就已经知道答案的。一般情况下，在对无限容器应用谓词时你要格外小心。它们可能会无止境的运行下去。

3.2.4 序列转换

转换函数用于对序列中的值进行转换。最简单的转换是映射函数 map。

```
(map f coll)
```

map 接受一个源容器 coll 和一个函数 f 作为参数，并返回一个新的序列。该序列

的所有元素，都是通过对 coll 中的每个元素调用 f 得到的。你可以使用 map 把一个容器的每个元素都用 HTML 标记给包起来。

```
(map #(format "<p>%s</p>" %) ["the" "quick" "brown" "fox"])
-> ("<p>the</p>" "<p>quick</p>" "<p>brown</p>" "<p>fox</p>")
```

还可以传入多个容器给 map。在这种情况下，f 必须是一个多参函数。map 会从每个容器分别取出一个值，作为参数来调用 f，直到数量最少的那个容器被耗尽为止。

```
(map #(format "<%s>%s</%s>" %1 %2 %1)
["h1" "h2" "h3" "h1"] ["the" "quick" "brown" "fox"])
-> ("<h1>the</h1>" "<h2>quick</h2>" "<h3>brown</h3>"
"<h1>fox</h1>")
```

另一个常用的转换是归纳函数 reduce。

```
(reduce f coll)
```

其中 f 是一个接受两个参数的函数。reduce 首先用 coll 的前两个元素作为参数来调用 f，然后用得到的结果和第三个元素作为参数，继续调用 f，以此类推。比如，你可以用 reduce 来让一些数字相加。

```
(reduce + (range 1 11))
-> 55
```

或者让它们相乘。

```
(reduce * (range 1 11))
-> 3628800
```

你可以使用 sort 或 sort-by 对容器进行排序。

```
(sort comp? coll)
(sort-by a-fn comp? coll)
```

sort 会依据元素的自然顺序对容器进行排序，sort-by 则会对每个元素调用 a-fn，再依据得到的结果序列来进行排序。

```
(sort [42 1 7 11])
-> (1 7 11 42)

(sort-by #(.toString %) [42 1 7 11])
-> (1 11 42 7)
```

如果不打算按照自然顺序排序，你可以为 sort 或 sort-by 指定一个可选的比较函数 comp。

```
(sort > [42 1 7 11])
-> (42 11 7 1)

(sort-by :grade > [{:grade 83} {:grade 90} {:grade 77}])
-> ({:grade 90} {:grade 83} {:grade 77})
```

所有过滤和转换的祖先都是列表解析（list comprehension）。列表解析使用集合记号法（set notation），基于一个已存在的列表来创建新的列表。换句话说，解析式描述了结果列表必须满足的性质。一般情况下，一个列表解析会包含以下内容。

- 输入列表。
- 输入列表中元素所对应的占位变量[①]。
- 作用于元素的谓词。
- 一个输出形式，它负责基于那些满足谓词要求的列表元素来产生输出。

当然，Clojure 把列表解析的概念泛化为了序列解析（sequence comprehension）。在 Clojure 中，是使用 for[②]宏来进行解析的。

```
(for [binding-form coll-expr filter-expr? ...] expr)
```

for 接受一个向量作为参数，该参数是由一系列 binding-form/coll-expr 和可选的 filter-expr 组成的，然后是依据表达式 expr 来产生结果序列。

列表解析比诸如 map 和 filter 这样的函数更加通用，而且，事实上它可以模拟之前的大多数过滤和转换函数。

你可以用列表解析来重写之前那个 map 的例子。

```
(for [word ["the" "quick" "brown" "fox"]]
  (format "<p>%s</p>" word))
-> ("<p>the</p>" "<p>quick</p>" "<p>brown</p>" "<p>fox</p>")
```

这样读起来几乎就像是英语一样：“For [each] word in [a sequence of words] format [according to format instructions]”。即按照格式要求，对一系列单词逐个进行格式化。

借助:when 子句，解析也可以用来模拟 filter 函数。你可以把 even?传给:when，用来过滤偶数。

```
(take 10 (for [n (whole-numbers) :when (even? n)] n))
-> (2 4 6 8 10 12 14 16 18 20)
```

① 这是数学意义上的“变量”，而不是命令式编程中的。你不可能改变它们。产生这种混淆，是我们自然语言的问题，对此，我诚挚的表达歉意。
② 列表解析的 for，与命令式编程中的那个 for 循环没有任何关系。

只要:while 字句的表达式保持为真，它就会继续进行求值。

```
(for [n (whole-numbers) :while (even? n)] n)
-> ()
```

当你有多个绑定表达式时，for 才会真正地发挥其威力。例如，你可以通过绑定 rank 和 file，用代数计数法来表示棋盘上的所有位置。

```
(for [file "ABCDEFGH" rank (range 1 9)] (format "%c%d" file rank))
-> ("A1" "A2" ...已省略... "H7 ""H8")
```

序列解析中，Clojure 会从按照右到左的顺序，来遍历绑定表达式。由于在绑定形式中，rank 列于 file 的右手边，所以 rank 会迭代的更快。如果你希望让 file 迭代的更快，可以反转绑定的顺序，把 rank 列在第一位。

```
(for [rank (range 1 9) file "ABCDEFGH"] (format "%c%d" file rank))
-> ("A1" "B1" ...已省略... "G8" "H8")
```

在很多语言中，转换、过滤和解析都是立即执行的。但千万不要认为 Clojure 也是如此。绝大多数的序列函数都不对元素进行遍历，除非你真的尝试要去使用它们。

3.3　惰性和无限序列

大多数 Clojure 序列都是惰性的，换句话说，直到真的需要时，元素才会被计算出来。使用惰性序列有很多好处。

- 你可以推迟那些实际上并不需要的昂贵计算。
- 你可以处理超出内存允许范围的庞大数据集。
- 你可以将 I/O 推迟至确实需要时才进行。

考虑下面的代码及随后的表达式。

src/examples/primes.clj

```
(ns examples.primes)
;; 取自于 clojure.contrib.lazy-seqs
; primes 无法被有效地编写为一个函数，这是因为它需要回顾整个序列。
; 不妨与 fibs 和 powers-of-2 做一下对比，它们仅需要为前 1 或者 2 个值准备固定的缓冲即可。
(def primes
  (concat
   [2 3 5 7]
    (lazy-seq
      (let [primes-from
            (fn primes-from [n [f & r]]
              (if (some #(zero? (rem n %))
                    (take-while #(<= (* % %) n) primes))
```

```
                (recur (+ n f) r)
                (lazy-seq (cons n (primes-from (+ n f) r)))))
            wheel (cycle [2 4 2 4 6 2 6 4 2 4 6 6 2 6 4 2
                          6 4 6 8 4 2 4 2 4 8 6 4 6 2 4 6
                          2 6 6 4 2 4 6 2 6 4 2 4 2 10 2 10])]
        (primes-from 11 wheel)))))
(use 'examples.primes)
```

```
(def ordinals-and-primes (map vector (iterate inc 1) primes))
-> #'user/ordinals-and-primes
```

ordinals-and-primes 包含类似于[5, 11]（11 是第 5 个素数）这样的结对元素。无论是序数还是素数，都是无限多的，但由于 ordinals-and-primes 是惰性的，所以它能装进内存没有任何问题。下面用 take 来提取你需要的那些。

```
(take 5 (drop 1000 ordinals-and-primes))
-> ([1001 7927] [1002 7933] [1003 7937] [1004 7949] [1005 7951])
```

什么时候你会更倾向于惰性序列呢？我认为大多数情况下都会如此。绝大多数序列函数返回的都是惰性序列。这就意味着你可以实际用到多少，就只需付出多少。更重要的是，于你而言，并不需要为惰性序列付出任何特别的努力。正如前面的那个示例，iterate、primes 和 map 都返回了惰性序列，所以 ordinals-and-primes 获得惰性是“免费的”。

惰性序列是 Clojure 中函数式编程的关键。关于创建和使用惰性序列，参见第 4.2 节以获得更多细节。

迫使序列求值

当你在 REPL 中查看很大的序列时，你可以使用 take 来阻止 REPL 对整个序列进行求值。在另外一些情况下，你遇到的问题可能正好相反。你创建的是一个惰性序列，但你希望强行对整个序列进行求值。这个问题通常是当生成序列的代码具有副作用时出现的。考虑下面的这个序列，它通过 println 植入了副作用。

```
(def x (for [i (range 1 3)] (do (println i) i)))
-> #'user/x
```

令 Clojure 新手们吃惊的是，这段代码竟然没有打印出任何东西。由于 x 的定义不会用到序列的任何一个元素，Clojure 自然也就不会为了得到它们而对解析式进行求值。不过，你可以使用 doall 来强制求值。

```
(doall coll)
```

doall 迫使 Clojure 遍历序列中的元素，并把这些元素作为结果返回。

```
(doall x)
| 1
| 2
-> (1 2)
```

你还可以使用 dorun。

```
(dorun coll)
```

dorun 同样会遍历容器中的元素，但它不会把穿过的元素保留在内存中。这样的结果是，dorun 可以遍历那些超过了内存容许范围的超大容器。

```
(def x (for [i (range 1 3)] (do (println i) i)))
-> #'user/x
(dorun x)
| 1
| 2
-> nil
```

返回值为 nil 是一个信号，提醒我们 dorun 并不持有整个序列的引用。dorun 和 doall 函数有助于你处理副作用，然而，Clojure 的其他大部分都不赞成副作用。所以你很少会用到这两个函数。

3.4　Java 亦可序化

对序列的 first/rest 抽象，可以套用到任何“多于一个”的事物之上。在 Java 世界中，这包括了以下内容。

- 容器 API
- 正则表达式
- 文件系统遍历
- XML 处理
- 关系型数据库结果集

Clojure 对这些 Java API 进行了包装，使得对于你要做的几乎所有事情来说，序列库均能适用。

3.4.1　序化 Java 容器

如果试图对 Java 容器应用序列函数，你会发现，这些 Java 容器的表现，和序列是

一模一样的。可作为序列使用的容器被称为可序化的。例如，数组就是可序化的。

```
; String.getBytes 方法返回一个字节数组
(first (.getBytes "hello"))
-> 104
(rest (.getBytes "hello"))
-> (101 108 108 111)
(cons (int \h) (.getBytes "ello"))
-> (104 101 108 108 111)
```

哈希表（Hashtable）和映射表（Map）同样也是可序化的。

```
; System.getProperties 方法返回一个哈希表
(first (System/getProperties))
-> #<Entry java.runtime.name=Java(TM) SE Runtime Environment>
(rest (System/getProperties))
-> (#<Entry sun.boot.library.path=/System/Library/... etc. ...
```

请牢记，即使底层的 Java 容器是可变的，经过包装后的序列却是不可变的。所以，你是不可能通过用 cons 把属性添加到(System/getProperties)中这种方式，来更新系统属性的。cons 会返回一个新的序列，而原来的属性集不会发生任何改变。

由于字符串是由字符组成的序列，所以它也是可序化的。

```
(first "Hello")
-> \H
(rest "Hello")
-> (\e \l \l \o)
(cons \H "ello")
-> (\H \e \l \l \o)
```

Clojure 会把容器自动包装为序列，但却不会自动拆包为它们的原始类型。对于大多数类型的容器而言，这种方式是合情合理的，但对于字符串而言，你恐怕还是希望能把结果转换回字符串。考虑一下字符串反转的功能。Clojure 提供了 reverse 函数。

```
; 结果可能不是你想要的
(reverse "hello")
-> (\o \l \l \e \h)
```

为了能把序列转换回字符串，可以使用(apply str seq)。

```
(apply str (reverse "hello"))
-> "olleh"
```

尽管 Java 容器是可序化的，但大多数情况下，它们并不比 Clojure 的内建容器更具优势。事实上，只有当你需要和遗留的 Java API 打交道时，才应该考虑使用 Java 容器。

3.4.2　序化正则表达式

Clojure 的正则表达式在其内部使用了 java.util.regex 库。从最底层的观点来看，这暴露出了 Java 匹配器（Matcher）的可变本性。你可以对一个正则表达式和一个字符串使用 re-matcher 来创建匹配器，然后在 re-find 上进行 loop，对匹配项进行迭代。

```
(re-matcher regexp string)
src/examples/sequences.clj
```

```
; 不要这样!
(let [m (re-matcher #"\w+" "the quick brown fox")]
  (loop [match (re-find m)]
    (when match
      (println match)
      (recur (re-find m)))))
| the
| quick
| brown
| fox
-> nil
```

更好的办法是，使用较为高级的 re-seq。

```
(re-seq regexp string)
```

re-seq 会把匹配结果暴露为一个不可变的序列。这样，就赋予了你 Clojure 序列函数的全部力量。赶快在 REPL 中试试这些表达式吧。

```
(re-seq #"\w+" "the quick brown fox")
-> ("the" "quick" "brown" "fox")
(sort (re-seq #"\w+" "the quick brown fox"))
-> ("brown" "fox" "quick" "the")
(drop 2 (re-seq #"\w+" "the quick brown fox"))
-> ("brown" "fox")
(map #(.toUpperCase %) (re-seq #"\w+" "the quick brown fox"))
-> ("THE" "QUICK" "BROWN" "FOX")
```

re-seq 是一个体现了如何通过良好抽象来减少代码膨胀的绝佳范例。正则表达式的匹配项并不是某种特殊的东西，无需借助特别的方法来对付他们。他们和其他所有东西一样，就是序列。相比你可能会误入歧途，去突击编写一些特定于正则的函数而言，从现有的序列函数中，你能获得的功能要多得多，感谢 Clojure 为我们提供了如此丰富的序列函数。

3.4.3 序化文件系统

你可以在文件系统上进行序化。作为开始，你可以直接调用 java.io.File。

```
(import '(java.io File))
(.listFiles (File. "."))
-> [Ljava.io.File;@1f70f15e
```

此处的“[Ljava.io.File...”，实际上就是调用 File 数组的 toString()方法得到的结果。尽管序列函数们会对上例自动调用 seq，但 REPL 却不会。

因此，你得要自己来。

```
(seq (.listFiles (File. ".")) )
-> (#<./concurrency> #<./sequences> ...)
```

如果这种对文件列表的默认打印格式不是你想要的，那么你也可以用 getName 把它们映射为字符串。

```
; 有点过头了
(map #(.getName %) (seq (.listFiles (File. "."))))
-> ("concurrency" "sequences" ...)
```

一旦你决定使用诸如 map 这样的函数，再调用 seq 就会显得多余。序列库中的函数会替你调用 seq，所以就不必你亲自来做了。所以，前面的代码可以这样简化。

```
 (map #(.getName %) (.listFiles (File. ".")))
-> ("concurrency" "sequences" ...)
```

通常情况下，你会期望能递归的遍历整个目录树。Clojure 通过 file-seq 提供了一个深度优先的遍历方式。如果是在本书的示例代码目录中执行 file-seq，你会看到很多文件。

```
(count (file-seq (File. ".")))
-> 104 ; 最终的数字可能更大!
```

如果你只想看到最近被修改的文件该怎么做呢？不妨编写一个谓词 recently-modified?，用来检查在最近的这半个小时里，文件是否被动过。

src/examples/sequences.clj

```
(defn minutes-to-millis [mins] (* mins 1000 60))
(defn recently-modified? [file]
  (> (.lastModified file)
     (- (System/currentTimeMillis) (minutes-to-millis 30))))
```

试一下看看。[①]

```
(filter recently-modified? (file-seq (File. ".")))
-> (./sequences ./sequences/sequences.clj)
```

3.4.4　序化流

你可以使用 line-seq 将 Java 的 Reader 以行的方式进行序化。为得到一个 Reader，你可以使用 Clojure 的 clojure.java.io 库。clojure.java.io 库提供了一个 reader 函数，能从流、文件、URL 或 URI 返回一个 Reader。

```
(use '[clojure.java.io :only (reader)])
; 致使 Reader 一直打开...
(take 2 (line-seq (reader "src/examples/utils.clj")))
-> ("(ns examples.utils" " (:import [java.io BufferedReader
InputStreamReader]))")
```

由于 Reader 代表的是非内存资源，需要被明确关闭，所以你应该把 Reader 的创建操作，包在一个 with-open 中。下面的这个表达式，使用了序列函数 count 来统计文件共有多少行，并利用 with-open 确保正确地关闭了 Reader。

```
(with-open [rdr (reader "src/examples/utils.clj")]
  (count (line-seq rdr)))
-> 64
```

为了让这个例子更有用一些，让我们添加一个仅对非空行进行计数的过滤器。

```
(with-open [rdr (reader "src/examples/utils.clj")]
  (count (filter #(re-find #"\S" %) (line-seq rdr))))
-> 55
```

同时把文件系统和单个文件的内容都当作序列，你就能很快地创建一些有趣的工具。让我们来新建一个定义了下面三个谓词的程序。

- non-blank?用于检测非空行。
- non-svn?用于检测非 Subversion 元数据的那些文件。
- clojure-source?用于检测 Clojure 源码文件。

然后，创建用于统计目录树中 Clojure 代码行数的 clojure-loc 函数，一路上，我们还会用到这几个序列函数：reduce、for、count 和 filter。

```
src/examples/sequences.clj
```

① 你得到的结果，可能与此处显示的不同。

```
(use '[clojure.java.io :only (reader)])
(defn non-blank? [line] (if (re-find #"\S" line) true false))
(defn non-svn? [file] (not (.contains (.toString file) ".svn")))
(defn clojure-source? [file] (.endsWith (.toString file) ".clj"))
(defn clojure-loc [base-file]
  (reduce
    +
    (for [file (file-seq base-file)
          :when (and (clojure-source? file) (non-svn? file))]
      (with-open [rdr (reader file)]
        (count (filter non-blank? (line-seq rdr)))))))
```

好了，现在让我们使用 clojure-loc 来看看 Clojure 自己有多少行 Clojure 代码吧。

```
(clojure-loc (java.io.File. "/home/abedra/src/opensource/clojure/clojure"))
-> 38716
```

clojure-loc 函数是面向特定任务的，但由于它仅由序列函数和几个简单谓词构成，所以你可以很轻松的对其进行调整，以适应其他大相径庭的任务。

3.4.5 序化 XML

Clojure 可以对 XML 数据进行序化。这里是在随后的例子中会用到的 XML。

data/sequences/compositions.xml

```
<compositions>
    <composition composer="J. S. Bach">
        <name>The Art of the Fugue</name>
    </composition>
    <composition composer="F. Chopin">
        <name>Fantaisie-Impromptu Op. 66</name>
    </composition>
    <composition composer="W. A. Mozart">
        <name>Requiem</name>
    </composition>
</compositions>
```

clojure.xml/parse 函数能解析 XML 文件、流或 URI，并以 Clojure 映射表的形式，返回数据的树形结构，嵌入在其内部的那些向量代表了子节点。

```
(use '[clojure.xml :only (parse)])
(parse (java.io.File. "data/sequences/compositions.xml"))
-> {:tag :compositions,
:attrs nil,
:content [{:tag :composition, ... etc. ...
```

你可以直接操作这个映射表，或是使用 xml-seq 函数，以序列的方式来查看这棵树。

```
(xml-seq root)
```

下面的示例，在 xml-seq 的基础之上，使用列表解析来提取作曲家信息。

src/examples/sequences.clj

```
(for [x (xml-seq
         (parse (java.io.File. "data/sequences/compositions.xml")))
      :when (= :composition (:tag x))]
  (:composer (:attrs x)))
-> ("J. S. Bach" "F. Chopin" "W. A. Mozart")
```

3.5　调用特定于结构的函数

Clojure 的序列函数允许你编写极为通用的代码。然而，有时候你也会希望能够更具体的针对某种特定的数据结构，以充分利用它们各自的特点。Clojure 包含了一些特定目标的函数，面向列表、向量、映射表、结构和集合。

接下来，我们会对这些特定于结构的函数中的一部分，进行一次快速浏览。Clojure 中特定于结构的完整函数列表，请参见 Clojure 网站的 Data Structures 部分[①]。

3.5.1　列表函数

Clojure 支持 peek 和 pop 这两个名称比较传统的函数，分别用于取出列表的第一个元素，和其余的那些元素。

```
(peek coll)
(pop coll)
```

下面传给 peek 和 pop 一个简单的列表。

```
(peek '(1 2 3))
-> 1
(pop '(1 2 3))
-> (2 3)
```

peek 等同于 first，但 pop 则与 rest 不同。如果是空序列，pop 会抛出一个异常。

```
(rest ())
-> ()
(pop ())
-> java.lang.IllegalStateException: Can't pop empty list
```

① http://clojure.org/data_structures。

3.5.2 向量函数

向量也支持 peek 和 pop，但它们是从向量的末尾开始处理元素的。

```
(peek [1 2 3])
-> 3
(pop [1 2 3])
-> [1 2]
```

get 返回索引位置的值，倘若索引超出了向量边界，则返回 nil。

```
(get [:a :b :c] 1)
-> :b
(get [:a :b :c] 5)
-> nil
```

向量自身也是函数。它接受一个索引作为参数并返回对应的值，或是当索引超过边界时，抛出一个异常。

```
([:a :b :c] 1)
-> :b
([:a :b :c] 5)
-> java.lang.ArrayIndexOutOfBoundsException: 5
```

assoc 会在指定的索引位置，关联一个新的值。

```
(assoc [0 1 2 3 4] 2 :two)
-> [0 1 :two 3 4]
```

subvec 会返回向量的一个子向量。

```
(subvec avec start end?)
```

若未指定 end，则默认为向量的末尾。

```
(subvec [1 2 3 4 5] 3)
-> [4 5]
(subvec [1 2 3 4 5] 1 3)
-> [2 3]
```

当然，你也可以通过组合 drop 和 take 来模拟 subvec。

```
(take 2 (drop 1 [1 2 3 4 5]))
-> (2 3)
```

不同之处在于，take 和 drop 非常通用，可用于任何序列。令一方面，对于向量而

言，subvec 要更快一些。每当你看到 subvec 这样特定于结构的函数，同时其功能又与序列库中的某个函数产生了重复时，那最有可能的理由就是性能。像 subvec 这样的函数，它们的文档中会包含有关于性能特征的描述。

3.5.3　映射表函数

Clojure 提供了几个用于从映射表中读取键和值的函数。keys 将所有的键作为序列返回，vals 则将所有的值作为序列返回。

```
(keys map)
(vals map)
```

尝试一下，从一个简单的映射表中取出键和值。

```
(keys {:sundance "spaniel", :darwin "beagle"})
-> (:sundance :darwin)
(vals {:sundance "spaniel", :darwin "beagle"})
-> ("spaniel" "beagle")
```

get 会返回键对应的值，或者是返回 nil。

```
(get map key value-if-not-found?)
```

用你的 REPL 来试验一下，当键存在和不存在时 get 的行为。

```
(get {:sundance "spaniel", :darwin "beagle"} :darwin)
-> "beagle"
(get {:sundance "spaniel", :darwin "beagle"} :snoopy)
-> nil
```

有一种比用 get 更简单的方法。因为映射表同时也是一个函数，以它自己的键作为参数。所以，你完全可以把 get 一脚踢开，把映射表自身放到函数位置，也就是形式的开头部分。

```
({:sundance "spaniel", :darwin "beagle"} :darwin)
-> "beagle"
({:sundance "spaniel", :darwin "beagle"} :snoopy)
-> nil
```

关键字同样也是函数。他接受一个容器作为参数，并在这个容器中查找其自身。由于:darwin 和:sundance 是关键字，所以也可以把前面那个形式中的元素调换一下顺序。

```
(:darwin {:sundance "spaniel", :darwin "beagle"} )
```

```
-> "beagle"
(:snoopy {:sundance "spaniel", :darwin "beagle"} )
-> nil
```

如果在映射表中查找一个键并返回了 nil，你无法确认究竟是键对应的值为 nil，还是这个键在映射表中根本就不存在。contains?函数就可以解决这个问题，它只是单纯的检测某个键是否存住。

```
(contains? map key)
```

下面创建了一个映射表，nil 是里面的一个合法值。

```
(def score {:stu nil :joey 100})
```

关键字:stu 的确存在，但如果看到结果是一个 nil，你也许就会得出错误的结论。

```
(:stu score)
-> nil
```

但如果使用了 contains?的话，你就能核实:stu 是否参加了这个比赛，即便他可能做的不算太好。

```
(contains? score :stu)
-> true
```

另外一种方法是调用 get，它可以传入可选的第三个参数，如果键未能找到，那么就会返回这个参数的值。

```
(get score :stu :score-not-found)
-> nil
(get score :aaron :score-not-found)
-> :score-not-found
```

这个默认的返回值:score-not-found 让人们能够知道，:aaron 不在映射表中，而:stu 则是存在的，只不过它的值为 nil 罢了。

如果映射表允许 nil 作为合法的值，那么就用 contains?或是三个参数的 get 来检测某个键是否存在吧。

Clojure 还提供了几个会构建新映射表的函数。

- assoc 返回新增了一个键值对的映射表。[①]
- dissoc 返回移除了某些键的映射表。

① 译注：assoc 函数也可以一次新增多个键值对。

- select-keys 返回一个映射表，仅保留了参数传入的那些键。
- merge 可以合并映射表。如果多个映射表包含了同一个键，那么最右边的获胜。

为了测试这些函数，让我们先创建一些歌曲的数据。

src/examples/sequences.clj

```
(def song {:name "Agnus Dei"
           :artist "Krzysztof Penderecki"
           :album "Polish Requiem"
           :genre "Classical"})
```

接下来，创建这个歌曲集的各种修改版本。

```
(assoc song :kind "MPEG Audio File")
-> {:name "Agnus Dei", :album "Polish Requiem",
:kind "MPEG Audio File", :genre "Classical",
:artist "Krzysztof Penderecki"}
(dissoc song :genre)
-> {:name "Agnus Dei", :album "Polish Requiem",
:artist "Krzysztof Penderecki"}
(select-keys song [:name :artist])
-> {:name "Agnus Dei", :artist "Krzysztof Penderecki"}
(merge song {:size 8118166, :time 507245})
-> {:name "Agnus Dei", :album "Polish Requiem",
:genre "Classical", :size 8118166,
:artist "Krzysztof Penderecki", :time 507245}
```

记住，song 本身是永远不会改变的。前面展示的每个函数都返回了一个新的容器。

最有趣的映射表构建函数是 merge-with。

```
(merge-with merge-fn & maps)
```

merge-with 与 merge 很类似，除了当两个或以上的映射表中有相同的键时，你能指定一个你自己的函数，来决定如何合并这个键对应的值。下面使用 merge-with 和 concat 在每个键之下构造一个由值组成的序列。

```
(merge-with
  concat
  {:rubble ["Barney"], :flintstone ["Fred"]}
  {:rubble ["Betty"], :flintstone ["Wilma"]}
  {:rubble ["Bam-Bam"], :flintstone ["Pebbles"]})
-> {:rubble ("Barney" "Betty" "Bam-Bam"),
:flintstone ("Fred" "Wilma" "Pebbles")}
```

始自三个不同的家庭成员容器（它们都以姓作为键），上面的代码把它们合并成了同样以姓作为键的一个容器。

3.5.4 集合函数

除了 clojure 命名空间中可用于集合的函数外，Clojure 还提供了一组位于 clojure.set 命名空间的函数。为了使用这些函数的非限定名，请在 REPL 中调用(use 'clojure.set)。为了运行随后的那些示例，你还需要定义以下变量。

src/examples/sequences.clj

```
(def languages #{"java" "c" "d" "clojure"})
(def beverages #{"java" "chai" "pop"})
```

第一组 clojure.set 函数执行由集合论而来的操作。

- union 返回的集合，包含了所有输入集合中的元素。
- intersection 返回的集合，其所有元素都曾同时出现于多个输入集合中。
- difference 返回的集合，其所有元素都出现于第一个输入集合，但却未出现于第二个中。
- select 返回所有元素都能与给定谓词相匹配的一个集合。

下面编写一个表达式，用来获得所有语言和饮料的并集。

```
(union languages beverages)
-> #{"java" "c" "d" "clojure" "chai" "pop"}
```

接下来，试着找出那些不同时也是一种饮料的语言。

```
(difference languages beverages)
-> #{"c" "d" "clojure"}
```

如果你很喜欢可怕的双关语，那么你一定会很高兴看到这样的事实：有些东西是一种语言的同时，也是一种饮料。

```
(intersection languages beverages)
-> #{"java"}
```

还有数量惊人的语言，它们不能忍受自己的名称长度超过一个字符。

```
(select #(= 1 (.length %)) languages)
-> #{"c" "d"}
```

并集和差集同为集合论的一部分，除此以外它们也是关系代数（relational algebra）的组成部分，而后者则是诸如 SQL 这样的查询语言的根基。关系代数包括六个基本的运算符：并集和差集（前面已经描述过了），加上重命名（rename）、选择（selection）、

投影（projection）和叉积（cross product）。

你可以通过与关系型数据库（见表 3-2）进行类比，来理解这些关系原语。

表 3-2　关系代数与关系型数据库

关系代数	数据库	Clojure 类型系统
关系	表	与集合类似的任何东西
元组（Tuple）	行	与映射表类似的任何东西

下面的示例需要用到一个内存数据库，用作音乐作品辑。在继续之前，让我们先加载这个数据库。

src/examples/sequences.clj

```
(def compositions
  #{{:name "The Art of the Fugue" :composer "J. S. Bach"}
    {:name "Musical Offering" :composer "J. S. Bach"}
    {:name "Requiem" :composer "Giuseppe Verdi"}
    {:name "Requiem" :composer "W. A. Mozart"}})
(def composers
  #{{:composer "J. S. Bach" :country "Germany"}
    {:composer "W. A. Mozart" :country "Austria"}
    {:composer "Giuseppe Verdi" :country "Italy"}})
(def nations
  #{{:nation "Germany" :language "German"}
    {:nation "Austria" :language "German"}
    {:nation "Italy" :language "Italian"}})
```

rename 函数可以用来给键（数据库的列）重新命名，基于一个映射表，把原来的名称改成新的。

```
(rename relation rename-map)
```

下面重命名这个作品辑，使用新的键 title 来替换 name。

```
(rename compositions {:name :title})
-> #{{:title "Requiem", :composer "Giuseppe Verdi"}
{:title "Musical Offering", :composer "J.S. Bach"}
{:title "Requiem", :composer "W. A. Mozart"}
{:title "The Art of the Fugue", :composer "J.S. Bach"}}
```

select 函数返回经谓词计算结果为真的那些映射表，这里的谓词与 SQL 语言中 SELECT 语句的 WHERE 部分非常类似。

```
(select pred relation)
```

下面写一个 select 表达式，用于找出所有标题为“Requiem”（安魂曲）的作品辑。

```
(select #(= (:name %) "Requiem") compositions)
-> #{{:name "Requiem", :composer "W. A. Mozart"}
{:name "Requiem", :composer "Giuseppe Verdi"}}
```

project 函数返回的那些映射表中，仅包含与参数匹配的键。

```
(project relation keys)
```

project 与指定了列的子集的 SQL SELECT 语句非常类似。让我们来写一个仅返回作品辑名称的投影吧。

```
(project compositions [:name])
-> #{{:name "Musical Offering"}
{:name "Requiem"}
{:name "The Art of the Fugue"}}
```

最后一个关系基本运算符是叉积，它是关系型数据库中各种类型的连接（join）的基础。叉积从不同的表中返回所有可能的行组合。在 Clojure 中，使用列表解析就能很轻松地做到。

```
(for [m compositions c composers] (concat m c))
-> ... 4 x 3 = 12 rows ...
```

虽然叉积理论上非常有趣，但通常情况下，你需要的是整个叉积中的某个子集。例如，你可能希望基于共享的键来连接集合。

```
(join relation-1 relation-2 keymap?)
```

你可以基于共享键:composer 来连接作品辑的名称和作曲家。

```
(join compositions composers)
-> #{{:name "Requiem", :country "Austria",
:composer "W. A. Mozart"}
{:name "Musical Offering", :country "Germany",
:composer "J. S. Bach"}
{:name "Requiem", :country "Italy",
:composer "Giuseppe Verdi"}
{:name "The Art of the Fugue", :country "Germany",
:composer "J. S. Bach"}}
```

如果两个关系中的键名不匹配，你可以传入一个 keymap，将 relation-1 中的键名映射到 relation-2 中对应的键。例如，你可以将使用:country 的 composers，与使用:nation 的 nations 相连接。示例如下。

```
(join composers nations {:country :nation})
-> #{{:language "German", :nation "Austria",
:composer "W. A. Mozart", :country "Austria"}
{:language "German", :nation "Germany",
:composer "J. S. Bach", :country "Germany"}
{:language "Italian", :nation "Italy",
:composer "Giuseppe Verdi", :country "Italy"}}
```

你可以把这些关系基本运算符组合到一起。也许你想知道所有创作了安魂曲的作曲家们，家乡都在哪些国家。你可以使用 select 找出所有的安魂曲，用 join 连接它们的作者，然后用 project 来缩小结果集，使其只包含国家名。

```
(project
(join
(select #(= (:name %) "Requiem") compositions)
composers)
[:country])
-> #{{:country "Italy"} {:country "Austria"}}
```

将 Clojure 关系代数与关系型数据库进行类比是很有启发意义的。但要记住，Clojure 的关系代数是一种通用的工具。你可以把它用在任意关系集合（set-relational）数据上。而且，当你使用它时，你还同时还可以借用整个 Clojure 的力量，当然还有任由你掌控的 Java。

3.6　小结

Clojure 把所有类型的容器统一成了一个抽象——序列。在面向对象编程主导了十多年之后，Clojure 的序列库的出现，不啻为一次“动词的复仇”。

Clojure 序列是使用函数式编程技术实现的：不可变数据、递归定义和惰性变现（lazy realization）。在下一章，你会直接看到如何使用这些技术，从而进一步增强 Clojure 的威力。

第 4 章

函数式编程

FP（Functional programming，函数式编程）是一个很大的题目，不可能仅用 21 天[①]，或是凭借书中短短的一章就能学得会。虽然如此，想要很快学会如何有效使用 Clojure 的惰性和递归技术却是有可能的。这样的话，你就能达到一个初级水平，而这也是我们本章将要达成的目标。

下面说明了我们将如何做到这一点。

- 在第 4.1 节中，你会快速概览关于 FP 的术语和概念。本节还提出了贯穿于整章的“Clojure FP 之六大规则”。
- 在第 4.2 节中，你会体验到惰性序列的强大威力。一开始你会使用一种糟糕的方法，来求解斐波那契数列，经过几次改进之后，最终将其提升为一个优雅的惰性方案。
- 和惰性序列本身同样酷的是，你实际上很少需要与它们直接打交道。在第 4.3 节中，你将会看到如何通过重新思考问题，直接运用第 3 章“一切皆序列”中介绍的序列库来解决它们。
- 还有在第 4.4 节中，我们会探索一些高级话题。不少程序员永远都不会用到此处讨论的技术。所以，如果你是一个 FP 的新手，跳过这一节也无妨。

4.1 函数式编程理念

函数式编程使得代码的编写、阅读、测试和重用都更容易了。具体方式如下。

① http://norvig.com/21-days.html。

4.1.1　纯函数

函数式程序构建于纯函数之上。纯函数没有副作用，也就是说，它不依赖于除参数以外的其他任何东西。此外，它影响外部世界的唯一途径，就是它的返回值。

数学函数都是纯函数。无论何时何地，2 加 2 始终都等于 4。当然，除了返回结果以外，求解过程也不会“做”任何其他的事情。

程序的输出显然是不纯粹的。例如，当你使用 println 时，你把数据推送给了输出流，外部世界就因此发生了改变。此外，println 会产生什么样的结果，还取决于函数之外的状态：标准输出流可能被重定向、被关闭或是遭到损坏。

一旦开始编写纯函数，你很快就会意识到，纯函数和不可变数据之间的联系是何等密切。考虑下面这个 mystery 函数。

```
(defn mystery [input]
  (if input data-1 data-2))
```

如果 mystery 是一个纯函数，那么无论它做什么，data-1 和 data-2 必须是不可变的！否则，数据的变化将导致函数对于相同的 input，却有不同的返回值。

只要一小块可变数据就会毁掉这一切，导致整个函数调用链都不再纯粹。所以，一旦许诺编写纯函数，最终，你在程序的大部分地方使用的都是不可变数据。

4.1.2　持久性数据结构

不可变数据是 Clojure 通向 FP 和处理状态的关键所在。站在 FP 的这一面，纯函数不能出现诸如更新可变对象状态这样的副作用。站在状态处理这一面，Clojure 的引用类型也需要借助不可变数据结构，来实现其并发保证。

然而美中不足的是性能。当所有的数据都不可变时，“更新”会转变为“创建一份原始数据的副本，再加上我的更改”。这样做的话，会迅速地把内存耗尽！试想，你有一个需要占用 5MB 内存的地址簿。然后，你进行了五次小小的更新。如果是可变的地址簿，你耗用的内存仍然为 5MB 左右。但如果每次更新时，你都不得不复制整个地址簿的话，那么这个不可变的版本将猛增至 25MB！

Clojure 数据结构并没有采用这种“复制一切”的天真方式。相反，所有的 Clojure

数据结构都是持久的（persistent）。在这里，所谓持久性，意味着新旧版本之间能高效的共享结构，通过这种方式，持久性数据结构能很好地维持其自身的旧副本。

通过列表，可以很容易地视觉化结构共享。考虑下面这个包含两个元素的列表 a。

```
(def a '(1 2))
-> #'user/a
```

然后，你可以创建一个列表 b，它比 a 多了一个元素。

```
(def b (cons 0 a))
-> #'user/b
```

可以看到，b 重用了所有 a 的结构，而不是建立它自己的私有副本，如图 4-1 所示。

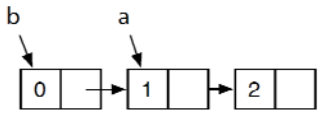


图 4-1　b 重用 a

只要可能，所有的 Clojure 数据结构都回采用共享。相较于简单的列表，其他结构的构造当然也会更加复杂。你如果对这些细节感兴趣，不妨阅读以下文章。

- 《理想的哈希树》[①]，Phil Bagwell 著。
- 《理解 Clojure 持久化向量的实现》[②]，Karl Krukow 著。

4.1.3　惰性和递归

函数式程序大量地使用了递归和惰性。函数对其自身进行调用时，无论是直接还是间接的，就会形成递归。有了惰性，对表达式的求值会被推迟至实际需要时才真正进行。对一个惰性表达式进行求值，也被称作表达式的变现（realizing）。

在 Clojure 中，函数和表达式都不是惰性化的。然而，序列则一般来说都是惰性的。由于进行序列操作的 Clojure 程序是如此之多，你也就可以获得一门全惰性语言的诸多益处。尤其是，你可以使用惰性序列创建复杂的表达式，然后仅当确实需要其中的某些元素时，才会付出相应的代价。

惰性技术还意味着纯函数。因为作为一个纯函数，无论什么时候被调用，返回的总是同样的东西，所以你也不必再为这个问题操心。反过来看，非纯函数就无法充分

① http://lampwww.epfl.ch/papers/idealhashtrees.pdf。
② http://tinyurl.com/clojure-persistent-vector。

发挥惰性技术的作用了。作为一名程序员，你必须控制好调用非纯函数的时机，因为如果调用的时间不同了，则其行为很可能也会不同！

4.1.4　引用透明性

惰性依赖于这样一种能力：在任何时刻，都可以用函数的结果来取代对其的调用。具备这种能力的函数之所以被称为是引用透明（Referential transparency）的，是因为对该函数的调用可被替换，却不会影响程序的行为。除了惰性之外，引用透明的函数还可以在下述两个方面受益。

- 快存（Memoization），自动缓存结果。
- 自动并行化，将函数移至另外一个处理器，甚至另外一台机器进行求值。

纯函数的定义就是引用透明的。然而其他的大部分函数都不是引用透明的，它们只能通过细致的代码审查，方能证明其安全性。

4.1.5　FP 的优势

呼，一下子出现了这么多专业术语，但我们承诺，FP 会让你的代码更容易编写、阅读、测试和组合。现在我们就要来说明怎么做。

你会发现函数式代码编写起来更容易，因为相关的信息都包含在函数参数当中，直接呈现在你面前。你不必担心全局范围、会话范围、应用范围或线程范围什么的。因为同样的原因，函数式代码也更容易阅读的多。

更容易阅读和编写的代码，自然也更容易测试，然而，函数式代码还为测试带来了额外的益处。随着项目变得越来越大，往往也需要付出很多的努力来建立适当的环境，以便运行测试。对于函数式代码而言，这样的问题要少得多，那是因为不存在参数以外的所谓“相关环境”。

函数式代码还提升了重用性。为了能重用代码，你需要做到以下几点。

- 发现和理解一块有用的代码。
- 将可重用代码与其他代码组合到一起。

函数式代码良好的可读性，有助于你发现和理解需要的函数，不过，能把代码组合到一起的这个优点，甚至比可读性还要更引人注目一些。

可组合性是一个很棘手的问题。多年来，程序员们使用封装（encapsulation）来尝试创建可组合的代码。封装竖立了一道防火墙，只允许通过公开的 API 来访问数据。

封装确实有些作用，但也不过是隔靴搔痒罢了。即便有了经过封装的对象，当你试图组合出整个系统时，却有太多潜在的相互作用足以令人大吃一惊。问题就是那些令人生厌的副作用。非纯函数妨碍了封装，因为它们允许来自外部世界的渗入（悄无声息的），并改变了你的代码的行为。另一方面，纯函数才是真正的封装良好，而且可以组合。无论把它们放到系统中的任何位置，它们的表现总是非常一致。

4.1.6 6 条规则

尽管 FP 的益处相当引人注目，但 FP 对于命令式编程风格而言，不啻为一种彻头彻尾的剧变，而后者还仍然统治着当今的编程世界。妙处在于，Clojure 还为 FP 带来了一种独特的方法，在学院派的纯粹性和运行于 Java 虚拟机上的这个现实之间，取得了平衡。这意味着需要一次学习很多东西。但不必恐惧。如果你还只是一个 FP 新手，下面这“Clojure FP 之 6 大规则”会帮助你从入门迈向精通，当然，是以 Clojure 的风格。

1．避免直接递归。Java 虚拟机无法优化递归调用，Clojure 的递归程序会撑爆它们的栈空间。

2．当产生的是标量（scalar values），或者是体积小还数量固定的序列时，你可以使用 recur。Clojure 会对显式的 recur 进行调用优化。

3．当产生个头大，或是大小可变的序列时，让它成为惰性的，而不要用递归。这样，你的调用者就只需要为他们实际需要的那一部分买单。

4．小心不要让一个惰性序列变现的太多，多的超出你的需要。

5．熟悉序列库。这样你就总能写出完全用不着 recur 或者惰性 API 的代码了。

6．细分。把看似简单的问题也尽可能划分为更小的块。这样你就能发现蕴藏于序列库中的解决方案。于是代码更通用，可重用性也会更好。

规则 5 和规则 6 尤为重要。如果你是 FP 新手，你甚至可以将这两条规则翻译为。“忽略本章吧，只要使用第 3 章中介绍的技术就好，直到你真的撞上了墙”。

如同大多数规则，这六条规则仅为指引，而非绝对。当 FP 对你来说开始变得越

来越趁手之后，你就会找到打破这些规则的理由。

现在，让我们开始编写函数式代码吧。

4.2　怎样偷个懒

函数式程序大量的使用了递归定义。递归定义由两个部分组成。

- 基础（basis），明确地列举出序列的部分成员。
- 归纳（induction），提供一些规则，规定了如何通过组合序列的成员，来产生更多的成员。

在本节中，我们面临的挑战是，要把递归定义转换为可以工作的代码。你可能会采用下面这几种方式来做。

- 简单的递归，让函数以某种方式来调用其自身，从而进行归纳。
- 尾递归，仅当函数执行到末尾时，才去调用其自身。尾递归使得一种非常重要的优化手段变得可行了。
- 惰性序列，实实在在地消除了递归，并且只有当需要时，才会去计算值。

选择正确的方法非常重要。递归定义的拙劣实现，会导致代码的性能表现极其糟糕，要么耗尽所有可用的栈空间然后挂掉；要么耗尽所有可用的堆内存然后挂掉；亦或是两者兼具。在 Clojure 中，选择惰性化往往是正确的做法。

我们会应用所有这些方法来求解斐波那契数，通过这种方式逐个对它们加以探讨。以意大利数学家比萨的莱昂纳多（又名斐波那契，生于 1170 年，卒于 1250 年）命名的斐波那契数，实际上早在公元前 200 年就为印度数学家所知晓了。斐波那契数有许多有趣的性质，并且一次又一次的出现在算法、数据结构，甚至于生物学中[①]。斐波那契有一个非常简单的递归定义。

- 基础：F0，第 0 个斐波那契数是 0。F1，第 1 个斐波那契数是 1。
- 归纳：对于 n>1，Fn 等于 $F_{n-1}+F_{n-2}$。

使用这个定义，前 10 个斐波那契数列举如下。

```
(0 1 1 2 3 5 8 13 21 34)
```

一开始，让我们用一个简单的递归来实现斐波那契数。下面的这个 Clojure 函数将会返回第 n 个斐波那契数。

① http://en.wikipedia.org/wiki/Fibonacci_number。

src/examples/functional.clj

```
1; 太糟了
2(defn stack-consuming-fibo [n]
3  (cond
4    (= n 0) 0
5    (= n 1) 1
6    :else (+ (stack-consuming-fibo (- n 1))
7             (stack-consuming-fibo (- n 2)))))
```

第 4 和第 5 行定义了基础，第 6 行则定义了归纳。stack-consuming-fibo 是递归实现的，因为它在第 6 行和第 7 行调用了自身。

测试表明，对于比较小的 n 值，stack-consuming-fibo 工作正常。

```
(stack-consuming-fibo 9)
-> 34
```

目前为止一切良好，但计算诸如 $F_{1000000}$ 这样的大斐波那契数时，就会出现问题。

```
(stack-consuming-fibo 1000000)
-> StackOverflowError clojure.lang.Numbers.minus (Numbers.java:1837)
```

因为是递归，对于 n>1 的情况，每次调用 stack-consuming-fibo 都会招致对其自身的两次调用。在 Java 虚拟机的层面，这些调用会被翻译为方法调用，而每次方法调用都会分配一个被称为栈帧（stack frame）的数据结构。[①]

stack-consuming-fibo 会创建深度与 n 成正比的栈帧，这样就迅速耗尽了 Java 虚拟机的栈空间，并引发了如前所示的 StackOverflowError 异常。而且它创建的栈帧总数是 n 的指数，因此即使栈没有溢出，其性能也极为糟糕。

Clojure 函数调用被标明是消耗栈空间（stack-consuming）的，因为它们使用了栈空间来分配栈帧。在 Clojure 中，你应该尽量避免采用 stack-consuming-fibo 这种消耗栈空间型的递归。

4.2.1 尾递归

函数式程序可以借助尾递归技术解决栈空间的使用问题。尾递归函数依然被定义为是递归的，但递归点必须位于尾部，即函数中返回某个值的那个表达式。语言随后

① 欲了解 Java 虚拟机是如何管理它的栈的，参见位于 http://tinyurl.com/jvm-spec-toc 的“Runtime Data Areas”（运行时数据区）。

即可进行尾部调用优化（tail-call optimization，TCO），把尾递归转换为迭代，于是就不再消耗栈空间了。

对斐波那契的定义 stack-consuming-fibo 并非尾递归，因为在调用了自己两次之后，它还调用了加法（+）。为了让 fibo 成为尾递归，你需要创建一个函数，它的参数携带了足够多的信息，能够让归纳可以继续前进，而不再需要去做一些额外的“善后”工作，这些额外的工作导致递归点远离了尾部位置。为了让 fibo 尾递归，对于 fibo 而言，它需要知道两个斐波那契数，再加上一个序数 n，当 n 递减为零时，新的斐波那契数也就计算出来了。你可以像下面这样编写 tail-fibo。

src/examples/functional.clj

```
1(defn tail-fibo [n]
2  (letfn [(fib
3           [current next n]
4            (if (zero? n)
5              current
6              (fib next (+ current next) (dec n))))]
7    (fib 0N 1N n)))
```

第 2 行引入了 letfn 宏。

```
(letfn fnspecs & body) ; fnspecs ==> [(fname [params*] exprs)+]
```

letfn 与 let 很像，只不过它是专门用来指定局部函数的。每个声明于 letfn 中的函数，都可以调用自身或是其他属于同一个 letfn 块的函数。第 3 行声明的 fib 有三个参数：当前斐波那契数 current、下一个斐波那契数 next 和代表剩余步数的数字 n。

当不再有剩余步数时，会在第 5 行返回 current，反之，则在第 6 行继续进行计算，并且把剩余步数减 1。最后，在第 7 行开启了递归，这里使用了基础值 0 和 1，再加上我们正要求解的斐波那契数的序号 n。

先用小一点的 n 值来试试 tail-fibo。

```
(tail-fibo 9)
-> 34N
```

但即便已经是尾递归了，在面对很大的 n 值时，它仍然执行失败了。

```
(tail-fibo 1000000)
-> StackOverflowError java.lang.Integer.numberOfLeadingZeros
(Integer.java:1054)
```

这里的问题来自于 Java 虚拟机。虽然诸如 Haskell 这样的函数式语言能够进行

TCO，但 Java 虚拟机却不是为函数式语言设计的。没有哪种直接运行在 Java 虚拟机上的语言，能够进行自动的 TCO。[①]

TCO 的缺乏是很不幸的，但对于函数式程序而言，这也算不上是什么致命的问题。Clojure 提供了几个务实的解决方法：使用 recur 进行显式自递归、惰性序列和使用 trampoline 进行显式互递归。

4.2.2 自递归与 recur

递归在 Java 虚拟机上有一种可以被优化的特殊情况，就是自递归。幸运的是，tail-fibo 就是这样一个例子：它直接调用其自身，而不必通过一系列中间函数。

在 Clojure 中，你可以借助 recure 把那种在尾部调用其自身的函数，转换为显式自递归。下面采用了这种方法，将 tail-fibo 转换为 recur-fibo。

src/examples/functional.clj

```
1; 好一些了，但还算不上很棒
2(defn recur-fibo [n]
3  (letfn [(fib
4            [current next n]
5            (if (zero? n)
6              current
7              (recur next (+ current next) (dec n))))]
8    (fib 0N 1N n)))
```

recur-fibo 和 tail-fibo 之间的关键区别是第 7 行，在那儿用 recur 替代了对 fib 的调用。

现在计算斐波那契数时，recur-fibo 就不再消耗栈空间了，此外，如果你有足够耐心等待的话，也可以拿个很大的 n 值来计算 F_n 看看。

```
(recur-fibo 9)
-> 34N
(recur-fibo 1000000)
-> 195 ... 208,982 other digits ... 875N
```

位于 output/f-1000000 中的示例代码，包含了 $F_{1000000}$ 完整的值。

① 在今天的 Java 虚拟机上，编程语言已经可以针对某些类型的递归提供自动尾部调用优化了，但还不是全部。由于尚无通用的解决方案，Clojure 强制你采用显式途径。一旦通用的尾部调用优化在 Java 虚拟机上得到广泛支持，Clojure 也同样会支持它。

recur-fibo 可以计算出一个斐波那契数。但是，如果你想要数字不止一个呢？多次调用 recur-fibo 显然是一种浪费，因为 recur-fibo 并不会为了迎接下一次调用，而进行任何的缓存。但是，如果要缓存的话，又该缓存多少个值？还有，要缓存哪些呢？这样的选择，应该交给函数的调用者去做，而不是本身的实现。

理想情况下，你会通过一个 API 来定义序列。它不会牵涉到任何特定客户关心的具体范围，反之，而是客户随后通过 take 和 drop 把他们想要的范围给拉出来。而这正是惰性序列的看家本领。

4.2.3　惰性序列

惰性序列是用 lazy-seq 宏创建出来的。

```
(lazy-seq& body)
```

仅当需要时，lazy-seq 才会去调用它的 body，也就是说，当 seq 被直接或是间接的调用时，lazy-seq 会为后续的调用缓存结果。你可以像下面这样，用 lazy-seq 来定义惰性的斐波那契数列。

src/examples/functional.clj

```
1(defn lazy-seq-fibo
2  ([]
3    (concat [0 1] (lazy-seq-fibo 0N 1N)))
4  ([a b]
5    (let [n (+ a b)]
6      (lazy-seq
7        (cons n (lazy-seq-fibo b n))))))
```

在第 3 行，无参版本的那个函数主体，把基础值[0 1]和随后将要计算得到的剩余部分连接起来并返回，而这些剩余部分的值，是通过调用那个有两个参数的版本来计算得到的。在第 5 行，两个参数版本的函数体计算出数列的下一个值 n，然后在第 7 行，把 n 和剩余部分的值进行连接。

最关键的是第 6 行，这使得它的主体惰性化了。如果没有这一句，第 7 行对 lazy-seq-fibo 的递归调用会立即生效，这样 lazy-seq-fibo 将会就会像它的前任们一样，一直递归下去直至栈空间被撑爆。这也说明了一个通用的模式：为了用惰性化来替代递归，可以把函数体中的递归部分用 lazy-seq 给包裹起来。

用小一点的值来试试这个 lazy-seq-fibo。

```
(take 10 (lazy-seq-fibo))
-> (0 1 1N 2N 3N 5N 8N 13N 21N 34N)
```

lazy-seq-fibo 对很大的值也仍然有效。下面用到了(rem ... 1000)，是为了避免撑爆屏幕，只打印出第一百万个斐波那契数的最后三位数字。

```
(rem (nth (lazy-seq-fibo) 1000000) 1000)
-> 875N
```

lazy-seq-fibo 采用的方法遵循了规则 3，即使用惰性化来实现无限序列。但通常情况下，并不需要你自己显式地调用 lazy-seq。根据规则 5，你应该重用已有的序列库函数，来返回惰性序列。考虑下面这种 iterate 的用法。

```
(take 5 (iterate (fn [[a b]] [b (+ a b)]) [0 1]))
-> ([0 1] [1 1] [1 2] [2 3] [3 5])
```

这个 iterate 起始于斐波那契数列的第一对数字：[0 1]。通过这种结对的运作方式，就让它携带了刚好足够的信息（两个值），用来计算斐波那契数列的下一个值。

斐波那契数正好是每对的第一个值。可以通过对整个序列应用 map first，来把它们给提取出来，下面的 fibo 定义是由 Christophe Grand 提出的。

src/examples/functional.clj

```
(defn fibo []
  (map first (iterate (fn [[a b]] [b (+ a b)]) [0N 1N])))
```

由于建立在返回惰性序列的 map 和 iterate 之上，fibo 返回的仍然是一个惰性序列。同时 fibo 也很简单，它是我们迄今为止见过的最短实现。但如果你已经习惯于编写命令式的、循环的代码，那么，怎样才能正确地选择这里的 fibo，而不是其他相对指出的方法呢？看起来相当不容易！需要学习递归的、惰性化的思维方式，还有 Java 虚拟机内部在递归上的限制，一下子要学习这么多东西确实挺吓人的。让前面列出的那些规则助你一臂之力吧。由于斐波那契数是无限的，根据规则 3，就可以准确的预测出，在 Clojure 中正确的做法是让其成为一个惰性序列，而根据规则 5，则应该让现有的序列库去做大部分的工作。

惰性化的定义会消耗一些栈空间和堆内存。但它们不会随序列的整体大小（可能是无限的）成比例地消耗资源。相反，当你遍历序列时，要消耗多少资源由你选择。如果你想要第一百万个斐波那契数，从 fibo 直接拿出来就行，无须为前面那一百万个值消耗栈或堆空间。

尽管天下没有免费的午餐。但借助惰性序列，你就相当于拥有了一份没有穷尽的

菜单，并且只有在上菜时，才需要为你点的菜品埋单。在编写 Clojure 程序时，对于任何大小会变化的序列和任何大型的序列，你都应该优先选择惰性序列而不是 loop/recur。

4.2.4　聊聊变现

惰性序列只有当它们被变现时才会显著的消耗资源，换句话说，就是作为序列的一部分在内存中被真正地实例化了。Clojure 尽可能的让序列惰性化，并且极力避免变现序列，直至绝对必要的时候。例如，take 函数根本就不去变现，它返回的仍然是一个惰性序列。你可以通过创建一个声称拥有前十亿个斐波那契数的变量来领会这一点。

```
(def lots-o-fibs (take 1000000000 (fibo)))
-> #'user/lots-o-fibs
```

lots-o-fibs 的创建几乎立刻就完成了，那是因为它几乎没有做任何事情。如果你调用了一个函数，而这个函数确实需要用到 lots-o-fibs 中的一些值，那么 Clojure 就会把这些值给计算出来。即使如此，它也仅会计算那些必须的。例如，下例会从 lots-o-fibs 取得第 100 个斐波那契数，而用不着去计算 lots-o-fibs 承诺提供的其他无数个斐波那契数。

```
(nth lots-o-fibs 100)
-> 354224848179261915075N
```

大多数序列函数返回的都是懒惰序列。如果你不确定某个函数返回的是不是惰性序列，这个函数的文档通常会告诉你答案。

```
(doc take)
-------------------------
clojure.core/take
([n coll])
Returns a lazy seq of the first n items in coll, or all items if
there are fewer than n.
```

然而，REPL 却不是惰性的。默认情况下，REPL 的打印器会把整个容器都给打印出来。这就是为什么我们要把这十亿个斐波那契数塞进 lots-o-fibs，而不是直接在 REPL 中对它们进行求值的原因。千万不要在 REPL 中输入下面的代码。

```
; 不要这么做
(take 1000000000 (fibo))
```

如果你输入了前面的这个表达式，REPL 将试图变现整个容器，并打印这十亿个斐波那契数。有可能你都等不到 Clojure 把所有内存都耗尽，就会因为觉得无聊而退

出 REPL。

为了方便使用惰性序列，你可以通过设置*print-length*的值，来配置打印器一次最多打印多少项。

```
(set! *print-length* 10)
-> 10
```

对于超过 10 个元素的容器，现在打印器就只会打印前 10 项，并在后面追加一个省略号。这样，你就可以放心大胆的打印十亿级别的斐波那契数列了。

```
(take 1000000000 (fibo))
-> (0N 1N 1N 2N 3N 5N 8N 13N 21N 34N ...)
```

甚至是所有的斐波那契数。

```
(fibo)
-> (0N 1N 1N 2N 3N 5N 8N 13N 21N 34N ...)
```

惰性序列真是太美妙了。它们仅做必要的事，而且大多数情况下你不必为此操心。如果什么时候你想强制一个序列完全变现，可以使用 doall 或者 dorun，第 3 章的“迫使序列求值”中有对它们的讨论。

4.2.5 丢弃头元素

还有最后一件事情是你在使用惰性序列时需要考虑的。惰性序列让你可以定义一个很大的（可能是无限的）序列，并且在某一时刻只有序列中很少的一部分会驻留在内存中。然而，如果你（或某些 API）无意中持有了这个序列中某个部分的引用，而你却忘了关照它们，那么这个聪明的策略将会失效。

最可能发生这种事情的情况是，你偶然地持有了一个序列的头（第一项）。在上一节的例子中，每种斐波那契数列的变体，都被定义成了一个返回序列的函数，是函数，而不是序列本身。

但你也可以把序列直接定义为顶级变量。

src/examples/functional.clj

```
; 持有了头部（应该避免!）
(def head-fibo (lazy-cat [0N 1N] (map + head-fibo (rest head-fibo))))
```

这个定义使用了 lazy-cat，它和 concat 非常相像，除了它的参数仅当需要时才会被

求值。这是一个非常漂亮的定义，递归地为（斐波那契数列的每个元素）和（斐波那契数列剩余的每个元素）求和以及映射。

对于较小的斐波那契数，head-fibo 运作良好。

```
(take 10 head-fibo)
-> (0N 1N 1N 2N 3N 5N 8N 13N 21N 34N)
```

但遇上大家伙可就不一样了。

```
(nth head-fibo 1000000)
-> java.lang.OutOfMemoryError: GC overhead limit exceeded
```

这里的问题是，顶级变量 head-fibo 持有着容器的头元素。这样就阻止了垃圾收集器去回收序列中的元素，即便你其实早就已经越过了那些元素。因此，你曾经用到的那部分斐波那契数序列都会被缓存起来，这些被 head-fibo 引用着的值的存活时间，很可能就和整个程序的生命一样长。

除非你就是希望在遍历序列时对它进行缓存，否则，你一定要格外小心，千万不要持有序列头元素的引用。正如 head-fibo 示例所展示的，通常情况下，你应该把惰性序列暴露为返回该序列的一个函数，而不是直接用变量来容纳这个序列。倘若这个函数的某个调用者想要显式地进行缓存，那么这个调用者可以随时创建它自己的变量。在使用惰性序列时，丢弃头元素往往是一个好主意。

4.3　懒上加懒

在语言的层面上，Clojure 惰性序列是一种相当棒的惰性形态。作为一名程序员，你甚至可以找到更懒的解决方案，完全不需要进行显式的序列操作。你往往可以通过组合现有的序列函数来解决问题，而不必在递归或者惰性序列这样的级别上弄脏你的手。

作为一个例子，你将会为了解决下面这个问题，实现数个解决方案。[①]下面给你一个由抛掷硬币的结果组成的序列，正面朝上用:h 表示，背面朝上则是:t。

```
[:h :t :t :h :h :h]
```

序列当中，连续两次都正面朝上的情况，发生了多少回呢？就前面这个例子而言，答案是有两回。第 3 和第 4 次抛掷都是正面朝上，第 4 和第 5 次抛掷也都是正面朝上。

① 向 Jeff Brown 致敬，他在早餐期间提出了这个问题，没有瞎聊，只有纯粹的学术讨论。

这个抛掷硬币的序列可能会非常大，但它毕竟是有限的。由于你寻求的是一个标量答案（计数），依据规则 2，使用 recur 是可以接受的。

src/examples/functional.clj

```
1(defn count-heads-pairs [coll]
2  (loop [cnt 0 coll coll]
3    (if (empty? coll)
4      cnt
5      (recur (if (= :h (first coll) (second coll))
6               (inc cnt)
7               cnt)
8        (rest coll)))))
```

由于这个函数的目的是计算某些东西，loop 引入了一个 cnt 绑定，并在第 2 行将其初始化为零。loop 还引入了对 coll 自己的绑定，以便我们每次都可以通过 recur 减少 coll 的数量。第 3 行提供了这个递归式的基础部分。如果抛掷硬币的序列为空，则必定不会再有连续正面朝上的可能了。

第 5 行是这个函数的精髓所在，如果容器 coll 的第一项和第二项都正面朝上（:h）的话，则 cnt 加 1。

下面尝试几个输入，可以看到 count-heads-pairs 的确与其描述相符。

```
(count-heads-pairs [:h :h :h :t :h])
-> 2
(count-heads-pairs [:h :t :h :t :h])
-> 0
```

尽管 count-heads-pairs 可以工作，但代码却颇为乏味。“连续两次”这个关键概念被样板的 loop/recur 代码给完全掩盖掉了。为了解决这个问题，你需要借助规则 5 和规则 6，对问题进行细分，以利用 Clojure 的序列库。

你将会遇到的第一个问题是，几乎所有的序列函数，都只会依次逐个处理序列中的元素。但这对我们而言完全没有价值，因为我们希望能连同其近邻来考察每一个元素。所以，让我们对序列进行一点转换。当你看到如下的语句。

```
[:h :t :t :h :h :h]
```

你应该在脑海中将其转换为由相邻元素结对组成的序列。

```
[[:h :t] [:t :t] [:t :h] [:h :h] [:h :h]]
```

让我们写一个名为 by-pairs 的函数来进行这种转换吧。因为 by-pairs 的输出会依据

输入的大小而发生变化，所以根据规则 3，你应该把这个序列构建为惰性的。

src/examples/functional.clj

```
1; 过于复杂了，更好的办法在后面……
2(defn by-pairs [coll]
3  (let [take-pair (fn [c]
4       (when (next c) (take 2 c)))]
5    (lazy-seq
6      (when-let [pair (seq (take-pair coll))]
7        (cons pair (by-pairs (rest coll)))))))
```

第 3 行定义了一个函数，用于从容器中获取第一对元素。第 5 行则确保递归式的求值是惰性的。

第 6 行是一个判定语句：如果接下来取出来的那一对，实际上不足两个元素，那么我们一定（几乎）已经处于列表的末尾，于是我们便含蓄地结束了。反之，如果我们确实得到了两个元素，那么，在接下来的第 7 行，我们继续构建这个序列，把这对元素，加入到由容器的剩余部分得到的其它元素对里。

检验一下看看 by-paris 是否有效。

```
(by-pairs [:h :t :t :h :h :h])
-> ((:h :t) (:t :t) (:t :h) (:h :h) (:h :h))
```

现在，你可以把抛掷硬币看作是由一对对结果组成的序列，这样就很容易用文字来描述 count-heads-pairs 了“对全都正面朝上的结果对进行计数”。

这段文字描述可以直接翻译成现有的序列库函数：“计数”就是 count，显然，“都是正面朝上的”则表明了应该使用 filter。

src/examples/functional.clj

```
(defn count-heads-pairs [coll]
  (count (filter (fn [pair] (every? #(= :h %) pair))
                 (by-pairs coll))))
```

相比基于 recur 的实现，这种写法的表达力要强很多，而且，这还使得设计意图非常清晰：我们正在对所有相邻且正面朝上的结果对进行计数。但我们还能让事情变得更加简单。Clojure 已经有比 by-pairs 更通用的版本了，它的名字是 partition。

```
(partition size step? coll)
```

partition 能把容器拆分为一个一个大小都是 size 的块。所以，你可以把一个朝上/

朝下的向量，拆分成结对序列。

```
(partition 2 [:h :t :t :h :h :h])
-> ((:h :t) (:t :h) (:h :h))
```

这样的结果与 by-pairs 并不十分一致，by-pairs 产生的是有重叠的结对。不过，partition 同样也可以做到重叠。可选的 step 参数指明了向下一个块移动之前，partition 应该在容器中向后移动多远。如果没有指定，则 step 就与 size 相同。为了让 partition 能像 by-pairs 那样工作，可以把 size 设置为 2，并把 step 设置为 1。

```
(partition 2 1 [:h :t :t :h :h :h])
-> ((:h :t) (:t :t) (:t :h) (:h :h) (:h :h))

(by-pairs [:h :t :t :h :h :h])
-> ((:h :t) (:t :t) (:t :h) (:h :h) (:h :h))
```

def 的形态

贯穿于本书，你会用到各式各样的 def 来创建变量，例如 defn、defmacro 和 defmulti。它们最终实际上都不过是对 def 这个特殊形式的包装罢了。

defonce 能够保证一个变量一定存在，且会为其设置根绑定值，前提是它还没有被设置过。

```
(defonce a-symbol initial-value?)
```

defn-与 defn 的作用类似，只不过，它产生的是一个私有函数，只能在定义它的命名空间中可见。

```
(defn- name & args-as-for-defn)
```

许多其他的 def 形式，也同样有以破折号结尾的变种，用于私有定义。

还有一处可以改善的地方，为了对两次都正面朝上的结对进行计数，采用了 count/filter 惯用法。这个惯用组合出现的如此频繁，以至于完全值得把它们封装到一个 count-if 函数中。

src/examples/functional.clj

```
(def ^{:doc "Count items matching a filter"}
 count-if (comp count filter))
```

comp 用于组合两个或更多的函数。

```
(comp f & fs)
```

组合而成的，是一个全新的函数，首先它用自己的参数调用最右端的函数，然后将获得的结果作为参数传给次右端的函数，以此类推。因此，count-if 首先会使用 filter

过滤，再把过滤结果交由 count 进行计数。

```
(count-if odd? [1 2 3 4 5])
-> 3
```

最后，你可以使用 count-if 和 partition 创建一个比 count-heads-pairs 更加通用的函数 count-runs。

src/examples/functional.clj

```
(defn count-runs
  "Count runs of length n where pred is true in coll."
 [n pred coll]
  (count-if #(every? pred %) (partition n 1 coll)))
```

count-runs 是个成功的组合：更简单的同时，也比上个版本的 count-heads-pairs 更加通用。你可以用它来为连续两次正面朝上的结果进行计数。

```
(count-runs 2 #(= % :h) [:h :t :t :h :h :h])
-> 2
```

此外，你也可以很容易地用它来为连续两次背面朝上的结果进行计数。

```
(count-runs 2 #(= % :t) [:h :t :t :h :h :h])
-> 1
```

或者，不是两次，而是连续三次抛出的都是正面朝上呢？

```
(count-runs 3 #(= % :h) [:h :t :t :h :h :h])
-> 1
```

如果你仍然希望保留那个名为 count-heads-pairs 的函数，你也可以用 count-runs 来实现它。

src/examples/functional.clj

```
(def ^{:doc "Count runs of length two that are both heads"}
  count-heads-pairs (partial count-runs 2 #(= % :h)))
```

这个新版的 count-heads-pairs 使用 partial 创建了一个新的函数。

```
(partial f & partial-args)
```

partial 会对一个函数进行部分应用（partial application）。当执行 partial 时，你可以指定一个函数 f 和该函数的部分参数。随后当你调用这个由 partial 创建的函数时，可再行指定剩余的那些参数。因此，下面这个表达是有效的。

```
(partial count-runs 1 #(= % :h))
```

相较下面这种说法而言，是另一种更具表现力的表述方式。

```
(fn [coll] (count-runs 1 #(= % :h) coll))
```

部分应用与部分求值（currying）有些类似，却不完全相同。

4.3.1 部分求值与部分应用

当对函数进行部分求值时，你会得到一个新的函数。该函数接受一个参数，并返回原来的那个函数，只不过有一个参数被固化了。部分求值又名柯里化，是以美国逻辑学家 Haskell Curry 命名的，其因在组合逻辑方面的工作而闻名于世。如果 Clojure 里有部分求值函数的话，它很可能会被实现成这样。

```
; 几乎就是部分求值了
(defn faux-curry [& args] (apply partial partial args))
```

部分求值的用途之一就是部分应用。此处是 Clojure 中的部分应用。

```
(def add-3 (partial + 3))
(add-3 7)
-> 10
```

这里则是部分求值，用的是我们的 faux-curry。

```
(def add-3 ((faux-curry +) 3))
(add-3 7)
-> 10
```

如果你需要的是部分应用，那么部分求值只不过是个过渡步骤罢了。事实上，我们的 faux-curry 并不是真正的部分求值。真正的部分求值，返回的是结果，而不是一个没有参数的函数（一旦所有参数都被固化了）。你可以借助 true?函数来辨明其间的差异。此时，true?函数只接受一个参数。

```
; 伪造的部分求值
((faux-curry true?) (= 1 1))
-> #<... mangled function name ...>
; 如果是真正的部分求值
((curry true?) (= 1 1))
-> true
```

由于 Clojure 函数可以接受变长的参数列表，Clojure 无从得知参数何时才算是全部都确定了。但作为程序员，你却很清楚参数是什么时候添加完毕的。一旦你固定了足够多的参数以后，调用这个函数就行。意思就是，在前面的那个表达式外面再增添一对额外的括号。

```
(((faux-curry true?) (= 1 1)))
-> true
```

部分求值在 Clojure 中的缺席，算不上是什么大问题。因为不管怎样，即使没有 curry，也还有 partial 可用，而且一般来讲，后者才是人们所期望的。事实上，许多程序员并未区分术语“部分求值”与“部分应用”。

在本节中，你看到了不少新形式。不要让这些细节掩盖了关键概念：相比直接编写递归或惰性代码，通过组合现有序列库中的函数，你能创建更为简单且更加通用的解决方案。你不必再为惰性化或者递归什么的感到烦心。因此，你应该工作在一个更高的抽象级别之上，让 Clojure 替你去打理惰性化和递归。

4.4　再议递归

Clojure 非常努力的在两者之间寻求着平衡：一个是函数式编程的力量，另一个是 Java 虚拟机的现实。一个例证是，通过 loop/recur 来支持显式的尾部调用优化。这是一个非常积极的选择。

但想要把两个世界中最好的部分融合到一起，总是要冒妥协的风险，而这很可能会让人感到不快，而且，问这样的问题也在情理之中：“Clojure 是否包含了刚开始看上去无关紧要，但却躲在暗处，将来会咬疼我的那种设计妥协？”

没有任何一门语言可以完全清楚地回答这个问题，但我们不妨通过探索一些更为复杂的递归式，来考虑这个问题。首先我们要考虑的是互递归（mutual recursion）。

当两个或更多个函数相互之间来回调用时，就发生了互递归。不再是 A 调用 A 再调用 A，你现在是 A 调用 B 再调用 A 了。作为一个简单的例子，你可以用互递归来定义 my-odd?和 my-even?。

src/examples/functional.clj

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

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

因为 my-odd?和 my-even?彼此调用了对方，你需要在真正定义着两个函数之前，先创建这两个变量。你可以用 def 来做这件事，不过 declare 宏允许你在一行代码里同时创建两个变量（没有初始绑定）。

用小一点的值来验证一下 my-odd?和 my-even?，看看是否有效。

```
(map my-even? (range 10))
-> (true false true false true false true false true false)

(map my-odd? (range 10))
-> (false true false true false true false true false true)
```

my-odd?和 my-even?会根据参数的大小，成正比的消耗栈帧，所以，当遇到很大的数字时，它们就玩完了。

```
(my-even? (* 1000 1000 1000))
-> StackOverflowError clojure.lang.Numbers$LongOps.equiv (Numbers.java:490)
```

这和那个促使我们引入 recur 的问题非常类似。但你无法用 recur 来修复这个问题，因为 recur 只能用于自递归，对互递归则无能力为。当然，odd/even 完全能被更高效的实现，完全用不着使用递归。Clojure 自己就是用 bit-and（按位与）来实现 odd?和 even?的。

```
; 摘自 core.clj
(defn even? [n] (zero? (bit-and n 1)))
(defn odd? [n] (not (even? n)))
```

我们之所以选择 odd/even 来演示，是因为他们比较简单。但别的递归问题可就不会这么简单了，往往找不到既优雅，同时还用不着递归的解决方案。我们将逐个探讨下面四种你可以用来解决那些复杂问题的方法。

- 转换为自递归。
- 采用 Trampolining[①]技术。
- 用惰性化取代递归。
- 用快存为递归抄条近路。

4.4.1 转换为自递归

对彼此独立但却存在某种联系的概念进行建模时，互递归往往是种很好的方式。例如，奇数性和偶数性是独立的概念，但它们显然涉及到了彼此。

你可以对同时涉及这些概念的部分进行独立抽象，把互递归转换为自递归。例如，

① 译注：Trampolining 的中文意思是在蹦床上弹跳，此处表示一种针对互递归的优化技术。

你可以认为，奇数性和偶数性都基于这样一个概念：奇偶性。下面定义了一个采用 recur 的函数 partity，当遇到偶数时返回 0，是奇数则返回 1。

src/examples/functional.clj

```
(defn parity [n]
  (loop [n n par 0]
    (if (= n 0)
      par
      (recur (dec n) (- 1 par)))))
```

下面用小一点的值来测试这个 parity。

```
(map parity (range 10))
-> (0 1 0 1 0 1 0 1 0 1)
```

至此，基于 partity 实现 my-odd?和 my-even?对你而言就是一件非常轻松的事情了。

src/examples/functional.clj

```
(defn my-even? [n] (= 0 (parity n)))
(defn my-odd? [n] (= 1 (parity n)))
```

奇偶性是一个直截了当的概念。不幸的是，许多互递归是无法简化为优雅的自递归的。假如在你尝试把互递归转换为自递归的过程中，发现代码中充斥着模糊了定义的条件表达式，那就不要采用这种方法了。

4.4.2　采用 Trampolining 技术

Trampolining 是一种用于优化互递归的技术。trampoline 就像是一种事后生效的 recur，由函数的调用者来施行，而不是实现者。因为在 trampoline 函数内部，调用者可以调用多个函数，所以 trampoline 函数可以用来优化互递归。

让 Clojure 的 trampoline 函数去调用你的互递归函数中的一员。

```
(trampoline f & partial-args)
```

如果 f 的返回值不是一个函数，那么 trampoline 函数好像只是直接地调用了 f 而已。让我们用几个 Clojure 函数，来体验一番 Trampolining 技术吧。

```
(trampoline list)
-> ()
(trampoline + 1 2)
-> 3
```

如果返回值是一个函数，那么 trampoline 会假定你打算递归的调用这个函数，并替你执行该调用。trampoline 管理着它自己的 recur，所以它会一直调用你的函数，直到返回的不再是函数为止。

回到前面的“尾递归”，你在那儿实现过一个 tail-fibo 函数。你也见到了那个函数是如何消耗栈空间的，并且还试过用 recur 来替代尾递归。现在，你有了另一种选择。你可以把 tail-fibo 的代码拿过来，把其中递归返回的那个分支封装到一个函数中，为应用 trampolining 技术做好准备。

这其实只需要添加一个字符#，用来引入一个匿名函数。

src/examples/trampoline.clj

```
1; 示例而已，不要写这样的代码。
-(defn trampoline-fibo [n]
-  (let [fib (fn fib [f-2 f-1 current]
-            (let [f (+ f-2 f-1)]
5             (if (= n current)
-               f
-               #(fib f-1 f (inc current)))))]
-    (cond
-      (= n 0) 0
10     (= n 1) 1
-      :else (fib 0N 1 2))))
```

与 tail-fibo 的原始版本相比，这里唯一的区别是第 7 行起始的#。试试让 trampoline-fibo 在 trampoline 上蹦蹦看看。

```
(trampoline trampoline-fibo 9)
-> 34N
```

由于 trampoline 替你做了 recur，所以它能妥善处理很大的输入值，不会抛出 StackOverflowError。

```
(rem (trampoline trampoline-fibo 1000000) 1000)
-> 875N
```

为了在 trampoline 和 recur 之间进行一个对比，我们使用 trampoline 移植了 tail-fibo。但因为类似 trampoline-fibo 这样的自递归，trampoline 没有任何优势可言，所以你还是应该优选 recur。但对于互递归而言，trampoline 就可以好好施展一番了。

考虑一下我们在第 4.4 节“再议递归”中展示的 my-odd?和 my-even?这两个互递归定义。你可以采取和转换 tail-fibo 相同的办法，使用 trampoline 对这两个消耗栈的蹩脚实现加以转换：在所有尾部递归调用前，简单地加上一个#符号。

src/examples/trampoline.clj

```
1(declare my-odd? my-even?)
-
-(defn my-odd? [n]
-  (if (= n 0)
5    false
-    #(my-even? (dec n))))
-
-(defn my-even? [n]
-  (if (= n 0)
10    true
-    #(my-odd? (dec n))))
```

与最初实现的唯一差别，就是在第 6 和第 11 行的#包装。通过这个改动，你现在可以 trampoline 很大的值 n，而不必担心会撑爆栈空间了。

```
(trampoline my-even? 1000000)
-> true
```

trampoline 是一种用来解决特定问题的特殊方案。它需要你把原始函数的返回值篡改为不一样的类型，用来指示递归。如果在本书介绍的其他技术中，还有某种技术能对特定递归提供更优雅的实现，那真是太棒了。但如果没有的话，那么你一定会庆幸在你的工具箱中，还有 trampoline 可用。

4.4.3　用惰性化替代递归

在所有本章讨论过的用于消除或者优化递归的技术中，你最常用到的很可能会是惰性化。

作为示例，我们将要实现由 Eugene Wallingford 开发，用于演示互递归的 replace 函数。（参见 http://www.cs.uni.edu/～wallingf/patterns/recursion.html）

replace 使用了一种称为 s 列表（s-list）的数据结构。这种列表可以同时包含符号与符号列表。replace 接受一个 s 列表、一个旧符号（oldsym）和一个新符号（newsym）作为参数，并使用新符号替换所有 s 列表中的旧符号。例如，下面的 replace 调用，会把所有出现的 b 替换为 a。

```
(replace '((a b) (((b g r) (f r)) c (d e)) b) 'b 'a)
-> ((a a) (((a g r) (f r)) c (d e)) a)
```

下面逐字逐句地对其 Scheme 版本进行了翻译，该版本来自 Wallingford 的论文。

我们把 Scheme 函数转换为 Clojure 函数，并重新命名为 replace-symbol，以避免和 Clojure 自己的 replace 函数发生冲突。虽然我们缩短了一些名称，以便更好地适应印刷页面，但保留了原来的代码结构。

src/examples/wallingford.clj

```
; 过于直白的移植，勿用
(declare replace-symbol replace-symbol-expression)
(defn replace-symbol [coll oldsym newsym]
  (if (empty? coll)
    ()
    (cons (replace-symbol-expression
            (first coll) oldsym newsym)
          (replace-symbol
            (rest coll) oldsym newsym))))
(defn replace-symbol-expression [symbol-expr oldsym newsym]
  (if (symbol? symbol-expr)
    (if (= symbol-expr oldsym)
     newsym
     symbol-expr)
    (replace-symbol symbol-expr oldsym newsym)))
```

replace-symbol 和 replace-symbol-expression 是两个互递归函数，所以传进去的结构如果嵌套过深的话，就会撑爆栈空间。为了说明这个问题，让我们写个 deeply-nested 函数，用它来创建一个深度嵌套的列表，列表最深处有个 bottom 元素。

src/examples/replace_symbol.clj

```
(defn deeply-nested [n]
  (loop [n n
         result '(bottom)]
    (if (= n 0)
      result
      (recur (dec n) (list result)))))
```

用小点儿的 n 值来试一下 deeply-nested。

```
(deeply-nested 5)
-> ((((((bottom))))))
(deeply-nested 25)
-> (((((((((((((((((((((((((bottom)))))))))))))))))))))))))
```

Clojure 提供了一个*print-level*，可以用来控制 Clojure 的打印器会深入一个嵌套数据结构到何种程度。把*print-level*设置为一个适度的值，这样打印器就不会疯狂地试图去打印一个深度嵌套的结构了。从下面可以看得出，当嵌套的更深时，打印器只是简单的停了下来，并打印出一个#。

```
(set! *print-level* 25)
-> 25
(deeply-nested 5)
-> ((((((bottom))))))
(deeply-nested 25)
-> (((((((((((((((((((((((((#)))))))))))))))))))))))))
```

现在，试着在不同的嵌套层次上，用 replace-symbol 把 bottom 变成 deepest。你可以看到，如果层次过深的话，就会撑爆栈空间。依赖于你的 Java 虚拟机实现，这可能会需要一个大于 10000 的值，如下所示。

```
(replace-symbol (deeply-nested 5) 'bottom 'deepest)
-> ((((((deepest))))))
(replace-symbol (deeply-nested 10000) 'bottom 'deepest)
-> java.lang.StackOverflowError
```

所有对 replace-symbol 的递归调用都位于 cons 之中。为了打破递归，你只需要把递归用 lazy-seq 给包装起来就行了。真的就是那么简单。下面是经过了改进的版本。由于过渡到惰性是如此之简单，以至于我们无法抵御这样的诱惑：用另一种 Clojure 味十足的方式，让这个函数的 Clojure 味也更浓一些。

src/examples/replace_symbol.clj

```
1(defn- coll-or-scalar [x & _] (if (coll? x) :collection :scalar ))
2(defmulti replace-symbol coll-or-scalar)
3(defmethod replace-symbol :collection [coll oldsym newsym]
4  (lazy-seq
5    (when (seq coll)
6      (cons (replace-symbol (first coll) oldsym newsym)
7        (replace-symbol (rest coll) oldsym newsym)))))
8(defmethod replace-symbol :scalar [obj oldsym newsym]
9  (if (= obj oldsym) newsym obj))
```

在第 4 行，lazy-seq 打破了递归，防止了遇到深度嵌套的结构时的栈溢出。另一处改进位于第 2 行。这里不再使用两个不同的函数来分别处理符号和列表，而是采用了多重方法 replace-symbol，其中的一个方法面向列表，另一个则面向符号（第 8 章“多重方法”有更多详细介绍）。这么做，不但消除了一个 if，还提高了可读性。

来确认一下这个改进版的 replace-symbol 能否处理深层嵌套。

```
(replace-symbol (deeply-nested 10000) 'bottom 'deepest)
-> (((((((((((((((((((((((((#)))))))))))))))))))))))))
```

惰性化是一个强大的盟友。你总能编写那些递归的，甚至是互递归的函数，然后

再用惰性化来打破递归。

4.4.4 用快存为递归抄条近路

为了演示更复杂的互递归，我们不妨考虑一下霍夫斯塔特的女性和男性序列。对霍夫斯塔特序列的第一次描述出现在《哥德尔、埃舍尔、巴赫：集异璧之大成》[Hof99]中。其女性与男性序列定义如下。[①]

```
F(0) = 1; M(0) = 0
F(n) = n - M(F(n-1)), n > 0
M(n) = n - F(M(n-1)), n > 0
```

这里提出了一个很直接的 Clojure 定义。

src/examples/male_female.clj

```
; 勿直接使用
(declare m f)
(defn m [n]
  (if (zero? n)
   0
    (- n (f (m (dec n))))))
(defn f [n]
  (if (zero? n)
   1
    (- n (m (f (dec n))))))
```

上面这个 Clojure 的定义易于阅读，且与其数学定义极为接近。然而，对于很大的 n 值而言这将是一场灾难。序列中的每个值都需要从头计算另外的两个值，而这又需要依次从头计算另外的两个值。在我们的一台 MacBook Pro 电脑[②]上，计算(m 250)会耗去超过半分钟的时间。

```
(time (m 250))
"Elapsed time: 38443.902083 msecs"
-> 155
```

那么想要保持干净的互递归定义，还想要具有不俗的性能，可能吗？答案是肯定的，只需要快存的一点点帮助就行。所谓快存，就是把过去的计算结果给缓存起来，用空间来换取时间。当你调用一个被快存了的函数，它首先在一个记录了输入历史和输出历史的映射表中检查你这次的输入。如果在映射表中发现了匹配的输入，就立刻返回其对应的输出，而不会傻呼呼的再计算一次。

① http://en.wikipedia.org/wiki/Hofstadter_sequence
② 3.06 GHz Intel Core 2 Duo, 4 GB 667 MHz DDR2 SDRAM, Ubuntu 10.10, SSD

下面使用 Clojure 的 memoize 函数，重新把 m 和 f 绑定到它们各自的快存版本上。

src/examples/memoized_male_female.clj

```
(def m (memoize m))
(def f (memoize f))
```

现在对于每一个 n 值，Clojure 仅需计算一次 F 和 M 即可。速度提升及其可观。同样是计算(m 250)，比之前快了成千上万倍。

```
(time (m 250))
"Elapsed time: 5.190739 msecs"
-> 155
```

而且，理所当然的，缓存一旦建立起来，被缓存的值几乎在一瞬间就被“计算”出来了。

```
(time (m 250))
"Elapsed time: 0.065877 msecs"
-> 155
```

然而，单独用快存还是不够的。只有在缓存得到填充的情况下，快存才会为递归提供捷径。如果你是从一个空缓存开始，并且让 m 或 f 去处理很大的数，那么在缓存被建立起来之前，栈空间就会被撑爆。

```
(m 10000)
-> java.lang.StackOverflowError
```

还有最后的一个绝招，为了保证缓存能万丈高楼平地起，我们把暴露方式由函数变成序列。下面用 m 和 f 来映射整个数字空间，通过这种方式创建出序列 m-seq 和 f-seq。

src/examples/male_female_seq.clj

```
(def m-seq (map m (iterate inc 0)))
(def f-seq (map f (iterate inc 0)))
```

现在调用者就可以通过从序列中提取第 n 个值（nth），来得到 M(n)或者 F(n)了。

```
(nth m-seq 250)
-> 155
```

这种做法的速度相当快，甚至对于很大的 n 值亦是如此。

```
(time (nth m-seq 10000))
"Elapsed time: 0.735 msecs"
-> 6180
```

我们把此处用到的方法总结一下。

- 以自然的方式定义一个互递归函数。
- 为了给递归找条捷径，对那些已经被计算过的值使用快存。
- 暴露为一个序列，这样依赖的那些值在使用之前就已经被缓存起来了。

这种方法会消耗堆空间，因为它需要缓存所有之前的值。如果这是一个问题，那么在某些情况下，你可以选择更加复杂的缓存策略来消除它。

4.5 小结

在本章中，你看到了 Clojure 对在支持函数式方面，是如何采取积极的态度，在学院派纯粹性和 Java 虚拟机上的有效性之间做出平衡的。Clojure 提供了包括使用 recur 的自递归、使用 trampoline 的互递归、惰性序列和快存在内的各种技术。更妙的是，面对各种类型的日常编程任务，你可以使用序列库，而不必为惰性序列定义显式循环。诸如 partition 这样的函数提供了清晰、富有表达力的解决方案，同时相比而言，这样的方案编写起来也更容易。

倘若 Clojure 是一门单纯的函数式语言，那我们也许会把注意力转向下一个挑战——身处一个没有可变状态的世界，并且很可能会开始探讨状态单子（state monad）。但是，Clojure 用其最具开创性的特性，引导我们走向了另外一个不同的方向：用显式的 API 来管理可变状态。Clojure 的引用模型提供了四种不同的语义，来对状态建模，而这就是我们即将在第 5 章“状态”中要讨论的内容。

第 5 章

状态

所谓状态，就是在某个时间点上一个标识所代表的值。

前面这个句子包含了太多的内容。首先让我们取出第一个词“值”。值是一种不可变的持久数据结构。当你能够彻底的用值来进行编程时，正如我们在第 4 章“函数式编程”中看到的那样，生活将会轻松很多。

时间的流逝会令事情变得愈发麻烦。今天的纽约洋基队和去年一样吗？1927 年呢？洋基队的名册是一个标识，它的值随着时间而发生改变。

更新标识并不会破坏旧的值。事实上，更新一个标识，无论如何也不会对已有的值造成影响。洋基队可以交易每一个球员，或是因为无聊而突然解散了一段时间，但这都无法改变这只伟大球队曾经留给我们的点点滴滴。

Clojure 的引用模型把标识和值清晰地区分开来。在 Clojure 中，几乎所有的东西都是值。为了加以标识，Clojure 提供了四种引用类型。

- 引用（Ref），负责协同地、同步地更改共享状态。
- 原子（Atom），负责非协同地、同步地更改共享状态。
- 代理（Agent），负责异步地更改共享状态。
- 变量（Var），负责线程内的状态。

这些 API 都会在本章中得以讨论。在本章结束的时候，我们还会开发一个示例应用。一个贪吃蛇游戏，用来演示如何把应用模型划分为不可变和可变的组件。

在我们开始之前，让我们回顾一下并发（concurrency）和并行（parallelism），此

外我们还会对传统锁定机制的困境进行一些评论。

5.1 并发、并行和锁定

所谓并发程序，是模拟了有许多事情同时发生。并行程序则是执行某种运算。这种运算可能是有顺序的，并且可以拆成单独的小块，而这些小块能同时执行，以加速整个执行过程。

有很多理由促使我们去编写并发或是并行程序。

- 几十年来，性能的提高来源于不断增强核心的计算能力。但是现在，以及不久的将来，性能的提高将主要来自于更多的核心数量。相比以往，现在硬件自身就能支持更多的并发，因此，系统也必须是并发的，才能充分发挥其能力。
- 为了能够及时完成那些代价高昂的计算任务，可能需要把它们放到多个核心上并行执行（或是多个处理器）。
- 那些因为等待资源而被阻塞的任务应该退开，让其他任务来使用空闲的处理器。
- 当执行长时间运行的任务时，用户界面也需要保持响应。
- 逻辑上独立的运算更容易实现，当然，这需要平台能够识别并充分利用它们的独立性。

有件事情在并发中是显而易见的：会有不止一个观察者（例如线程）在窥视着你的数据。对于那种把值和标识交织[①]在一起的语言来说，这是个非常大的问题。这样的语言对待一块数据，就像对待只有一行的银行总账一样。每次新的操作都会抹去历史记录，潜在的破坏了系统中每个其他线程的工作。

虽然是并发使得这个挑战被凸显了出来，但认为只有在并发时才会有多个观察者参与进来，是完全错误的。如果你的程序中有两个变量都引用了相同的数据，那么这些变量就是不同的观察者。如果你的程序允许可变性，那么你就必须认真考虑和状态相关的问题。

可变的语言趋向于通过锁定和防御性复制（defensive copying）来应对这一挑战。继续那个总账的类比：银行雇佣了警卫（锁），在有人使用总账时，都对其活动进行监督，不允许任何人在别人使用总账的期间进行修改。

当性能变得相当糟糕时，银行甚至会要求总账的读者制作一份他们自己的总账副本，这样他们就会让开道，令交易可以继续。这些副本必须仍由警卫来负责监督！

① http://www.infoq.com/presentations/Simple-Made-Easy

这种模型听起来就令人生厌，甚至在实现细节的层面上，事情还会变得更糟。选择锁定什么和在哪儿锁，实在是一项艰难的任务。如果你搞错了，各种糟糕的事情都有可能发生：线程间的竞争条件（Race conditions）可能会破坏数据，死锁（Deadlocks）可能会使整个程序完全的停止运作。《Java 并发编程实践》讨论了上述问题，还包括这些问题的详细解决方案。这是一本了不起的书，但却很难读懂，而且它也没有启发你自问一句：“难道就没有别的办法了吗？”

Clojure 的状态和标识模型解决了这些问题。大部分程序代码是函数式的。代码库中只有很小的一部分，才会真正明显地受益于可变性，也才有必要去选择那四种引用模型之一。使用这些模型，你可以把你的模型分为两层。

- 函数式模型，没有可变状态。你的大多数代码通常位于这一层，它们易于阅读、易于测试并易于并行化。
- 引用模型，你发现程序中如果使用可变状态（尽管有其缺点）处理起来会更加方便的那些部分。

让我们开始 Clojure 的状态之旅吧，首先是最为著名的 Clojure 引用模型：软事务内存。

5.2　引用与软事务内存

Clojure 中的大多数对象都是不可变的。当你真的想要可变数据时，你必须明确地表示出来。比如说，你可以像下面这样创建一个可变的引用（ref），让它指向不可变对象。

```
(ref initial-state)
```

举个例子，可以为音乐播放列表中的当前歌曲创建一个引用。

```
(def current-track (ref "Mars, the Bringer of War"))
-> #'user/current-track
```

引用包装并保护了对其内部状态的访问。要读取引用的内容，你可以调用 deref。

```
(deref reference)
```

deref 函数可以缩写为读取器宏@。下面试着用 deref 和@对 current-track 解引用。

```
(deref current-track)
-> "Mars, the Bringer of War"
@current-track
-> "Mars, the Bringer of War"
```

注意在这个例子中，Clojure 采用的模型与真实世界是多么的一致。一首歌曲是一个不可变的实体。当你听完这首歌曲时，它并不会变成另外的一首歌。但是，当前歌曲是一个实体的引用，它会发生变化。

5.2.1 ref-set

你可以使用 ref-set 来改变一个引用所指向的位置。

```
(ref-set reference new-value)
```

下面调用 ref-set 来收听另外一首歌曲。

```
(ref-set current-track "Venus, the Bringer of Peace")
-> java.lang.IllegalStateException: No transaction running
```

哎呀，出错了。因为引用是可变的，你必须在更新它们时施以保护。在许多语言中，你可能会使用锁来达到目的。而在 Clojure 中，你可以使用事务。事务被包裹在 dosync 之中。

```
(dosync& exprs)
```

把你的 ref-set 用 dosync 包起来，就一切正常了。

```
(dosync (ref-set current-track "Venus, the Bringer of Peace"))
-> "Venus, the Bringer of Peace"
```

这个 current-track 引用现在指向了另外一首歌曲。

5.2.2 事务的属性

和数据库事务一样，STM 事务也具有一些重要的性质。

- 更新是原子的（atomic）。如果你在一个事务中更新了多个引用，所有这些更新的累积效果，在事务外部看来，就好像是在一个瞬间发生的。
- 更新是一致的（consistent）。可以为引用指定验证函数。如果这些函数中的任何一个失败了，整个事务都将失败。
- 更新是隔离的（isolated）。运行中的事务，无法看到来自于其他事务的局部完成结果。

数据库提供了一个额外的保证：更新是持久的（durable）。因为 Clojure 的是内存中事务，所以无法保证更新是可持久的。在 Clojure 中如果你想要可持久的事务，那么

你应该使用一个数据库。

总之，以上四种事务的性质被称为 ACID。数据库提供了 ACID，而 Clojure 的 STM 则提供了 ACI。

如果你在一个事务中改变了多个引用，在事务外部的任何代码看来，这些变化都被协同（coordinated）为“在同一时间发生”。所以你可以确信，对 current-track 和 current-composer 的更新是被协同的。

```
(def current-track (ref "Venus, the Bringer of Peace"))
-> #'user/current-track
(def current-composer (ref "Holst"))
-> #'user/current-composer
(dosync
  (ref-set current-track "Credo")
  (ref-set current-composer "Byrd"))
-> "Byrd"
```

由于这些更新都处于事务中，其他任何线程都不会看到那种更新到一半，歌曲和作曲家对应不上的情况。

5.2.3　alter

那个 current-track 示例是故意弄得很容易的，因为在更新该引用时，与其之前的状态没有任何联系。接下来让我们构建一个更复杂的示例，需要在事务中更新现有的信息。一个简单的聊天应用刚好符合这个要求。首先，创建一个记录类型用来表示消息，它有一个发送者和一些文本。

src/examples/chat.clj

```
(defrecord Message [sender text])
```

现在，你可以通过实例化这个记录来创建消息。

```
(user.Message. "Aaron" "Hello")
-> #:user.Message{:sender "Aaron", :text "Hello"}
```

这个聊天应用程序的用户希望首先看到的是最新消息，所以列表是非常合适的数据结构。下面创建一个引用 messages，指向一个初始为空的列表。

```
(def messages (ref ()))
```

现在，你需要一个函数，用来把新消息添加到 messages 的前端。你可以简单的用

deref 获取消息列表，用 cons 把这条新消息加进去，然后把更新过的列表用 ref-set 放回 messages。

```
; 糟糕的主意
(defn naive-add-message [msg]
  (dosync (ref-set messages (cons msg @messages))))
```

但还有一个更好的选择。为什么不在同一个步骤中就完成读取和更新呢？Clojure 的 alter 能在事务中对引用对象应用一个更新函数。

```
(alter ref update-fn & args...)
```

alter 会返回这个引用在事务中的新值。当事务成功完成后，引用将获得它在事务中的最后一个值。用 alter 来替代 ref-set 能使代码更具可读性。

```
(defn add-message [msg]
  (dosync (alter messages conj msg)))
```

注意这里的更新函数用了 conj（“conjoin”的缩写），而不是 cons。这是因为 conj 的参数顺序更适用于 alter。

```
(cons item sequence)
(conj sequence item)
```

alter 函数调用它的 update-fn 时，把当前引用的值作为其第一个参数，这正是 conj 所期望的。如果你打算编写自己的更新函数，那么它们也应该遵循与 conj 相同的结构。

```
(your-func thing-that-gets-updated & optional-other-args)
```

试着添加几条消息，观察代码是否按照预期那样工作了。

```
(add-message (user.Message. "user 1" "hello"))
-> (#:user.Message{:sender "user 1", :text "hello"})

(add-message (user.Message. "user 2" "howdy"))
-> (#:user.Message{:sender "user 2", :text "howdy"}
    #:user.Message{:sender "user 1", :text "hello"})
```

alter 是 Clojure STM 的骨干，也是用于更新引用的主要手段。不过如果你了解一些关于 STM 是如何工作的，在某些场景下，你也许就能优化你的事务了。

5.2.4 STM 的工作原理：MVCC

Clojure 的 STM 采用了一种名为多版本并发控制（Multiversion Concurrency

Control，MVCC）的技术，这种技术也被用在了几个主要的数据库中。下面说明了在 Clojure 中，MVCC 是如何运作的。

事务 A 启动时会获取一个“起始点”，这个起始点就是一个简单的数字，被当作 STM 世界中的唯一时间戳。在事务 A 中访问任何一个引用，实际上访问的是这个引用与起始点相关的一份高效副本。Clojure 的持久性数据结构（第 4.1.2 节“持久性数据结构”）使得提供这些高效的私有副本相当廉价。

在事务 A 中，对引用进行操作时依赖（以及返回）的这个私有副本的值，被称为事务内的值。

在任意时间点，如果 STM 检测到其他事务设置或更改了某个引用，而事务 A 正好也想要设置或更改，那么事务 A 将被迫重来。如果你在 dosync 块中抛出了一个异常，那么事务 A 会终止，而非重试。

事务 A 一旦提交，它一直以来那些私有的写操作就会暴露给外部世界，而且是在这个事务时间轴的一个点上瞬间发生的。

有时候 alter 背后的措施有点过于谨慎了。假如你并不在意事务进行到一半时，有另一个事务也更改了其中的一个引用，又会怎样呢？倘若即便是在这样的情况下，你无论如何也想要提交你的更改，那么借助 commute 就能获得超过 alter 的性能表现。

5.2.5　commute

commute 是一种特殊的 alter 变体，允许更多并发。

```
(commute ref update-fn & args...)
```

当然，这需要进行权衡。之所以名为 commute，是因为它们必须是可交换的（commutative）。也就是说，更新操作必须能以任何的次序出现。这就赋予了 STM 系统对 commute 重新排序的自由。

为了使用 commute，只要在你的 add-message 实现中，用 commute 简单地替换掉 alter 即可。

```
(defn add-message-commute [msg]
  (dosync (commute messages conj msg)))
```

commute 会返回引用的新值。然而，因为重排了顺序，你从 commute 那儿看到该

引用的最后一个事务内值，却未必总与其事务终结时的值相符。如果当你尝试进行 commute 时，另一个事务也偷偷地更改了引用，STM 将不会重启你的整个事务。相反，它只会简单的再次运行你的 commute 函数，当然那是乱序的。而你的事务也永远不会见到 commute 函数最终运行时，那个引用的值究竟是什么。

因为 Clojure 的 STM 会在你的背后悄悄地重新排定 commute 的顺序，所以你只有在不关心顺序的情况下，才可以使用它们。准确地来讲，聊天程序其实并不符合这个条件。消息列表肯定是有顺序的，所以如果两条消息的添加顺序颠倒了，结果列表将无法正确地反映消息到达的顺序。

但从实践角度而言，聊天信息的更新是可交换的，没有太大问题。基于 STM 的消息重排，很可能发生在微秒甚至更短的时间尺度上。对于聊天应用的用户而言，那些由于网络和人为延迟造成的消息重排，时间尺度相比而言要大得多。想想看，在一个在线聊天室里，你有多少次“多嘴”，只是因为别人的消息还没有到达你这儿罢了。由于这些更严重的顺序问题是不确定的，所以对于聊天应用而言，完全有理由忽略 STM 的那些点乱序问题，这实在是无足轻重。

5.2.6 alter 优先

许多更新都是不可交换的。例如，考虑一个计数器，会返回一系列递增的数字。你可以使用这个计数器来构建系统中的唯一标识。这个计数器可以简单地引用一个数字。

src/examples/concurrency.clj

```
(def counter (ref 0))
```

你不应该使用 commute 来更新计数器。commute 返回的是它被调用的那一刻，计数器的事务内值，但重新排序可能会导致事务结束时，其值与之前并不相同。这会导致多个调用者得到相同的计数值。作为替代，你应该使用 alter。

```
(defn next-counter [] (dosync (alter counter inc)))
```

试着多调用 next-counter 几次，以验证计数器能按照预期那样工作。

```
(next-counter)
-> 1
(next-counter)
-> 2
```

一般情况下，你应该首选 alter 而不是 commute。alter 的语义更容易理解，也更容易避免出错。另一方面，commute 还要求你小心仔细的考虑事务语义。如果你在能用 commute 的地方使用了 alter，最遭的事情也不过就是可能导致性能下降。但如果你在应该用 alter 的地方误用了 commute，就会引入相当微妙的故障，很难通过自动化测试进行检测。

5.2.7　为引用添加验证

数据库事务通过各种完整性检查来保持一致性。你也可以通过在创建引用时指定一个验证函数，让 Clojure 的事务内存来做类似的事情。

```
(ref initial-state options*)
; options 包括:
; :validator 一个验证函数
; :meta 一个元数据映射表
```

ref 的选项中，包含一个可选的验证函数，它可以通过抛出异常来阻止事务完成。需要注意的是，options 不是一个映射表，它只是一连串拼接进函数调用中的键值对罢了。

下面继续那个聊天室的例子，给 messages 添加一个验证函数，以保证所有消息的:sender 和:text 都是非空值。

src/examples/chat.clj

```
(def validate-message-list
  (partial every? #(and (:sender %) (:text %))))

(def messages (ref () :validator validate-message-list))
```

这个验证，就像是数据库事务中表的主键约束一样。如果违反了约束，整个事务都会回滚。下面用不符合约束条件的消息来试一下看看，比如说就传给它一个简单的字符串。

```
(add-message "not a valid message")
-> java.lang.IllegalStateException: Invalid reference state

@messages
-> ()
```

符合约束条件的消息则没有问题。

```
(add-message (user.Message. "Aaron" "Real Message"))
-> (#:user.Message{:sender "Aaron", :text "Real Message"})
```

对于需要协同访问的共享状态而言，引用是非常适合的，但并不是所有的任务都需要进行协同。对于更新一块独立的数据而言，原子是更好的选择。

5.3 使用原子进行非协同、同步的更新

相比引用，原子是一种更加轻量级的机制。在事务中对多个引用进行更新会被协同，而原子则允许更新单个的值，不与其他的任何事物协同。

你可以使用 atom 来创建原子，它的函数签名与 ref 非常类似。

```
(atom initial-state options?)
; options 包括:
; :validator 一个验证函数
; :meta 一个元数据映射表
```

回到我们那个音乐播放器的例子，你可以使用原子替代引用来保存 current-track。

```
(def current-track (atom "Venus, the Bringer of Peace"))
-> #'user/current-track
```

对原子解引用就可以得到它的值，这和引用是一样的。

```
(deref current-track)
-> "Venus, the Bringer of Peace"

@current-track
-> "Venus, the Bringer of Peace"
```

原子并不参与事务，因而不需要 dosync。要为一个原子设置值，简单的调用 reset!即可。

```
(reset! an-atom newval)
```

例如，你可以把 current-track 设置为“Credo”。

```
(reset! current-track "Credo")
-> "Credo"
```

如果打算使用原子对 current-track 和 current-composer 两者进行协同更新，应该怎么做呢？简短的回答是：“你不能。”这就是引用和原子之间的区别。如果你需要协同访问，那么还是使用引用吧。

比较周全的答案是：“你可以这么做，如果你愿意改变对问题建模的方式的话。”让我们看看把歌曲的标题和作曲家都放进一个映射表，并把这整个映射表保存到一个原子中，会是怎样？

```
(def current-track (atom {:title "Credo" :composer "Byrd"}))
-> #'user/current-track
```

现在你就可以在单个 reset!中一块儿更新它们了。

```
(reset! current-track {:title "Spem in Alium" :composer "Tallis"})
-> {:title "Spem in Alium", :composer "Tallis"}
```

也许你喜欢连续收听同一位作曲家的歌曲。要是这样的话，你一定希望改变的只是歌曲的标题，而保持相同的作曲家。swap!会帮助你实现目标。

```
(swap! an-atom f & args)
```

swap!通过在 an-atom 当前的值上调用函数 f 来更新 an-atom，此外还可以传入额外的一些参数 args。

为了修改歌曲标题，使用 swap!和 assoc 来更新:title。

```
(swap! current-track assoc :title "Sancte Deus")
-> {:title "Sancte Deus", :composer "Tallis"}
```

swap!返回了新的值。如果其他线程也试图修改同一个原子的话，有可能会导致 swap!被再次调用。因此，你传给 swap!的函数应该是没有副作用的。

引用和原子执行的都是同步更新。当更新函数返回时，值就已经改变了。如果你不需要这样的控制级别，并且能够容忍一段时间之后才进行异步更新，那么 agent 代理就是你想要的。

5.4　使用代理进行异步更新

有的应用程序会有这样一些任务，任务之间只需要很少地协同就能彼此独立进行。Clojure 提供了代理来支持这种风格的任务。

代理和引用有很多共同点。和引用一样，你可以通过包装初始状态来创建代理。

```
(agent initial-state)
```

下面创建了一个计数器的代理，并把初始计数值设置为 0。

```
(def counter (agent 0))
-> #'user/counter
```

一旦得到了一个代理，你就可以向它 send 一个函数，来更新其状态。send 把函数

update-fn 放进线程池里的某个线程中开始排队，等待随后执行。

```
(send agent update-fn & args)
```

向代理进行发送，和对引用进行交换非常相像。下面告诉计数器 counter，准备好要自增（inc）了。

```
(send counter inc)
-> #<clojure.lang.Agent@23451c74: 0>
```

注意，调用 send 不会返回代理的新值，而是返回了代理本身。这是因为 send 并不知道那个最新的值是什么。当 send 把 inc 放入队列，排队等待执行之后，它就立即返回了。

尽管 sned 对代理的新值究竟是什么毫不知情，但 REPL 有可能会知道。依赖于代理线程或 REPL 线程谁最先执行，在前面那段输出信息的冒号后面，你可能会看到 1，也可能会看到 0。

就像引用一样，你可以用 deref 或是@来检查代理当前的值。等一会儿你再去检查这个计数器的值时，几乎可以肯定线程池中的那个 inc 任务已经完成，值增长为 1。

```
@counter
-> 1
```

如果 REPL 与代理线程之间的竞争条件让你烦闷不已，有一个解决方案。如果你希望确保代理已经完成了你发送给他的动作，你可以调用 await 或者 await-for。

```
(await & agents)
(await-for timeout-millis & agents)
```

这两个函数会导致当前线程阻塞，直到所有发自当前线程或代理的动作全部完成。如果超过了超时时间，await-for 会返回空，否则会返回一个非空值。await 没有超时时间，所以一定要小心：await 愿意永远等下去。

5.4.1 代理的验证与错误处理

代理与引用还有另一个共同之处。那就是代理也可以携带一个验证函数。

```
(agent initial-state options*)
; options 包括：
; :validator 一个验证函数
; :meta 一个元数据映射表
```

下面使用验证函数来重新创建那个计数器，以确保它一定是个数字。

```
(def counter (agent 0 :validator number?))
-> #'user/counter
```

试着用一个非数字的值来设置这个代理，这里传入的更新函数忽略了当前代理的值，只是简单地返回了一个字符串。

```
(send counter (fn [_] "boo"))
-> #<clojure.lang.Agent@4de8ce62: 0>
```

由于 send 仍然是立刻就返回了，所以目前看起来一切都还不错。等这个代理在一个池中线程（pooled thread）里试着更新它自己时，它就会进入异常状态。一旦你试图对这个代理解引用时，就能察觉到出错了。

```
@counter
-> java.lang.Exception: Agent has errors
```

为了找出具体的错误，可以调用 agent-errors，它会返回一个序列。该序列中包含了代理执行动作期间抛出的错误。

```
(agent-errors counter)
-> (#<IllegalStateException ...>)
```

一旦代理出错，所有对该代理的后续查询尝试，都会返回错误。你可以通过调用 clear-agent-errors，令这个代理再次可用。

```
(clear-agent-errors agent)
```

这样，代理就会回到出错之前的状态。下面清除计数器 counter 的错误，并且验证其状态是否与错误发生前相同。

```
(clear-agent-errors counter)
-> nil

@counter
-> 0
```

现在你已经学会了代理的基本知识，下面让我们结合引用和事务来运用它们。

5.4.2　在事务中包含代理

事务应当没有副作用，因为 Clojure 有可能会重试事务任意多次。然而，有时候你会希望当事务成功时，能做一些有副作用的事情。代理提供了一种解决方案。如果你在事务中向代理发送了一个动作，那么这个动作仅当事务成功时才会真正被发送，且仅发送一次。

作为一个很有用的例子，不妨考虑当事务成功时，让一个代理去写文件。你可以把“commute”一节中聊天室的例子与一个代理相结合，对聊天室消息进行自动备份。首先，创建一个 backup-agent，它保存了将要写入的文件名。

src/examples/concurrency.clj

```
(def backup-agent (agent "output/messages-backup.clj"))
```

然后，创建一个 add-message 的修改版本。这个新的 add-message-with-backup 函数应该要做两件额外的事情。

- 在 let 绑定中抓取 commute 的返回值，它是当前消息的数据库。
- 在同一个事务中，向代理发送把消息数据库写入文件的动作。简单起见，让这个动作函数返回文件名，以便该代理在下一次备份时能使用相同的文件名。

```
(defn add-message-with-backup [msg]
  (dosync
    (let [snapshot (commute messages conj msg)]
      (send-off backup-agent (fn [filename]
                               (spit filename snapshot)
                               filename))
      snapshot)))
```

这个新函数还有另一个关键区别：它和代理通讯时，调用的是 send-off 而不是 send。send-off 是 send 的一个变体，针对可能会造成阻塞的动作，比如写文件。send-off 的动作会得到属于它们自己的可扩展线程池。不要用 send 来发送可能会阻塞的函数，否则你可能会不必要的妨碍到其他代理工作。

下面试着使用 add-message-with-backup 来添加几条消息。

```
(add-message-with-backup (user.Message. "John" "Message One"))
-> (#:user.Message{:sender "John", :text "Message One"})

(add-message-with-backup (user.Message. "Jane" "Message Two"))
-> (#:user.Message{:sender "Jane", :text "Message Two"}
    #:user.Message{:sender "John", :text "Message One"})
```

你不妨检查一下内存中的 messages 和备份文件 messages-backup，以验证它们是否包含了同样的结构。

你可以用各种方式来增强本例中的备份策略。比如你可以提供选项，令备份频率低于更新频率，或是仅备份那些自上次备份以来发生了变化的信息。

由于 Clojure 的 STM 提供了 ACID 中的 ACI，另外由于写文件可视为提供了 D

（durability，持久性），就诱使人们认为 STM 加上一个用来备份的代理，就相当于是个数据库了。事实并非如此。Clojure 的事务仅承诺会发送一个动作给代理，它并没有真正的在 ACI 保护伞之下执行这个动作。所以，举个例子，一个事务可能已经完成，然后有人在代理写入数据库之前，就把电源线给拔掉了。其寓意非常简单。如果解决你的问题需要一个真正的数据库，那就使用真正的数据库吧。

5.4.3　统一的更新模型

如你所见，引用、原子和代理都提供了基于它们当前的状态，通过应用其他函数来更新这些状态的函数。这种处理共享状态的统一模型是 Clojure 的核心概念之一。表 5-1 总结了这种统一的模型及各种辅助函数。

表 5-1　统一的模型及辅助函数

更新机制	引用函数	原子函数	代理函数
应用函数	alter	swap!	send-off
函数（交换）	commute	不适用	不适用
函数（非阻塞）	不适用	不适用	send
简单设置	ref-set	reset!	不适用

统一的更新模型，是迄今为止用来更新引用、原子和代理的最重要的方式。另一方面，其他每个辅助函数，都是源于其特殊语义的优化和选项。

- commute 优化的时机出现在进行协同更新时。因为只有引用提供了协同的更新，所以 commute 只对引用才有意义。
- 对引用和原子的更新发生在访问它们的线程中，所以它们没有提供调度选项。而代理则是在一个线程池中稍后更新，故产生了与调度相关的阻塞或非阻塞选项。

变量（Var），是 Clojure 的最后一种引用类型，是全然不同的东西。它们并未参与到统一的更新模型当中，相反，是用于管理线程内的私有状态。

5.5　用变量管理线程内状态

当你调用 def 或者 defn 时，你就创建了一个动态变量[①]，也常常被简称为变量。到目前为止，本书所有的例子中，你都传了一个初始值给 def，此初始值便成为了该变

① 译注：这种说法适用于 Clojure 1.3.0 之前的版本。自 1.3.0 开始，用 def 和 defn 等定义的默认情况下不再是动态变量，不能使用 binding 来进行动态绑定，除非像本节中的 foo 变量一样，明确指定了 dynamic 元数据。

量的根绑定。例如，下面的代码为 foo 创建了一个根绑定 10。

```
(def ^:dynamic foo 10)
-> #'user/foo
```

这个对 foo 的绑定在所有的线程间共享。你可以在你自己的线程中检查 foo 的值。

```
foo
-> 10
```

你也可以在其他的线程中检查 foo 的值。下面创建一个新线程，传入了一个用来打印 foo 的函数。还有，可别忘了启动这个线程哦。

```
user=> (.start (Thread. (fn [] (println foo))))
-> nil
| 10
```

前面的例子中，调用 start()返回了 nil，随后 foo 的值在一个新的线程中被打印了出来。

大多数变量都甘愿保持它们的根绑定永不改变。然而，你可以借助 binding 宏，为一个变量创建线程内的绑定。

```
(binding [bindings] & body)
```

绑定具有动态范围。换句话说，在 binding 创建的这个范围内，线程执行过程中需要经过的任何地方，绑定都是可见的，直到该线程离开了该范围。同时对于其他线程而言，绑定也是不可见的。

在结构上，binding 与 let 看起来非常相像。下面为 foo 创建一个线程内绑定，并检查它的值。

```
(binding [foo 42] foo)
-> 42
```

为了观察 binding 和 let 之间的差异，让我们先创建一个简单的函数，用来打印 foo 的值。

```
(defn print-foo [] (println foo))
-> #'user/print-foo
```

现在，从 let 和 binding 中都分别调用一下 print-foo 试试看。

```
(let [foo "let foo"] (print-foo))
| 10
```

```
(binding [foo "bound foo"] (print-foo))
| bound foo
```

正如你看到的，let 不会对其形式的外部造成影响，所以第一个 print-foo 打印的仍然是根绑定 10。另一方面，binding 会对所有始于其形式中的调用链产生影响，所以第二个 print-foo 打印的是 bound foo。

5.5.1　远距离作用

打算用于动态绑定的变量有时又被称为特殊变量。为它们命名时的好习惯是，在其首尾加上一对星号。例如，Clojure 对线程范围内的选项使用了动态绑定，譬如标准输入输出流*in*、*out*和*err*。动态绑定使得远距离作用成为可能。

当改变一个动态绑定时，无需变动任何参数，你就能改变那些位于远处的函数的行为。

有一类远距离作用是，临时性地增加函数的行为。在一些语言中，这被归类为面向方面编程，而在 Clojure 中，这只不过是动态绑定的一个副产品罢了。作为例子，假想你有一个执行某种昂贵计算的函数。为了模拟这种情况，编写一个名为 slow-double 的函数，它会在休眠 1/10 秒后将其输入翻倍。

```
(defn ^:dynamic slow-double [n]
  (Thread/sleep 100)
  (* n 2))
```

接下来，编写一个名为 calls-slow-double 的函数，它每次都会用[1 2 1 2 1 2]中的一项来调用 slow-double。

```
(defn calls-slow-double []
  (map slow-double [1 2 1 2 1 2]))
```

接下来对调用 calls-slow-double 进行计时。由于其内部调用了 6 次 slow-double，所以应该会耗用 6/10 秒多一点。注意，你必须用 dorun 过一遍它的结果，否则，Clojure 的 map 函数会弄巧成拙的立刻返回一个惰性序列。

```
(time (dorun (calls-slow-double)))
| "Elapsed time: 601.418 msecs"
-> nil
```

读过那段代码后，你也许会说 calls-slow-double 之所以很慢，是因为它一遍又一遍的在做同样的工作。1 乘以 2 就是 2，不管你做多少次都是如此。

类似于 slow-double 这种类型的计算，是非常好的快存目标。当你对函数进行了快存时，它就会维护一块缓存，把过往输入映射至过往输出。如果后续调用命中了缓存，它们几乎是立即返回的。因此，你用空间（那块缓存）换取了时间（相同的输入，也重新进行计算）。

Clojure 提供了 memoize，它接受一个函数作为参数，并返回该函数的快存版本。

```
(memoize function)
```

于快存而言，slow-double 是很合适的对象，但此时它仍然还是未快存的，诸如 calls-slow-double 这样的客户已经使用了它那个缓慢的未快存版本。使用动态绑定，这就不再是个问题了。只需要创建 slow-double 的一个快存版本的绑定，并在这个绑定的内部调用 calls-slow-double 即可。

```
(defn demo-memoize []
  (time
    (dorun
      (binding [slow-double (memoize slow-double)]
        (calls-slow-double)))))
```

使用 slow-double 的快存版本，calls-slow-double 足足快了三倍，只用了大约 2/10 秒就完成了。

```
(demo-memoize)
"Elapsed time: 203.115 msecs"
```

这个例子展示了远距离作用的威力以及危险。通过对诸如 slow-double 这样的函数进行动态绑定，你可以对类似 calls-slow-double 这样的函数毫不知情，甚至在不允许的情况下，更改它们的行为。用诸如 let 这样的词法绑定形式，就能很容易地对类似更改的整个作用范围一览无遗。而动态绑定则没有那么简单。它会改变别的文件中其他形式的行为，即便那个地方距离你进行绑定的源码位置非常遥远。

偶尔用之，动态绑定能产生巨大的威力。但它绝不应该成为你进行扩展或是重用的主要机制。使用了动态绑定的函数就不再是纯函数了，并且会迅速丧失 Clojure 函数式风格的诸多益处。

5.5.2 用于 Java 回调 API

一些 Java API 依赖于回调事件处理器。诸如 Swing 这样的 GUI 框架，使用事件处理器来对用户的输入做出响应。而诸如 SAX 这样的 XML 解析器，也依赖于需要用户

实现的回调处理器接口。

要记住，这些回调处理器是用可变对象编写的。此外，它们往往还是单线程的。在 Clojure 中，满足此类 API 最好的折衷办法就是使用动态绑定。这会牵涉到可变引用，感觉上就好像是“变”量一样，但因为它们仅在一个单线程的环境中使用，所以也不会出现任何的并发问题。

Clojure 提供了用于设置线程内动态绑定的特殊形式 set!。

```
(set! var-symbol new-value)
```

set!应该很少会被用到。事实上，在整个 Clojure 核心代码中，只有一处用到了 set!，那就是 ContentHandler，SAX 的 Clojure 实现。

ContentHandler 接收回调作为解析器，以处理会随时到来的 XML 数据流。在现实的场景中，ContentHandler 需要保持追踪 XML 流已解析至何处：当前已打开元素的堆栈、当前的字符数据等等。

用 Clojure 的话来说，你可以认为所谓 ContentHandler 的当前位置，就是一个可变的指针，指向了一个不可变 XML 流中的某个特定点。因为一切都发生在单个线程中，所以没有必要在 ContentHandler 中使用引用类型。作为替代，Clojure 的 ContentHandler 使用了动态绑定变量和 set!。相关细节如下。

```
; 节选自 Clojure 的 xml.clj，聚焦动态变量的使用
(startElement
 [uri local-name q-name #^Attributes atts]
  ; 细节省略
  (set! *stack* (conj *stack* *current*))
  (set! *current* e)
  (set! *state* :element))
nil)
(endElement
 [uri local-name q-name]
  ; 细节省略
  (set! *current* (push-content (peek *stack*) *current*))
  (set! *stack* (pop *stack*))
  (set! *state* :between)
 nil)
```

SAX 解析器在它每遇到一个 XML 的开始标记时，就会调用 startElement。回调处理器随即更新三个线程内的变量。*stack*是一个堆栈，保存了嵌套着当前元素的所有父元素。*current*是当前元素，*state*对其内部是何种内容保持追踪（其中最重要的恐怕就是字符类型数据了，此处并未列出相关细节）。

与 startElement 相对，endElement 把*stack*的栈顶弹出，并放置到*current*中。

值得一提的是，这种编程风格是行业惯例：对于可变对象，如果是单线程程序，那么可以不必在意并发的可能性。作为一种明确的特殊情况，Clojure 允许这种风格的存在，并且仅当以互操作为目的时，你才应该采用这种风格。

ContentHandler 对 set!的使用，并没有使可变数据外泄到其他的 Clojure 程序中。Clojure 利用 ContentHandler 的实现构建了不可变的 Clojure 数据结构。

现在你已经看到了四种不同的状态处理模型。此外由于 Clojure 是建立在 Java 之上的，你也可以使用 Java 基于锁定的模型。表 5-2 总结了这些模型，以及它们的用途。

表 5-2 Java 基于锁定的模型

模型	用途	函数
引用和 STM	协同的、同步的更新	纯粹的
原子	非协同的、同步的更新	纯粹的
代理	非协同的、异步的更新	任意
变量	线程内的动态范围	任意
Java 锁定	协同的、同步的更新	任意

接下来让我们把这些模型结合起来，设计一个麻雀虽小，但却五脏俱全的应用吧。

5.6 Clojure 贪吃蛇

贪吃蛇游戏的特征是，有一条由玩家控制的蛇，在游戏网格中为了吞食苹果而来回游走。每当你的蛇吞下一个苹果时，它就会伸长一小段，同时另一个新的苹果出现。如果你的贪吃蛇达到一定的长度，你就赢了。但是如果你的贪吃蛇越过了它自己的身体，你就输了。

开始建立你自己的贪吃蛇游戏之前，先花一分钟时间来体验一下已完成的版本吧。在本书的 REPL 中，输入以下内容。

```
(use 'examples.snake)

(game)
-> [#<Ref clojure.lang.Ref@65694ee6>
#<Ref clojure.lang.Ref@261ae209>
#<Timer javax.swing.Timer@7f0df737>]
```

选中贪吃蛇游戏窗口，并使用方向键控制你的贪吃蛇。

通过把贪吃蛇游戏分为三个层次，我们的设计将充分利用 Clojure 的函数式本质，以及 Clojure 对显式可变状态的支持能力。

- 函数式模型，使用纯函数尽可能多地为游戏进行建模。
- 可变模型，处理游戏的可变状态。可变模型将会用到一种或是多种本章讨论的引用模型。由于可变模型测试起来要难得多，所以我们会让这个部分保持尽可能小。
- GUI，使用 Swing 来绘制游戏，并接收来自用户的输入。

这些层次的划分，使得这个贪吃蛇游戏构建、测试和维护起来，都非常容易。

为了完成这个示例，请把你的代码添加到示例代码目录中的文件 reader/snake.clj 中。当打开这个文件时，你会发现已经使用 import 和 use 导入了几个 Swing 类与 Clojure 库，这些类和库你随后就会用到。

src/reader/snake.clj

```
(ns reader.snake
  (:import (java.awt Color Dimension)
           (javax.swing JPanel JFrame Timer JOptionPane)
           (java.awt.event ActionListener KeyListener))
  (:use examples.import-static))
(import-static java.awt.event.KeyEvent VK_LEFT VK_RIGHT VK_UP VK_DOWN)
```

现在，你可以开始构建函数式模型了。

5.6.1　函数式模型

首先，让我们创建一组常量，用来描述时间、空间和运动。

```
(def width 75)
(def height 50)
(def point-size 10)
(def turn-millis 75)
(def win-length 5)
(def dirs { VK_LEFT  [-1 0]
            VK_RIGHT [ 1 0]
            VK_UP    [ 0 -1]
            VK_DOWN  [ 0 1]})
```

width 和 height 设置了游戏板的大小，point-size 则用来把游戏点位转换为屏幕像素。turn-millis 是这个游戏的心跳，控制每隔多少毫秒更新一次游戏面板。win-length 是在你赢得游戏之前，贪吃蛇需要吃到多少段的长度（5 是个有些乏味的小数字，但于测试而言却非常合适）。dirs 把四个方向的矢量等价物映射为符号常量。由于 Swing

已经为不同的方向定义了 VK_常量，我们将重用它们，而不是再自己定义一份。

贪吃蛇的其他实现

有不止一种方法可以实现贪吃蛇游戏。你可能会有兴趣将本节展示的贪吃蛇游戏，与其他版本进行一些比较。

- http://planet.plt-scheme.org/package-source/dvanhorn/snake.plt/1/0/main.ss
  大卫·范·霍恩（David Van Horn）的版本，使用 Typed Scheme（一种类型化的 Scheme 方言，是 PLT Scheme 版本 4 的一部分）语言编写，不存在任何可变状态。
- http://www.plt1.com/1069/smaller-snake/
  杰里米·里德（Jeremy Read）编写了一个 Java 的版本，设计为“差不多小的刚好能用 Java 做出来，并仍然可读”。
- http://www.plt1.com/1070/even-smaller-snake/
  阿布舍克·雷迪（Abhishek Reddy）编用 Clojure 编写了一个极小的版本，只用了 35 行。其设计目标是简洁到极致。
- http://www.ccs.neu.edu/home/cce/acl2/worm.html
  戴尔·瓦尔兰科特（Dale Vaillancourt）的蠕虫游戏。
- http://www.ociweb.com/mark/programming/ClojureSnake.html
  马克，福克曼（Mark Volkmann）编写的 Clojure 版本，旨在提高可读性。

以上每种贪吃蛇游戏都有其独特的风格。你的风格呢？

接下来，为游戏创建一些基本的数学函数。

```
(defn add-points [& pts]
  (vec (apply map + pts)))

(defn point-to-screen-rect [pt]
  (map #(* point-size %)
       [(pt 0) (pt 1) 1 1]))
```

add-points 函数把几个点加到一起。你可以用 add-points 来计算移动游戏对象的新位置。例如，你可以把一个位于坐标[10 10]的对象左移一步。

```
(add-points [10 10] [-1 0])
-> [9 10]
```

point-to-screen-rect 简单的把游戏中的一个点，转换为屏幕上的一个矩形。

```
(point-to-screen-rect [5 10])
-> (50 100 10 10)
```

下面，让我们编写一个函数，用来创建一个新苹果。

```
(defn create-apple []
  {:location [(rand-int width) (rand-int height)]
   :color (Color. 210 50 90)
   :type :apple})
```

每个苹果占据一个点，用:location 表示，并保证其一定位于游戏面板之上。相比而言，贪吃蛇则要略微复杂一些。

```
(defn create-snake []
  {:body (list [1 1])
   :dir [1 0]
   :type :snake
   :color (Color. 15 160 70)})
```

由于一条贪吃蛇可以占据面板上的多个点，所以它有个:body，保存了一系列的点。此外，蛇总是朝某个方向在运动，这用:dir 来表示。

接下来，创建一个函数用来移动贪吃蛇。这应该是个纯函数，会返回一条新的蛇。同时，它还应该接受一个 grow 选项，允许贪吃蛇在吃掉一个苹果后增加长度。

```
(defn move [{:keys [body dir] :as snake} & grow]
  (assoc snake :body (cons (add-points (first body) dir)
                           (if grow body (butlast body)))))
```

move 使用了一个相当复杂的绑定表达式。{:keys [body dir]}这个部分，令贪吃蛇的身体和方向都拥有了自己的绑定，同时:as snake 这个部分则绑定了整条贪吃蛇。函数接下来还做了以下几件事情。

1．add-points 创建了一个新的点，这个点是贪吃蛇根据其运动方向产生位移后，头部应该到达的位置。

2．cons 把这个新建的点添加到贪吃蛇的前端。如果蛇加长了，整个原来的贪吃蛇都会被保留下来，否则的话，就只保留原来的贪吃蛇中除最后一段以外的其他部分（butlast）。

3．assoc 返回一条新的贪吃蛇，这是旧蛇的一个副本，但:body 被更新过了。

下面通过移动和加长一条贪吃蛇，来测试一下 move。

```
(move (create-snake))
-> {:body ([2 1]), ; etc.
(move (create-snake) :grow)
-> {:body ([2 1] [1 1]), ; etc.
```

接下来编写一个 win?函数，用来检测贪吃蛇是否赢得了这场游戏。

```
(defn win? [{body :body}]
  (>= (count body) win-length))
```

下面用不同的蛇体长度来测试 win?。注意 win?仅绑定了:body，所以你并不需要用一条“真正”的蛇，任意一个具有:body 的东西都行。

```
(win? {:body [[1 1]]})
-> false

(win? {:body [[1 1] [1 2] [1 3] [1 4] [1 5]]})
-> true
```

如果贪吃蛇的头部撞上了身体的其他部分，它就输了。编写一个 head-overlaps-body?函数来对这种情况进行检测，并用这个函数来定义 lose?。

```
(defn head-overlaps-body? [{[head & body] :body}]
  (contains? (set body) head))

(def lose? head-overlaps-body?)
```

让我们分别用重叠和不重叠的贪吃蛇身体来测试一下 lose?。

```
(lose? {:body [[1 1] [1 2] [1 3]]})
-> false

(lose? {:body [[1 1] [1 2] [1 1]]})
-> true
```

如果贪吃蛇的头部占据了一个苹果的位置，那么就意味着这个苹果被吃掉了。定义一个 eats?函数来检测这种情况。

```
(defn eats? [{[snake-head] :body} {apple :location}]
  (= snake-head apple))
```

注意 eats?函数的函数体是何等之干净。所有事情都在绑定中完成了：{[snake-head] :body}把 snake-head 绑定到了贪吃蛇:body 的第一个元素之上，{apple :location}则把 apple 绑定到了苹果的:location 之上。在 REPL 中测试一下 eats?。

```
(eats? {:body [[1 1] [1 2]]} {:location [2 2]})
-> false

(eats? {:body [[2 2] [1 2]]} {:location [2 2]})
-> true
```

最后，你还需要某种途径来让贪吃蛇转向，更新它的:dir。

```
(defn turn [snake newdir]
  (assoc snake :dir newdir))
```

turn 会返回一条更新了方向的新蛇。

```
(turn (create-snake) [0 -1])
-> {:body ([1 1]), :dir [0 -1], ; etc.
```

目前为止，你已经编写的所有这些代码，都属于贪吃蛇游戏中函数式模型的一部分。因为没有任何局部变量或是可变状态，所以这个部分理解起来非常容易。正如你在下一节将看到的，游戏中可变状态的数量相当之少（甚至可以不使用任何可变状态就能实现贪吃蛇游戏，只不过本演示无意于此）。

5.6.2　使用 STM 建立可变模型

贪吃蛇游戏的可变状态，只在三种情况下会发生改变。

- 游戏可被重置为初始状态。
- 每一轮，贪吃蛇都会更新其位置。如果它吃了一个苹果，则一个新的苹果会被放进来。
- 贪吃蛇可以转向。

我们会把每种这些变化实现为函数，这些函数会在一个事务中修改 Clojure 引用。这样一来，改变贪吃蛇和苹果的位置将是同步和协同的。

reset-game 函数很简单。

```
(defn reset-game [snake apple]
  (dosync (ref-set apple (create-apple))
          (ref-set snake (create-snake)))
  nil)
```

你可以传入一些引用，并随后检查它们，以此来测试 reset-game，看这些引用能否被解引用为一条贪吃蛇和一个苹果。

```
(def test-snake (ref nil))
(def test-apple (ref nil))

(reset-game test-snake test-apple)
-> nil
@test-snake
-> {:body ([1 1]), :dir [1 0], ; etc.

@test-apple
-> {:location [52 8], ; etc.
```

update-direction 甚至还要更加简单；它仅仅是对那个函数式的 turn 进行了一次微不足道的包装。

```
(defn update-direction [snake newdir]
  (when newdir (dosync (alter snake turn newdir))))
```

来试着把你的 test-snake 转朝“向上”的方向。

```
(update-direction test-snake [0 -1])
-> {:body ([1 1]), :dir [0 -1], ; etc.
```

最复杂的变化函数是 update-positions。如果贪吃蛇吃掉了苹果，一个新的苹果会被创建出来，并且这条贪吃蛇也会变长。否则，这条贪吃蛇只是简单的向前移动。

```
(defn update-positions [snake apple]
  (dosync
    (if (eats? @snake @apple)
      (do (ref-set apple (create-apple))
          (alter snake move :grow))
      (alter snake move)))
  nil)
```

为了测试 update-positions，先要重置一下游戏。

```
(reset-game test-snake test-apple)
-> nil
```

然后，把苹果挪到马上就要被贪吃蛇咬到的地方。

```
(dosync (alter test-apple assoc :location [1 1]))
-> {:location [1 1], ; etc.
```

现在，在调用了 update-positions 之后，你应该会有一条更大的，有两节长的贪吃蛇。

```
(update-positions test-snake test-apple)
-> nil

(:body @test-snake)
-> ([2 1] [1 1])
```

这就是贪吃蛇世界中所有的可变状态了：三个函数，大约十几行代码。

5.6.3 贪吃蛇的 GUI

贪吃蛇的 GUI 包括以下功能：绘制屏幕对象、响应用户输入和装配各种 Swing 组件。由于贪吃蛇和苹果被绘制为简单的点，所以绘画函数很简单。fill-point 函数用来填充一个点。

```
(defn fill-point [g pt color]
  (let [[x y width height] (point-to-screen-rect pt)]
    (.setColor g color)
    (.fillRect g x y width height)))
```

多重函数 paint 知道应该如何来绘制贪吃蛇和苹果。

```
1 (defmulti paint (fn [g object & _] (:type object)))
2
3(defmethod paint :apple [g {:keys [location color]}]
4  (fill-point g location color))
5
6(defmethod paint :snake [g {:keys [body color]}]
7  (doseq [point body]
8    (fill-point g point color)))
```

paint 接受两个必需的参数：g 是 java.awt.Graphics 的一个实例，object 是要被绘制的对象。defmulti 包含一个可选的剩余参数，以便将来 paint 的实现可以选择接受更多地参数。对多重方法的深入描述，参见第 8.2 节“定义多重方法”。在第 3 行，paint 的:apple 方法绑定了苹果的位置 location 和颜色 color，并使用它们在屏幕上绘制一个点。在第 6 行，:snake 方法绑定了贪吃蛇的身体 body 和颜色 color，并使用 doseq 来绘制蛇体的每一个点。

UI 部分的精髓是 game-panel 函数，它创建了一个带有事件处理器的 Swing JPanel，用来绘制游戏、在每次定时器滴答时进行更新、以及对用户输入做出响应。

```
1 (defn game-panel [frame snake apple]
-  (proxy [JPanel ActionListener KeyListener] []
-    (paintComponent [g]
-      (proxy-super paintComponent g)
5      (paint g @snake)
-      (paint g @apple))
-    (actionPerformed [e]
-      (update-positions snake apple)
-      (when (lose? @snake)
10       (reset-game snake apple)
-        (JOptionPane/showMessageDialog frame "You lose!"))
-      (when (win? @snake)
-        (reset-game snake apple)
-        (JOptionPane/showMessageDialog frame "You win!"))
15     (.repaint this))
-    (keyPressed [e]
-      (update-direction snake (dirs (.getKeyCode e))))
-    (getPreferredSize []
-      (Dimension. (* (inc width) point-size)
20                (* (inc height) point-size)))
-    (keyReleased [e])
-    (keyTyped [e])))
```

game-panel 函数很长，但很简单。它使用 proxy 来创建带有 Swing 回调方法的面板。

- Swing 调用 paintComponent（第 3 行）来绘制面板。paintComponent 则调用

proxy-super 来启用普通 JPanel 的行为，随后绘制贪吃蛇和苹果。

- 当每次定时器滴答时，Swing 还会调用 actionPerformed（第 17 行）。actionPerformed 会更新贪吃蛇和苹果的位置。如果游戏结束，它还会显示一个提示对话框并重置游戏。最后，它用(.repaint this)触发重绘事件。
- 为了对键盘输入做出响应，Swing 会调用 keyPressed（第 16 行）。keyPressed 调用 update-direction 以改变贪吃蛇的方向。如果键盘输入的不是方向键，dirs 函数返回 nil，且 update-direction 什么都不做。
- 游戏面板忽略了 keyReleased 和 keyTyped。

game 函数用来创建一个新游戏。

```
1 (defn game []
-  (let [snake (ref (create-snake))
-        apple (ref (create-apple))
-        frame (JFrame. "Snake")
5       panel (game-panel frame snake apple)
-        timer (Timer. turn-millis panel)]
-    (doto panel
-      (.setFocusable true)
-      (.addKeyListener panel))
10   (doto frame
-      (.add panel)
-      (.pack)
-      (.setVisible true))
-    (.start timer)
15   [snake, apple, timer]))
```

在第 2 行，game 函数创建了所有必须的游戏对象：可变模型对象 snake 和 apple，还有 UI 组件 frame、panel 及 timer。第 7 和第 10 行执行 panel 和 frame 的样板初始化。第 14 行通过开启定时器来启动游戏。

第 15 行返回了一个包含贪吃蛇、苹果和定时器的向量。这是为了便于在 REPL 中进行测试：你可以使用这些对象来移动贪吃蛇和苹果，或是启动和停止游戏。

来吧，再玩一次这个游戏，你已经拥有它了。要启动游戏，可以在 REPL 中 use 这个贪吃蛇游戏库，并运行 game 函数。如果你是自己敲入的这些代码，你可以用你自己挑选的库名（本教程中是 examples.reader）；否则的话，你可以使用完整的示例 examples.snake。

```
(use 'examples.snake)

(game)
-> [#<Ref clojure.lang.Ref@6ea27cbe>
```

```
#<Ref clojure.lang.Ref@6dabd6b0>
#<Timer javax.swing.Timer@32f60451>]
```

游戏窗口可能会出现在你的 REPL 窗口后面。如果发生这种状况，用你的操作系统来定位游戏窗口吧。

对于这个贪吃蛇游戏，还可以有很多改进。如果贪吃蛇到达了屏幕的边缘，也许它应该立即转向，以避免从视野中消失。或者（严厉的爱）也许你会直接输掉这场游戏！通过适合你个人风格的改进，让这个贪吃蛇游戏成为你自己的作品吧。

5.6.4　没有引用的贪吃蛇

我们选择使用引用来实现贪吃蛇游戏的可变模型，这样我们就可以对蛇和苹果协同更新了。别的方法也同样有效。例如，你可以把贪吃蛇和苹果的状态合并到单个 game 对象中。只有一个对象，就不再需要协同了，这样你可以用一个原子来加以替代。

文件 examples/atom-snake.clj 演示了这种方法。其中，像 update-positions 这样的函数也都成为了函数式模型中的一部分，并会返回一个带有已更新状态的新 game 对象。

src/examples/atom_snake.clj

```
(defn update-positions [{snake :snake, apple :apple, :as game}]
  (if (eats? snake apple)
    (merge game {:apple (create-apple) :snake (move snake :grow)})
    (merge game {:snake (move snake)})))
```

注意，解构让获取 game 的内部结构变得何等容易：通过参数列表，贪吃蛇和苹果都得到了绑定。

实际上，可变的更新现在全都是原子 swap!了。我们发现，这简单到足以把他们留在 UI 函数 game-panel 中，如同此处摘录所示。

```
(actionPerformed [e]
  (swap! game update-positions)
  (when (lose? (@game :snake))
    (swap! game reset-game)
    (JOptionPane/showMessageDialog frame "You lose!"))
```

同样，还有其他的可能性。克里斯·豪泽（Chris Houser）在本书示例代码的基础上开了一个分支[①]，演示了如何用一个代理进行 Thread/sleep，来替代 Swing 的定时器，

① http://github.com/Chouser/programming-clojure。

而且还在每个游戏游戏回合中使用一个新的代理来更新游戏状态。

5.7 小结

Clojure 的引用模型是这门语言最具创新性的部分，结合了软事务内存、代理、原子和动态绑定技术。正如你在本章中看到的，对于各种各样存在状态的系统，这些技术赋予了 Clojure 强大的抽象能力。这也使得 Clojure 成为少数几门适合于未来新一代多核计算机硬件的语言之一。

接下来，我们将会看到 Clojure 的新特性之一。有人将其称为“表达问题”[①]的一种解决方案。我们把它叫做协议。

① http://en.wikipedia.org/wiki/Expression_problem。

第 6 章

协议和数据类型

抽象为代码重用打下了基础。Clojure 语言自身对序列、容器和可调用性进行了抽象。传统上，这些抽象就意味着：定义 Java 接口并用 Java 类实现之。一开始，Clojure 提供了 proxy 和 genclass，使得不再依靠 Java 也能达成这一目标，但自从引入了协议之后，情况又发生了变化。

- 协议（Protocol）作为 Java 接口的替代物，适于高性能的多态方法调度。
- 数据类型（Datatype）作为 Java 类的替代物，可用来创建抽象定义的实现，无论该抽象是由协议还是接口定义的。

协议和数据类型提供了一种高性能、灵活的机制，使得采用 Clojure 进行编程时，不必再为了抽象化编写 Java 接口，或是为了具体化而编写 Java 类。使用协议和数据类型，你可以创建新的抽象，以及实现了这些抽象的新类型，更为甚者，你还可以将新的抽象扩展至现有类型。

本章中，我们将会探讨 Clojure 使用协议和数据类型的抽象之道。首先，我们会重新实现一份我们自己的 spit 和 slurp 函数，它们都是 Clojure 的内建函数。然后，我们会开个小差，去构建一个 CryptoVault，在那里你能学到关于扩展 Java 标准库的知识。最后，我们将会把所有东西都放到一起，使用记录和协议来定义音符及其序列。通过这些练习，你无疑将会体验到 Clojure 可组合抽象的强大力量。

6.1 针对抽象编程

Clojure 的输入输出函数 spit（吐）和 slurp（吃）建立在两个抽象之上：读和写。

这意味着你可以对各种各样的源和目标使用它们，包括文件、URL 链接和套接字，并且它们还能被任何人扩展，以支持其他新的类型，无论是那些类型是已经存在的，还是新定义的。

- slurp 函数接受一个输入源作为参数，读取其内容，并以字符串的形式返回。
- spit 函数接受一个输出目标和一个值作为参数，并把值转换为字符串，然后写入到输出目标。

我们将从这两个函数的基础版本开始，即只能从文件读以及向文件写。然后我们会对这个基础版本进行几次重构，探索几种不同的途径，来支持更多的数据类型。通过这样的做法，让能充分感受到面向通用和灵活的抽象编程，是何等实用，特别是在这个过程中，你还能感受到 Clojure 的协议和数据类型的强大威力。

我们自己编写的 spit 和 slurp 版本，分别叫做 expectorate 和 gulp，能用于数种现存的数据类型。我们还会创建一种新的数据类型 CryptoVault，也能很好地用于我们自己的那两个函数，如同能够很好的用于 spit 和 slurp 一样。

gulp 和 expectorate

gulp（吞）函数是 Clojure 中 slurp 函数的简化版本，而 expectorate（咳出，先不要管它夸张的名字），则是 Clojure 中 spit 函数的缩减版。让我们先写一个 gulp 的基础版本，它只能对 java.io.File 进行读取。

src/examples/gulp.clj

```
(ns examples.gulp
  (:import (java.io FileInputStream InputStreamReader BufferedReader)))
(defn gulp [src]
  (let [sb (StringBuilder.)]
    (with-open [reader (-> src
                           FileInputStream.
                           InputStreamReader.
                           BufferedReader.)]
      (loop [c (.read reader)]
        (if (neg? c)
          (str sb)
          (do
            (.append sb (char c))
            (recur (.read reader))))))))
```

gulp 函数从给定的 File 对象创建一个 BufferedReader，随后在它上面进行

loop/recur，每次读取一个字符，并追加到一个 StringBuilder 中，直至到达到输入的末尾，最后返回一个字符串。而基础版本的 expectorate 函数，甚至比 gulp 还要更小些。

src/examples/expectorate.clj

```
(ns examples.expectorate
  (:import (java.io FileOutputStream OutputStreamWriter BufferedWriter)))

(defn expectorate [dst content]
  (with-open [writer (-> dst
                         FileOutputStream.
                         OutputStreamWriter.
                         BufferedWriter.)]
    (.write writer (str content))))
```

它创建了一个 BufferedWriter 文件，把 content 参数的值转换为字符串，并写入 BufferedWriter。

但如果我们还想支持其他类型，例如套接字、URL 链接和基本输入输出流，又该怎么做呢？我们需要对 gulp 和 expectorate 加以改进，使其能从文件以外的其他数据类型中，生成 BufferedReaders 和 BufferedWriters。所以，让我们创建两个新函数 make-reader 和 make-writer，它们负责以下行为。

- make-reader 函数从输入源生成一个 BufferedReader。
- make-writer 函数从输出目标生成一个 BufferedWriter。

```
(defn make-reader [src]
  (-> src FileInputStream. InputStreamReader. BufferedReader.))
(defn make-writer [dst]
  (-> dst FileOutputStream. OutputStreamWriter. BufferedWriter.))
```

就像我们基础版本的 gulp 和 expectorate 函数一样，make-reader 和 make-writer 只对文件有效，但不久就会有所改变。现在，让我们使用这两个新函数对 gulp 和 expectorate 进行一下重构。

src/examples/protocols.clj

```
(defn gulp [src]
  (let [sb (StringBuilder.)]
    (with-open [reader (make-reader src)]
      (loop [c (.read reader)]
        (if (neg? c)
          (str sb)
          (do
            (.append sb (char c))
            (recur (.read reader))))))))
```

```
(defn expectorate [dst content]
  (with-open [writer (make-writer dst)]
    (.write writer (str content))))
```

现在，我们只需要更新 make-reader 和 make-writer，就可以使 gulp 和 expectorate 支持更多的源和目标了。一种支持更多类型的途径是，使用 cond 或者 condp 语句来恰当的处理各种不同类型。例如，下面这个版本的 make-reader，把调用 FileInputStream 的构造函数，替换成了一个 condp 语句，根据给定的输入创建 InputStream，因为无论是文件、套接字、URL 链接还是 InputStream 自己，都是一种 InputStream。

```
(defn make-reader [src]
  (-> (condp = (type src)
        java.io.InputStream src
        java.lang.String (FileInputStream. src)
        java.io.File (FileInputStream. src)
        java.net.Socket (.getInputStream src)
        java.net.URL (if (= "file" (.getProtocol src))
                       (-> src .getPath FileInputStream.)
                       (.openStream src)))
    InputStreamReader.
    BufferedReader.))
```

下面是采用了相同策略的 make-writer。

```
(defn make-writer [dst]
  (-> (condp = (type dst)
        java.io.OutputStream dst
        java.io.File (FileOutputStream. dst)
        java.lang.String (FileOutputStream. dst)
        java.net.Socket (.getOutputStream dst)
        java.net.URL (if (= "file" (.getProtocol dst))
                       (-> dst .getPath FileOutputStream.)
                       (throw (IllegalArgumentException.
                                "Can't write to non-file URL"))))
    OutputStreamWriter.
    BufferedWriter.))
```

这种方法的问题是，它是封闭的：没有人可以不重写 make-reader 和 make-writer，就能支持新的源和目标类型。我们需要的是一种开放的方案，即便代码已然写就，不可更改，其他同伴也能增加对新类型的支持。我们需要的是两个抽象；一个用于读，一个用于写。

6.2 接口

在 Java 中，对这种形式进行抽象时通常采用的机制是接口。接口机制提供了一种手段，能够根据调用时传入的第一个参数的类型，对接口定义中的指定抽象函数进行

调用分派，调度至某个特定的实现。在 Java 中，调用时的第一个参数是隐式的，它就是实现了接口的那个对象。

接口的优势列举如下。

- 一种数据类型可以实现多个接口。
- 接口仅提供规范，而非实现。这就让多接口的实现避免了那些与多重继承相关的问题。

接口的缺点是现有数据类型无法被扩展，为了实现新的接口，你必须重写它们。

在 Clojure 中，我们可以使用 definterface 宏来创建 Java 接口。它接受一个名称，以及一个或多个方法的签名。

```
(definterface name & sigs)
```

让我们来创建自己的抽象，把那些可以读出和可以写入的东西，抽象为一个接口，我们将其称为 IOFactory。

```
(definterface IOFactory
  (^java.io.BufferReader make-reader [this])
  (^java.io.BufferedWriter make-writer [this]))
```

这会创建一个名为 IOFactory 的接口。它有两个抽象函数，make-reader 和 make-writer。任何实现这个接口的类，都必须包含 make-reader 和 make-writer 函数。它们把数据类型自身的实例作为参数，并分别返回一个 BufferedReader 和一个 BufferedWriter。

遗憾的是，一个类支持哪些接口，是由其作者在设计时决定的；一旦定义了某个 Java 类，要想让它支持新的接口就只有重写它。因此，我们没有办法扩展 File、Socket 和 URL 这些类，以实现 IOFactory 接口。

和那两个基于 condp 的 make-reader 和 make-writer 一样，我们的接口也是封闭的，只有作者能对其扩展；其他人则不能。这是所谓“表达问题（expression problem）”[①]的一部分。幸运的是，对于这个问题，Clojure 提供了解决方案。[②]

6.3　协议

Clojure 解决表达问题的解决方案之一，就是协议。协议提供了一种灵活的抽象机

① http://lambda-the-ultimate.org/node/2232。
② http://www.ibm.com/developerworks/java/library/j-clojure-protocols/?ca=drs-。

制，仅提供规范而非实现，以及允许数据类型实现多种协议，充分地吸收了接口中最棒的那个部分。此外，协议旨在解决接口的关键弱点，允许对现有数据类型进行非侵入式的扩展，以支持新的协议。

下面列举了协议的长处。

- 数据类型可以实现多种协议。
- 协议仅提供了规范，而不是实现，这就让多接口的实现避免了那些与多重继承相关的问题。
- 现有的数据类型可被扩展，无需修改该数据类型，即可实现新的接口。
- 协议的方法名位于命名空间当中，所以当多个同伴扩展了同一个现有类型时，不存在名称冲突的风险。

defprotocol 宏的工作方式与 definterface 很像，只不过现在我们可以对现有数据类型进行扩展，以实现我们的新抽象了。

```
(defprotocol name & opts+sigs)
```

让我们把 IOFactory 重新定义为一个协议，而不再是一个接口。

```
(defprotocol IOFactory
  "某种我们可以读出和写入的事物的协议。"
  (make-reader [this] "Creates a BufferedReader.")
  (make-writer [this] "Creates a BufferedWriter."))
```

注意，我们可以为整个协议添加注释，如同可以对它的每个方法都添加注释一样。现在，让我们对 java.io.InputStream 和 java.io.OutputStream 加以扩展，实现我们的 IOFactory 协议。

我们可以使用 extend 函数，把一个现有类型与一个协议关联起来，并提供所需函数的实现，在此上下文中，往往把这些函数称为方法。extend 的参数分别是：打算扩展的类型名称、将要实现的协议名称和一个方法实现的映射表，其中映射表的键是方法名的关键字版本。

```
(extend type & proto+mmaps)
```

下面是 InputStream 的 make-reader 实现，只是用 BufferedReader 把传入的值包装了一下。

src/examples/protocols.clj

```
(extend InputStream
  IOFactory
```

```
  {:make-reader (fn [src]
                  (-> src InputStreamReader. BufferedReader.))
   :make-writer (fn [dst]
                  (throw (IllegalArgumentException.
                           "Can't open as an InputStream.")))})
```

同样，OutputStream 的 make-writer 实现也只是把给定的输入，包装成了一个 BufferedWriter。因为不能向一个 InputStream 写入，或是从一个 OutputStream 中读出，它们各自的 make-writer 和 make-reader 实现，都会抛出 IllegalArgumentExceptions 异常。

```
(extend OutputStream
  IOFactory
  {:make-reader (fn [src]
                  (throw
                    (IllegalArgumentException.
                      "Can't open as an OutputStream.")))
   :make-writer (fn [dst]
                  (-> dst OutputStreamWriter. BufferedWriter.))})
```

我们可以使用 extend-type 宏来扩展 java.io.File 类型，实现我们的 IOFactory 协议，它提供了比 extend 更为清晰的语法。

```
(extend-type type & specs)
```

它接受要扩展的类型名称，和一份或多份规格作为参数，每份规格包括一个协议名称，以及相应的方法实现。

```
(extend-type File
  IOFactory
  (make-reader [src]
    (make-reader (FileInputStream. src)))
  (make-writer [dst]
    (make-writer (FileOutputStream. dst))))
```

请注意，我们创建了一个 InputStream，具体而言，是一个来自于我们文件的 FileInputStream，随后递归地调用了 make-reader。这会被分派给早先定义的那些 InputStreams 实现。我们对 make-writer 方法也使用了相同的递归模式，其余那些类型的方法也一样。

我们还可以使用 extend-protocol 宏，一次就对剩下的所有那些类型全部进行扩展。

```
(extend-protocol protocol & specs)
```

它接受协议名称作为参数，后面紧跟一个或多个类型名称，以及它们各自的方法

实现。

```
(extend-protocol IOFactory
  Socket
  (make-reader [src]
    (make-reader (.getInputStream src)))
  (make-writer [dst]
    (make-writer (.getOutputStream dst)))
  URL
  (make-reader [src]
    (make-reader
      (if (= "file" (.getProtocol src))
        (-> src .getPath FileInputStream.)
        (.openStream src))))
  (make-writer [dst]
    (make-writer
      (if (= "file" (.getProtocol dst))
        (-> dst .getPath FileInputStream.)
        (throw (IllegalArgumentException.
                 "Can't write to non-file URL"))))))
```

现在，让我们把它们全都放到一块儿。

```
(ns examples.io
  (:import (java.io File FileInputStream FileOutputStream
                        InputStream InputStreamReader
                        OutputStream OutputStreamWriter
                        BufferedReader BufferedWriter)
           (java.net Socket URL)))
(defprotocol IOFactory
  "A protocol for things that can be read from and written to."
  (make-reader [this] "Creates a BufferedReader.")
  (make-writer [this] "Creates a BufferedWriter."))

(defn gulp [src]
  (let [sb (StringBuilder.)]
    (with-open [reader (make-reader src)]
      (loop [c (.read reader)]
        (if (neg? c)
          (str sb)
          (do
            (.append sb (char c))
            (recur (.read reader))))))))
(defn expectorate [dst content]
  (with-open [writer (make-writer dst)]
    (.write writer (str content))))
(extend-protocol IOFactory
  InputStream
  (make-reader [src]
```

```
  (-> src InputStreamReader. BufferedReader.))
(make-writer [dst]
  (throw
    (IllegalArgumentException.
      "Can't open as an InputStream.")))
OutputStream
(make-reader [src]
  (throw
    (IllegalArgumentException.
      "Can't open as an OutputStream.")))
(make-writer [dst]
  (-> dst OutputStreamWriter. BufferedWriter.))
File
(make-reader [src]
  (make-reader (FileInputStream. src)))
(make-writer [dst]
  (make-writer (FileOutputStream. dst)))
Socket
(make-reader [src]
  (make-reader (.getInputStream src)))
(make-writer [dst]
  (make-writer (.getOutputStream dst)))
URL
(make-reader [src]
  (make-reader
    (if (= "file" (.getProtocol src))
      (-> src .getPath FileInputStream.)
      (.openStream src))))

(make-writer [dst]
  (make-writer
    (if (= "file" (.getProtocol dst))
      (-> dst .getPath FileInputStream.)
      (throw (IllegalArgumentException.
               "Can't write to non-file URL"))))))
```

6.4　数据类型

我们已经展示了在 Clojure 中，如何使用协议来扩展现有类型，以实现新的抽象，但如果我们想要创建一种新的类型呢？这就到数据类型登场的时候了。

数据类型提供了下面这些东西。

- 一个独一无二的类，无论是命名的还是匿名的。
- 结构，无论是显式的作为字段还是隐式的作为闭包。
- 字段可以有类型提示，并且可以是原始类型。

- 可以有选择的实现协议或是接口中指定的抽象方法。
- 默认的不可变性。
- 与映射表统一（通过记录）。

我们将使用 deftype 宏来定义一个新的数据类型，称为 CryptoVault，会实现两个协议，其中包括 IOFactory。

现在 gulp 和 expectorate 函数已经支持现有的几个 Java 类了，接下来让我们创建同样得到支持的这个新类型：CryptoVault。你会传入下列参数，来创建一个 CryptoVault 的实例：实现了 clojure.java.io.IOFactory 协议（不是我们自己定义的那个）的对象、存储了密钥的路径和一个密码。“咳出”到 CryptoVault 中的内容将会被加密，并写入那个 IOFactory 对象，另外，在“吞”回来的时候，还会对内容进行解密。

我们使用 deftype 来创建这个新的类型。

```
(deftype name [& fields] & opts+specs)
```

它接受类型的名称和字段向量作为参数。数据类型与 Java 类的命名约定相同，即帕斯卡命名法（PascalCase）或驼峰命名法（CamelCase）。

```
user=> (deftype CryptoVault [filename keystore password])
user.CryptoVault
```

一旦定义了这个类型，我们就能像下面这样创建 CryptoVault 实例了。

```
user=> (def vault (->CryptoVault "vault-file" "keystore" "toomanysecrets"))
#'user/vault
```

还可以使用访问 Java 对象字段的前缀点号（prefix-dot）语法，来访问它的字段。

```
user=> (.filename vault)
"vault-file"
user=> (.keystore vault)
"keystore"
user=> (.password vault)
"toomanysecrets"
```

现在，我们已经定义了这个基本的 CryptoVault 类型，接下来让我们添加几个方法，来为其添加一些行为。数据类型只能实现那些在协议或者接口中指定的方法，所以首先我们要创建一个名为 Vault（保险库）的协议。

```
(defprotocol Vault
  (init-vault [vault])
  (vault-output-stream [vault])
```

```
  (vault-input-stream [vault]))
```

这个协议包含了三个函数：init-vault、vault-output-stream 和 vault-input-stream，无论什么类型的保险库，都必须实现它们。

我们可以使用 deftype，内联地定义新类型的方法，只需要像之前那样，传入类型名称和包含字段的向量，随后是协议名称及一个或多个方法体。

src/examples/cryptovault.clj

```
(ns examples.cryptovault
  (:use [examples.io :only [IOFactory make-reader make-writer]])
  (:require [clojure.java.io :as io])
  (:import (java.security KeyStore KeyStore$SecretKeyEntry
                                   KeyStore$PasswordProtection)
           (javax.crypto KeyGenerator Cipher CipherOutputStream
                                      CipherInputStream)
           (java.io FileOutputStream)))
(deftype CryptoVault [filename keystore password]
  Vault
  (init-vault [vault]
    ... 在此定义方法主体 ...)

  (vault-output-stream [vault]
    ... 在此定义方法主体 ...)

  (vault-input-stream [vault]
    ... 在此定义方法主体 ...)

  IOFactory
  (make-reader [vault]
    (make-reader (vault-input-stream vault)))
  (make-writer [vault]
    (make-writer (vault-output-stream vault))))
```

注意，可以内联地定义多个协议的方法，我们同时定义了 Vault 和 IOFactory 协议的方法，尽管 Vault 的那些方法主体都被省略了，但接下来我们就会对它们加以描述。

init-vault 方法会生成一个 AES（Advanced Encryption Standard，高级加密标准）密钥，将其放进一个 java.security.KeyStore 中，并把这个密钥库的数据写入由 CryptoVault 的 keystore 字段指定的文件中，然后对其进行口令保护。

```
(init-vault [vault]
  (let [password (.toCharArray (.password vault))
        key (.generateKey (KeyGenerator/getInstance "AES"))
        keystore (doto (KeyStore/getInstance "JCEKS")
                   (.load nil password)
                   (.setEntry "vault-key"
                              (KeyStore$SecretKeyEntry. key)
```

```
                                (KeyStore$PasswordProtection. password)))]
    (with-open [fos (FileOutputStream. (.keystore vault))]
      (.store keystore fos password))))
```

vault-output-stream 和 vault-input-stream 这两个方法都用到了同一个函数：vault-key，它会读取与该 CryptoVault 关联的密钥库，提取用于加密的 AES 密钥，并对这个保险库的内容进行解密。

```
(defn vault-key [vault]
  (let [password (.toCharArray (.password vault))]
    (with-open [fis (FileInputStream. (.keystore vault))]
      (-> (doto (KeyStore/getInstance "JCEKS")
            (.load fis password))
        (.getKey "vault-key" password)))))
```

vault-output-stream 方法使用 vault-key 来初始化一个 AES 密码对象，根据保险库的文件名创建一个 OutputStream，然后使用那个密码和这个 OutputStream 来创建一个 CipherOutputStream 的实例。

```
(vault-output-stream [vault]
  (let [cipher (doto (Cipher/getInstance "AES")
                 (.init Cipher/ENCRYPT_MODE (vault-key vault)))]
    (CipherOutputStream. (io/output-stream (.filename vault)) cipher)))
```

vault-input-stream 方法的工作方式与 vault-output-stream 类似，只不过它返回的是一个 CipherInputStream。

```
(vault-input-stream [vault]
  (let [cipher (doto (Cipher/getInstance "AES")
                 (.init Cipher/DECRYPT_MODE (vault-key vault)))]
    (CipherInputStream. (io/input-stream (.filename vault)) cipher)))
```

只要提供保存数据的位置、密钥库文件名和保护密钥库的口令，就可以创建出一个 CryptoVault 的实例。

```
user=> (def vault (->CryptoVault "vault-file" "keystore" "toomanysecrets"))
#'user/vault
```

如果密钥库尚未初始化，那就调用 init-vault 方法。

```
user=> (init-vault vault)
nil
```

然后，就可以像使用任何其他的源、目标那样，用 gulp 和 expectorate 来操作 CryptoVault 了。

```
user=> (expectorate vault "This is a test of the CryptoVault")
nil
user=> (gulp vault)
"This is a test of the CryptoVault"
```

我们同样可以扩展 CryptoVault，使其支持 clojure.java.io/IOFactory 协议，这样它就可以用于内建的 spit 和 slurp 函数了。我们自己的 IOFactory 只有两个方法，而这个版本的 IOFactory 则有四个方法。还好，在一个名为 default-streams-impl 的映射表中，有这些方法的默认实现。我们只重写它的 make-input-stream 和 make-output-stream 这两个方法。只要把我们自己的实现放进该映射表，并将这个映射表传给 extend 函数即可。

```
(extend CryptoVault
 clojure.java.io/IOFactory
  (assoc clojure.java.io/default-streams-impl
    :make-input-stream (fn [x opts] (vault-input-stream x))
    :make-output-stream (fn [x opts] (vault-output-stream x))))
```

就是这样，现在我们可以使用 slurp 和 spit 来读写一个 CryptoVault 了。

```
user=> (spit vault "This is a test of the CryptoVault using
spit and slurp")
nil

user=> (slurp vault)
"This is a test of the CryptoVault using spit and slurp"
```

下面让我们把所有这些代码拼在一起，放进一个.clj 文件中。在你的工程目录中建立一个 src/examples/datatypes 子目录，并创建名为 vault.clj 的文件。

src/examples/cryptovault_complete.clj

```
(ns examples.cryptovault-complete
  (:require [clojure.java.io :as io]
            [examples.protocols.io :as proto])
  (:import (java.security KeyStore KeyStore$SecretKeyEntry
                                   KeyStore$PasswordProtection)
           (javax.crypto Cipher KeyGenerator CipherOutputStream
                                CipherInputStream)
           (java.io FileInputStream FileOutputStream)))
(defprotocol Vault
  (init-vault [vault])
  (vault-output-stream [vault])
  (vault-input-stream [vault]))
(defn vault-key [vault]
  (let [password (.toCharArray (.password vault))]
    (with-open [fis (FileInputStream. (.keystore vault))]
      (-> (doto (KeyStore/getInstance "JCEKS")
            (.load fis password))
        (.getKey "vault-key" password)))))
```

```
(deftype CryptoVault [filename keystore password]
  Vault
  (init-vault [vault]
    (let [password (.toCharArray (.password vault))
          key (.generateKey (KeyGenerator/getInstance "AES"))
          keystore (doto (KeyStore/getInstance "JCEKS")
                     (.load nil password)
                     (.setEntry "vault-key"
                                (KeyStore$SecretKeyEntry. key)
                                (KeyStore$PasswordProtection. password)))]
      (with-open [fos (FileOutputStream. (.keystore vault))]
        (.store keystore fos password))))
  (vault-output-stream [vault]
    (let [cipher (doto (Cipher/getInstance "AES")
                   (.init Cipher/ENCRYPT_MODE (vault-key vault)))]
      (CipherOutputStream. (io/output-stream (.filename vault)) cipher)))

  (vault-input-stream [vault]
    (let [cipher (doto (Cipher/getInstance "AES")
                   (.init Cipher/DECRYPT_MODE (vault-key vault)))]
      (CipherInputStream. (io/input-stream (.filename vault)) cipher)))

  proto/IOFactory
  (make-reader [vault]
    (proto/make-reader (vault-input-stream vault)))
  (make-writer [vault]
    (proto/make-writer (vault-output-stream vault))))
(extend CryptoVault
  clojure.java.io/IOFactory
  (assoc io/default-streams-impl
    :make-input-stream (fn [x opts] (vault-input-stream x))
    :make-output-stream (fn [x opts] (vault-output-stream x))))
```

6.5 记录

面向对象编程中的类，往往都可以归入两种不同的类别：一些代表了编程构件，例如字符串、套接字、输入流和输出流；另外一些则代表了应用的领域信息，例如员工和采购单。

不幸的是，使用类来对应用的领域信息建模时，这些信息却被设值函数（setters）和取值函数（getters）。这样特定于类的微型语言给掩盖住了。你无法采用某种通用的方法来进行信息处理，此处还总是伴随着不断扩散的非必要特异性，并且重用性极度贫弱。参阅 Clojure 文档中数据类型的部分[①]以获得更多信息。

① http://clojure.org/datatypes。

出于这个原因，Clojure 总是鼓励使用映射表来对此类信息进行建模，即便是结构化的数据类型也要如此，于是记录（record）便登场了。记录是一种数据类型，就像用 deftype 创建的那些类型一样，它还实现了 PersistentMap，因此，它可以像其他任何映射表（绝大部分）那样来使用。此外，因为记录同样也是真正的类，所以他们可以通过协议来支持基于类型的多态。有了记录，我们就能两全其美了：能够实现协议的映射表。

有什么能比用记录来播放音乐更自然的呢？所以，让我们创建一种代表一个音符的记录，具有音高、八度音阶和持续时间这几个字段；然后我们会使用 JDK（Java Developer's Kit，Java 开发工具包）内建的迷笛合成器（MIDI synthesizer）来播放这些音符序列。

由于记录也是映射表，所以我们能够使用 assoc 和 update-in 函数来更改单个音符的属性，并且我们可以使用 map 和 reduce 来创建或是转换整个音符序列。这使得我们能访问 Clojure 的整个容器 API。

我们将用 defrecord 宏来创建一种音符记录 Note，该宏看起来就像是 deftype。

```
(defrecord name [& fields] & opts+specs)
```

音符记录有三个字段：音高 pitch，八度音阶 octave 和持续时间 duration。

```
(defrecord Note [pitch octave duration])
-> user.Note
```

音高采用如同:C、:C#和:Db 这样的关键字来表示，它们分别代表音符 C、C 升和 D 降。每个音高能在不同的八度播放；例如，中央 C 在第四个八度。持续时间表示音符的长度；全音符用 1 表示，二分音符是 1/2，四分音符是 1/4，十六分音符是 1/16。例如，我们可以用音符记录来表示第四个八度的二分音符。

```
(->Note :D# 4 1/2)
-> #user.Note{:pitch :D#, :octave 4, :duration 1/2}
```

我们可以像对待其他数据类型那样对待记录，使用点语法来访问它们的字段。

```
(.pitch (->Note :D# 4 1/2))
-> :D#
```

此外记录也像映射表一样。

```
(map? (->Note :D# 4 1/2))
-> true
```

所以我们可以使用关键字来访问它们的字段。

```
(:pitch (->Note :D# 4 1/2))
-> :D#
```

我们也可以使用 assoc 和 update-in 函数来创建修改过的记录。

```
(assoc (->Note :D# 4 1/2) :pitch :Db :duration 1/4)
-> #user.Note{:pitch :Db, :octave 4, :duration 1/4}

(update-in (->Note :D# 4 1/2) [:octave] inc)
-> #user.Note{:pitch :D#, :octave 5, :duration 1/2}
```

记录是开放的，所以我们可以把额外的字段关联进一条记录中。

```
(assoc (->Note :D# 4 1/2) :velocity 100)
-> #user.Note{:pitch :D#, :octave 4, :duration 1/2, :velocity 100}
```

我们使用可选的:velocity 字段，来表示音符播放的力度。

使用记录时，assoc 和 update-in 都会返回一条新的记录，但是分离函数 dissoc 则有所不同；如果被分离的是前例中力度那样的可选字段，那么它会返回一条新的记录，但如果字段是通过 defrecord 指定的，就像音高、八度音阶或者持续时间那样，那么它就会返回一个普通的映射表。

换句话说，如果你从一条给定类型的记录中，移除了一个必需字段，它就不再是该类型的一条记录，而是会简单的变成一个映射表。

```
(dissoc (->Note :D# 4 1/2) :octave)
-> {:pitch :D#, :duration 1/2}
```

注意在这种情况下，dissoc 返回的是映射表，而不是记录。记录与映射表之间的区别之一是，与映射表不同，记录不是关键字的函数。

```
((->Note. :D# 4 1/2) :pitch)
-> user.Note cannot be cast to clojure.lang.IFn
```

之所以抛出了 ClassCastException 异常，是因为记录不像映射表那样，实现了 IFn 接口。这是故意形成的一种风格差异，能使代码更加可读。

当访问一个容器时，你应该首先放置这个容器。当访问在概念上被当作一条数据记录的映射表时，你首先应该放置关键字，哪怕这条记录实际上被实现为普通的映射表。现在我们有自己的基本音符记录了，让我们再添加一些方法，这样我们就能用 JDK 内建的迷笛合成器来播放它们了。我们将从创建一个包含了三个方法的 MidiNote 协议开始。

src/examples/protocols.clj

```
(defprotocol MidiNote
  (to-msec [this tempo])
  (key-number [this])
  (play [this tempo midi-channel]))
```

为了能用迷笛合成器播放我们的音符，我们需要把它的音高和八度音阶转化为一个迷笛键编号（MIDI key number），并将它的持续时间转化为毫秒。这里我们定义了 to-msec、key-number 和 play，将用于创建我们自己的 MidiNote。

- to-msec 返回音符的持续时间（以毫秒为单位）。
- key-number 返回这个音符对应的迷笛键编号。
- play 用给定的节奏在给定的通道播放这个音符。

现在，让我们对音符记录 Note 进行扩展，让它支持 MidiNote 协议。

```
(import 'javax.sound.midi.MidiSystem)
(extend-type Note
  MidiNote
  (to-msec [this tempo]
    (let [duration-to-bpm {1 240, 1/2 120, 1/4 60, 1/8 30, 1/16 15}]
      (* 1000 (/ (duration-to-bpm (:duration this))
                 tempo))))
```

to-msec 函数基于给定的节奏（用每分钟的节拍数表示，简称 bpm），把全音符、二分音符、四分音符等，转化为毫秒表示的音符持续时间。

```
(key-number [this]
  (let [scale {:C 0,  :C# 1, :Db 1,  :D 2,
               :D# 3, :Eb 3, :E 4,   :F 5,
               :F# 6, :Gb 6, :G 7,   :G# 8,
               :Ab 8, :A 9,  :A# 10, :Bb 10,
               :B 11}]
    (+ (* 12 (inc (:octave this)))
      (scale (:pitch this)))))
```

key-number 函数把用于表达音高的关键字，映射为范围从 0 到 11 的一个数字，然后用这个数字连同给定的八度音阶，找出对应的迷笛键编号。①

```
(play [this tempo midi-channel]
  (let [velocity (or (:velocity this) 64)]
    (.noteOn midi-channel (key-number this) velocity)
    (Thread/sleep (to-msec this tempo)))))
```

① 注意，有不止一个音高会被映射为 1、3、6、8 和 10。

最后，播放方法play需要一个音符、节奏和一条迷笛通道；接下来，发送一条noteOn消息给这条通道；然后在音符持续的这段时间休眠。即使当前线程处于休眠，该音符也仍然会继续播放，只有当下一个音符被送往通道时才会停止。

现在我们需要一个函数来设置迷笛合成器，并播放一系列的音符。

```
(defn perform [notes & {:keys [tempo] :or {tempo 120}}]
  (with-open [synth (doto (MidiSystem/getSynthesizer) .open)]
    (let [channel (aget (.getChannels synth) 0)]
      (doseq [note notes]
        (play note tempo channel)))))
```

演奏函数 perform 需要一个包含音符的序列和一个可选的节奏值，打开迷笛合成器，从中获得通道，然后调用每个音符的播放方法 play。

现在所有工作已经准备就绪，让我们用音符记录 Note 的序列来创作一曲吧。

```
(def close-encounters [(->Note :D 3 1/2)
                       (->Note :E 3 1/2)
                       (->Note :C 3 1/2)
                       (->Note :C 2 1/2)
                       (->Note :G 2 1/2)])
-> #'user/close-encounters
```

本例中，我们的“乐曲”包含了 5 个音符，它们是在影片《第三类亲密接触》中用来向外星飞船问好的。要播放它们，只要把这个序列传给演奏函数 perform 即可。

```
(perform close-encounters)
 -> nil
```

我们也可以用 for 来动态地生成音符序列。

```
(def jaws (for [duration [1/2 1/2 1/4 1/4 1/8 1/8 1/8 1/8]
                pitch [:E :F]]
            (Note. pitch 2 duration)))
-> #'user/jaws

(perform jaws)
-> nil
```

这次得到的是《大白鲨》中的鲨鱼主题——一个由 E 和 F 音符交替组成的序列，这些音符从二分音符，到四分音符，再到八分音符，逐步加快。

由于音符是记录，而记录又是类映射表的，所以我们能用 Clojure 中任何可用于映射表的函数来操作它们。例如，我们可以贯穿《接触》的音符序列，用 update-in 函数对每个音符进行映射，以升高或降低它们的八度音阶。

```
 (perform (map #(update-in % [:octave] inc) close-encounters))
-> nil

(perform (map #(update-in % [:octave] dec) close-encounters))
-> nil
```

或者我们可以创建一个音符序列，这些音符的可选字段:velocity 逐步增大。

```
 (perform (for [velocity [64 80 90 100 110 120]]
           (assoc (Note. :D 3 1/2) :velocity velocity)))
-> nil
```

得到的是一个越来越有力的 D 音序列。序列操作是 Clojure 特有的一个优势，所以用编程方式来创建和操作由音符记录组成的序列，具有无限的可能性。

让我们把迷笛音符协议 MidiNote、音符记录 Note 和播放函数 perform 放到同一个 Clojure 源文件中，并将其命名为 src/examples/midi.clj，这样我们以后还可以再次使用它们。

src/examples/midi.clj

```
(ns examples.datatypes.midi
  (:import [javax.sound.midi MidiSystem]))
(defprotocol MidiNote
  (to-msec [this tempo])
  (key-number [this])
  (play [this tempo midi-channel]))

(defn perform [notes & {:keys [tempo] :or {tempo 88}}]
  (with-open [synth (doto (MidiSystem/getSynthesizer).open)]
    (let [channel (aget (.getChannels synth) 0)]
      (doseq [note notes]
        (play note tempo channel)))))

(defrecord Note [pitch octave duration]
  MidiNote
  (to-msec [this tempo]
    (let [duration-to-bpm {1 240, 1/2 120, 1/4 60, 1/8 30, 1/16 15}]
      (* 1000 (/ (duration-to-bpm (:duration this))
                 tempo))))
  (key-number [this]
    (let [scale {:C 0, :C# 1, :Db 1, :D 2,
                 :D# 3, :Eb 3, :E 4, :F 5,
                 :F# 6, :Gb 6, :G 7, :G# 8,
                 :Ab 8, :A 9, :A# 10, :Bb 10,
                 :B 11}]
      (+ (* 12 (inc (:octave this)))
        (scale (:pitch this)))))
  (play [this tempo midi-channel]
    (let [velocity (or (:velocity this) 64)]
```

```
        (.noteOn midi-channel (key-number this) velocity)
        (Thread/sleep (to-msec this tempo)))))
```

6.6 具体化

通过 reify（具体化）宏，你可以创建一个匿名实例，这个实例的数据类型实现了一种协议或是一个接口。注意你是通过闭包来访问数据，而不是通过声明。这是因为它根本就没有成员声明。

```
(reify& opts+specs)
```

reify，就像 deftype 和 defrecord 一样，需要一个或多个协议、接口的名称，以及一系列的方法主体作为参数。但与 deftype 和 defrecord 不同，它不需要名称以及字段组成的向量；通过 reify 产生的数据类型实例，没有显式的字段，而是依靠闭包。

让我们创作一些约翰·凯奇风格（John Cage–style）[①]的即兴音乐[②]吧，或者，更好的是，创建一个即兴音乐的生成器。我们将会使用 reify 来创建一个迷笛音符 MidiNote 的实例，每次该实例的播放方法 play 被调用时，都会播放出一个不同的随机音符。

src/examples/generator.clj

```
(import '[examples.datatypes.midi MidiNote])
(let [min-duration 250
      min-velocity 64
      rand-note (reify
    MidiNote
    (to-msec [this tempo] (+ (rand-int 1000) min-duration))
    (key-number [this] (rand-int 100))
    (play [this tempo midi-channel]
      (let [velocity (+ (rand-int 100) min-velocity)]
        (.noteOn midi-channel (key-number this) velocity)
        (Thread/sleep (to-msec this tempo)))))]
  (perform (repeat 15 rand-note)))
```

我们要做的第一件事情，是从 examples.midi 命名空间中导入（是 import，而不是 use 或者 require）我们的迷笛音符协议 MidiNote。接下来我们绑定了两个值，最短持续时间 min-duration 和最小力度 minvelocity，我们会把它们用在迷笛音符 MidiNote 的方法实现中。然后，我们用 reify 创建了匿名类型的一个实例，这种类型实现了迷笛音符协议 MidiNote，每次它的 play 被调用时，它都会选择一个随机的音符、持续时间和

① http://en.wikipedia.org/wiki/John_Cage。
② http://en.wikipedia.org/wiki/Aleatoric_music。

力度。最后，我们使用 repeat 函数创建了一个包含 15 个音符的序列（序列由 rand-note 的单个实例组成），并演奏这个序列。瞧，这下子你就有了一个虚拟的约翰·凯奇啦！

6.7　小结

从编程中的通用抽象，到 Clojure 提供的一些（但不是全部）具体抽象机制，本章我们讨论了许多的主题。我们探讨了在 Clojure 中使用协议来创建具体的抽象，以及这个过程中一些有趣的东西！

但还有更多。Clojure 实现的宏，易于学习并易于正确地在常见任务中运用，此外，对于那些与宏相关的更为困难的任务来说，它也足够强大。在下一章中，你将看到 Clojure 是如何把宏带到主流程序设计中来的。

第 7 章

宏

宏赋予 Clojure 以强大的力量。用大多数的编程技术，你都是在那门语言的“内部”构建功能。当你编写宏的时候，更准确的说法是，你正在为这门语言“添加特性”。这是一种很强大，同时又很危险的能力，所以你应该遵循第 7.1 节“何时使用宏”中的规则，至少也要等你拥有足够多的经验后，才去决定何时可以采取一些通融。第 7.2 节“编写流程控制宏”是体验宏的一个开始，它会引导你为 Clojure 添加一项新的特性。

虽然很强大，但宏并不总是很简单。Clojure 致力于让宏尽可能的简单，包含了诸多便利，帮助解决编写宏时产生的许多常见问题。第 7.3 节“让宏更加简单”中描述了这些问题，并展示了 Clojure 是如何缓解它们的。

宏和其他编程风格是如此的不同，以至于你很难确信该什么时候使用它们。没有比来自社区的经验分享更好的指南了，第 7.4 节“宏的分类”，会基于 Clojure 自身，以及分享库中对宏的使用，介绍了一种宏在 Clojure 中的分类法。

7.1 何时使用宏

宏俱乐部有两条规则，外加一个例外。

宏俱乐部的第一条规则是，不要编写宏。宏相当复杂，它们需要你仔细的思考宏展开期和编译期之间的相互影响。如果你能把它写成一个函数，那么使用宏之前请务必三思。

宏俱乐部的第二条规则是，如果宏是封装一种模式的唯一方法，那就编写它们吧。所有的语言都提供了一些方法来封装模式，但没有宏的话，这些机制都是不完整的。在大多数语言中，每当你说：“要是这门语言有 X 特性的话，我的生活就会轻松得多”时，你就会感受到这种不完整。在 Clojure 中，你只需要用宏去实现这个特性 X 就可以。

上述规则的一个例外是，你可以编写这样的一种宏，它们相较其等价函数而言，能让调用者的生活变得更加轻松。但为了理解这个例外，你需要一些编写宏的实践，并将它们与函数进行比较。所以，让我们从一个例子开始吧。

7.2　编写流程控制宏

作为语言的一部分，Clojure 提供了特殊形式 if。

```
(if (= 1 1) (println "yep, math still works today"))
| yep, math still works today
```

某些语言还提供了一个 unless，与 if 正好相反（几乎是）。unless 执行一个测试，然后只有测试结果逻辑为假时，才执行它的主体部分。

Clojure 没有 unless，但它有一个名为 when-not 的等价宏。为了能从一个简单的例子开始，让我们先假装 when-not 不存在，并创建一个 unless 的实现。为了遵循宏俱乐部的规则，首先试着将 unless 写成一个函数。

src/examples/macros.clj

```
; 这注定要失败...
(defn unless [expr form]
  (if expr nil form))
```

检验一下，看看当 unless 的测试表达式 expr 为 false 时，它能否正确求值。

```
(unless false (println "this should print"))
| this should print
```

迄今为止一切正常。那么我们再勤快一点儿，也测试一下为 true 的情况。

```
(unless true (println "this should not print"))
| this should not print
```

显然是出问题了。这里的问题是，在参数被传入函数之前，Clojure 便对所有这些参数进行了求值，所以，甚至在 unless 还没有见到之前，println 就已经被调用了。事

实上，在进入 unless 函数之前，前面对 unless 的两次调用，都过早的调用了 println。为了说明这一点，把一句 println 添进 unless 中。

```
(defn unless [expr form]
  (println "About to test...")
  (if expr nil form))
```

现在你能清楚的看到，函数的参数们总是在传递给 unless 之前就被求值了。

```
(unless false (println "this should print"))
| this should print
| About to test...

(unless true (println "this should not print"))
| this should not print
| About to test...
```

宏解决了这个问题，因为它们不会对参数立即求值。相反，一个宏的参数何时（以及要不要）求值由你决定。

当 Clojure 遇到一个宏，会分两步对其进行处理。首先，它展开（执行）这个宏，并将展开结果替换回程序当中。这就是所谓的“宏展开期”。然后，它继续正常的“编译时间”。

为了实现 unless，你需要编写一些 Clojure 代码，在宏展开期完成下列转化。

```
(unless expr form) -> (if expr nil form)
```

然后，你还需要使用 defmacro 来告诉 Clojure 你的代码是宏，这看起来很像 defn。

```
(defmacro name doc-string? attr-map? [params*] body)
```

因为 Clojure 代码同时也是 Clojure 数据，所以你实际上已经拥有了编写 unless 所需的全部工具。下面编写的这个 unless，用列表来构造 if 表达式。

```
(defmacro unless [expr form]
  (list‘if expr nil form))
```

unless 的主体部分是在宏展开期执行的，产生出一个 if 形式用于编译。如果你在 REPL 中输入这个表达式。

```
(unless false (println "this should print"))
```

接下来 Clojure 会（你是看不到的）把这个 unless 形式展开。

```
(if false nil (println "this should print"))
```

然后，Clojure 编译并执行这个已展开的 if 形式。下面用 true 和 false 来验证一下这个 unless 工作是否正常。

```
(unless false (println "this should print"))
| this should print
-> nil

(unless true (println "this should not print"))
-> nil
```

祝贺你，这是你编写的第一个宏。unless 看起来可能很简单，但不妨想想看：你刚才所做的，对于大多数语言来说根本就不可能。在没有宏的语言中，特殊形式会成为障碍。

7.2.1　特殊形式、设计模式和宏

Clojure 没有为代码提供什么特殊的语法。代码都是由数据结构组成的。于普通函数如此，于特殊形式和宏亦是如此。

考虑一门拥有更多语法变化的语言，就说 Java 吧。[①] 编写 Java 代码的机制中，最灵活的是实例方法。设想你正在编写一个 Java 程序。如果你在一些实例方法中发现了某种反复出现的模式，那么把这种模式封装起来，就能让你在整个 Java 语言中随意使用了。

目前一切良好。但是 Java 也有大量的特殊形式（尽管他们通常不叫这个名字）。与 Clojure 的特殊形式就是 Clojure 数据不同，每种 Java 的特殊形式都有着它们自己的语法。举个例子，if 在 Java 中是一种特殊形式。如果你发现了一种反复出现的模式涉及到 if，想要对其进行封装，可惜你没有任何办法能够做到。你不可能创建一个 unless，所以你只能使用一种 if 的惯用法来模拟 unless。

```
if (!something) ...
```

这看起来似乎是一个相对次要的问题。Java 程序员当然能学会在脑海中将 if (!foo) 翻译为 unless (foo)。但问题是不仅仅只有 if：这门语言中每种不同的语法形式，都抑制了你把涉及该形式的那些模式给封装起来的能力。

另外一个例子，Java 的 new 也是一种特殊形式。多态性对于 new 无效，所以你必

① 我们并非特别针对 Java，并试图去贬低它；只不过，讨论的是一门具体的语言会更容易很多，而 Java 又是一门众所周知的语言。

须去模拟多态，例如用类方法这样的惯用法。

```
Widget w = WidgetFactory.makeWidget(...)
```

这个惯用法就有些笨重了。它引入了一整个全新的类，WidgetFactory。但在问题域中，这个类没有任何意义，它的存在只是为了解决构造器这个特殊形式的局限性而已。与 unless 惯用法不同，“多态实例化”这种惯用法实在太过于复杂，以至于实现的解决方案不止一种。因此，这种惯用法更为贴切的称呼是“设计模式”。

维基百科是这么定义设计模式的：“它是对软件设计中普遍存在（反复出现）的各种问题，所提出的解决方案。”它接下来还继续表示：“设计模式并非某种最终设计，不可以被直接地（特别强调）转换为代码。”

这里就是宏大展拳脚的地方了。宏提供了一个间接层，这样你就能自动化任何反复出现的模式的公共部分了。宏，还有代码即数据，两者的结合让你可以对你的语言重新进行设计，自动地封装模式。

当然，这种争论并不完全是在同一个方向。很多人可能会争辩说，拥有丰富的特殊语法形式，能让一门编程语言更容易学习或是阅读。对此我们并不赞同，即便我们赞同，我们也更愿意舍弃花样繁多的语法，换回强大的宏系统。一旦你习惯了代码即数据，这种能使设计模式自动化的能力，就会获得巨大的回报。

7.2.2 宏展开

当你创建 unless 宏时，你引述（quote）了符号 if。

```
(defmacro unless [expr form]
  (list 'if expr nil form))
```

但你并没有引述其他任何符号。想要理解为什么会这样，需要你仔细地思考在宏展开期都发生了些什么。

- 通过引述[①]if，你阻止了 Clojure 在宏展开期间，直接对 if 进行求值。相反，展开 unless 时，会剥去引述，留下符号 if 准备接受编译。
- 因为 expr 和 form 是宏的参数，所以不必引述它们。在宏展开期，Clojure 会直接把它们替换掉，而不是对它们求值。
- 你无需引述 nil，因为对 nil 求值得到的仍然是它自身。

① http://en.wikipedia.org/wiki/Design_pattern_(computer_science)。

思考什么东西需要引述这个问题，很快就会变得复杂起来。幸运的是，你不必为了这件事情而挖空心思。Clojure 包含了几个诊断函数，这样你就能在 REPL 中测试宏展开了。

macroexpand-1 函数向你展示在宏展开期间都会发生什么。

```
(macroexpand-1 form)
```

下面用 macroexpand-1 来证明 unless 展开后，是一个合理的 if 表达式。

```
(macroexpand-1 '(unless false (println "this should print")))
-> (if false nil (println "this should print"))
```

宏是复杂的野兽，虽然可以用 macroexpand-1 来测试它们，但我们不能夸大其价值。让我们返回去，尝试一些 unless 的错误版本。这里就是一个错误的引述了 expr 的版本。

```
(defmacro bad-unless [expr form]
  (list 'if 'expr nil form))
```

当展开 bad-unless 时，你会看到它生成了符号 expr，而非真正的测试表达式。

```
(macroexpand-1 '(bad-unless false (println "this should print")))
-> (if expr nil (println "this should print"))
```

如果你尝试使用这个 bad-unless 宏，Clojure 会抱怨说，它无法解析符号 expr。

```
(bad-unless false (println "this should print"))
-> java.lang.Exception: Unable to resolve symbol: expr in this context
```

有时候，宏会展开为别的宏。发生这种情况时，Clojure 会继续展开所有的宏，直至仅剩下普通的代码。例如，..宏会递归的展开。生成一个句点操作符调用，再包装进另外一个..中，直到处理了所有的剩余参数。你可以用下面的宏展开来对其进行观察。

```
(macroexpand-1 '(.. arm getHand getFinger))
-> (clojure.core/.. (. arm getHand) getFinger)
```

如果你想看看一直展开下去是什么样子。这时可以使用 macroexpand。

```
(macroexpand form)
```

如果你对一次..调用使用 macroexpand，它会递归的展开，直至仅剩余句点操作符。

```
(macroexpand '(.. arm getHand getFinger))
-> (. (. arm getHand) getFinger)
```

即便 arm、getHand 和 getFinger 并不存在，这也不是一个问题。这会儿你只是展开，并不打算编译和执行它们。

另一个递归的宏是 and。倘若在调用 and 时你传入了两个以上的参数，它会展开为包含另外一个 and 的调用，只不过少了一个参数。

```
(macroexpand '(and 1 2 3))
-> (let* [and__3585__auto__ 1]
     (if and__3585__auto__ (clojure.core/and 2 3)
       and__3585__auto__))
```

这一次，macroexpand 没有完全展开。macroexpand 仅对你传给它的顶层形式有效。由于在展开原 and 期间创建的那个新 and，是嵌套在形式内部的，macroexpand 就不会再对其进行展开了。

7.2.3 when 与 when-not

你的 unless 宏可以进行些许改进，使其能支持执行多个形式，以避免出现下面这个错误。

```
(unless false (println "this") (println "and also this"))
-> java.lang.IllegalArgumentException: \
Wrong number of args passed to: macros$unless
```

不妨思考一下如何编写这个改良的 unless。你可能需要捕获一个可变的参数列表，并在列表前面放置一个 do，以便每个形式都能得以执行。Clojure 借助其 when 和 when-not 宏，正确的提供了这种行为方式。

```
(when test & body)
(when-not test & body)
```

when-not 就是那个你正在寻找的改良版 unless。

```
(when-not false (println "this") (println "and also this"))
| this
| and also this
-> nil
```

鉴于你已经练习过编写 unless，现在阅读 when-not 的源码应该不会有任何问题。

```
; 来自于 Clojure 的 core.clj
(defmacro when-not [test & body]
  (list 'if test nil (cons 'do body)))
```

当然，你还可以使用 macroexpand-1 来查看 when-not 是如何工作的。

```
(macroexpand-1 '(when-not false (print "1") (print "2")))
-> (if false nil (do (print "1") (print "2")))
```

when 与 when-not 相对，仅当其测试表达式 test 为真时，才执行传给它的那些形式。请注意，when 与 if 有两点不同。

- if 可以有一个 else 子句，但 when 没有。这反映了英语的用法，因为没人会用英语说：“when...else.”
- 正因为 when 没有使用它的第二个参数作为 else 字句，所以它可以接受可变的参数列表，并将所有这些参数放进一个 do 中全部执行。

你并不真的需要一个 unless 宏。用 Clojure 提供的 when-not 就好。在你开始编写自己的宏之前，请务必检查一下，看看别人是否已经写了你需要的那个宏。

7.3　让宏更加简单

unless 宏是一个极其简单的例子，但是大多数的宏都会更为复杂。在本节中，我们会构建一组越来越复杂的宏，并随之介绍 Clojure 的特性。表 7-1 中总结了本节中介绍的特性，供你参考之用。

表 7-1　Clojure 的特性

形式	描述
foo#	自动符号生成（Auto-gensym）：在一个被语法引述的片段内部，创建前缀为 foo 的唯一名称
(gensym prefix?)	创建一个唯一的名称，可以有一个可选的前缀
(macroexpand form)	用 macroexpand-1 反复的展开形式，直至得到的形式不再是一个宏为止
(macroexpand-1 form)	显示 Clojure 将如何展开形式
(list-frag? ～@form list-frag?)	解引述拼接：用在语法引述内部，把一个解引述的列表拼接到模板中
`form	语法引述：对形式进行引述，但允许在内部进行解引述，所以形式扮演了模板的角色。形式内部的符号会被解析，这有助于防止意外的符号捕捉
～form	解引述：用于语法引述内部，被替换为解引述后的值

首先，让我们构建一个宏的副本。我们将把它命名为 chain，因为它会链式的调用一系列方法。这里有一些对 chain 的简单展开如表 7-2 所示。

表 7-2 chain 的展开

宏调用	展开
(chain arm getHand)	(. arm getHand)
(chain arm getHand getFinger)	(. (. arm getHand) getFinger)

从实现仅调用一个方法这种最简单的情况开始。这个宏只需要造出一个简单的列表即可。

src/examples/macros/chain_1.clj

```
; chain，重新实现 Clojure 的..宏
(defmacro chain [x form]
  (list '. x form))
```

chain 需要支持任意数量的参数，所以其余的实现应该定义一个递归。列表操作变得更复杂了，因为你需要构建两个列表，并把它们连接到一起。

src/examples/macros/chain_2.clj

```
(defmacro chain
  ([x form] (list '. x form))
  ([x form & more] (concat (list 'chain (list '. x form)) more)))
```

用 macroexpand 测试一下 chain，以确保它生成了正确的展开式。

```
(macroexpand '(chain arm getHand))
-> (. arm getHand)

(macroexpand '(chain arm getHand getFinger))
-> (. (. arm getHand) getFinger)
```

同写出来的代码一样，chain 宏工作良好，但用来处理多个参数的那个表达式，实在是太难阅读了。

```
(concat (list 'chain (list '. x form)) more)))
```

chain 的定义，徘徊于宏代码和要生成的主体代码之间。两者交织在一起，令整段代码都难以阅读。而这还仅仅只是一个形式的主体，不过一行代码而已。随着宏的形式变得愈发复杂，诸如 list 和 concat 这样的装配函数将会迅速掩盖住宏真正的含义。

解决这类问题的一个方案是采用模板语言。如果宏可以由模板创建，你就能采用“填空”的方式来创建它们。这样 chain 的定义可能看起来会像。

```
; 假想的模板语言
(defmacro chain
  ([x form] (. ${x} ${form}))
  ([x form & more] (chain (. ${x} ${form}) ${more})))
```

在这种假想的模板语言中，${}会在宏展开式中被参数给替换掉。

请注意，这种定义是何等之易读，且又何等清晰地向我们展示出，展开式将会是什么样子的。

7.3.1　语法引述、解引述和解引述拼接

Clojure 并未引入某种分裂的语言来支持宏模板。语法引述符，其实就是一个反引号（`），与普通引述的工作方式非常相似。但是在语法引述的列表内部，解引述符（～，波浪线）会再次关停引述。总的效果看起来就像一种这样的模板。

src/examples/macros/chain_3.clj

```
(defmacro chain [x form]
 `(. ~x~form))
```

测试一下这个新版的 chain，看它能否正确的生成单个方法的调用。

```
(macroexpand‘(chain arm getHand))
-> (. arm getHand)
```

遗憾的是，这种语法引述/解引述的方法，不太适用于 chain 那个多参数的变种。

src/examples/macros/chain_4.clj

```
; 不适用
(defmacro chain
  ([x form] `(. ~x ~form))
  ([x form & more] `(chain (. ~x ~form) ~more)))
```

当你展开这个 chain 时，会发现括号并不完全正确。

```
(macroexpand '(chain arm getHand getFinger))
-> (. (. arm getHand) (getFinger))
```

chain 的最后一个参数 more，是一个由多个参数组成的列表。当你把 more 放进那个宏“模板”中时，因为它是一个列表，所以当然会被一对括号括住。但你其实并不想要这些括号，你希望 more 的元素能被拼接进它的父列表中。这种需要是如此常见，以至于存在一个专门针对它的读取器宏：解引述拼接符（～@）。下面来重写 chain，用解引述拼接符来拼接参数 more。

src/examples/macros/chain_5.clj

```
(defmacro chain
  ([x form] `(. ~x ~form))
```

```
  ([x form & more] `(chain (. ~x ~form) ~@more)))
```

现在，展开式应该就是正确的了。

```
(macroexpand '(chain arm getHand getFinger))
-> (. (. arm getHand) getFinger)
```

许多宏都遵循了 chain 的模式。

- 宏的主体由一个语法引述符（`）开始，把整体作为模板。
- 嵌入单个的参数时，带上一个解引述符（~）。
- 对于 more 这样的多参数，用解引述拼接符（~@）进行拼接。

迄今为止我们创建的宏都非常简单，简单到足以躲开 let 或者 binding，无须创建任何绑定。接下来就让我们创建一个会用到它们的宏吧。

7.3.2 在宏内部创建名称

Clojure 有个可以对表达式计时的 time 宏，向控制台输出执行该表达式所用的时间。

```
(time (str "a" "b"))
| "Elapsed time: 0.06 msecs"
-> "ab"
```

让我们构建一个叫做 bench 的 time 变种，旨在收集性能数据。bench 不再输出到控制台，而是返回一个映射表，该映射表同时包含原始表达式的返回值及其所用时间。

开始编写宏最好的方法是，先手工编写期望中的展开式。bench 展开后应该是这个样子的。

```
; (bench (str "a" "b"))
; 应展开为
(let [start (System/nanoTime)
      result (str "a" "b")]
  {:result result :elapsed (- (System/nanoTime) start)})

-> {:elapsed 61000, :result "ab"}
```

let 把 start 绑定至开始时间，然后执行待基准测试的表达式，执行结果绑定至 result。最后，这个形式返回一个包含结果 result 和运行时间（自开始时间 start 以来）的映射表。

有这个展开式在手，你现在就可以回过头去编写能生成这个展开式的宏了。试着

用上一节中介绍的语法引述和解引述技术，编写这个基准测试宏 bench。

src/examples/macros/bench_1.clj

```
; 这是行不通的
(defmacro bench [expr]
  `(let [start (System/nanoTime)
         result ~expr]
     {:result result :elapsed (- (System/nanoTime) start)}))
```

如果你试着调用这个版本的 bench，Clojure 会大声抱怨道。

```
(bench (str "a" "b"))
-> java.lang.Exception: Can't let qualified name: examples.macros/start
```

Clojure 指责你尝试用 let 绑定了一个限定名，而这是不合法的。可以调用 macroexpand-1 来确认这个问题。

```
(macroexpand-1 '(bench (str "a" "b")))
-> (clojure.core/let [examples.macros/start (System/nanoTime)
     examples.macros/result (str "a" "b")]
     {:elapsed (clojure.core/- (System/nanoTime) examples.macros/start)
      :result examples.macros/result})
```

在语法引述的形式中，当遇上一个符号时，它会把该符号解析为一个完全限定名。乍一看，这不免让人有些恼火，因为你是想要创建几个局部名称，具体来说就是 start 和 result。但 Clojure 之所以采取这种方式，实际上是让你免受一种被称为“符号捕获”的险恶的 bug 之害。

如果 bench 的宏展开式允许使用非限定符号 start 和 result，那么随后在某个地方（这两个名称在目标范围内早已被绑定为别的什么东西了）使用 bench 时，会发生什么事呢？宏将会捕获这些名称，并将它们绑定为为不同的值，从而产生离奇的结果。如果 bench 捕获的是它自己的符号，大多数时候都会显得很正常。比如用 1 加上 2 会得到 3。

```
(let [a 1 b 2]
  (bench (+ a b)))

-> {:result 3, :elapsed 39000}
```

直到有一天，你不幸地挑选了一个像 start 这样，与 bench 内部名称有冲突的局部名称。

```
(let [start 1 end 2]
  (bench (+ start end)))

-> {:result 1228277342451783002, :elapsed 39000}
```

bench 捕获了符号 start，并将其绑定为(System/nanoTime)。出乎意料的，1 加上 2 似乎等于 1228277342451783002 了。

Clojure 宏在解析名称方面的强硬态度，有助于你免受“符号捕获”之害，但你的 bench 目前仍然无法工作。这时候，你需要以某种方式来引入局部名称，原则上它们应该是独一无二的，不可能与调用者使用的任何名称产生冲突。

Clojure 提供了一个读取器形式来创建唯一的局部名称。在语法引述的形式中，你可以为非限定名称添加一个井号#（或是使用 auto-gensym），这样 Clojure 就会创建一个自动生成的符号：基于原来的名称，加上一个下划线和一个唯一标识号。在 REPL 中试一下看看。

```
`foo#
foo__1004
```

借助这种由你支配的自动生成符号，很容易就能实现正确的 bench。

```
(defmacro bench [expr]
  `(let [start# (System/nanoTime)
         result# ~expr]
     {:result result# :elapsed (- (System/nanoTime) start#)}))
```

赶快在 REPL 中测试一下。

```
(bench (str "a" "b"))
-> {:elapsed 63000, :result "ab"}
```

Clojure 令唯一名称的生成变得非常容易，但如果你铁了心要做，也仍然可以强制执行符号捕获。本书的示例代码中包含了一个邪恶的 evil-bench，它混合了语法引述、引述和解引述，并引发了符号捕获。除非你已经非常透彻的理解了宏，否则千万不要使用符号捕获。

7.4 宏的分类

现在你已经编写了一些宏，我们可以用更为详细的依据，来重申一下宏俱乐部的规则了。

宏俱乐部的第一条规则是，不要编写宏。宏非常复杂。如果对你而言，Clojure 中已经没有任何宏可以称得上是复杂的话，就来我的公司应聘吧。[①]

宏俱乐部的第二条规则是，如果宏是封装某种模式的唯一途径，就编写之。正如你看到的那样，那些抗拒封装的模式（在该编程语言中，试图对那些模式进行封装是

① http://thinkrelevance.com。

不合法的）往往与特殊形式有关。所以，第二条规则也被称为“特殊形式规则”。

特殊形式有作为程序员的你所不具有的几个特权。

- 特殊形式提供了最基本的流程控制结构，例如 if 和 recur。所有的流程控制宏可能最终都会调用某个特殊形式。
- 特殊形式提供了对 Java 的直接访问。每当你要从 Clojure 中调用 Java，你至少会通过一个诸如.或 new 这样的特殊形式来达到目的。
- 名称是通过特殊形式得以创建及被绑定的，无论是通过 def 定义变量、通过 let 创建词法绑定、还是通过 binding 创建动态绑定。

特殊形式之所以如此强大，是因为它们不是函数。但它们也无法做到一些函数可以做到的事情。你不能 apply 一个特殊形式，把特殊形式保存到变量中，或是在使用序列库时把特殊形式用作过滤器。简而言之，特殊形式不是语言中的一等公民。

特殊形式的特殊性可能是一个重大的问题，并会导致在你的代码中产生重复和难以维护的模式。但是宏却巧妙的解决了这个问题，因为你可以使用宏来生成特殊形式。从实践角度来理解，在宏展开期间，所有的语言特性都是一等特性。

生成特殊形式的宏往往是最难编写的，但同时也是最值得一试的。就像被施加了魔法一般，这样的宏就好像是在往语言中添加新的特性。

宏俱乐部规则的例外是调用方的便利性：你可以编写任何相较其等价函数而言，能让调用者的生活变得更加轻松的宏。因为宏并不对它们的参数求值，因此调用者就可以向宏传入原始代码，而不必将这些代码包装到一个匿名函数中。或者，调用者可以传入未转义的名称，而不是被引述的符号或是字符串。

我们已经考察了 Clojure 和分享库中的宏，它们几乎所有都遵循了宏俱乐部的规则。此外，它们还可以被归入到表 7-3 中的一或多个类别，这个表格展示了 Clojure 宏的分类法。

表 7-3　Clojure 宏的分类法

理由	类别	示例
特殊形式	有条件的求值	when、when-not、and、or、comment
特殊形式	定义变量	defn、defmacro、defmulti、defstruct、declare
特殊形式	Java 互操作	..、doto、import-static
调用方的便利性	延迟求值	lazy-cat、lazy-seq、delay
调用方的便利性	对求值进行包装	with-open、dosync、with-out-str、time、assert
调用方的便利性	避开 lambda 表达式	（与“对求值进行包装”相同）

下面让我们依次查看每个类别。

7.4.1 有条件的求值

因为宏并不会立即对它们的参数求值，所以它们可以用于创建自定义的流程控制结构。在 7.2 节“编写流程控制宏”中，你已经见过了作为代表的 unless。

从事有条件的求值的宏，阅读和编写起来往往都相当简单。他们总是遵循共通的形式：对某个参数（条件）求值；然后，基于求值结果，选择性的对其他参数进行求值。Clojure 的 and 就是一个很好的例子。

```
1 (defmacro and
2   ([] true)
3   ([x] x)
4   ([x & rest]
5     `(let [and# ~x]
6      (if and# (and ~@rest) and#))))
```

and 采用了递归定义。零参和单参的主体设置了基本事实。

- 对于没有参数的情况，返回 true。
- 对于一个参数的情况，返回这个参数。

对于两个及以上的参数，and 使用其第一个参数作为条件，在第 5 行对其进行求值。然后，如果条件为真，通过递归的对 rest 应用 and 宏自身，and 接着对剩余参数进行求值。

为了在遭遇第一个非真值后能进行短路求值，and 必须是一个宏。毫无意外的，and 有一个近亲 or。它们有着一样的签名。

```
(and& exprs)
(or& exprs)
```

不同的是，and 在遇到第一个逻辑假时会停下来，而 or 则是遇到第一个逻辑真时停下。

```
(and 1 0 nil false)
-> nil

(or 1 0 nil false)
-> 1
```

任何时间都会短路求值的冠军是注释宏 comment。

```
(comment& exprs)
```

comment 从不对它的任何参数求值，有些时候会被用在源码文件末尾，演示 API 的用法。

例如，Clojure 的 inspector 库就用下面的这个 comment 作为结束，演示了 inspector 应该如何使用。

```
(comment
(load-file "src/inspector.clj")
(refer 'inspector)
(inspect-tree {:a 1 :b 2 :c [1 2 3 {:d 4 :e 5 :f [6 7 8]}]})
(inspect-table [[1 2 3][4 5 6][7 8 9][10 11 12]])
)
```

注意它缺少了代码缩进。对于大多数 Clojure 代码而言，这不是标准的做法，但对注释而言却很有用，因为其目的是让人对它的主体引起注意。

7.4.2　创建变量

Clojure 通过特殊形式 def 来创建变量。任何其他能创建变量的形式，最终都会调用 def。因此，举例来说，defn、defmacro 和 defmulti 它们自己也都是宏。

为了演示编写创建变量的宏，我们将会查看作为 Clojure 组成部分的两个宏：defstruct 和 declare。

Clojure 提供了一个名为 create-struct 的低级别函数用来创建结构。

```
(create-struct& key-symbols)
```

下面用 create-struct 来创建一个 person 结构。

```
(def person (create-struct :first-name :last-name))
-> #'user/person
```

create-struct 工作正常，但从视觉角度来看，它比较嘈杂。考虑到你常常希望能立即定义一个新结构，你通常会调用 defstruct，它结合了 def 和 create-struct，成为单个操作。

```
(defstruct name & key-symbols)
```

defstruct 是一个很简单的宏，并且已经是 Clojure 的一部分了。

```
(defmacro defstruct
  [name & keys]
  `(def ~name (create-struct ~@keys)))
```

这个宏看起来是如此简单，以至于你可能禁不住想要尝试把它写成一个函数。可惜你做不到，因为 def 是一个特殊形式。你必须在宏展开期生成 def；你无法在运行期“动态”地调用 def。

defstruct 使得一句单行代码更容易阅读了，但有些宏还可以将多行代码浓缩进单个形式当中。不妨考虑一下前置声明的问题。你现在正在编写一个程序，需要对变量 a、b、c 和 d 进行前向引用（forward references）。你可以不加参数的调用 def，来定义没有初始绑定的变量名。

```
(def a)
(def b)
(def c)
(def d)
```

但是，这相当乏味，而且浪费了大量的垂直空间。declare 宏则可以接受一个可变的名称列表，为你定义这些名称。

```
(declare& names)
```

现在，你可以在一个紧凑的形式中定义所有这些名称了。

```
(declare a b c d)
-> #'user/d
```

Clojure 内建了 declare 的实现。

```
(defmacro declare
  [& names] `(do ~@(map #(list 'def %) names)))
```

让我们由内到外的来分析一下 declare。匿名函数#(list‘def %)负责生成单个的 def。让我们在 REPL 中单独测试一下这个形式。

```
(#(list 'def %) 'a)
-> (def a)
```

map 对每个传入的符号调用一次其内部函数。同样，你可以在 REPL 中测试这个形式。

```
(map #(list 'def %) '[a b c d])
-> ((def a) (def b) (def c) (def d))
```

最后，领头的 do 让整个展开式成为了单个合法的 Clojure 形式。

```
`(do~@(map #(list 'def %) '[a b c d]))
-> (do (def a) (def b) (def c) (def d))
```

在上面的这个测试中，替换'[a b c d]是手工的进行的，这与用 macroexpand-1 对整个宏进行测试是等价的。

```
(macroexpand-1 '(declare a b c d))
-> (do (def a) (def b) (def c) (def d))
```

Clojure 中最有趣的部分，就是那些能展开为涉及到 def 的特殊形式的宏。这里我们只探索了其中很少的一部分，但你可以阅读它们当中任何一个的源码。它们中的大多数位于 Clojure 源码分发包的文件 src/clj/clojure/core.clj 中。

7.4.3　Java 互操作

Clojure 程序通过特殊形式.（句点）、new 和 set!来调用 Java。然而，地道的 Clojure 代码往往使用诸如..（链式成员访问）和 doto 这样的宏来简化调用 Java 的形式。

你（或是其他人）可以通过编写宏来扩展 Clojure 调用 Java 的方式。考虑下面的场景。你正在编写用到几个 java.lang.Math 中常量的代码。

```
Math/PI
-> 3.141592653589793
(Math/pow 10 3)
-> 1000.0
```

在较长的代码段中，这个 Math/前缀将很快就成为分散注意力的东西，所以如果你能简单的称它们为 PI 和 pow 将会很棒。Clojure 没有提供任何可以直接做到这一点的方法，但你可以自己动手定义一串变量。

```
(def PI Math/PI)
-> #'user/PI
(defn pow [b e] (Math/pow b e))
-> #'user/pow
```

斯图尔特·塞拉（Stuart Sierra）用 import-static 宏自动化了这个样版。

```
(examples.import-static/import-static class & members)
```

import-static 导入一个 Java 类的静态成员，作为在本地命名空间中的名称出现。下面使用 import-static 来从 Math 中导入那些你想要的成员。

```
(use '[examples.import-static :only (import-static)])
(import-static java.lang.Math PI pow)
-> nil

PI
-> 3.141592653589793
```

```
(pow 10 3)
-> 1000.0
```

7.4.4 延迟求值

Clojure 中的大多数序列都是惰性的。当你正在构建一个惰性序列时，常常会希望对一些形式加以组合。这些形式被延迟到迫不得已的时候，才会进行求值。由于不是立即求值，所以就需要一个宏了。

在第 3.3 节“惰性和无限序列”中，你已经看到了一个这样的宏：lazy-seq。另外的一个例子是 delay。

```
(delay& exprs)
```

当你创建了一个 delay，它会持有 exprs，并且不对它们做任何事，直到被迫不得不做的那一刻。下面试着创建一个 delay，并通过休眠来模拟一个长时间的计算。

```
(def slow-calc (delay (Thread/sleep 5000) "done!"))
-> #'user/slow-calc
```

为了真正地执行 delay，你得用 force 强迫它去做。

```
(force x)
```

来试着强迫你的 slow-calc 执行几次。

```
(force slow-calc)
-> "done!"
(force slow-calc)
-> "done!"
```

当你第一次用 force 强迫执行一个 delay 时，它会执行其表达式，并把结果缓存下来。随后的 force 只是简单的返回那个缓存的值罢了。

实现了惰性和延迟求值的宏，都调用了 clojure.jar 中的 Java 代码。在你自己的代码中，你不应该直接调用此类 Java API。尽管现在是把惰性/延迟求值的宏作为公共 API，并把 Java 类作为实现细节，但这种方式随时都可能会发生改变。

7.4.5 对求值进行包装

许多宏都是对一组形式进行包装，在这些形式得以求值之前和（或）之后，添加

某些特殊的语义。你已经见过了这种宏的几个例子。

- time 启动一个计时器，对形式求值，然后报告执行它们用掉了多长时间。
- let 和 binding 建立绑定，对一些形式求值，然后拆除这些绑定。
- with-open 接受一个打开的文件（或是其他资源），执行一些形式，然后确保这个资源会在一个 finally 块中得以关闭。
- dosync 在一个事务中执行一些形式。

包装宏的另外一个例子是 with-out-str。

```
(with-out-str& exprs)
```

with-out-str 把*out*临时地绑定到一个新的 StringWriter 上，对 exprs 进行求值，然后返回写入到*out*中的字符串。with-out-str 立刻使得用 print 和 println 来构建字符串变得容易得多。

```
(with-out-str (print "hello, ") (print "world"))
-> "hello, world"
```

with-out-str 的实现结构非常简单，可以作为编写类似的宏的模板。

```
1 (defmacro with-out-str
2  [& body]
3 `  (let [s# (new java.io.StringWriter)]
4     (binding [*out* s#]
5          ~@body
6          (str s#))))
```

包装宏通常会接受可变数量的参数（第 2 行），它们是那些即将被求值的形式。然后，这些宏会执行下面的三个步骤。

（1）设置：为求值创建一些特殊的上下文，必要时用 let（第 3 行）和 binding（第 4 行）引入一些绑定。

（2）求值：对形式求值（第 5 行）。由于通常都有可变数量的形式，所以要用“解引述拼接”来插入它们：～@。

（3）拆除：将执行上下文恢复正常，并视情况返回一个值（第 6 行）。

当编写一个包装宏时，你总要问问自己是否需要一个 finally 块来正确的实现拆除步骤。对于 with-out-str 而言，答案是不，因为无论是 let 还是 binding 都会照料好它们自己的清理工作。然而，如果你还通过 Java API 设置了某种全局或是线程局部的状态，那么你就需要一个 finally 块来重置这个状态。

这次对可变状态的讨论导致了另外一种意见。任何代码，当在包装宏内部执行时，如果行为会发生改变的话，显然不会是一个纯函数。print 和 println 根据*out*的值，行为会有所不同，所以它们不是纯函数。诸如 with-out-str 这样的宏设置了一个绑定，就意味在某个方面改变了非纯函数的行为。

不是所有的包装宏都会改变它们包装的函数的行为。你已经见过了 time，它只是为一个函数的执行计时。另外一个例子是 assert。

```
(assert expr)
```

assert 会测试一个表达式，如果表达式不是逻辑为真，它就抛出一个异常。

```
(assert (= 1 1))
-> nil

(assert (= 1 2))
-> java.lang.Exception: Assert failed: (= 1 2)
```

像 assert 和 time 这样的宏，违背了宏俱乐部的第一条规则，但这是为了避免不必要的 lambda 表达式。

7.4.6 避免 Lambda 表达式

由于历史的原因，匿名函数通常被称为 lambda 表达式。有的时候宏可以被一个函数调用所取代。这个函数调用把参数包在了一个 lambda 表达式当中。例如，第 7.3.1 节“语法引述、解引述和解引述拼接”中的那个 bench，不一定非得是一个宏。你也可以把它写作一个函数。

```
(defn bench-fn [f]
  (let [start (System/nanoTime)
        result (f)]
    {:result result :elapsed (- (System/nanoTime) start)}))
```

但是，如果你想要调用 bench-fn，你就必须传给它一个函数，该函数包装了你想要执行的形式。下面的代码显示了这当中的区别。

```
; 宏
(bench (+ 1 2))
-> {:elapsed 44000, :result 3}

; 函数
(bench-fn (fn [] (+ 1 2)))
-> {:elapsed 53000, :result 3}
```

对于像 bench 这样的任务，宏和匿名函数非常相近，可以相互替代。两者都防止了立即执行一个形式。然而，匿名函数的方式需要调用方做更多的工作。所以，为此打破了第一条规则，编写一个宏而不是一个函数，是没有任何问题的。

另外一个更倾向于让 bench 成为一个宏的原因是，bench-fn 并非一个完美无缺的替代品，它在运行期增加了一次调用匿名函数的开销。由于 bench 的目的是对事物计时，所以你应该避免产生这种开销。

7.5　小结

Clojure 的宏能让你自动化代码中的模式。因为它们能在宏展开期间对源码进行变换，所以你可以在你的代码中，用宏来抽象出任何一种模式。你不会受限于 Clojure。借助宏，你可以对 Clojure 进行扩展，让它深入到你的问题域当中。

第 8 章

多重方法

Clojure 的多重方法提供了一种灵活的方式，将一个函数和一组输入关联起来。这与 Java 的多态机制非常类似，但更为通用。当你调用一个 Java 方法时，Java 通过检查对象的类型，来选择执行其特定的实现。而当你调用 Clojure 的多重方法时，Clojure 则是通过检查一个函数的返回值，来选定要执行多重方法的哪个特定实现。注意，那是一个由你任选的函数，Clojure 会把传递给多重方法的所有参数，都传给这个函数。

本章一开始，在没有多重方法的情况下，你会发现自己对它们有多么渴望。然后，你会构建一系列越来越复杂的多重方法，刚开始是用多重方法来模拟多态机制，随后会使用多重方法来实现各种不同的特设分类法（ad hoc taxonomies）。

多重方法在 Clojure 中的使用频度，往往会比多态在面向对象语言中的使用频度要低得多。但是，使用它们的地方，往往都是代码中的关键功能所在。第 8.5 节“何时应该使用多重方法”探讨了在一些开源的 Clojure 项目中是如何使用多重方法的，并提供了一些指引，以便你在自己的项目中能更好地运用它们。

如果你是顺序阅读本书的，那么一旦你读完了本章，恭喜你，所有的 Clojure 关键特性你都体验了一遍。

8.1 没有多重方法的日子

领会多重方法最好的办法就是，花上几分钟时间，体验一下缺少它们的生活。让我们开始吧。Clojure 可以用 print/println 打印任何东西。不过，这会儿先假装这些函数

不存在，你需要自己构建一种通用的打印机制。下面马上开始创建一个 my-print 函数，它能向标准输出流*out*打印一个字符串。

src/examples/life_without_multi.clj

```
(defn my-print [ob]
  (.write *out* ob))
```

接下来，创建一个 my-println，它只是简单地调用了 my-print，并添加一个换行符。

src/examples/life_without_multi.clj

```
(defn my-println [ob]
  (my-print ob)
  (.write *out* "\n"))
```

在 REPL 中测试时，换行能让 my-println 的输出更容易阅读。在本节的剩余部分，你会对 my-print 进行修改，并且通过调用 my-println 来测试它们。下面让我们用字符串测试一下 my-println。

```
(my-println "hello")
| hello
-> nil
```

相当不错，但是 my-println 对于诸如 nil 这样的非字符串，就不太给力了。

```
(my-println nil)
-> java.lang.NullPointerException
```

不过，这算不上是什么大问题。只要使用 cond 为 nil 增加一个特殊情况处理即可。

src/examples/life_without_multi.clj

```
(defn my-print [ob]
  (cond
    (nil? ob) (.write *out* "nil")
    (string? ob) (.write *out* ob)))
```

借助这个恰到好处的条件语句，你这回可以很顺利地打印 nil 了。

```
(my-println nil)
| nil
-> nil
```

当然，还有大量的类型是这个 my-println 无法处理的。你如果试图去打印一个向量，cond 的所有条件都无法匹配，于是程序就打印不出任何东西。

```
(my-println [1 2 3])
-> nil
```

你知道现在该怎么做。只要为 cond 添加另外一个针对向量的条件子句即可。实现这个会稍稍有些复杂，所以你可能会希望把实际的打印操作，分离为一个辅助函数，比如 my-print-vector。

src/examples/life_without_multi.clj

```
(require '[clojure.string :as str])
(defn my-print-vector [ob]
  (.write *out*"[")
  (.write *out* (str/join " " ob))
  (.write *out* ")"))

(defn my-print [ob]
  (cond
    (vector? ob) (my-print-vector ob)
    (nil? ob) (.write *out* "nil")
    (string? ob) (.write *out* ob)))
```

确认一下，看看现在你能不能打印向量了。

```
(my-println [1 2 3])
| [1 2 3]
-> nil
```

my-println 现在支持 3 种类型：字符串、向量和 nil。并且你还为新类型准备了路线图：向 my-println 中的那个 cond 添加新的子句。但这是一种拙劣的路线图，因为它把两件事情混到一块儿了：挑选某种实现的决策过程，以及这种特定实现的细节。

通过把类似 my-print-vector 这样的辅助函数抽取出来，在一定程度上能改善这种状况。然而，随后每次当你想要为 my-println 添加新功能时，你又不得不分别对两个地方做出修改。

- 创建一个特定于类型的新辅助函数。
- 修改现有的 my-println 函数，增加一个新的条件，并调用这个特定功能的辅助函数。

你真正需要的是这样一种方式，它能在你向系统中添加新功能时，无需修改任何已有的代码，只要在某个位置添加新的代码即可。Clojure 通过协议（第 6.3 节“协议”）提供了这种方式，还有就是接下来要介绍的多重方法。

8.2 定义多重方法

为了定义一个多重方法，可以使用 defmulti。

```
(defmulti name dispatch-fn)
```

name 是这个多重方法的名称，Clojure 会用方法的参数去调用 dispatch-fn，以选定多重方法的某个特定方法（实现）。

考虑上一节中的 my-print。它接受一个参数，就是那个将要被打印出来的东西，同时你希望能基于这个参数的类型选择某种特定的实现。所以，需要 dispatch-fn 函数能够接受一个参数，并返回该参数的类型。Clojure 有一个内建的函数符合这些描述，名为 class。下面使用 class 创建一个名为 my-print 的多重方法。

src/examples/multimethods.clj

```
(defmulti my-print class)
```

这时候，你已经描述了多重方法应如何去选择某种特定方法，但实际上那种特定方法尚不存在。不出所料，试图调用 my-print 将会失败。

```
(my-println "foo")
-> java.lang.IllegalArgumentException: \
No method for dispatch value
```

为了向 my-println 添加一个特定方法，可以使用 defmethod。

```
(defmethod name dispatch-val & fn-tail)
```

name 是多重方法的名称，指明了该实现的归属。Clojure 把 dispatch-val 与执行 defmulti 调度函数得到的结果进行匹配，以选定一个方法。fn-tail 则如同一个普通的函数，包含了参数及主体部分。

下面创建了一个与字符串相匹配的 my-print 实现。

src/examples/multimethods.clj

```
(defmethod my-print String [s]
  (.write *out* s))
```

现在，让我们用字符串参数作为参数，来调用 my-println。

```
(my-println "stu")
| stu
-> nil
```

接下来，创建一个与 nil 匹配的 my-print。

src/examples/multimethods.clj

```
(defmethod my-print nil [s]
  (.write *out* "nil"))
```

注意，你已经解决了上一节中提出的问题。现在每个 my-println 的实现都彼此分离，而不再是塞进一个巨大的 cond 当中。多重方法的实现方法，可以存在于你的源码中的任何地方，并且你可以在任意时候增添一个新的实现，无需触碰原来的代码。

8.2.1 调度是继承感知的

多重方法的调度知晓 Java 继承性。为了理解这一点，接下来我们将创建一个用来处理数字 Number 的 my-print，把数字的 toString 表示打印出来。

src/examples/multimethods.clj

```
(defmethod my-print Number [n]
  (.write *out* (.toString n)))
```

下面用整数测试一下这个 Number 的实现。

```
(my-println 42)
| 42
-> nil
```

42 其实是整数（Integer）[①]，而非数字（Number）。多重方法非常聪明，知道整数是一种数字，并做出正确的匹配。在内部调度时，它使用了 isa?函数。

```
(isa? child parent)
```

isa?知晓 Java 的继承关系，所以它知道整数就是数字。

```
(isa? Integer Number)
-> true
```

isa?并不仅限于继承。正如你随后将在第 8.4 节“创建特设分类法”中看到的，其行为可以在运行期动态扩展。

8.2.2 多重方法的默认值

如果 my-print 能有某种备用表示法，可用于任何尚未明确定义的类型，那会是一件很不错的事情。Clojure 已经为你想到了。你可以使用:default 作为调度值，当任何更具体的实现方法都无法成功匹配时，就由它负责对这种情况进行处理。下面使用:default，创建一个 my-println，它会把 Java 的 toString 值打印出来，而且还用#<>对

① 译注：在译者的 64 位 Windows 机器上，它其实是 Long 型的，用(type 42)可以验证。

输出内容进行了包裹。

src/examples/multimethods.clj

```
(defmethod my-print :default [s]
  (.write *out* "#<")
  (.write *out* (.toString s))
  (.write *out* ">"))
```

现在试着用这个默认方法，让 my-println 随便打印一些东西。

```
(my-println (java.sql.Date. 0))
-> #<1969-12-31>
(my-println (java.util.Random.))
-> #<java.util.Random@1c398896>
```

在一种很罕见的情形下，:default 在你的领域中已经有了某种特定的含义。遇到这种情况，你也可以使用下面的这种备用签名来创建多重方法。

```
(defmulti name dispatch-fn :default default-value)
```

可以用 default-value 来指定你自己的默认值。也许你会喜欢把它叫做:everything-else。

src/examples/multimethods/default.clj

```
(defmulti my-print class :default :everything-else)
(defmethod my-print String [s]
  (.write *out* s))
(defmethod my-print :everything-else [_]
  (.write *out* "Not implemented yet..."))
```

现在，任何找不到其他匹配的调度值，都会匹配到:everything-else 上。

就像你对 my-print 所做的那样，根据第一个参数的类型来调度多重方法，只是目前最为常见的一种调度方式而已。事实上，这种被称为多态的机制，对于许多面向对象语言来说，也是它们唯一的动态调度方式。

Clojure 的调度则要通用的多。让我们把 my-print 变得稍许复杂一些，使得仅凭普通多态性的能力，将无法应对。

8.3　调度不简单

Clojure 的 print 函数，能把各种各样“序列的”事物作为列表打印出来。如果你希望 my-print 也能做同样的事情，可以添加一个方法，调度到 Java 继承体系中高层的

容器接口之上，比如说 Collection。

src/examples/multimethods.clj
```
(require '[clojure.string :as str])
(defmethod my-print java.util.Collection [c]
  (.write *out* "(")
  (.write *out* (str/join " " c))
  (.write *out* ")"))
```

现在，用不同的序列试试，可以看到它们获得了不错的打印效果。

```
(my-println (take 6 (cycle [1 2 3])))
| (1 2 3 1 2 3)
-> nil

(my-println [1 2 3])
| (1 2 3)
-> nil
```

可作为完美主义者，你实在是难以忍受用圆括号来打印向量。这不像是它们应有的方括号语法。所以，又添加了另一个 my-print 方法，用于处理向量。由于所有的向量都实现了 IPersistentVector，所以下面的代码按理说应该可以工作。

src/examples/multimethods.clj
```
(defmethod my-print clojure.lang.IPersistentVector [c]
  (.write *out* "[")
  (.write *out* (str/join " " c))
  (.write *out* "]"))
```

但是它没有。相反，现在打印向量会抛出一个异常。

```
(my-println [1 2 3])
-> java.lang.IllegalArgumentException: Multiple methods match
dispatch value: class clojure.lang.LazilyPersistentVector ->
interface clojure.lang.IPersistentVector and
interface java.util.Collection,
and neither is preferred
```

现在的问题是，对于向量而言，有两个调度值都与之匹配：Collection 和 IPersistentVector。许多语言对方法调度进行了限制，例如通过禁止多重继承，以确保不会发生此类冲突。Clojure 采用了不同的方法。你可以创建冲突，并且你可以用 prefer-method 来解决它们。

```
(prefer-method multi-name loved-dispatch dissed-dispatch)
```

当你对一个多重方法调用 prefer-method 时，就相当于告诉它，每当出现冲突时，

应该优先选择调度值 loved-dispatch，而非 dissed-dispatch。因为你想要向量版本的 my-print 胜过了那个容器的版本，把你的期望告诉多重方法就可以。

src/examples/multimethods.clj

```
(prefer-method
 my-print clojure.lang.IPersistentVector java.util.Collection)
```

现在，你应该能把向量和其他序列都路由到正确的方法实现了。

```
(my-println (take 6 (cycle [1 2 3])))
| (1 2 3 1 2 3)
-> nil

(my-println [1 2 3])
| [1 2 3]
-> nil
```

许多语言要么建立了复杂的规则，要么武断地进行了限制，以便解决它们系统中对于函数调度的二义性。Clojure 则允许采用更简单的办法：如果出现了歧义，不必担心，用 prefer-method 来解决它。

8.4　创建特设分类法

多重方法允许你创建特设的分类法，在你探索那些尚未显式声明的类型关系时，这会很有帮助。

举个例子，考虑一个金融应用，涉及到支票和储蓄账户。我们需要为账户定义一个 Clojure 结构，并使用 tag 来区分上述两种账户。让我们把代码放进命名空间 examples.multimethods.account 中。要做到这一点，你需要在你的 classpath[①]中创建一个名为 examples/multimethods/account.clj 的文件，然后输入下列代码。

src/examples/multimethods/account.clj

```
(ns examples.multimethods.account)

(defstruct account :id :tag :balance)
```

现在，你将要创建两个不同的支票账户，分别标记为::Checking 和::Savings。采用首字母大写的名称是一种 Clojure 惯例，用来表示该关键字扮演了类型的角色。名称中的双冒号（::）使得对该关键字的解析，是在当前命名空间中进行的。为了查看命名空间解析时都发

① 注意，本书的示例代码中包含了本例的一份完整版本，并已经放进了 classpath 中。为了完成你自己的例子，只需要简单地移走或是重命名那个挡道的完整示例即可。

生了什么，不妨在 REPL 中分别输入:Checking 和::Checking，并加以比较。

```
:Checking
-> :Checking

::Checking
-> :user/Checking
```

把关键字放入一个命名空间中，有助于防止与其他人的代码产生名称冲突。当你想在另一个命名空间中使用::Savings 或者::Checking 时，需要对它们进行完全限定。

```
(struct account 1 ::examples.multimethods.account/Savings 100M)
-> {:id 1, :tag :examples.multimethods.account/Savings, :balance 100M}
```

完整的名称很快就会显得冗长而且乏味，所以你可以使用 alias 来为其指定一个较短的别名。

```
(alias short-name-symbol namespace-symbol)
```

下面使用 alias 来创建别名 acc。

```
(alias 'acc 'examples.multimethods.account)
-> nil
```

现在有别名 acc 可用了，下面创建了两个顶级的测试对象，储蓄账户和支票账户。

```
(def test-savings (struct account 1 ::acc/Savings 100M))
-> #'user/test-savings

(def test-checking (struct account 2 ::acc/Checking 250M))
-> #'user/test-checking
```

注意，数字末尾的那个 M 并不意味着你拥有数百万美元，而是用来表示 BigDecimal（任意精度小数）的字面量。

支票帐户的利率是 0，储蓄账户则是 5%。让我们创建一个多重方法 interest-rate，它基于:tag 进行调度，就像下面这样。

src/examples/multimethods/account.clj

```
(defmulti interest-rate :tag)
(defmethod interest-rate ::acc/Checking [_] 0M)
(defmethod interest-rate ::acc/Savings [_] 0.05M)
```

检查一下你的 test-savings 和 test-checking，确保这个 interest-rate 符合预期。

```
(interest-rate test-savings)
-> 0.05M
```

```
(interest-rate test-checking)
-> 0M
```

账户还要缴纳年服务费，计费规则如下。

- 普通支票账户支付 25 美元的服务费。
- 普通储蓄账户支付 10 美元的服务费。
- 高级账户不收取任何费用。
- 余额达到或超过 5000 美元的支票账户，即为高级。
- 余额达到或超过 1000 美元的储蓄账户，即为高级。

现实中，规则还会更加复杂。例如高级状态可能会由随时间变化的平均余额来决定，也可能会有其他的获准方式。但为了说明关键所在，上述规则已经足够复杂了。

当然，你也可以用一大串的条件逻辑来实现 service-charge，但 premium 感觉上就像是一种类型，尽管对于账户而言，并没有明确的 premium 标签。让我们创建一个多重方法 account-level，它会返回::Premium 或者::Basic。

src/examples/multimethods/account.clj

```
(defmulti account-level :tag)
(defmethod account-level ::acc/Checking [acct]
  (if (>= (:balance acct) 5000) ::acc/Premium ::acc/Basic))
(defmethod account-level ::acc/Savings [acct]
  (if (>= (:balance acct) 1000) ::acc/Premium ::acc/Basic))
```

赶快测试一下 account-level，确保支票和储蓄账户需要不同的余额水平才能达到::Premium 状态。

```
(account-level (struct account 1 ::acc/Savings 2000M))
-> :examples.multimethods.account/Premium

(account-level (struct account 1 ::acc/Checking 2000M))
-> :examples.multimethods.account/Basic
```

现在，你可能会禁不住想要拿 account-level 作为调度函数，来实现 service-charge。

src/examples/multimethods/service_charge_1.clj

```
; 糟糕的方法
(defmulti service-charge account-level)
(defmethod service-charge ::Basic [acct]
  (if (= (:tag acct) ::Checking) 25 10))
(defmethod service-charge ::Premium [_] 0)
```

出现在::Basic 的 service-charge 中的条件逻辑，正好就是类型驱动的那种条件，而

多重方法可以帮助我们避免它。这里的问题是，你已经通过 account-level 来进行调度了，可现在你却需要通过:tag 同样也能进行调度。没问题，你可以同时基于它们两者进行调度。下面编写的这个 service-charge，其调度函数既调用了 account-level，也调用了:tag，并把它们的结果放进一个向量中返回。

src/examples/multimethods/service_charge_2.clj

```
(defmulti service-charge (fn [acct] [(account-level acct) (:tag acct)]))
(defmethod service-charge [::acc/Basic ::acc/Checking] [_] 25)
(defmethod service-charge [::acc/Basic ::acc/Savings] [_] 10)
(defmethod service-charge [::acc/Premium ::acc/Checking] [_] 0)
(defmethod service-charge [::acc/Premium ::acc/Savings] [_] 0)
```

这个版本的 service-charge 依靠两种不同的分类法来进行调度：账户固有的:tag，和外部定义的 account-level。下面用几个账户来试一下，以验证 service-charge 能否按预期那样工作。

```
(service-charge {:tag ::acc/Checking :balance 1000})
-> 25

(service-charge {:tag ::acc/Savings :balance 1000})
-> 0
```

注意，我们甚至都没花功夫去创建一个“真正”的账户用于上述测试。像 account 这样的结构，其实就是被优化过适于存储特定字段的映射表，所以，如果你觉得会更方便的话，使用一个普通的映射表也完全没有问题。

特设类型的继承

你还可以对 service-charge 做出另外一个改进。由于所有高级账户都有相同的服务费，为::Savings 和::Checking 提供两个单独的 service-charge 方法感觉有些多余。要是有一个::Account 父类型就好了，这样你就能定义一个多重方法，让它为所有种类的::Account 匹配::Premium。Clojure 允许你使用 derive（派生）来定义任意的父子关系。

```
(derive child parent)
```

使用 derive，你能把::Savings 和::Checking 都指定为::Account 类型。

src/examples/multimethods/service_charge_3.clj

```
(derive ::acc/Savings ::acc/Account)
(derive ::acc/Checking ::acc/Account)
```

当你开始使用 derive 时，isa?会非常的有用。除了懂得 Java 继承以外，isa?还清楚所有的 derive 派生关系。

```
(isa? ::acc/Savings ::acc/Account)
-> true
```

现在，Clojure 已经知道储蓄和支票都属于账户，你可以定义一个 service-charge，使用单个方法来处理::Premium。

src/examples/multimethods/service_charge_3.clj

```
(defmulti service-charge (fn [acct] [(account-level acct) (:tag acct)]))
(defmethod service-charge [::acc/Basic ::acc/Checking] [_] 25)
(defmethod service-charge [::acc/Basic ::acc/Savings] [_] 10)
(defmethod service-charge [::acc/Premium ::acc/Account] [_] 0)
```

乍一看，你可能会认为 derive 和 isa?只不过是 Clojure 通过 Java 继承，简单复制过来的现成功能。可事实并非如此。Java 的继承关系会永远定格在你定义类的那一刻。而 derive 的派生关系，能在任何你需要的时候创建它们，并且可以在现有对象不知情，甚至不同意的情况下实施。所以，当你在现有对象之间发现有用的关系时，可以直接派生这种关系，而不用去动那些原有对象的源码，也无须创建那些无聊的“包装”类。

如果这么多种定义多重方法的方式，让你有些摸不着头脑的话，不必担心。在实践中，大多数 Clojure 代码对多重方法的使用都相当谨慎。接下来，让我们找一些开源的 Clojure 代码来看看，以便从中获得如何更好地使用多重方法的灵感。

8.5　何时应该使用多重方法

多重方法极为灵活，并且这种灵活性来自于选择。你要怎么选择，应该何时使用多重方法，而不是其他的一些技术呢？我们从两个方向来接近这个问题，不妨如下这么问。

- 用 Clojure 创建的项目，都在什么地方用到了多重方法？
- 它们在什么地方避开了多重方法？

最明显的一点是，多重方法非常罕见——大约每 1000 行代码才有一行。所以，你构建了一个 Clojure 应用，如果很少用到，甚至没有用到多重方法，那么你完全不必担心遗漏了什么重要的东西。一个 Clojure 程序没有定义多重方法，就像一个面向对象程序没有用到多态一样，一点也不奇怪。

很多多重方法是基于类来调度的。这种调度是最容易理解和实现的方式。我们已

经用 my-print 示例对其进行了详细的讨论，所以此处不再累述。

Clojure 多重方法会依据除类以外的其他东西来调度，那是相当罕见的。我们可以直接从 Clojure 中找一些例子。clojure.inspector 和 clojure.test 库就使用了不寻常的调度函数。

8.5.1 Inspector

Clojure 的 inspector 库，使用了 Swing 来为数据创建简单的视图。例如，你可以用它来获取系统属性的树形视图。

```
(use '[clojure.inspector :only (inspect inspect-tree)])
(inspect-tree (System/getProperties))
-> #<JFrame ...>
```

inspect-tree 返回（并且显示）了一个 JFrame 窗口，它用树形视图来展示任何树形结构的东西。所以，你也可以传一个嵌套的映射表给 inspect-tree。

```
(inspect-tree {:clojure {:creator "Rich" :runs-on-jvm true}})
-> #<JFrame ...>
```

树形结构由能够回答以下两个问题的节点构成。

- 谁是我的孩子？
- 我是一个叶子节点吗？

所有“树”、“节点”和“叶子”这些树形结构的概念，听起来就像是面向对象设计中类或是接口的候选。但是 inspector 库并没有这么做。相反，它使用特设的方式为现有类型添加了一套“树形结构”类型系统，使用的调度函数名为 collection-tag。

```
; 来自于Clojure 代码：clojure/inspector.clj
(defn collection-tag [x]
  (cond
    (instance? java.util.Map$Entry x) :entry
    (instance? clojure.lang.IPersistentMap x) :map
    (instance? java.util.Map x) :map
    (instance? clojure.lang.Sequential x) :seq
    :else :atom))
```

collection-tag 返回以下几个关键字中的一个：:entry、:map、:seq 和:atom。它们充当了这个树形结构世界中的类型系统。然后 collection-tag 函数被用来调度三个不同的多重方法，基于这个树形结构类型系统，来选定特定的实现。

```
(defmulti is-leaf collection-tag)

(defmulti get-child
  (fn [parent index] (collection-tag parent)))

 (defmulti get-child-count collection-tag)
; 简便起见，方法实现已省略
```

树形结构的类型系统，是围绕着现有 Java 类型系统来添加的。现有的那些对象，并没打算要成为树形结构中的一员，更没有为此进行任何准备；但 inspector 库为它们做了。树形结构示范了一种强大的重用风格。你可以在现有代码中发现新的类型关系，并且无须修改那些原始代码，就可以非常简单地对这些关系加以利用。

8.5.2　clojure.test

Clojure 中的单元测试库 clojure.test，允许你使用 is 宏来编写几种不同类型的断言。你可以对任意函数进行断言，称其为真。例如，10 不是一个字符串。

```
(use :reload '[clojure.test :only (is)])
(is (string? 10))

FAIL in clojure.lang.PersistentList$EmptyList@1 (NO_SOURCE_FILE:2)
expected: (string? 10)
actual: (not (string? 10))
-> false
```

尽管你可以使用任意的函数，但 is 只了解少数特定的函数，知道如何为它们提供更加详细的错误信息。例如，你可以用 instance?检查字符串是不是 Collection 的一个实例。

```
(is (instance? java.util.Collection "foo"))

FAIL in clojure.lang.PersistentList$EmptyList@1 (NO_SOURCE_FILE:3)
expected: (instance? java.util.Collection "foo")
actual: java.lang.String
-> false
```

is 同样也认识=。让我们证实一下，力量（power）并不等同于智慧（wisdom）。

```
(is (= "power" "wisdom"))

FAIL in clojure.lang.PersistentList$EmptyList@1 (NO_SOURCE_FILE:4)
expected: (= "power" "wisdom")
actual: (not (= "power" "wisdom"))
-> false
```

在内部，is 使用了一个名为 assert-expr 的多重方法，它不是依据类型，而是依据其第一个参数的真实身份来进行调度。

```
(defmulti assert-expr (fn [form message] (first form)))
```

第一个参数是个符号，代表了要用什么函数来进行检查，也就相当于另外的一套特设类型系统。只不过这次有三种类型：=、instance?和其他所有东西。

各种 assert-expr 方法，提供了那些你可能会从 is 中调用的函数的具体错误信息。因为多重方法是开放的，所以你也可以添加自己的 assert-expr 方法，为那些你经常会传给 is 的函数提供更好的错误信息。

8.5.3 反例

正如在第 8.4 节“创建特设分类法”中看到的，你经常可以使用多重方法，来把你的函数主流程中，那些基于类型的分支给提取出来。为了找出在什么地方不应该使用多重方法的反例，我们在 Clojure 的核心代码中，查找那些没有使用多重方法来提取的类型分支。

一个简单的例子是 Clojure 的 class，这是对底层 Java 的 getClass 方法一个空值安全（null-safe）的包装函数。去掉注释和元数据后，class 如下所示。

```
(defn class [x]
(if (nil? x) x (. x (getClass))))
```

你可以编写你自己版本的 class，让它变成一个多重方法，并基于 identity 来调度。

src/examples/multimethods.clj

```
(defmulti my-class identity)
(defmethod my-class nil [_] nil)
(defmethod my-class :default [x] (.getClass x))
```

任何需要检查空值的函数，都可以采用这种方式编写。但我却发现，原来的那个 class 函数，要比这个多重方法版本的更容易阅读。这就是所谓“正因为存在例外，才证明了有普遍规律”。尽管 class 的分支是基于类型的，但采用了分支的版本要更加易于阅读。

当决定是否要创建一个函数或是多重方法时，不妨采用下列一般性规则。

- 如果一个函数的分支是基于一种类型或是多种类型，可以考虑多重方法。
- 所谓类型，是那种你发现它们是什么，它们就是什么的东西。不必非得是明确的 Java 类或者数据标签。
- 你应该能做到，不参考 defmulti 定义，就能对 defmethod 的调度值进行解释。

- 不要仅仅为了处理可选参数或者递归，而使用多重方法。

当存有疑问时，这两种风格都试一下，选择看起来更具可读性的那个。

8.6　小结

多重方法支持任意方式的调度。通常它是基于类型关系的。有时候这些类型是正式的，就像 Java 类那样。其他的时候，它们是非正式、特别设置的，形成于系统中对象的性质。

第 9 章

极尽 Java 之所能

Clojure 对 Java 的支持，强大且犀利。说它强大，是因为它带来了 Lisp 语法丰富的表现力，并为 Java 量身订做了一些语法糖。说它犀利，则是因为它可以直达底层。Clojure 代码不需要任何通向 Java 的特殊转换层，会直接编译为字节码。

Clojure 对 Java，以及 Java 库相当包容。地道的 Clojure 代码会直接调用 Java 库，而不会试图把天下万物都包装的好似 Lisp 一般。这令许多 Clojure 新手感到非常惊讶，但这样做却相当务实。只要那个地方 Java 没问题，那么 Clojure 就不会去动它。

本章中，你将会看到 Clojure 访问 Java 是何等的方便、优雅和快速。

- 调用 Java 简单而且直接。Clojure 提供了语法扩展，用于访问任何你能够触及的 Java 代码：类、实例、构造器、方法和字段。尽管通常你会直接调用 Java 代码，但你也可以用更加函数式的风格对 Java API 进行包装。
- Clojure 运行飞快，这与很多其他 JVM 上的动态语言并不相同。因为有对原始类型和数组的定制支持，再加上类型提示，使得 Clojure 编译器生成的代码，与 Java 编译器生成的代码相同。
- Clojure 的异常处理用起来非常容易。更妙的是，必须显式进行的异常处理相当罕见。Clojure 的异常原语与 Java 是相同的。然而，Clojure 并不要求你去处理已检查异常（checked exception），并且使用 with-open 习语也使得资源清理工作变得非常轻松。

9.1　异常处理

在 Java 中出现异常处理代码，原因包括以下三方面。

- 包装已检查异常（如果你对已检查异常并不熟悉，不妨翻到下一页“已检查异常”看看）。
- 用一个 finally 块来清理非内存资源，比如说文件和网络句柄。
- 对问题做出响应：忽略这个异常、重试操作、把异常转换为非异常的结果等。

在 Clojure 中，事情是类似的，但要简单得多。通过 try 和 throw 这两个特殊形式，就为你提供了 Java 中 try、catch、finally 和 throw 具备的所有能力。但是你不应该经常性的使用它们，理由如下。

- 在 Clojure 中你不必处理已检查异常。
- 你可以使用诸如 with-open 这样的宏来封装资源清理。

让我们看看在实践中它是什么样子的。

9.1.1　保持简单的异常处理

Java 程序常常在抽象边界对已检查异常进行包装。一个很好的例子是 Apache Ant，它倾向于用一个 Ant 级别的构建异常来对底层异常（例如输入/输出异常）进行包装。

```
// 与 Ant 类似的代码（清晰起见，进行了简化）
try {
   newManifest = new Manifest(r);
} catch (IOException e) {
   throw new BuildException(...);
}
```

在 Clojure 中，你不会被强迫去处理已检查异常。你不必捕获它们，也不必声明你要抛出它们。所以，前面的代码会转化成下面这样。

```
(Manifest. r)
```

没有了对异常的包装，使得地道的 Clojure 代码与地道的 Java 代码相比，更易于阅读和编写。也就是说，在 Clojure 中，没有谁去阻止你显式地捕获、包装和重新抛出异常。只不过它不是必须的而已。只有当你打算以一种有意义的方式来做出反应时，才应该去捕获异常。

已检查异常

Java 的已检查异常，必须在每个可能产生它们的方法中，显式地进行捕获，或者是重新抛出。乍一看这似乎是个不错的主意：已检查异常可以利用类型系统严格地描述错误处理，并借助编译器来强制执行。然而，如今大多数 Java 程序员认为已检查异常是一种失败的尝试，因为它们在代码膨胀和可维护性方面的代价，大大超过了它们带来的益处。更多关于已检查异常的历史，请参阅罗德·瓦尔德霍夫（Rod Waldhoff）的文章及其随附链接（http://tinyurl.com/checked-exceptions-mistake）。

9.1.2 清理资源

垃圾收集（Garbage Collection）会清理内存中的资源。如果你使用了诸如文件句柄这种不受垃圾收集管控的外部资源，那么一旦发生了异常，就需要你确保把它们都清理干净了。在 Java 中，这通常是在 finally 块中处理的。

如果你需要释放的这种资源，遵循惯例提供了一个 close 方法，那么 Clojure 的 with-open 宏就派上用场了。

```
(with-open [name init-form] & body)
```

在内部，with-open 创建了一个 try 块，把 name 设置为 init-form 的结果，然后运行 body 中的那些形式。最重要的是，with-open 会在一个 finally 块中，关闭绑定到 name 的那个对象。

with-open 一个很好的例子是 clojure.string 中的 spit 函数。

```
(clojure.core/spit file content)
```

spit 只是简单的把一个字符串写入文件。来试一下看看。

```
(spit "hello.out" "hello, world")
-> nil
```

现在你可以找找看，应该会有一个内容为“hello, world”的文件 hello.out。

spit 的实现非常简单。

```
; 来自于clojure.core
(defn spit
  "Opposite of slurp. Opens f with writer, writes content, then
```

```
   closes f. Options passed to clojure.java.io/writer."
  {:added "1.2"}
  [f content & options]
  (with-open [^java.io.Writer w (apply jio/writer f options)]
    (.write w (str content))))
```

spit 在 f 之上创建了一个 PrintWriter，它几乎可以是任何能够写入的东西：文件、URL、URI，或是 Java 的任意一种 Writer 和输出流。随后它把 content 打印至 writer。最后，with-open 保证了在 spit 的末尾，writer 会被关闭。

如果你还想做更多地一些事情，而不仅仅只是在 finally 块中进行关闭操作，那么，Clojure 的 try 形式看起来是这个样子的。

```
(try expr* catch-clause* finally-clause?)
; catch-clause -> (catch classname name expr*)
; finally-clause -> (finally expr*)
```

因此，它可以按照如下语法使用。

```
(try
  (throw (Exception. "something failed"))
(finally
  (println "we get to clean up")))
| we get to clean up
-> java.lang.Exception: something failed
```

上面这段代码同时也演示了 Clojure 的 throw，throw 只是简单地把任何传给它的异常给抛了出去。

9.1.3　响应异常

异常处理中最有意思的部分，就是可以让代码中的 catch 块，去试图对问题做出响应。举个简单的例子，考虑编写一个函数，来测试某个特定的类在运行期间是否可用。

src/examples/interop.clj

```
; 不是对调用者友好的做法
(defn class-available? [class-name]
  (Class/forName class-name))
```

这种做法对调用者而言不是非常友好。调用者想要的，不过是一个“是”或者“否”的答案，但得到的却是一个异常。

```
(class-available? "borg.util.Assimilate")
-> java.lang.ClassNotFoundException: borg.util.Assimilate
```

一种更友好的方式是，用一个 catch 块来返回 false。

src/examples/interop.clj

```
(defn class-available? [class-name]
  (try
    (Class/forName class-name) true
    (catch ClassNotFoundException _ false)))
```

现在调用者的体验就要好很多了。

```
(class-available? "borg.util.Assimilate")
-> false

(class-available? "java.lang.String")
-> true
```

Clojure 把需要的一切东西都给了你，用来抛出异常、捕获异常以及清理资源。与此同时，Clojure 让异常乖乖待在它们应该待的地方。异常是很重要，但尚未重要到要让你的主线代码被它们主导。

Clojure 的设计目标是让你把事情做好，同时在这个过程中享受乐趣。然而，能把事情做好的一个重要部分，就是能够充分发挥你使用的平台的全部潜能。在本书的其他章节中，我们已经见过了地道的 Clojure 行事方式。本章中，我们将用 Java 风格来做。

为了能提供 Java 平台的全部能力，Clojure 做了几件事情。

- 类型提示及推断，人们可以在有性能需要的地方，与静态类型语言建立关联。
- AOT（Ahead-of-time，提前）编译，使得 Clojure 程序可以作为平等的参与者，加入到这个以二进制产物为中心的 Java 生态系统当中。
- 虽然作为灵活的实现工具，reify、defrecord 和 deftype 更为可取，但 Clojure 也提供了一些交互形式，让你可以访问 Java 互操作中那些丑陋的部分。
- Clojure（以及 Clojure 分享库）中有一整套针对常见任务的“已含电池”的库，而且还在迅速成长。此外，如果这些库中尚未包含你需要的某些东西，你也可以随时调用原始的 Java 库。
- 如果对于你正在解决的问题而言，本章当中的这些技术显得过于丑陋或是不必要，那真是太棒了！离它远远地。但是，如果你需要榨干哪怕最后一点点的性能，或是需要与一个古老、丑陋的库相处得更好，那么本章为你而生。

9.2　与整数搏斗

Clojure 为整数类型提供了三组不同的运算。

- 未经检查（unchecked）运算
- 默认运算
- 升格（promoting）运算

表 9-1 给出了这些运算类型的一个采样。

表 9-1　Clojure 的运算类型

未经检查的	默认	升格
unchecked-add	+	+'
unchecked-subtract	-	-'
unchecked-multiply	*	*'
unchecked-inc	inc	inc'
unchecked-dec	dec	dec'

那些未经检查的运算与 Java 中的原始数学计算完全对应。它们执行速度快，但却极度危险，因为它们会默默地溢出，并得出不正确的答案。在 Clojure 中，只有在这种罕见的情形下，你才应该使用未经检查的运算：产生溢出就是期望中的行为；或是性能至关重要，并且你确信不可能溢出，或者哪怕溢出了也无所谓。

```
(unchecked-add 9223372036854775807 1)
-> -9223372036854775808
```

出于性能考虑，默认运算在可能的情况下，使用的是 Java 基本类型，但会始终进行溢出检查，并在造成溢出时抛出异常。

```
(+ 9223372036854775807 1)
-> ArithmeticException integer overflow
```

升格运算会在溢出发生时，自动地由基本类型升格为大数。这使得它可以处理任意范围的数字，但会造成显著的性能开销。由于基本类型和大数没有共同的基类型，使用升格运算来进行数学计算，阻碍了把基本类型用作返回类型。

```
(+' 9223372036854775807 1)
-> 9223372036854775808N
```

Clojure 依赖 Java 的高精度小数类 BigDecimal，来表示任意精度的小数。详情请参阅联机文档[①]。BigDecimal 提供了任意的精度，但代价颇高：BigDecimal 的数学计

① http://docs.oracle.com/javase/6/docs/api/java/math/BigDecimal.html

算要明显慢于 Java 浮点数。

Clojure 有它自己的大整数类 BigInt，用来处理与 Java 的 BigInteger 间的转换。相对于直接使用 Java 的 BigInteger，Clojure 的 BigInt 有一些性能方面的改进。同时它也对 BigInteger 一些毛边进行了包装。尤其是，它正确地实现了 hashCode。这就使得相等性优先于表示法，而你在这门语言的几乎每种抽象中都能看到这一点。

在底层，Clojure 使用了 Java 的 BigInteger。性能的差异来自于 BigInt 是如何对待其值的。BigInt 由一个 Long 部分和一个 BigInteger 部分组成。当传入 BigInt 的值小到足以被视为 Long 时，它就是一个 Long。当对 BigInt 进行数值运算时，如果运算结果小到足以被视为 Long 时，它还是一个 Long。这使得用户只有在绝对必要的情况下，才需要为添加那个溢出提示（N）而付出 BigInteger 的代价。

9.3 性能优化

在 Clojure 中调用 Java 所采用的惯例技术，已经在第 2.5 节“调用 Java”中做过描述。对于百分之九十的场景而言，那样生成的代码已经足够快了。不过，当你需要时，也可以进行局部调整以提高性能。这些调整并不会导致外部调用者需要更改调用你的代码的方式，所以，你可以无偿地让你的代码继续工作，并使它运行加快。

9.3.1 使用基本类型提升性能

在前面的章节中，函数的参数没有携带任何类型信息。Clojure 只是简单地做了正确的事。依赖于你的视点，这可能是优点也可能是弱点。它是优点，因为你的代码会因此干净并且简单，还能够利用鸭子类型（duck typing）。但它同时也是弱点，因为代码的读者无法确定数据类型，并且为了把事情做对，也会带来一定的性能开销。

考虑一个能计算从 1 到 n 的数字之和的函数。

```
; 仅用于演示性能，切勿写这样的代码
(defn sum-to [n] (loop [i 1 sum 0]
   (if (<= i n) (recur (inc i) (+ i sum)) sum)))
```

你可以用一个较小的输入值来验证这个函数。

```
(sum-to 10)
=> 55
```

让我们看一下 sum-to 是如何执行的。为了给一次操作计时，你可以使用 time 函数。在做基准测试时，你会倾向于进行多次测量，这样就可以排除任何外部附加的启动开销；因此，你可以在 dotimes 宏的内部调用 time。

```
(dotimes bindings & body)
```

dotimes 会把名称绑定到整数上，从零到 n-1，重复地执行其 body。使用 dotimes，你可以收集五次 sum-to 的计时如下。

```
(dotimes [_ 5] (time (sum-to 10000)))
|  "Elapsed time: 0.149 msecs"
|  "Elapsed time: 0.126 msecs"
|  "Elapsed time: 0.194 msecs"
|  "Elapsed time: 0.279 msecs"
-> "Elapsed time: 0.212 msecs"
```

为了提高速度，你可以提示其参数和返回值的类型为长整形。Clojure 的类型推断会把这个提示传播给其内部的所有运算，以及这个函数内部的函数调用。

```
(defn ^long integer-sum-to [^long n]
  (loop [i 1 sum 0]
    (if (<= i n)
      (recur (inc i) (+ i sum))
      sum)))
```

integer-sum-to 确实更快了。

```
(dotimes [_ 5] (time (integer-sum-to 10000)))
|  "Elapsed time: 0.044 msecs"
|  "Elapsed time: 0.023 msecs"
|  "Elapsed time: 0.025 msecs"
|  "Elapsed time: 0.023 msecs"
-> "Elapsed time: 0.02 msecs"
```

Clojure 的原始数学计算仍然是精确的，因为它会对溢出进行检查并且抛出异常。但还能更快吗？Java 程序员们可以使用破坏性的超快速数学计算：算术运算拥有极限性能，但可能会悄无声息的溢出并损坏数据。

Clojure 通过未检查的函数家族，打开了通向 Java 算术语义的大门。也许通过使用未检查版本的+，能让你得到一个运行更快的 unchecked-add 函数。

```
(defn ^long unchecked-sum-to [^long n]
  (loop [i 1 sum 0]
    (if (<= i n)
      (recur (inc i) (unchecked-add i sum))
     sum)))
```

可这个 unchecked-sum-to 并没有显著的变快。

```
(dotimes [_ 5] (time (unchecked-sum-to 10000)))
|  "Elapsed time: 0.039 msecs"
|  "Elapsed time: 0.018 msecs"
|  "Elapsed time: 0.014 msecs"
|  "Elapsed time: 0.015 msecs"
-> "Elapsed time: 0.015 msecs"
```

优先级非常重要！基本类型提示能让某些运算显著的变快。然而，切换至 Java 的未检查语义通常是一个亏本生意。往往是你得到了些许微不足道的性能提升，但却伴随着明天数据就会损坏的可能性。

那么，Clojure 到底是为什么还要提供这些操作呢？两个原因如下。

- 有的时候你实际上想要的就是 Java 语义。未经检查的运算主要用于与其他期望这种行为的库进行交互。
- 不进行尝试的话，没人知道（或是相信）它们能否加快执行速度。

首先考虑正确性，然后仅当需要时才对速度进行优化。发生溢出时，integer-sum-to 将会抛出一个异常。这是个问题，但很容易被检测出来。

```
(integer-sum-to 10000000000)
-> java.lang.ArithmeticException: integer overflow
```

但 unchecked-sum-to 在发生溢出时，却将会悄无声息的失败。它能在一段特意安排的程序中保持安静，但却会灾难性的损坏数据。

```
(unchecked-sum-to 10000000000)
-> -5340232216128654848 ; WRONG!!
```

考虑到正确性和性能这两个关注点之间存在的矛盾，你通常应该倾向于如同最初的 sum-to 那样，简单且未经修饰的代码。如果性能分析定位到了瓶颈所在，那时候再要求 Clojure 在需要的地方使用原始类型也不迟。

为了演示 Clojure 中的各种整数运算，示例 sum-to 故意设计的非常简单。在真实的 Clojure 程序中，通过使用 reduce 来实现 sum-to，会有更强的表达力。对序列进行求和的过程，与对其前两项求和是一致的，将相加结果与下一项相加，以此类推。这的确就是(reduce + ...)提供的循环模式。使用归纳函数 reduce，你可以把 sum-to 重写为一行代码。

```
(defn better-sum-to [n]
  (reduce + (range 1 (inc n))))
```

这个例子同时也说明了一个更通用的重点：从选择合适的算法开始。对从 1 到 n 的数字进行求和，可被直接计算如下。

```
(defn best-sum-to [n]
  (/ (* n (inc n)) 2))
```

即便没有使用性能提示，这也比那个基于反复相加的实现要快得多。

```
(dotimes [_ 5] (time (best-sum-to 10000)))
|  "Elapsed time: 0.029 msecs"
|  "Elapsed time: 0.0040 msecs"
|  "Elapsed time: 0.0040 msecs"
|  "Elapsed time: 0.0040 msecs"
-> "Elapsed time: 0.0030 msecs"
```

性能是一个微妙的话题。千万不要为了速度而编写丑陋的代码。首先选择合适的算法，并让你的代码能够正常工作。如果你遇到了性能问题，再通过分析来定位问题。然后才引入必要的复杂性来解决那些问题。

9.3.2　添加类型提示

Clojure 中，可以为函数的参数、let 绑定、变量名和表达式添加类型提示。这些类型提示出于以下三个目的。

- 对关键性能路径进行优化。
- 文档化所需类型。
- 在运行期强制要求所需类型。

例如，考虑下面的这个函数，它会返回关于一个 Java 类的信息。

```
(defn describe-class [c]
  {:name (.getName c)
   :final (java.lang.reflect.Modifier/isFinal (.getModifiers c))})
```

你可以通过把特殊变量*warn-on-reflection*设置为真，来询问 Clojure，看它可以推断出多少类型信息。

```
(set! *warn-on-reflection* true)
-> true
```

set!末尾的那个感叹号是一种习惯表示法，指出 set!会更改可变状态。在第 5.5.2 节“用于 Java 回调 API”中，有关于 set!的详细描述。*warn-on-reflection*被设置为真之后，编译 describe-class 将会产生以下警告。

```
Reflection warning, line: 87
- reference to field getName can't be resolved.

Reflection warning, line: 88
- reference to field getModifiers can't be resolved.
```

这些警告表明，Clojure 无法获知 c 的类型。你可以使用元数据语法^Class 来提供一个类型提示，以解决这个问题。

```
(defn describe-class [^Class c]
  {:name (.getName c)
   :final (java.lang.reflect.Modifier/isFinal (.getModifiers c))})
```

放置了类型提示之后，反射警告随之消失。这样，Clojure 编译出来的代码，会和 Java 编译出来的代码完全相同。此外，试图用非 Class 的对象来调用 describe-class，将会抛出一个 ClassCastException 异常。

```
(describe-class StringBuffer)
-> {:name "java.lang.StringBuffer", :final true}

(describe-class "foo")
-> java.lang.ClassCastException: \
   java.lang.String cannot be cast to java.lang.Class
```

如果你发现 ClassCastException 提供的有用信息太少，那是因为你使用的还是 Java 6 之前的版本。有更好的错误报告，是你让 Clojure 代码运行在 Java 6 或者更新版本之上的诸多理由之一。

当你提供了类型提示，Clojure 会插入一个恰当的类型转换，以避免因为对 Java 方法进行反射调用而导致变慢。不过，如果你的函数并没有真的调用任何带有提示对象的 Java 方法，那么 Clojure 就不会插入转换。考虑下面这个 wants-a-string 函数。

```
(defn wants-a-string [^String s] (println s))
-> #'user/wants-a-string
```

你也许期望 wants-a-string 会对非字符串的参数发出抱怨。可事实上，它仍然表现的极为开心。

```
(wants-a-string "foo")
-> foo

(wants-a-string 0)
-> 0
```

Clojure 会辩解道，wants-a-string 从来就没有把它的参数真正的当作字符串来使用

（println 很乐意尝试打印任意类型的参数）。由于不需要调用字符串方法，Clojure 就不会试图把 s 转换为一个字符串。

当你需要速度时，类型提示会让 Clojure 的代码向下编译至与 Java 产生的代码相同。不过，通常你并不需要类型提示。首先确保代码正确，然后再去考虑让它更快。

9.4　在 Clojure 中创建 Java 类

Clojure 的所有对象都实现了合理的 Java 接口。

- Clojure 的数据结构实现了来自于 Java 容器 API 中的接口。
- Clojure 的函数实现了 Runnable 和 Callable。

除了这些通用接口，你偶尔也会需要一些特定于领域的接口。通常它们来自于回调处理器，面向诸如 Swing 或是一些 XML 解析器这样的事件驱动 API。Clojure 可以根据需要，仅用几小行必须的 Java 代码，就轻松地生成一次性代理，或是磁盘上的类。

9.4.1　创建 Java 代理

为了与 Java 互操作，你常常需要实现 Java 接口。一个不错的例子是，用简单的 XML（SAX）解析器 API 来解析 XML 文档。作为这个例子的准备工作，需要导入下面的这些类。在我们完成本章之前，会用到它们。

```
(import '(org.xml.sax InputSource)
        '(org.xml.sax.helpers DefaultHandler)
        '(java.io StringReader)
        '(javax.xml.parsers SAXParserFactory))
```

要使用 SAX 解析器，你需要实现一种回调机制。最简单的办法往往是扩展 DefaultHandler 类。在 Clojure 中，你可以使用 proxy 函数来扩展一个类。

```
(proxy class-and-interfaces super-cons-args & fns)
```

作为一个简单的例子，下面使用 proxy 来创建一个 DefaultHandler，它会把调用 startElement 的详情给打印出来。

```
(def print-element-handler
  (proxy [DefaultHandler] []
    (startElement [uri local qname atts]
      (println (format "Saw element: %s" qname)))))
```

proxy 生成了一个代理类的实例。proxy 的第一个参数是[DefaultHandler]，一个由超类和超接口组成的向量。这第二个参数[]，是为其基类构造函数准备的参数向量。在这个例子中不需要任何参数。

在设置了代理之后，接下来就是零或多个代理方法的实现。前面所示的代理有一个方法。它的名称是 startElement，接受四个参数，并把参数 qname 的名称打印出来。

现在，你需要的只是一个解析器，可以把这个处理器传给它。这需要哼哧哼哧的越过一大堆 Java 工厂方法和构造函数。为了在 REPL 中的探索能更轻松一些，你可以创建一个解析 XML 字符串的函数。

```
(defn demo-sax-parse [source handler]
  (.. SAXParserFactory newInstance newSAXParser
    (parse (InputSource. (StringReader. source)) handler)))
```

现在解析起来就容易得多了。

```
(demo-sax-parse "<foo>
  <bar>Body of bar</bar>
  </foo>" print-element-handler)
  | Saw element: foo
  | Saw element: bar
```

前面的这个例子，演示了如何通过创建 Clojure 代理，来处理 Java XML 接口的这种机制。你可以采用类似的方法来实现你自己的自定义 Java 接口。不过，如果你要做的只是 XML 处理的话，clojure.data.xml 库已经能很好地支持 XML 了，并且它还能和任意与 SAX 兼容的 Java 解析器一同工作。

代理机制十分通用，只要你希望，它可以生成任意类型的 Java 对象，且立即可用。有时候对象非常简单，以至于你可以把整个对象都塞进一行代码中。下面的代码创建了一个新的线程，然后在这个新线程里创建了一个 Runnable 的动态子类。

```
(.start (Thread.
  (proxy [Runnable] [] (run [] (println "I ran!")))))
```

在 Java 中，你必须为每个接口的每个方法都提供一份实现。而在 Clojure 中，你终于可以摆脱它们了。

```
(proxy [Callable] []) ; 没有任何方法的代理 (??)
```

如果你省略了某个方法的实现，Clojure 会为其提供一个默认的，抛出一个 UnsupportedOperationException 异常。

```
(.call (proxy [Callable] []))
-> java.lang.UnsupportedOperationException: call
```

对于诸如 Runnable 和 Callable 这样只有一个方法的接口而言，这样的默认实现没有太大意义，但是当你实现一些较大的接口，而且并不关心其中的某些方法时，这就相当便利了。

至此，你已经在本章中看到了如何使用 proxy 来创建 Java 接口的实现。当你需要时，这是非常强大的，不过 Clojure 往往已经为你代劳了。例如，函数自动的实现了 Runnable 和 Callable。

```
; 普通用途：调用一个匿名函数
(#(println "foo"))
foo
; 通过 Runnable 的 run 来调用
(.run #(println "foo"))
foo
; 通过 Callable 的 call 来调用
(.call #(println "foo"))
foo
```

这使得把 Clojure 函数传给其他线程变得非常容易。

```
(dotimes [i 5]
  (.start
    (Thread.
      (fn []
        (Thread/sleep (rand 500))
        (println (format "Finished %d on %s" i (Thread/currentThread)))))))
```

对于诸如 XML 和线程回调这样的一次性任务而言，Clojure 代理用起来既快捷又方便。如果你需要一个长期存在的类，你同样可以在 Clojure 中生成新的命名类。

9.4.2　使用 Java 容器

对于大多数用途而言，Clojure 容器都可以取代 Java 容器。Clojure 容器是并发安全的，具有良好的性能特征，并且实现了相应的 Java 容器接口。所以，在使用 Clojure 时你通常应该优选 Clojure 自己的容器，甚至在方便的时候，你也可以把它们传回给 Java。

如果你选择使用 Java 容器，Clojure 当然也不会阻止你。从 Clojure 的角度来看，Java 容器与其他类是一样的，所有的 Java 互操作形式都有效。但是，Java 容器是专为基于锁的并发而设计的。它们并不提供 Clojure 容器那样的并发保证，也就无法从 Clojure 的软事务内存获得益处。

一个需要特别对待的 Java 容器是数组。在 Java 中，数组有它自己的语法和它自己的字节码指令。Java 数组并没有实现任何 Java 接口。所以，Clojure 容器也就无法伪装成数组。Java 容器同样也不能！Java 平台让数组在每一个方面都成为特例，所以 Clojure 也是一样。

Clojure 提供了 make-array 来创建 Java 数组。

```
(make-array class length)
(make-array class dim & more-dims)
```

make-array 接受一个类型，以及一或多个维度上的尺寸作为参数。如果是一维的字符串数组，你可能会这么写。

```
(make-array String 5)
-> #<String[] [Ljava.lang.String;@45a270b2>
```

这个有点怪异的输出，是 Java 为数组提供的 toString()实现：[Ljava.lang.String;就是所谓“一维的字符串数组”的 JVM 规范编码。在 REPL 中，这种表示法没有太大用处，所以你可以用 seq 把 Java 数组包装成 Clojure 序列。这样 REPL 就能逐项的打印整个数组了。

```
(seq (make-array String 5))
-> (nil nil nil nil nil)
```

Clojure 还包括一个名如 int-array 的函数家族，用来创建 Java 基础类型的数组。你可以在 REPL 中发出下列命令，来查阅这个数组函数家族的文档。

```
(find-doc "-array")
```

Clojure 提供了一组针对 Java 数组的低级操作，包括 aset、aget 和 alength。

```
(aset java-array index value)
(aset java-array index-dim1 index-dim2 ... value)
(aget java-array index)
(aget java-array index-dim1 index-dim2 ...)
(alength java-array)
```

下面使用 make-array 来创建一个数组，然后使用 aset、aget 和 alength 来操作这个数组。

```
(defn painstakingly-create-array []
  (let [arr (make-array String 5)]
    (aset arr 0 "Painstaking")
    (aset arr 1 "to")
    (aset arr2 "fill")
```

```
    (aset arr 3" in")
    (aset arr 4 "arrays")
   arr))

(aget (paintakingly-create-array) 0)
-> "Painstaking"

(alength (painstakingly-create-array))
-> 5
```

大多数情况下你会发现，诸如 to-array 这样的高级函数要更容易使用，可以直接由任何一个容器创建出数组。

```
(to-array sequence)
```

用 to-array 创建出来的，总是一个 Object 的数组。

```
(to-array ["Easier" "array" "creation"])
-> #<Object[] [Ljava.lang.Object;@1639f9e3>
```

当调用接受可变参数列表的 Java 方法时，to-array 同样也很有用，比如说 String/format。

```
; 仅作示例。更好的做法是用 clojure.core/format
(String/format "Training Week: %s Mileage: %d"
               (to-array [2 26]))
-> "Training Week: 2 Mileage: 26"
```

to-array 的表亲 into-array，能创建出类型比 Object 更为具体的数组。

```
(into-array type? seq)
```

你可以向 into-array 传入一个显式的类型，作为其可选的第一个参数。

```
(into-array String ["Easier", "array", "creation"])
-> #<String[] [Ljava.lang.String;@391ecf28>
```

如果你省略了类型参数，into-array 将会根据序列中的第一项，来猜测应该是何种类型。

```
(into-array ["Easier" "array" "creation"])
-> #<String[] [Ljava.lang.String;@76bfd849>
```

如你所见，这个数组包含的是字符串，而不是 Object。如果你想要对一个 Java 数组中的每一项都进行变换，而又不想把它转换为一个 Clojure 序列，那么你可以使用 amap。

```
(amap a idx ret expr)
```

amap 会对数组 a 进行克隆，并把这个复制品绑定到你用 ret 指定的名称上。然后，它会对 a 中的每个元素都执行一次 expr，同时还把 idx 绑定到了当前元素的索引上。最后，amap 返回那个克隆数组。你可以使用 amap 把数组中的所有字符串都转换为大写。

```
(def strings (into-array ["some" "strings" "here"]))
-> #'user/strings

(seq (amap strings idx _ (.toUpperCase (aget strings idx))))
-> ("SOME" "STRINGS" "HERE")
```

上面的 ret 参数被设置成了下划线（_），表示它在这个映射表达式中并不需要。此外，用 seq 包装起来，只是为了便于在 REPL 中打印结果而已。areduce 与 amap 类似。

```
(areduce a idx ret init expr)
```

amap 会产生一个新的数组，areduce 则能产生你想要的任何东西。一开始，ret 被设置为 init，随后每次都被设置为调用 expr 得到的返回值。areduce 通常用于编写那种，以某种方式对一个容器进行“总结”的函数。例如，下面这个调用，能找出字符串数组中，最长的那个字符串的长度。

```
(areduce strings idx ret 0 (max ret (.length (aget strings idx))))
-> 7
```

amap 和 areduce 都是用来与 Java 数组交互的专用宏。

9.5 真实世界的例子

虽然讨论各种不同的互操作情况，以及学习如何使用 Java 的原始形式来挤出一点额外性能都有帮助，但你仍然需要一些实用的动手知识。在接下来的这个例子中，我们将要构建一个应用，来测试网站的可用性（availability），目标是检查网站是否正确地返回了 HTTP 200 OK 响应。如果收到的不是我们期望的响应，那么它会被标记为不可用。

我们将要再次使用构建工具 Leiningen。如果你还没有安装 Leiningen，请参阅第 1.2 节“Clojure 编程快速入门”。下面让我们从创建一个新项目开始吧。

```
lein new pinger
```

打开你的 project.clj 文件，并修改其内容以符合我们将要进行的工作。请确保 Clojure 更新到了最新的版本。[①]

① 译注：本书编写时 Clojure 的最新发布版本是 1.3.0，翻译本书时，1.5.0 即将发布。不过不用担心，本书的所有示例在 Clojure 的最新版本下都能够顺利通过。所以，你把下例中的 1.3.0 改为 1.4.0，甚至 1.5.0 都没有问题。

```
(defproject pinger "0.0.1-SNAPSHOT"
  :description "A website availability tester"
  :dependencies [[org.clojure/clojure 1.3.0]])
```

然后运行 lein deps 命令来获取依赖库。

```
lein deps
```

首先，我们需要编写一些代码，连接到一个 URL 地址，并捕获响应码。我们可以通过使用 Java 的 URL 类来实现。

```
(ns pinger.core
  (:import (java.net URL HttpURLConnection)))
(defn response-code [address]
  (let [conn ^HttpURLConnection (.openConnection (URL. address))
        code (.getResponseCode conn)]
    (when (< code 400)
      (-> conn .getInputStream .close))
    code))
```

在 REPL 中试一下看看。

```
(response-code "http://google.com")
-> 200
```

现在，让我们创建一个函数，它会根据 response-code 判断指定的 URL 地址是否可用。在当前语境中，我们把“可用”定义为返回了 HTTP 200 响应码。

```
(defn available? [address]
  (= 200 (response-code address)))


(available? "http://google.com")
-> true

(available? "http://google.com/badurl")
-> false
```

接下来，我们需要采用一种方式来启动程序，让它不时地去检查我们关注的 URL 列表，并报告那些 URL 地址的可用性。下面让我们创建一个-main 函数。

```
(defn -main []
  (let [addresses '("http://google.com"
                    "http://amazon.com"
                    "http://google.com/badurl")]
    (while true
      (doseq [address addresses]
        (println (available? address)))
      (Thread/sleep (* 1000 60)))))
```

在这个例子中，我们创建了一个地址列表（两个正确的和一个错误的），并使用一个不会退出的 while 循环，以获得永不停息的程序执行效果。它会持续不断地每隔一分钟就检查一次这些地址，直到整个程序被终止。因为我们要导出-main 函数，所以不要忘了把:gen-class 添加到你的命名空间声明中。

```
(ns pinger.core
  (:import (java.net URL))
  (:gen-class))
```

现在，基础已经搭建好了，接下来还得告诉 Leningen 我们的主函数在哪儿。打开 project.clj 并添加下面这个:main 声明。

```
(defproject pinger "0.0.1-SNAPSHOT"
  :description "A website availability tester"
  :dependencies [[org.clojure/clojure "1.3.0"]]
  :main pinger.core)
```

是时候把我们的程序编译为 JAR 文件，并运行一下看看了。执行下列命令。

```
lein uberjar
java -jar pinger-0.0.1-SNAPSHOT-standalone.jar
true
false
true
```

你应该能看到程序启动了，并且一直不停地运行，直到你按下 Ctrl-C 把它给终止了。

9.5.1 真正的持续循环

一个始终为真的 while 循环会持续运行，直至程序终止，但实际上为了得到这个效果，这么做并非最整洁的方法，因为它并未考虑到应该干净的关闭。我们可以使用调度线程池（scheduled thread pool），它会采用类似 while 循环的方式，启动并执行那些那些你打算要执行的命令，只不过，它支持更加高级的控制。接下来，让我们在 src 目录中创建一个名为 scheduler.clj 的文件，并输入下列代码。

```
(ns pinger.scheduler
  (:import (java.util.concurrent ThreadPoolExecutor
             ScheduledThreadPoolExecutor TimeUnit)))
(defn scheduled-executor
  "Create a scheduled executor."
  ^ScheduledThreadPoolExecutor [threads]
  (ScheduledThreadPoolExecutor. threads))

(defn periodically
```

```
  "Schedules function f to run on executor e every 'delay'
   milliseconds after a delay of 'initial-delay' Returns
   a ScheduledFuture."
  ^ScheduledFuture [e f & {:keys [initial-delay delay]}]
  (.scheduleWithFixedDelay
    e f
    initial-delay delay
    TimeUnit/MILLISECONDS))
(defn shutdown-executor
  "Shutdown an executor."
  [^ThreadPoolExecutor e]
  (.shutdown e))
```

这个命名空间提供了用于创建和关闭 Java 对象 ScheduledExecutor 的函数。它还定义了一个名为 periodically 的函数，用于接受一个执行者（executor）、一个函数、一个初始延时和一个重复延时作为参数。它会在等待初始延时之后，首次执行函数 f，然后依据重复延时，反复持续执行这个函数。它会一直运行下去，直至线程池被关闭。

让我们更新一下 pinger.core 的程序，利用刚才这些调度代码，让-main 函数只需要调用一个函数就启动循环。用下面这些函数替代那个旧的-main。

```
(defn check []
  (let [addresses '("http://google.com"
                    "http://google.com/404"
                    "http://amazon.com")]
    (doseq [address addresses]
      (println (available? address)))))

(def immediately 0)
(def every-minute (* 60 1000))

(defn start [e]
  "REPL helper. Start pinger on executor e."
  (scheduler/periodically e check
                          :initial-delay immediately
                          :delay every-minute))

(defn stop [e]
  "REPL helper. Stop executor e."
  (scheduler/shutdown-executor e))

(defn -main []
  (start (scheduler/scheduled-executor 1)))
```

记得更新命名空间声明，确保导入了调度器 scheduler。

```
(ns pinger.core
  (:import (java.net URL))
  (:require [pinger.scheduler :as scheduler])
  (:gen-class))
```

前例中并非所有东西都是必须的，但它们使得代码更加可读。增加了启动函数 start 和停止函数 stop，使得在 REPL 中进行交互操作更为容易，这么做能获得的益处实在是太大了，所以你应该从这个例子开始进行扩展。最后一件要做的事情，就是运行 lein uberjar 命令，并执行 JAR 文件。这个程序的功能，和之前那个应该完全相同。

9.5.2 日志

至此，我们已经制作了一个程序，能够定期检查一系列网站的可用性。然而，它还缺乏这样一种能力：追踪程序已经做了些什么事情，并通知我们某个站点是何时失效的。我们可以用日志系统同时解决这两个问题。对于 Java 应用程序来说，有大把的选择可以用来做日志，在这个例子中，我们会采用 log4j。它是一个真正可以使用的日志系统，并且还能赋予我们邮件通知的能力。这真是太棒了，因为当我们发现一个网站不再可用时，就可以发送一封邮件来告警。为此，我们需要把 log4j 和 mail 这两个库加入我们的应用。为了能更容易的利用 log4j，我们还需要加入 clojure.tools.logging。下面打开你的 project.clj 文件，并加入 clojure.tools.logging、log4j 和 mail。

```
(defproject pinger "0.0.1-SNAPSHOT"
  :description "A website availability tester"
  :dependencies [[org.clojure/clojure "1.3.0"]
                 [org.clojure/tools.logging "0.2.3"]
                 [log4j "1.2.16"]
                 [javax.mail/mail "1.4.1"]]
  :main pinger.core)
```

另外记得要用 Leiningen 获取依赖库。

```
lein deps
```

Clojure 日志库最棒的一点是，会使用任何在 classpath 中的出现的标准 Java 日志库，所以不需要在 log4j 和你的应用之间进行额外的连接操作。只要在项目的根目录下，创建一个名为 resources 的目录，Leiningen 就会自动把这个目录中的内容添加到 classpath 中。这正是你的 log4j 属性文件所需要的。接下来让我们在 resources 目录中创建一个名为 log4j.properties 的文件，并添加下列内容。

```
log4j.rootLogger=info, R, email
log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R.File=logs/pinger.log
log4j.appender.R.MaxFileSize=1000KB
log4j.appender.R.MaxBackupIndex=1
log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%d{ISO8601} %-5p [%c] - %m%n
```

```
log4j.appender.email=org.apache.log4j.net.SMTPAppender
log4j.appender.email.SMTPHost=localhost
log4j.appender.email.From=system@yourapp.com
log4j.appender.email.To=recipient@yourapp.com
log4j.appender.email.Subject=[Pinger Notification] - Website Down
log4j.appender.email.threshold=error
log4j.appender.email.layout=org.apache.log4j.PatternLayout
log4j.appender.email.layout.conversionPattern=%d{ISO8601} %-5p [%c] - %m%n
```

这样就设置好了输出至 pinger.log 文件的标准日志。此外，当出现任何级别为 error 的日志时，还会发送一封邮件提醒。在我们的例子中，如果某个网站没有回复 HTTP 200 响应，或是检查该站点的过程中抛出了异常，就会激发邮件提醒。别忘了更改邮箱地址信息，这样才能在你的环境中生效。

让我们更新一下代码，加入日志功能。这里的目标是把所有 println 语句替换成日志消息。下面打开 core.clj，把 clojure.tools.logging 中的 info 和 error 函数添加到你的命名空间声明中，并创建一个函数用来记录结果。

```
(ns pinger.core
  (:import (java.net URL))
  (:require [pinger.scheduler :as scheduler]
            [clojure.tools.logging :as logger])
  (:gen-class))
...
(defn record-availability [address]
  (if (available? address)
    (logger/info (str address " is responding normally"))
    (logger/error (str address " is not available"))))
```

另外还要更新 check 函数，以反映上述变化。

```
(defn check []
  (let [addresses '("http://google.com"
                    "http://google.com/404"
                    "http://amazon.com")]
    (doseq [address addresses]
      (record-availability address))))
```

再次重新构建你的程序，并试一下看看。你应该会注意到有一个新创建的 logs 目录，你可以查看一下程序的执行状况。另外，你应该也注意到了，有一封带有错误信息的邮件送达你的邮箱。如果你得到了一个在端口 25“connection refused”（连接被拒绝）的错误，那么就需要在你的机器上设置一个邮件传输代理，以启用邮件发送功能。好了，现在你已经有办法通知大伙儿，网站失效了！

9.5.3 配置

我们硬编码了待监测站点列表，然而这是不可行的！我们需要一种能从某些外部资源获得待监测站点的方法。我们可以使用的方式包括属性文件、数据库，或是 web 服务。为了便于说明，我们会采用属性文件。接下来让我们在应用的根目录下，创建一个名为 pinger.properties 的文件，并添加以下内容。

```
urls=http://google.com,http://amazon.com,http://google.com/badurl
```

我们还需要采用某种方式来加载这个文件，并创建一个保存了站点的容器，提供给 check 函数。让我们在 src 目录中创建一个名为 config.clj 的文件。

```
(ns pinger.config
  (:use [clojure.java.io :only (reader resource)])
  (:require [clojure.string :as str])
  (:import (java.util Properties)))

(defn load-properties [src]
  (with-open [rdr (reader src)]
    (doto (Properties.)
      (.load rdr))))

(defn config
 []
  (load-properties (resource "pinger.properties")))
```

只要 pinger.properties 在 classpath 中，这个 config 函数就会把 pinger.properties 读入一个 Java 属性对象中。我们剩下的工作，就是获取 urls 属性，并放进一个列表中。把下面的这个函数添加至 config 命名空间。

```
(defn urls [conf]
  (str/split (.get conf "urls") #","))
```

最后，更新 core.clj 中的 check 函数，以利用这个新的配置函数。

```
(ns pinger.core
  (:import (java.net URL))
  (:require [pinger.scheduler :as scheduler]
            [clojure.tools.logging :as logger]
            [pinger.config :as config])
  (:gen-class))

...

(defn check []
  (doseq [address (config/urls (config/config))]
    (record-availability address)))
```

重新构建你的应用，然后试一下看看。请别忘了要把根目录也放进 classpath 中，这样应用才能找到 pinger.properties。

```
java -cp .:pinger-0.0.1-standalone.jar pinger.core
```

现在我们成功地获得了我们想要的东西。在这个例子中，我们介绍了下述内容。

- 使用 Java 的 URL 对象来检查网站，以查看它是否可用。
- 使用 Java 的 ScheduledTheadPoolExecutor 对象来创建定期执行的任务。
- 使用 log4j 和 clojure.tools.logging 来发送错误通知。
- 使用 Java 的属性系统进行配置。
- 使用 Leiningen 来创建可执行的独立 JAR 文件。

通过扩展这个例子，我们能做到不少事情。我们可以通过增加需求，来重新定义所谓网站可用的含义，比如说要求必须存在某些 HTML 元素，或是要求响应必须在某个特定的时间内返回，以确保满足某种 SLA（service level agreement，服务水平协议）。你不妨试试看，把它们加进这个例子中，看看你还能想到什么。

9.6　小结

我们讨论了 Clojure 与 Java 的相处之道。我们甚至用一种有趣的方式，把它们两者进行了混合。既然我们已经开始体验 Clojure 语言以外的东西，那么，我们可能遇到一个不错的时机，可以开始讨论那些用来构建真实世界中 Clojure 应用的库了。下一章我们就打算这么做。

第 10 章

搭建应用

现在，你已经学会了 Clojure 语言的基本知识，是时候把 Clojure 用在你自己的项目中了。但是，当你冲出门去为你的杀手级 Clojure 应用开始工作时，你很快就会发现，在整个 Clojure 生态系统中，需要你能够灵活运用的东西有很多，语言知识仅仅只是一部分而已。你会遇到下面的问题。

- 我应该使用什么工具来组织项目和依赖项？
- 什么才是良好的代码编写流程？
- 我如何才能确保自己的代码是正确的？
- 我怎样才能保持代码灵活且易于维护？
- 我需要什么样的库？
- 我如何才能把 Clojure 放到万维网上？

对于这几个问题，永远都没有唯一的、放之四海皆准的答案。Clojure 运行于 Java 虚拟机之上。这是一个庞大的生态系统，有成百上千个流派正处于全盛时期。但你总得从某个地方开始。本章中，我们会为你提供这些问题的临时答案。这些答案对于我们而言非常有效，在较短的时期内，你不妨也使用它们。随着个人偏好以及感受的演变，你可能会适应本章中的这些方法，或者将它们抛弃，用对你而言最合适的方法加以替代。

作为示例应用，我们将会实现一个万维网版本的 Clojure 电码译员游戏（Clojurebreaker game，）。在这个电码译员游戏中，电码制造者（程序）创建一个正序（N-ordered）的彩钉密码。然后电码译员（人类玩家）提交猜测的结果。电码制造者对玩家猜测的结果计分规则如下。

- 每当有一个颜色和位置都正确的彩钉，就得到一枚黑色钉。
- 每当有一个颜色正确但位置有误的彩钉，就得到一枚白色钉。

全都猜对了，或是猜错次数达到了额定的上限之后，游戏结束。

虽然过程中我们会向你展示所有的这些代码，但本章真正关注的并非代码。而是关于如何处理问题的一种风格，以及关于如何在 Clojure 生态系统中交付解决方案的一些细节。让我们开始吧。

10.1　为电码译员游戏计分

作为 Clojure 程序员，一个你经常会提出的问题是：“为了解决这个问题，我在什么地方必须用到状态？”或者，更好一点的是：“对于这个问题，有多少是我无须借助任何状态就能解决的？”

对于电码译员游戏（以及其他许多游戏）而言，游戏逻辑本身就是个纯粹的函数。它接受一个密码和一个猜测结果，并返回得分。因为很早就识别出了这个事实，因此我们获得了两个好处。

- 编写独立的计分函数的测试，将会非常容易。
- 我们可以轻松地开始实现计分功能，甚至于不必考虑系统其余的部分应如何工作。

计分本身又可以分为两个部分：对精确匹配进行计数；以及对顺序错乱的匹配进行计数。每个部分都可以有它们自己的函数。让我们先从精确匹配开始。具体来说，为了方便在 REPL 中进行尝试，我们会为彩钉挑选一组恰当的表现形式：:r（红色）、:g（绿色）、:b（蓝色）和:y（黄色）四种颜色。这个函数会返回精确匹配的彩钉数量，这样我们随后就可以在单独的步骤中，用黑色钉来展示结果。我们认为这个函数的样子应该是这样的。

clojurebreaker/snippets.clj

```
(defn exact-matches
  "Given two collections, return the number of positions where
   the collections contain equal items."
 [c1 c2])
```

等等——文档中并没有说明任何关于游戏、颜色，或是关键字的内容。这究竟是怎么回事？尽管有些调用者（例如，game 函数）最终会关心游戏状态的表现形式，但

exact-matches 函数本身并不关心。所以，让我们尽可能的保持通用吧。负责任的 Clojure 设计的一个关键部分，就是要从数据的角度来思考，而不是一逮到机会，就倾泻出大量具体的对象。

当被描述为一个面向数据的通用函数时，exact-matches 这个名字听起来就像是某个可能已经存在的函数似的。于是，通过在相关的命名空间（clojure.core 和 clojure.data）中进行搜索之后，我们发现和 exact-matches 最接近是 clojure.data 中的 diff。diff 会递归的比较两个数据结构，并返回一个三元组：只存在于 a 的、只存在于 b 的、和两者共有的。这个“两者共有的”部分，不就是我们想要的精确匹配吗。赶快在 REPL 中试一下看看。

```
(require '[clojure.data :as data])
(data/diff [:r :g :g :b] [:r :y :y :b])
-> [[nil :g :g] [nil :y :y] [:r nil nil :b]]
```

[:r nil nil :b]中那些非 nil 的元素，就是当 r/g/g/b 与 r/y/y/b 进行比较时，得到的精确匹配。有 diff 在手，exact-matches 的实现就微不足道了。

clojurebreaker/src/clojurebreaker/game.clj

```
(defn exact-matches
  "Given two collections, return the number of positions where
  the collections contain equal items."
 [c1 c2]
  (let [[_ _ matches] (data/diff c1 c2)]
    (count (remove nil? matches))))
```

同样，我们在 REPL 中用这个例子再进行一次测试。

```
(exact-matches [:r :g :g :b] [:r :y :y :b])
2
```

现在，让我们把注意力转移到错序匹配上。要计算出结果，我们需要知道每种颜色的彩钉，在密码和猜测结果中都分别都有多少个。听起来这似乎就是 frequencies 函数要干的活。

```
(def example-secret [:r :g :g :b])
(frequencies example-secret)

-> {:r 1, :g 2, :b 1}

(def example-guess [:y :y :y :g])
(frequencies example-guess)

-> {:y 3, :g 1}
```

要把这两句 frequencies 变成 unordered-matches，我们还需要做两件额外的事情。

- 只考虑同时出现于密码和猜测结果中的键。
- 仅对重叠部分进行计数（即从属于每个键的最小的那个值）。

同样，我们希望这些操作已经存在了，并很乐意交给它们去做。你可以用 select-keys 来筛选出你需要的键。

```
(select-keys (frequencies example-secret) example-guess)
-> {:g 2}

(select-keys (frequencies example-guess) example-secret)
-> {:g 1}
```

此外你可以使用 merge-with 来对这两个频次映射表的重叠部分进行计数。

```
(merge-with min {:g 1} {:g 2})
-> {:g 1}
```

把 frequencies、select-keys 和 merge-with 组合起来，就得到了下面的这个 unordered-matches 定义。

clojurebreaker/src/clojurebreaker/game.clj

```
(defn unordered-matches
  "Given two collections, return a map where each key is an item
  in both collections, and each value is the number of times the
  value occurs in the collection with fewest occurrences."
 [c1 c2]
  (let [f1 (select-keys (frequencies c1) c2)
        f2 (select-keys (frequencies c2) c1)]
    (merge-with min f1 f2)))
```

当然，我们应该在 REPL 中验证一下。

```
(unordered-matches [:r :g :g :b] [:y :y :y :g])
-> {:g 1}
```

这真是太棒了，只是还有一个细微的差异。unordered-matches 计算的匹配数没有顾及到顺序，然而这个游戏现在想要知道的，只是那些顺序不正确的匹配。尽管这个游戏看起来并不需要 unordered-matches，但把它编写出来也是一场胜利，原因如下。

- unordered-matches 做了一整件事情。为了编写“顺序错误”的匹配，我们将不得不在 unordered-matches 的内部，重新实现 exact-matches。
- 刚才编写的那两个简单的函数，我们只要把它们组合起来，就可以获得“顺序错误”的语义，这正是我们需要的。只要从 unordered-matches 的结果中减去 exact-matches 的结果即可。

随着这两个原语就绪，计分操作 score 只需要简单的将它们给混合起来。

clojurebreaker/src/clojurebreaker/game.clj

```
(defn score
 [c1 c2]
  (let [exact (exact-matches c1 c2)
        unordered (apply + (vals (unordered-matches c1 c2)))]
    {:exact exact :unordered (- unordered exact)}))
```

在 REPL 中庆祝一下。

```
(score [:r :g :g :b] [:r :y :y :g])
-> {:exact 1, :unordered 1}
```

“什么才是良好的代码编写流程？”此刻，对于这个问题，我们已经展示了部分答案。概括如下。

- 对问题进行分解，识别出纯函数。
- 学习标准库，这样你就能发现那些已经写好的功能。
- 不要让实体泛滥（数据就得当作数据来用）。
- 在 REPL 中做彻底的测试。

根据我们的经验，程序员在第一次尝试这种流程时喜欢犯两个典型的错误。

- 编码过多
- 把测试搞复杂了

每当你搞不清楚某个形式会产生什么样的行为时，就说明你一次编写了太多的代码，但还没有完全测试和理解这个形式的所有子形式。许多开发者都有一种直觉：“写 X 行代码，就要进行测试”。这个 X 是为了完成某些实质性的事情，所需的最少代码行数。在 Clojure 中，X 明显要小于 1。这也就是为什么我们一再强调，要从里到外地在 REPL 中构建函数。

“把测试搞复杂了”则更加微妙，我们会在下一节中讨论它。

10.2 测试计分器

上一节，我们在 REPL 使用迭代的方式，开发了计分函数 score，并且看到对于少量的几个样例输入，它是正确的。但这并不足以保证其质量水平，我们需要进行更充分的测试！当人们说“测试”时，总是意味着某些东西，让我们从梳理这些含义开始吧。所谓测试，包括了下述内容。

- 想清楚代码是否正确。
- 在开发环境中单步调试代码，这样你可以看到正在发生的一切事情。
- 手工构造输入，以覆盖不同的代码路径。
- 手工构造输出，以匹配手工构造的输入。
- 用各式各样的输入来运行代码。
- 验证结果的正确性。
- 让结果的验证自动化。
- 对测试进行组织，以便将来可以自动地开展回归测试。

很难说这个清单已经足够详尽了，但它足以指出本节的要点所在。简而言之，测试往往是复杂的过程，但它其实是可以变得简单的。

传统的单元测试方法会把上面列出的许多测试任务交织到一起。例如，输入、输出、执行和验证往往被裹在一起，放进单个的测试方法当中。另一方面，我们之前进行的最小 REPL 测试又过于简单。那么，可不可以既从前面关于测试的某些思路中获益，而又不必背负单元测试的复杂性呢？让我们试试看吧。

10.2.1　手工构造输入

我们已经见过用几个手工构造的输入，让计分函数 score 工作的情景了。那么我们究竟需要多少输入，才能让自己信服，这个函数是正确的呢？在完美的世界中，我们只要对所有可能的输入进行测试就可以了，但从运算量的角度来看，这几乎总是不可行的。但值得庆幸的是，对于电码译员游戏的计分问题而言，本质上，它与为各种不同数量的颜色，或是不同数量的彩钉计数是相同的。鉴于此，事实上我们的确可以为该游戏的较小版本，生成“所有”可能的输入。

有一个名为“计数组合学”（enumerative combinatorics）的数学分支，涉及不同的形成模式（forming patterns）方法。Clojure 中，math.combinatorics 库就有我们需要的函数，可以用来生成所有可能的输入。如果你的 project.clj 中没有下面这行，那就把它添加到键:dependencies 之下吧。

```
[org.clojure/math.combinatorics "0.0.1"]
```

selections 函数接受两个参数（一个容器和一个尺寸），并根据传入的尺寸，返回容器元素所有可能的组合。下面用一个只有三个颜色，且只有两列的微型电码译员游戏来试一下。

```
(require '[clojure.math.combinatorics :as comb])
(comb/selections [:r :g :b] 2)
-> ((:r :r) (:r :g) (:r :b)
    (:g :r) (:g :g) (:g :b)
    (:b :r) (:b :g) (:b :b))
```

因此，selections 可以为我们创造出密码或是玩家的猜测结果。让我们为 score 函数生成一些输入怎么样？只要基于前面 selections 的结果，再用 selections 选出成对的组合即可。

```
(-> (comb/selections [:r :g :b] 2)
    (comb/selections 2))
-> (一共输出了 81 对游戏位置，为简洁起见此处省略)
```

让我们把它放进一个命名函数中。

clojurebreaker/src/clojurebreaker/game.clj

```
(defn generate-turn-inputs
  "Generate all possible turn inputs for a clojurebreaker game
  with colors and n columns"
 [colors n]
  (-> (comb/selections colors n)
      (comb/selections 2)))
```

好啦，输入这就生成了。接下来本应考虑输出了，但我们打算跳过这个过程（原因嘛，很快就会明白了），并把我们的注意力转向计分器的运行，当然，这次用的是我们生成的输入。

10.2.2 运行测试

我们将要编写一个函数，它接受一个由输入组成的序列，并返回由这些输入和调用 score 得到的结果组成的序列。我们还不打算承诺如何去验证运行测试得到的结果。也许会让一个人去读它。也许会有一个验证程序处理这些结果。总之，无论采用哪种方式，用来展现结果的最佳途径，可能都是映射表，这个映射表应该有密码（:secret）、猜测结果（:guess）和得分（:score）这几个键。

这个函数要做的就是调用 score，并创建上面提到的那个结果映射表。

clojurebreaker/src/clojurebreaker/game.clj

```
(defn score-inputs
  "Given a sequence of turn inputs, return a lazy sequence of
```

```
  maps with :secret, :guess, and :score."
 [inputs]
  (map
    (fn [[secret guess]]
      {:secret (seq secret)
       :guess (seq guess)
       :score (score secret guess)})
   inputs))
```

在 REPL 中试一下看看。

```
(->> (generate-turn-inputs [:r :g :b] 2)
     (score-inputs))
-> ({:secret (:r :r), :guess (:r :r),
     :score {:exact 2, :unordered 0}}
    {:secret (:r :r), :guess (:r :g),
     :score {:exact 1, :unordered 0}}
   ;; 简洁起见，其余省略
```

如果有人打算要阅读这个测试报告，你可能会使用 print-table 来把得分结果格式化为文字表格，以获得更好的阅读体验。进行到这儿，让我们生成一个更大的游戏（四种颜色，有四列），并把表格打印到一个文件中。

```
(use 'clojure.pprint)
(require '[clojure.java.io :as io])
(with-open [w (io/writer "scoring-table")]
  (binding [*out* w]
    (print-table (->> (generate-turn-inputs [:r :g :b :y] 4)
                      (score-inputs)))))
-> nil
```

如果你去查阅这个名为 scoring-table 的文件，你应该能看到 65,536 组不同的密码、猜测结果以及得分。

10.2.3　验证输出

至此，我们之所以要跳过手工构造输出的原因已经真相大白。这段程序已经为我们做了。我们只需要决定花费多少努力来验证它们。这里有一些我们可能会采用的方法。

- 让一个人类的电码译员专家，去阅读该游戏的一个微型变种的整个输出表。这么做的好处是能够检查的比较彻底，但有可能错过一些仅会出现于更大的游戏当中的逻辑错误。
- 从更大的游戏中随机挑选出一组结果，并由人类专家对其验证。

因为验证这个步骤，与生成输入和运行程序是分开的，所以我们可以独立的设计和编写这些步骤，甚至很可能会在不同的时间进行。

此外，验证者对于输入是如何生成的一无所知。而单元测试呢，输入和输出几乎同时诞生于同一个程序员的脑海中。如果那个程序员系统性的弄错了某些东西，那么这样的测试只不过是把错误当作正确进行编码罢了。当验证的输出是穷举或者随机产生的，这样的事情就不可能发生。

稍后我们还会回到程序化的验证上来，但首先让我们转向回归测试。

10.2.4 回归测试

你是多么想要拥有这样一个回归套件，它能做的验证，比之前你已经做过的那些努力还要全面。没问题。

- 编写一个程序，其运行结果是不应该发生变化的。
- 运行一次这个程序，把结果保存到（有个合适名称的）文件中。
- 每次更改了这个程序以后，就运行它一次，并将结果与之前保存的文件进行比较。如果发现出现了不同，那就说明这个程序出问题了。

关于这种回归方法，好的方面是，即使你从来没有对结果进行过任何的验证，它依然有效。当然，你还是应该要做验证，因为这样能帮助你在定位问题时缩小范围。没有做验证的话，回归出错也许只能告诉你：代码被破坏了，需要修复。

编写一个程序，要产生完全相同的输出，究竟有多难？只要这个程序调用的是纯函数就可以，而这正是我们的 score-inputs 函数所做的。

编写这种类型的回归测试，并把它放进持续集成的构建中并不困难。如果你这样做了，不妨考虑一下把它贡献给你正在使用的测试框架吧。

现在，对于“我如何才能确保自己的代码是正确的？”，我们已经部分回答了这个问题，概括如下。

- 创建体积小、可组合的部件（大部分应该是纯函数）。
- 在 REPL 中进行彻底的测试。
- 编写测试代码时，保持生成输入、执行和验证输出都是独立的步骤。

最后一点尤其重要，值得用某个库去支持它。所以，在继续前进之前，我们打算引入 test.generative，一个致力于简化测试的库。

10.3　test.generative

test.generative 把测试划分为三个关键的步骤。

- 生成测试输入。
- 调用测试函数。
- 验证结果。

这三个步骤的每一步，都通过函数来实现，然后通过 defspec 宏，它们会被组合成完整的测试。

让我们安装 test.generative，并带它出去兜个风吧。在你的 project.clj 中添加下面这行。

```
[org.clojure/test.generative "0.1.3"]
```

现在重新加载你的项目，并和我们一块儿，在 REPL 中生成一些测试数据吧。

10.3.1　生成数据

在 REPL 中，用 require 导入 generators 命名空间如下。

```
(require '[clojure.test.generative.generators :as gen])
```

generators 命名空间包含一些函数，用来为各种常见数据类型生成伪随机值。例如，对应每一种 Java 基础类型，都存在一个同名的生成函数。来选几个试试。

```
(gen/int)
-> 977378563

(gen/char)
-> \A

(gen/boolean)
-> true
```

注意，你得到的结果可能与此处显示的不同，因为这些值是随机的。

你也可以为各种 Clojure 容器类型生成随机值。这些函数会根据参数化的类型来创建容器。示例如下。

```
(gen/vec gen/boolean)
-> [false false true true true false]

(gen/hash-map gen/byte gen/int)
-> {32 -504310803, 100 424501842, 5 1439482147, 37 1161641068}
```

除了这些基本类型，你还可以使用几个“旋钮”来控制这些类型应如何生成。你可以选择分布的概率。

```
(gen/geometric 0.02)
-> 10
```

你也可以使常量或是某种分布，来控制容器的大小。

```
(gen/list gen/int 2)
-> (-1029960512 1985289448)

(gen/list gen/int (gen/uniform 0 5))
-> (315829211)
```

还有其他一些好东西，不过我们不想夺走你探索的乐趣。你可以对这个命名空间使用 dir，并自己摸索一下别的那些标准生成函数吧。

为了测试电码译员游戏的计分功能，我们希望能有一个函数，可以生成随机的密码（或者猜测结果）。沿用那些表示彩钉颜色的关键字，这个函数可能会像下面这样。

clojurebreaker/test/clojurebreaker/game_test.clj

```
(defn random-secret
 []
  (gen/vec #(gen/one-of :r :g :b :y) 4))
```

在我们开始考虑验证之前，可别忘了要在 REPL 中试试这个 random-secret 哦。

10.3.2 程序化验证

程序化验证，不应该仅仅只对一些手工编写的特定值进行抽样检查。当我们可以很容易地检查一组输出的有效性，并把它们保存为回归测试时，这样的抽样检查也算是一种可怜的应时之选。相反，真正的程序化验证，应该保证输入数据与输出数据之间的逻辑恒定。

下面是几个电码译员游戏计分函数的恒定属性。

- 计分是对称的。如果你反转了密码和猜测结果参数，分数应该是一样的。
- 精确匹配和错序匹配的总数，必须大于或等于零，并小于或等于彩钉的数量。
- 如果你打乱了猜测结果中彩钉的顺序，得分中精确匹配和错序匹配的总和不会发生改变。

让我们把这几个不变式编码为布尔函数 secret、guess 和 score。

clojurebreaker/test/clojurebreaker/game_test.clj

```
(ns clojurebreaker.game-test
  (:use [clojure.test.generative :only (defspec) :as test])
  (:require [clojure.test.generative.generators :as gen]
            [clojurebreaker.game :as game]
            [clojure.math.combinatorics :as comb]))

(defn matches
  "Given a score, returns total number of exact plus
   unordered matches."
  [score]
  (+ (:exact score) (:unordered score)))

(defn scoring-is-symmetric
  [secret guess score]
  (= score (game/score guess secret)))

(defn scoring-is-bounded-by-number-of-pegs
  [secret guess score]
  (< 0 (matches score) (count secret)))
(defn reordering-the-guess-does-not-change-matches
  [secret guess score]
  (= #{(matches score)}
    (into #{} (map
                #(matches (game/score secret %))
                (comb/permutations guess)))))
```

虽然我们常常会先在 REPL 中输入代码，然后再把代码粘贴到适当的文件中，不过这回我们采用了另一种方法，一开始就把前面的这些不变式函数添加到了源文件当中。但这并不意味着我们失去了 REPL 立即可得的特性。在 REPL 中，我们可以重新加载这个命名空间，并用 in-ns 进入其中。

```
(require :reload 'clojurebreaker.game-test)
(in-ns 'clojurebreaker.game-test)
```

现在，我们已经身处我们想要使用的命名空间当中，我们可以尽情访问其所有公共的名称和别名。这就非常便于手工构造一些样本数据，对这些验证函数进行一次快速的合理性检查。

```
(def secret [:r :g :g :b])
(def guess [:r :b :b :y])

(scoring-is-symmetric secret guess
  (game/score secret guess))
-> true

(scoring-is-bound-by-number-of-pegs
  secret guess (game/score secret guess))
-> true

(reordering-the-guess-does-not-change-matches
  secret guess (game/score secret guess))
-> true
```

有了测试数据生成器和这些验证函数在手，接下来我们就准备在一个 defspec 中，把所有这一切都给连接起来。

10.3.3 defspec

defspec 需要三个必不可少的参数。

- 规格的名称。
- 要测试的函数。
- 要生成的参数。

然后是零个或者多个执行验证的主体形式。下面是一个微不足道的例子。

clojurebreaker/snippets.clj

```
(defspec closed-under-addition
  +'
  [^long a ^long b]
  (assert (integer? %)))
```

这些参数看起来就像是普通的 Clojure 参数，但其中有一个变形。普通函数的参数，其类型提示是可选的。然而规格的参数，则必须进行类型标记，以提示应该如何生成测试参数。

test.generative 中的 generate-test-data 函数，接受一份参数的规范，并据此生成一个无限的测试数据序列。你可以脱离任何测试，直接调用 generate-test-data 来单独生成参数。

```
(take 3
  (test/generate-test-data '[long long]))

-> ([-5025562857975149833 -5843495416241995736]
    [5694868678511409995 5111195811822994797]
    [-6169532649852302182 -1782466964123969572])
```

和一个普通的函数一样，defspec 的主体可以通过名称来访问参数。此外，它还可以访问%，%持有着调用测试函数得到的返回值。在 closed-under-addition 的例子中，唯一的验证“形式”仅检查了相加的结果是否为一个整数，它甚至都没花功夫看输入参数一眼。

```
(assert (integer? %))
```

而更加精细的验证，通常都会同时涉及到输入参数和结果。

10.3.4　运行测试

规格仍然是函数，所以你可以简单的通过函数调用来运行它们。如果你在阅读上一节的时候，已经在 REPL 中对 closed-under-addition 规格求过值，那么现在就可以调用它了，下面传入几个整数。

```
(closed-under-addition 1 2)
-> nil
```

但更有趣的事情是，用自动生成的输入来运行这个规格。test.generative 中有三个辅助函数可以用来做这件事情。

- test-vars 接受一个或多个变量，并运行这些变量所指向的规格。
- test-dirs 接受一或多个目录，并找出这些目录中的所有规格，然后运行它们。
- test-namespaces 接受一或多个命名空间，并找出这些命名空间中的所有规格，然后运行它们。

下面让我们来运行一下 closed-under-addition 规格看看。

```
(require '[clojure.test.generative :as test])
(test/test-vars #'closed-under-addition)

-> [#<core$future_call$reify__5684@603a3e21: :pending>
    #<core$future_call$reify__5684@fc519e2: :pending>]
```

调用 test-vars 会返回一定数量的 future 对象（前面输出的是两个）。在随后的大约十秒钟内什么也没有发生，然后 REPL 会输出类似于下面这样的信息。

```
{:iterations 179766,
  :msec 10228,
  :var #'clojurebreaker.game-test/closed-under-addition,
  :seed 43}
{:iterations 156217,
  :msec 10249,
  :var #'clojurebreaker.game-test/closed-under-addition,
  :seed 42}
```

在后台，test.generative 为你的每个处理器核心都创建了一个 future 对象。每个 future 都会在十秒钟内尽可能多的运行测试。那个后来的输出，显示了在每个线程上都进行了多少次迭代；完成每个线程都耗去了多少时钟时间；为每个测试的变量生成数据时采用的随机种子是什么。

test.generative 对外暴露了几个动态变量，用来自定义测试的运行。你可以绑定

test/*msec*以改变测试运行的时长，或者绑定 test/*cores*以改变运行测试所用的核心数量。你还可以绑定 test/*verbose*来把测试运行过程中所有输入都打印至*out*。

10.3.5 规格执行失败时

当某个规格失败时，test.generative 提供了额外的输出信息来帮助你重现问题。下面让我们来编写一个规格，由于两数之和小于其中的任意一个加数，将会导致断言失败。

clojurebreaker/snippets.clj

```
(defspec incorrect-spec
  +'
  [^long a ^long b]
  (assert (< a %))
  (assert (< b %)))
```

如果你运行 incorrect-spec，它很快就会失败，并显示下面这样的错误。

```
{:form (#'clojurebreaker.game-test/incorrect-spec
        -5025562857975149833 -5843495416241995736),
 :iteration 0,
 :seed 42,
 :error "Assert failed: (< a %)",
 :exception #<AssertionError java.lang.AssertionError:
             Assert failed: (< a %)>}
```

键:form 指出了引发失败的那个输入，iteration 则告诉你究竟是哪次迭代导致了失败。因为用伪随机生成测试数据时使用的那个种子是清楚明白的，所以你只需要简单的运行整个测试就可以复现该故障（有可能后来还会添加日志或者附加调试器什么的）。

如果你不希望仅为了回到故障点而重复运行整个测试的话，那么你可以通过调用 failures 函数来访问一个容器，这个容器中的成员都是引发错误的那些形式。

```
(test/failures)
-> ({:form
     (#'clojurebreaker.game-test/incorrect-spec
       -5025562857975149833 -5843495416241995736)}
     {:form
      (#'clojurebreaker.game-test/incorrect-spec
        -5027215341191833961 -2715953330829768452)}
```

有了这个，就可以通过遍历容器，轻而易举地对曾经失败的输入重新求值。

```
(eval (:form (first (test/failures))))
-> AssertionError Assert failed: (< a %)
```

至此，我们已经编写并运行了几个规范，并且也看到了当情况变糟时应如何进行探索。接下来我们会把这些技术用于测试我们游戏的计分函数。

10.3.6　计分函数的产生式测试

在前面的“程序化验证”一节中，我们创建了几个函数，对电码译员游戏计分中的不变式进行了验证。现在让我们把它们添加到一个规格当中。

clojurebreaker/test/clojurebreaker/game_test.clj

```
(defspec score-invariants
  game/score
  [^{:tag `random-secret} secret
   ^{:tag `random-secret} guess]
  (assert (scoring-is-symmetric secret guess %))
  (assert (scoring-is-bounded-by-number-of-pegs secret guess %))
  (assert (reordering-the-guess-does-not-change-matches secret guess %)))
```

这个规范展示了一点点 defspec 的新语义：语法引述（`）符号的使用。对 random-secret 的语法引述，标志着生成函数是由我们自行编写的自定义版本。参数规范中对名称的引述，有以下两条规则。

- 如果一个名称没有进行语法引述，那么该名称被解释为命名空间 clojure.test.generative.generators 中的内建生成器。这些名称恰好就是源自 clojure.core 的影子类型。这也是为什么这些内建的生成器，看起来就好像是类型提示一样。
- 如果一个名称进行了语法引述，那么对该名称的解释，将会遵循 defspec 所处命名空间的规则。这就允许你使用普通的命名空间工具，来管理你自己的生成器了。

关键时刻终于到来了。当我们运行这个规格时，不变式仍然能够保持其不变性吗？

```
#'clojurebreaker.game-test/score-invariants
-> [#<core$future_call$reify__5684@590e130c: :pending>
    #<core$future_call$reify__5684@2b04a681: :pending>]

;; 十秒钟过后
{:iterations 1794, :msec 10002, :seed 42,
 :var #'clojurebreaker.game-test/score-invariants}
{:iterations 1787, :msec 10001, :seed 43,
 :var #'clojurebreaker.game-test/score-invariants}
```

看起来不错。你还能想到其他有趣的不变式用来测试吗？

此刻，我们只是碰到了 test.generative 的一些皮毛而已。而且，虽然我们喜欢把输入生成、执行和验证加以分离，但在 Clojure 中，使用你更熟悉的单元测试、TDD 和 BDD 方法当然也没有任何问题。如果你对它们有兴趣，不妨调查以下这些库。

- clojure.test[①]内建于 Clojure，是一个朴素的单元测试库，拥有测试套件。
- Lazytest[②]是一个相当通用的库，基于几个泛化的测试陈述，支持不同的测试风格。
- Midje[③]强调测试的可读性，并同时允许自顶向下和自底向上两种不同风格的测试。

test.generative 中很多想法的灵感来自于 QuickCheck[④]，一个最初使用 Haskell 编写，现在已经被移植到很多语言的测试库。如果你发现 test.generative 有点意思，那当然也应该要了解一下 QuickCheck。它有相当长的发展历史，并且它的许多能力是目前这个 test.generative 尚不具备的。

现在我们对自己的计分函数更有信心了，随后让我们开始考虑在万维网上运行电码译员游戏吧。

10.4 创建用户界面

有了坚实的基础，为游戏引入界面并令其可玩，应该是一件轻而易举的事情。让我们从一个基本的万维网应用开始。对于我们的应用而言，web 框架 nori 是一个相当不错的基础。创建一个新的 noir 项目非常容易。

首先，你需要为 leiningen 安装 lein-noir 插件。

```
$ lein plugin install lein-noir 1.2.0
```

现在，你可以生成我们的应用并启动它了。

```
$ lein noir new clojurebreaker
$ cd clojurebreaker
$ lein run
```

让你的浏览器指向 http://localhost:8080。你会得到默认的 noir 欢迎页面，目前这对你来说还算不上很有趣，但它能确保你进行了正确的设置。

① http://clojure.github.com/clojure/clojure.test-api.html。
② https://github.com/stuartsierra/lazytest。
③ https://github.com/marick/Midje。
④ http://en.wikipedia.org/wiki/QuickCheck。

10.4.1　该是时候处理状态了

这里是我们唯一一个需要引入状态的地方。因为我们希望能有多个玩家同时享受这个游戏。此外我们还希望能在多个请求之间维持游戏的正确答案。因此，我们就得找个地方保存这个值。有个好消息，我们不必在自己的代码中管理它。我们可以把它放进浏览器的会话中，这样就能让我们的应用代码摆脱状态管理了。

首先让我们打开 src/clojurebreaker/views/welcome.clj。把页面/welcome 改为/。如果会话中缺少某些东西，我们需要一种方法把它们给放进去。幸运的是，noir 有 session/put!和 session/get。这就是我们想要的。

```
(defpage "/" []
  (when-not (session/get :game )
    (session/put! :game (.nextInt (java.util.Random.) 1000000)))
  (common/layout
   [:p "Welcome to clojurebreaker.
        Your current game id is " (session/get :game )]))
```

如果你刷新浏览器，现在会看到会话中出现了一个你的游戏标识。如果你继续刷新的话，这个标识不会发生改变。但如果你清除了会话，并再次刷新的话，就应该会看到一个新的游戏标识。我们会一直使用这种技术，不过我们并不需要随机生成的标识。那只不过是为了证明会话存储可以正常工作罢了。

在 src/clojurebreaker/models 中创建一个新文件 game.clj。我们会在这里放置一个必要的函数，用于生成新的游戏密码，并把结果返回给视图。

```
(ns clojurebreaker.models.game)

(defn create []
  (vec (repeatedly 4 (fn [] (rand-nth ["r" "g" "b" "y"])))))
```

让我们在 REPL 中试一下，确保它产生的结果是我们需要的。

```
(in-ns 'clojurebreaker.models.game)
clojurebreaker.models.game=> (dotimes [_ 5] (println (create)))
| [g y g b]
| [g r r r]
| [r y g r]
| [b y y b]
| [b g r g]
-> nil
```

启动游戏需要的一切东西都已经就绪。我们只需要回到视图部分，把这些代码连接起来即可。请记住，test.generative 同样也可以用来创建游戏密码，所以完全可以用它替代前面的那个函数。此处只是为了演示另外一种创建密码的方式而已。

```
(defpage "/" []
  (when-not (session/get :game)
    (session/put! :game (game/create)))
  (common/layout
   [:p "Welcome to clojurebreaker.
        Your current game solution is " (session/get :game)]))
```

另外，别忘了在你的命名空间声明中，用 require 导入 clojurebreaker.models.game，并用“:as game”为其创建别名 game。当你刷新浏览器时，就会看到游戏的正确答案被打印出来了。如果你发现显示的仍然是会话标识，那就需要清除浏览器的会话数据，并再次刷新。试试看新打开一个浏览器并访问该应用。你应该会看到一个与之前不一样的正确答案。

10.4.2 玩家界面

拥有了游戏状态以后，我们还要为玩家提供一个界面。就从创建游戏面板开始吧。我们需要向命名空间声明中添加 hiccup.form-helpers，导入 hiccup 其余的那些部分。现在我们会用到的是 noir 的 defpartial 宏。

```
(defpartial board []
  (form-to [:post "/guess"]
    (text-field "one")
    (text-field "two")
    (text-field "three")
    (text-field "four")
    (submit-button "Guess")))
```

在这里，defpartial 非常之有用，因为它用起来，就好像是个普通的 Clojure 函数一样。这样当我们连接计分部分时，它就能派上大用场。下面在 REPL 中试一下看看。

```
clojurebreaker.models.game=> (in-ns 'clojurebreaker.views.welcome)
clojurebreaker.views.welcome=> (board)
| "<form action=\"/guess\" method=\"POST\">
|  <input id=\"one\" name=\"one\" type=\"text\" />
|  <input id=\"two\" name=\"two\" type=\"text\" />
|  <input id=\"three\" name=\"three\" type=\"text\" />
|  <input id=\"four\" name=\"four\" type=\"text\" />
|  <input type=\"submit\" value=\"Guess\" /></form>"
```

现在我们只需要进行一下连接。

```
(defpage "/" []
  (when-not (session/get :game)
    (session/put! :game (game/create)))
  (common/layout (board)))
```

因为我们已经验证过，会话部分可以工作，所以，只要移除之前的显示内容，并把它们替换为这个真正的游戏界面即可。现在我们已经有办法让用户进行猜测了。嗯，至少我们已经有一种方式，可以让用户在浏览器中输入了。让我们把服务端也连起来吧。

回到视图，我们需要创建另外一个页面来对 post 到/guess 的请求进行响应。在这儿我们需要做以下几件事。

- 接受四个输入，并把它们传递给计分函数。
- 确定玩家是否赢得了游戏（四个精确匹配）。如果确实如此，则对他们的胜利表示祝贺，并显示开始新游戏的按钮。
- 如果用户尚未取胜，则返回当前精确匹配和无序匹配的数量，以及最后一组输入，并把这些内容显示给用户。

为了对请求进行计分，我们需要加入本章早些时候创建的那些游戏计分函数。让我们把 exact-matches、unordered-matches 和 score 函数添加到我们的模型（model）中。一旦这几个计分函数就绪，我们就可以把 post 的处理器与之相连，并且完成这个游戏第一阶段的开发。

clojurebreaker/src/clojurebreaker/views/welcome.clj

```
(defpage [:post "/guess"] {:keys [one two three four]}
  (let [result (game/score (session/get :game) [one two three four])]
    (if (= (:exact result) 4)
      (do (session/remove! :game)
        (common/layout
          [:h2 "Congratulations, you have solved the puzzle!"]
          (form-to [:get "/"]
                   (submit-button "Start A New Game"))))
      (do (session/flash-put! result)
          (render "/" {:one one
                       :two two
                       :three three
                       :four four
                       :exact (:exact result)
                       :unordered (:unordered result)})))))
```

在这里，我们能检测到玩家是否赢得了游戏并进行了恰当的处理。这儿有几个来自于 noir 中的新玩意儿，是我们还没有介绍过的。defpage 后面的那个:keys 解构，用

来处理从浏览器传过来的参数。session/remove!则正确地做了你认为它应该做的事情。有趣的部分是从 session/flash-put!开始的。它会向会话中添加一些东西，供紧跟着的下一个请求消费，然后从此消失。它与 Ruby on Rails 中的 flash 方法非常相似。最后，渲染函数 render 使用紧随其后的参数来调用页面路由。

只要再做少量的修改，我们就可以准备体验这个游戏啦。我们的页面路由“/”，要能够接受刚才编写的那个 render 传过来的参数。让我们来解决这个问题。首先修改局部页面 board，让其接受并解构传给它的参数，然后渲染结果。

clojurebreaker/src/clojurebreaker/views/welcome.clj

```
(defpartial board [{:keys [one two three four exact unordered]}]
  (when (and exact unordered)
    [:div "Exact: " exact " Unordered: " unordered])
  (form-to [:post "/guess"]
           (text-field "one" one)
           (text-field "two" two)
           (text-field "three" three)
           (text-field "four" four)
           (submit-button "Guess")))
```

这儿我们修改了面板的渲染内容，以响应玩家做出的猜测。我们会在面板上显示猜测的计分结果，还有玩家上一次的猜测内容。这样他们就可以做出调整并继续游戏。让我们在 REPL 中体验一下，看看给定几个输入，它会生成什么样的输出。

```
clojurebreaker.views.welcome=> (board {:one "r"
                                       :two "b"
                                       :three "y"
                                       :four "g"})

| "<form action=\"/guess\" method=\"POST\">
|  <input id=\"one\" name=\"one\" type=\"text\" value=\"r\" />
|  <input id=\"two\" name=\"two\" type=\"text\" value=\"b\" />
|  <input id=\"three\" name=\"three\" type=\"text\" value=\"y\" />
|  <input id=\"four\" name=\"four\" type=\"text\" value=\"g\" />
-> <input type=\"submit\" value=\"Guess\" /></form>"

clojurebreaker.views.welcome=> (board {:one "r"
                                       :two "b"
                                       :three "y"
                                       :four "g"
                                       :exact 2
                                       :unordered 0})

| "<div>Exact: 2 Unordered: 0</div>
|  <form action=\"/guess\" method=\"POST\">
```

```
|  <input id=\"one\" name=\"one\" type=\"text\" value=\"r\" />
|  <input id=\"two\" name=\"two\" type=\"text\" value=\"b\" />
|  <input id=\"three\" name=\"three\" type=\"text\" value=\"y\" />
|  <input id=\"four\" name=\"four\" type=\"text\" value=\"g\" />
-> <input type=\"submit\" value=\"Guess\" /></form>"
```

只需要对/页面的定义进行一点微小的改动，我们就可以准备开始游戏了！

clojurebreaker/src/clojurebreaker/views/welcome.clj

```
(defpage "/" {:as guesses}
  (when-not (session/get :game)
    (session/put! :game (game/create)))
  (common/layout (board (or guesses nil))))
```

这里只是接收了来自 render 的参数，所以我们可以再把它们传给局部页 board。由于 board 的工作方式与标准 Clojure 函数非常类似，所以整个过程微不足道。

在我们带着这个游戏准备出去兜风之前，还有一件事情要做。我们还需要把 math.combinatorics 添加到 noir 项目的 project.clj 文件中。

clojurebreaker/project.clj

```
(defproject clojurebreaker "0.1.0-SNAPSHOT"
            :description "Clojurebreaker game for Programming Clojure 2nd Edition"
            :dependencies [[org.clojure/clojure "1.3.0"]
                           [org.clojure/math.combinatorics "0.0.1"]
                           [org.clojure/test.generative "0.1.3"]
                           [noir "1.2.0"]]
            :main clojurebreaker.server)
```

重新运行 lein deps 和 lein run 命令，你现在可以尽情的玩这个电码译员游戏了。

10.5　部署代码

现在，你有了一个完整的应用，是时候把它放在某个地方，供全世界乐享了。把应用发布到生产环境可能是一项长期而艰巨的任务，涉及到添加代码或是移动部件什么的。幸运的是，我们将会跳过所有这些步骤，直达重点。现在有一种便利的服务，完全支持 Clojure 应用的部署，并且让这个过程简单的要死。它们把自己叫做 Heroku。[①]

为了获得一个 Heroku 账号，你有几个初始步骤需要进行设置，但部署的过程就像

① http://heroku.com。

往源码控制仓库中签入代码一样简单。事实上，那就是你要做的所有工作。部署，就是一个简单的 git push 命令而已。

为了开始部署，你可能需要先注册一个 Heroku 账号。这相当简单的，而且完全免费。Heroku 确实有付费产品，但这个例子应该用不着你花费一毛钱。完成注册后，你还需要在 Heroku 平台上创建一个应用。

10.5.1 Procfile

Heroku 在你的程序中使用一个特殊的文件，来让它知道应该如何启动这个程序。这个文件被称为 Procfile（进程文件）。把包含下述内容的 Procfile 添加到项目的根目录下。

```
web: lein run
```

如果你想要确保所有东西都设置正确了，不妨安装 foreman gem，并运行 foreman start 命令。如果能成功的在本机启动你的应用，那么在 Heroku 上应该也不成问题。

```
foreman start

14:35:28 web.1 | started with pid 34538
14:35:33 web.1 | Starting server...
14:35:33 web.1 | 2011-12-09 14:35:33.042:INFO::Logging to STDERR
14:35:33 web.1 | Server started on port [5000].
14:35:33 web.1 | You can view the site at http://localhost:5000
14:35:33 web.1 | #<Server Server@63ce15f6>
14:35:33 web.1 | 2011-12-09 14:35:33.044:INFO::jetty-6.1.26
14:35:33 web.1 | 2011-12-09 14:35:33.070:INFO::Started SocketConnector@0.0.0.0:5000
```

10.5.2 Heroku 库

Heroku 提供了一个漂亮的途径，来与其平台进行编程方式的交互。这里唯一的麻烦是，如果你从来没有做过，那么它还需要一些额外的设置。你还需要 Ruby 编程语言，连同 RubyGems 一起。你系统中的包管理器应该能为你提供这些东西。如果没有，你可以随时访问 http://ruby-lang.org。

安装 Heroku 库是通过 RubyGems 来进行的。

```
gem install heroku
```

这能提供一个 heroku 命令，你可以用它进行所有基于 Heroku 的交互操作。

10.5.3　Git

下一步，你需要在电码译员游戏的代码基（codebase）内，创建一个 git 仓库。同样，你的系统包管理器应该为你提供一个适当版本的 git。如果没有，你可以访问 http://git-scm.org。

一旦安装了 git，你只需要简单的在电码译员游戏根目录中，执行 git init 命令即可。

```
git init
-> Initialized empty Git repository in~/clojurebreaker/.git/
```

这条命令会为你的应用创建初始 git 骨架。接下来你需要在本地提交代码。

```
git add .
git commit -m "Initial commit"

[master (root-commit) dd4f8a8] initial commit
13 files changed, 65860 insertions(+), 0 deletions(-)
create mode 100644 .gitignore
create mode 100644 project.clj
create mode 100644 resources/public/css/reset.css
create mode 100644 src/clojurebreaker/game.clj
create mode 100644 src/clojurebreaker/models/game.clj
create mode 100644 src/clojurebreaker/server.clj
create mode 100644 src/clojurebreaker/views/common.clj
create mode 100644 src/clojurebreaker/views/welcome.clj
```

10.5.4　安置你的应用

至此，你已经在本地准备好了要推送到 Heroku 上的代码。现在是时候创建一个 Heroku 应用来容纳你的程序了。

```
heroku create --stack cedar

Creating freezing-waterfall-3937... done, stack is cedar
http://freezing-waterfall-3937.herokuapp.com/ |
git@heroku.com:freezing-waterfall-3937.git
Git remote heroku added
```

那个--stack 参数指定了 Heroku 上支持 Clojure 应用的平台。这个应用会得到一个 Heroku 提供的随机名称，和一个你可以访问的 URL 地址，通过这个地址可以看到你运行中的程序。如果你想，可以随后对其重新命名。

10.5.5 部署

最后一步是运行 git push 命令，剩下的工作就交给 Heroku 去照料吧。

```
git push heroku master

Counting objects: 24, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 230.67 KiB, done.
Total 24 (delta 0), reused 0 (delta 0)

-----> Heroku receiving push
-----> Clojure app detected
-----> Installing Leiningen
       Downloading: leiningen-1.5.2-standalone.jar
       Downloading: rlwrap-0.3.7
       Writing: lein script
-----> Installing dependencies with Leiningen

... 省略输出 ...

-----> Discovering process types
       Procfile declares types -> web
-----> Compiled slug size is 12.8MB
-----> Launching... done, v4
       http://stormy-water-3888.herokuapp.com deployed to Heroku
```

你也可以使用 heroku 命令来访问你的应用。

```
heroku open
```

这样会打开一个浏览器，并把你直接带到你部署的应用那儿。第一次访问时，可能要花费一些时间来加载，但后续的请求会快得多。你可以在 Heroku 开发中心找到一份额外的 Heroku/Clojure 教程，地址是 http://devcenter.heroku.com/articles/clojure-web-application。

享受你这个 Clojureweb 应用的乐趣吧。让它变得更好，赋予它一些个性，然后分享给全世界！

10.6 小结

恭喜你。在很短的时间里，你已经走了很长的一段路。你学到了很多想法的组合：Lisp、Java、函数式编程和显式并发，正是它们让 Clojure 显得如此伟大。此外，在本

章中，你还见到了一种（由许多步骤组成的）可行的工作流程，并用它开发了一个完整的 Clojure 应用。

我们仅仅触及了 Clojure 巨大潜力的一点皮毛，我们希望你会迈出下一步，成为 Clojure 社区中活跃的一员：加入邮件列表[①]，参与到 IRC 的网上讨论中[②]。Clojure 社区友好而且热情，我们非常乐意听到你的消息。

① http://groups.google.com/group/clojure。
② #clojure on irc.freenode.net。

附录

编辑器

编辑器对 Clojure 的支持，正在快速发展当中，所以在你读到这里的时候，一些信息可能已经过时了。最新的信息，请参阅 Clojure 社区 wiki 上的入门指南[①]页面。那里有一些关于各种开发环境的子页面。

Clojure 代码简洁且富有表现力，其结果是，编辑器对它的影响，并不像对其他语言那样明显。然而，你一定会希望你使用的编辑器，至少能够正确地缩进代码，并可以进行括号匹配。

写这本书的时候，我们使用的是 Emacs，再加上杰弗里·楚（Jeffrey Chu）的 clojure-mode[②]。Emacs 对 Clojure 的支持相当不错，但如果你不是一个 Emacs 用户的话，你可能会倾向于从这里列出的编辑器中挑选一个你熟悉的作为开始。

编辑器	项目名称	项目 URL 地址
Eclipse	Counterclockwise	http://code.google.com/p/counterclockwise/
Emacs	clojure-mode	http://github.com/jochu/clojure-mode
IntelliJ IDEA	La Clojure	http://plugins.intellij.net/plugin/?id=4050
jEdit	jedit modes	http://github.com/djspiewak/jedit-modes/tree/master/
NetBeans	enclojure	http://enclojure.org
TextMate	textmate-clojure	https://github.com/swannodette/textmate-clojure
Vim	VimClojure	http://www.vim.org/scripts/script.php?script_id=2501

① http://dev.clojure.org/display/doc/Getting+Started。
② http://github.com/jochu/clojure-mode。

参考书目

[Goe06]	布莱恩·戈茨（Brian Goetz）。《JAVA 并发编程实践》（Java Concurrency in Practice）。Addison-Wesley, Reading, MA, 2006.
[Hof99]	侯世达（Douglas R. Hofstadter）。《哥德尔、艾舍尔、巴赫：集异壁之大成》（Gödel, Escher, Bach: An Eternal Golden Braid）。Basic Books, New York, NY, USA, 20th Anniv, 1999.
[McC06]	史蒂夫·麦康奈尔（Steve McConnell）。《软件估算——“黑匣子”揭秘》（Software Estimation: Demystifying the Black Art）。Microsoft Press, Redmond, WA, 2006.

欢迎来到异步社区！

异步社区的来历

异步社区（www.epubit.com.cn）是人民邮电出版社旗下 IT 专业图书旗舰社区，于 2015 年 8 月上线运营。

异步社区依托于人民邮电出版社 20 余年的 IT 专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与 POD 按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。

社区里都有什么？

购买图书

我们出版的图书涵盖主流 IT 技术，在编程语言、Web 技术、数据科学等领域有众多经典畅销图书。社区现已上线图书 1000 余种，电子书 400 多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户帐户中的积分可以用于购书优惠。100 积分 =1 元，购买图书时，在 

里填入可使用的积分数值，即可扣减相应金额。

特 别 优 惠

购买本书的读者专享**异步社区购书优惠券**。

使用方法：注册成为社区用户，在下单购书时输入 57AWG 使用优惠码，然后点击“使用优惠码”，即可享受电子书 8 折优惠（本优惠券只可使用一次）。

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。

社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得 100 积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于 Markdown 的写作环境，喜欢写作的您可以在此一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握 IT 圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：

异步社区	微信服务号	微信订阅号	官方微博	QQ 群：368449889

社区网址：www.epubit.com.cn

官方微信：异步社区

官方微博：@ 人邮异步社区，@ 人民邮电出版社 - 信息技术分社

投稿 & 咨询：contact@epubit.com.cn

Clojure编程经典之作，Clojure之父强烈推荐

Clojure是一种基于Java虚拟机的动态编程语言。如果你要转移到函数式编程，或是要应对来自并发的挑战，那么，Clojure是你的首选语言。Clojure具备以下优点：

- 简单，是这门优雅语言的核心；
- 独到的并发机制和函数式风格；
- Lisp的强大；
- 充分发挥JVM的全部能力；
- 代码量往往比Java少，且易于维护。

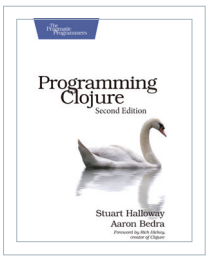
本书是介绍Clojure的经典之作。它完全覆盖了Clojure 1.3版本提供的新特性，而且为了体现Clojure新生概念的重要意义，作者重新组织和编写了一些章节。

两位作者通过本书展示了如何从头构建一个应用，帮助读者充分了解Clojure程序的完整开发流程，还帮助读者运用Clojure思想为问题的各个部分找出解决方案。

通过阅读本书，你能够学会运用Clojure的方式进行思考，充分利用这门语言的优势，快速构建强大的程序。

Stuart Halloway　Clojure/core成员、Relevance公司CTO。

Aaron Bedra　Clojure/core成员，在担任Relevance公司开发者期间，他成为了一名技术领袖、演讲家和作家。他对Clojure语言及其支持库贡献颇多，同时也是Clojure社区中的活跃分子。Aaron牵头开发了一些商业的Clojure项目。目前他在Groupon工作。



Clojure是目前最有趣的编程语言之一，同时，作为学习Clojure的最佳途径，本书增加了许多最新的信息和丰富实用的范例，此外还有大量关于如何学习和使用Clojure的提示，这真是太管用了。

—— Ola Bini
loke编程语言之父，ThoughtWorks公司开发者

被Clojure给吓住啦？在读过本书之后，你就不再会这样了。它文风清晰、平易近人且令人愉悦，带你循序渐进地学习这门语言。

—— Tim Berglund
August Technology Group的创始人及负责人

作者凭借这本组织良好且易于阅读的著作，为学习Clojure语言设计了一条最为平坦的路线。本书用简单有效的示例来展示了Clojure独一无二的特性。

—— Chris Houser
Clojure的主要贡献者及程序库作者

Clojure是Java虚拟机上一门美丽、优雅且极其强大的编程语言。在本书中，两位经验丰富的开发者帮助你学习和掌握这门语言。你将会乐在其中，并从中获益。

—— Venkat Subramaniam
Agile Developer公司创始人

The Pragmatic Programmers

ISBN 978-7-115-30847-4

分类建议：计算机／程序设计
人民邮电出版社网址：www.ptpress.com.cn　封面设计：任文杰