The Pragmatic Programmers

TURING 图灵程序设计丛书

Scala与Clojure函数式编程模式

Java虚拟机高效编程

学会函数式方案，最大程度简化编程
掌握21个模式，从面向对象从容过渡到函数式编程

【美】Michael Bevilacqua-Linn 著
赵震一 译

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

Functional Programming Patterns in Scala and Clojure
Write Lean Programs for the JVM

Scala与Clojure函数式编程模式

Java虚拟机高效编程

【美】Michael Bevilacqua-Linn 著
赵震一 译

人民邮电出版社
北京

图书在版编目（CIP）数据

Scala与Clojure函数式编程模式 : Java虚拟机高效编程 / (美) 贝维拉夸林 (Bevilacqua-Linn, M.) 著 ; 赵震一译. -- 北京 : 人民邮电出版社, 2015.5
(图灵程序设计丛书)
ISBN 978-7-115-38894-0

Ⅰ. ①S… Ⅱ. ①贝… ②赵… Ⅲ. ①JAVA语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2015)第065935号

内 容 提 要

本书向读者展示了如何采用函数式方案来替代或最大程度地简化在面向对象编程中所使用的许多常用模式，同时还介绍了一些在函数式世界中广泛使用的模式。主要内容包括：函数式编程简介，Java、Scala、Clojure 三种语言中的 Tinyweb 对比，函数式编程范式如何替换面向对象编程模式，几种主要的函数式编程范式。

本书适合所有程序员和对函数式编程感兴趣的读者阅读。

◆ 著　[美] Michael Bevilacqua-Linn
译　赵震一
责任编辑　朱　巍
责任印制　杨林杰

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷

◆ 开本：800×1000　1/16
印张：13.5
字数：319千字　2015年5月第1版
印数：1－3 000册　2015年5月北京第1次印刷
著作权合同登记号　图字：01-2014-2964号

定价：49.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第 0021 号

译 者 序

编程世界就好比江湖，各种技术与思想有如各种内外家功夫在历史的舞台上纷呈登场，各领风骚。如今，自 C、C++传承而来的以 Java 为代表的命令式语言一派可谓如日中天、门徒万千。多年来，这几门语言一直占据着 TIOBE 编程语言排行榜前几名，而很多“没落”的语言却在最近这几十年里逐渐淡出了人们的视线。在命令式语言中，随着面向对象编程的流行而兴起的对设计模式的探讨始终是 OO 程序员群体中的热门话题，设计模式的相关问题也一度成为面试官遴选候选者的硬性指标。

然而，江湖上逐渐出现了一种说法，“有些设计模式事实上是用来弥补语言本身不足的”，也就是说有人认为：大部分设计模式或许并没有存在的必要，只要将设计模式融入语言设计之中，使语言拥有足够的表现力，那么很多模式将不再需要或可以得到极大的简化。或许你对这一说法还将信将疑，不置可否。但一个“没落贵族”已经再现在你的面前，它就是函数式编程语言。

函数式编程语言曾一度在江湖上销声匿迹，或许大部分人只在象牙塔中领略过它们的风采。它的“没落”并不是没有原因的，比如早期人们对它在性能表现上的质疑等。然而，凭借着大数据时代的东风，拥有着函数式血统的继承者们再次站上了浪潮之巅，并在江湖上留下了它们的足迹。

Google 的论文 *MapReduce: Simplified Data Processing on Large Clusters* 表明，MapReduce 编程模型的抽象是受到了 Lisp 等函数式语言中的 map 和 reduce 原语的启发而设计出来的。

分布式实时计算系统 Storm 的核心代码是采用 Clojure 编写的，而当《程序员》杂志问及 Storm 创始人 Nathan Marz 为何会选择 Clojure 时，Nathan 回答道：“Clojure 是我迄今用过的最好的语言。它允许我轻松地使用不可变性及函数式编程等技术，令我的效率大幅提高。基于 Lisp 的动态特性，我总能将 Clojure 塑造成符合最佳抽象的状态。假如没有 Clojure，Storm 倒不会有什么不同，但实现过程将痛苦得多”。

新一代分布式计算系统 Spark 不仅在实现中选择了 Scala，其提供的首选编程语言也是 Scala。而从 RDD 所提供的 API 来看，你不难发现这便是典型的函数式编程，其充分利用了函数式编程的简洁性和不可变性。

……

处于大数据时代的你，估计对以上的事迹已有所耳闻。凭借着当今大规模的分布式计算能力和无限的水平扩展能力，原本函数式编程在单机性能上的不足已不再是人们最为诟病的问题。尤其是在当代高性能 JVM 的平台之上，像 Scala 和 Clojure 等函数式语言的性能已不可同日而语。而我们更为看重的是其简洁的声明性编程风格与丰富的语言表现力，而其对不可变性的推崇和

“无副作用”函数也让程序的编写变得更加简单，从而进一步解放了程序员的生产力，让“码农”们可以节省下更多的时间去喝喝咖啡，做些更有创造力的事情。这也是为什么技术社区对 Java 8 加入 Lambda 表达式支持的呼声如此之高的主要原因。

回到设计模式的话题，GoF 的设计模式是基于面向对象编程而总结提出的，它的确指引了一代 OO 程序员的成长。然而，一旦你了解了更多的编程范式之后，你就会发现原来 OO 的模式在很多其他范式的语言中并不再是必需的，尤其是在函数式编程语言之中，它的关注点是拥有头等支持的函数，而不再是对象。而其声明性的编程风格和丰富的语言表现力让你可以不再样本化地照搬照抄设计模式，一些内置的指令和语法糖让你可以像编写诗歌一样，以简洁、精炼、优雅的方式来编写代码。相比于命令式风格的语言，函数式语言解决相同问题的代码量可以减少到前者的一半，有时甚至是数量级的精简。所以，在函数式语言中，我们可以采用一些更为简洁的方式来替代原来在面向对象编程中的设计模式。而在纯函数式编程的实践中，先行者们又沉淀出了一些专注于函数式编程的模式，它们是函数式编程中的最佳实践。

如果你是一名 Java 程序员，那么本书就是为你而准备的。这是一部专注于采用 Scala 和 Clojure 来讲解函数式编程模式的武功秘籍。Scala 和 Clojure 都是基于 JVM 平台的函数式语言。由 Scala 设计者 Martin Odersky 所开设的名为 Functional Programming Principles in Scala 的课程成为了 Coursera 上通过率最高的编程课程之一。自 2012 年起，已经有至少 5 万人参加了该课程的学习，而其中有 1 万人通过了课程认证，其受欢迎度可见一斑。而相比拥有一部分面向对象血统的 Scala，Clojure 的函数式基因则显得更纯粹一些，它出自名门 Lisp 一脉。Paul Graham 曾在《黑客与画家》中对 Lisp 的生产力推崇备至，其江湖地位也无需置疑。作者 Michael Bevilacqua-Linn 采用了这两种 JVM 语言来对比地讲解函数式编程模式，其中前半部分结合了面向对象的模式，列举了一些面向对象模式在函数式编程中的替代方案。而后半部分则重点讲述了在函数式编程世界中所独具的编程模式。通过阅读本书，你可以领略到各种函数式编程思想与技巧，为从原来普通的 Java 命令式编程顺利过渡到函数式编程打好坚实的基础，从而在 JVM 上编写出更加精益的程序。

作为译者，我很荣幸能为你翻译并呈现这部在函数式编程领域的优秀书籍。在此，我要感谢人民邮电出版社的编辑朱巍，是你的信任让我有了翻译这本书的机会，也是你的理解给了我较为充分的翻译时间。还要感谢图灵社区的编辑陈婷婷和李鑫，在全书的翻译工作中给予了逐字的审校与润色，并不断向我反馈翻译中出现的问题，让本书的质量得到了保证。我还要感谢我的父母和岳母，在本书的翻译期间，我太太正处于待产阶段，并最终为我们的家庭带来了新的成员，而我却因为工作和翻译的原因，并没有给予他们母子足够多的照顾，全靠有你们帮我分担。最后要感谢我的太太陈祥奎，是你的理解和包容才让我得以做好自己喜欢的这些事，而在本书的翻译期间，你给我带来了可爱的儿子赵彧知。正是因为有了你们，才让我在工作中有了更大的动力。

整个翻译过程也是一个学习过程。感谢在此过程中给予我支持和理解的所有朋友和同事。由于个人能力所限，在对原书的理解及译文的表述中或许会存在一些不当之处，请读者给予理解与反馈。如发现任何翻译上的问题，欢迎与我联系，我的电子邮箱是 emsn1026@gmail.com。

赵震一，2015 年 2 月于杭州

前　言

这是一本有关模式和函数式编程的书，其中的函数式编程采用了 Scala 和 Clojure 两种语言进行描述。在本书中，我们向读者展示了如何采用函数式方案来替代或最大程度地简化在面向对象编程中所使用的诸多通用模式，同时还介绍了一些在函数式世界中广泛使用的模式。

相比原来单一地采用面向对象编程，如果能结合使用这些函数式模式，那么程序员将变得更有效率，同时也能以一种更简洁且更具声明性风格的方式来解决问题。如果你是一名 Java 程序员，希望了解函数式编程能为你的工作效率带来多大的提升，或者你是一名刚刚开始使用 Scala 和 Clojure 的新手，尚不能玩转这些函数式的问题解决方案，那么本书就是为你而准备的。

在开始深入探讨这些内容之前，我想先来讲一个故事。尽管出于保护隐私的目的修改了一些名字，但这确实是一个真实的故事。

函数式编程故事一则

作者：Michael Bevilacqua-Linn，软件消防员

网站的服务器并没有宕机，但是大量的警报却接踵而来。我们对这个问题进行跟踪并定位到了对我们使用过的第三方 API 的一组变更。而正是这一组变更导致了我们这一端主要的数据问题。换句话说，我们并不知道这组变更具体改变了什么，也无法找到任何能说明这些变更的人。现在的情况就是，事实表明与该 API 交互的系统也使用了这些遗留代码，而唯一知道这些代码的运作方式的家伙却正在度假。这是一个大型系统：有足足 50 万行的 Java 和 OSGi 代码量。

大量寻求支持的电话如潮水般蜂拥而至，失望的客户不顾昂贵的花费，打来支持电话以寻求我们的帮助。必须快速修复这个问题。我启动了一个 Clojure REPL，开始毫无头绪地用它来对问题 API 进行诊断。

我的老板把头探进了我的办公室。“进展如何？”他问道。“哦，正在处理。”我回应。十分钟之后，老板的老板来了。“怎么样了？”他问道。“在处理。”我回应道。又是十分钟过去了，大老板（老板的老板的老板）来了。“情况如何？”他问道。“还在搞呢。”我回应道。接下来我清净了半小时，直到 CTO 出现。“还在搞呢。”在他开口之前我就先开了腔。

一个小时后，我找出了变更的问题所在。我立马提出了一个方案，该方案可以在遗

> 留代码的开发者回来彻底修复该问题前保持数据不被污染。我将编写的用于解决该问题的小程序转交给了运维团队，他们将该程序运行在了一个 JVM 的安全区域。至此，潮水般的支持电话戛然而止，每个人都松了一口气。
>
> 在大约一周之后的全体会议上，大老板对我上次编写的用于处理故障的 Java 程序表示了感谢。而我则微笑着告诉他："那并不是 Java。"

REPL 是 Clojure 的交互式编程环境，它在这个故事里帮了很大的忙。然而，大多数的语言（尤其是非函数式的语言）都拥有相似的交互式编程环境，所以这并不是这个故事的全部。

本书将要介绍两种模式：模式 21 "领域特定语言" 和模式 15 "操作链模式"。在圆满处理这场故障中，它们起到了关键作用。

在早些时候，我就编写了一个领域特定语言的小型实例来配合这些特别的 API，即使这些 API 很庞大，很难从中识别出问题所在，这一方式也能帮助我快速地对这些 API 进行探查。除此之外，函数式编程所依赖的强大的数据转换工具，例如我们将在模式 15 中看到的例子，可以帮助我快速地编写代码来收拾故障引起的烂摊子。

本书组织结构

我们将从一段对模式的介绍开始，并讨论这些模式是如何与函数式编程联系起来的。然后，我们将会看到一个扩展示例，一个名为 TinyWeb 的小型 Web 框架。我们首先会向读者展示一个使用 Java 编写的、采用了传统面向对象模式的版本，然后逐块重写 TinyWeb，即采用 Scala 将其重构成一个混合了面向对象和函数式风格的版本。最后，使用 Clojure 来编写一个纯函数式风格的版本。

TinyWeb 扩展示例有以下几个目的。首先，它将使我们了解如何将本书所讨论的多个模式全面结合起来；同时，也可以利用它对 Scala 和 Clojure 的基础知识进行介绍。最后，由于我们会将 TinyWeb 从 Java 版本逐步转换成 Scala 和 Clojure 的版本，因此有机会来探知如何轻松地将 Java 代码与 Scala 及 Clojure 进行集成。

本书剩余内容组织为两个大的章节。首先是第 3 章，它对面向对象模式的函数式替代方案进行了描述。这一部分挑选了一些重要的面向对象模式，并展示了如何采用简洁的函数式解决方案来替代它们。

Peter Norvig 是 Lisp 经典著作 *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp* [Nor92]的作者，他目前在 Google 担任研究主任。Peter 是一个技术全才，天资聪慧，他在 *Design Patterns in Dynamic Languages* 一书中指出：动态语言（像 Lisp 这样极富表现力的语言）中的设计模式可以将传统面向对象的模式无痕化[①]。

遗憾的是，似乎没有多少主流软件开发世界中的人读过 Norvig 的著作。但是既然我们能够

① http://norvig.com/design-patterns/

采用一些更简单的东西来替代某个复杂的模式，就表明我们这样做是有意义的。这样做可以使代码变得更加简洁与便于理解，从而更易于维护。

第二部分是第 4 章，该部分内容描述了函数式世界中一些原生模式。这些模式规模不一，从由一两行代码组成的微型模式到处理整个项目的大型模式都有所涉及。

有的时候，这些模式拥有头等的语言支持，这就意味着已经有人通过努力工作替我们实现了这些模式。就算遇到尚未实现的场景，我们通常也可以采用一个非常强大的模式，即模式 21“领域特定语言”来自行实现。也就是说，函数式模式相比面向对象模式更加轻量级。你仍然需要在使用相应的模式之前理解该模式，但是其实现将变得非常简单，甚至只需要几行代码。

模式模板

在本书中所介绍的模式都会使用如下的格式，当然也有例外。举个例子，如果某个模式没有任何其他通用的名字，它将不会具有“别名”部分。而“函数式替代方案”一节则只适用于第 3 章。

目的

本节提供了对模式的目的及其所能解决的问题的快速介绍。

概述

在这一小节中，你将会看到相关模式的更深层次的动机及对该模式运作方式的介绍。

别名

这一小节列举了该模式一些其他的常见叫法。

函数式替代方案

在这一小节你将会看到我们是如何采用函数式编程技术来替代当前模式的——有的时候，面向对象模式可以由基本的函数式语言特性来替代，有时则可以由一些更简单的模式来替代。

范例代码

这一小节包含了模式的示例——对于面向对象模式来说，在介绍如何使用 Clojure 和 Scala 替代该模式之前，我们首先会采用类图或 Java 代码来展示一个面向对象解决方案的框架。而函数式模式只采用 Clojure 和 Scala 来进行展示。

讨论

该部分会对当前模式中比较有意思的地方进行总结和讨论。

延展阅读

该部分罗列了关于当前模式更多信息的参考资料。

相关模式

该部分提供了一些与当前模式相关的其他模式，这些模式都是本书所讨论的内容。

为什么选择Scala和Clojure

本书中的很多模式也可以通过采用其他具备函数式特性的语言得以应用，但是我们会专注于采用 Clojure 和 Scala 来实现。之所以这样做有几个原因，其中最首要的就是这两门语言都是适合在生产环境中使用的实用语言。

Scala 和 Clojure 都运行在 Java 虚拟机（JVM）之上，因此它们可以与现有的 Java 代码库很好地进行互操作，并且可以毫无问题地部署在 JVM 的基础设施中。这使得它们成为与现有的 Java 代码库友好并存的理想选择。最后，Scala 和 Clojure 都具备函数式特性，但是它们之间又存在着明显的差异。对两者的学习可以帮助我们拓宽函数式编程范式的视野。

Scala 是一门混合了面向对象与函数式特性的语言。同时它又是一门静态类型的语言，结合了一个非常精妙且具有本地类型推断的类型系统，通常允许我们在代码中省略对类型的显式标注。

Clojure 是 Lisp 的一门现代方言。它拥有 Lisp 强大的宏系统和动态类型，但是 Clojure 的实现者又为其添加了在早前的 Lisp 中未曾见过的新特性。其中最重要的特性便是通过使用引用类型、软件事务内存系统以及高效的不可变数据结构来处理状态变化的独特方式。

虽然 Clojure 不是一门面向对象的语言，但是它给予了我们不少在面向对象语言中常见的优良特性，只是实现方式与我们所熟知的有所不同。举个例子，我们可以通过 Clojure 的多重方法和协议来获取多态机制，还可以通过 Clojure 的特别层级来获得层级结构。

在介绍模式的同时，我们还会对这两门语言以及它们的特性进行探索，所以本书也不失为对 Scala 和 Clojure 的一个不错的入门介绍。如果你想对这两门语言有更深入的了解，那么我会向你推荐我最喜欢的几本书，其中关于 Clojure 的书有《Clojure 程序设计》[Hal09]和《Clojure 编程乐趣》[FH11]，关于 Scala 的书有《Scala 程序设计：Java 虚拟机多核编程实战》[Sub09]和 *Scala In Depth* [Sue12]。

如何阅读本书

最佳的阅读起点是从第 1 页，即第 1 章开始阅读，本章为函数式编程及其相关的模式提供了概览。接下来便是第 2 章，本章对 Scala 和 Clojure 中的基本概念进行了介绍，并展示了如何将本书中的模式组合起来。

在此之后，你便可以根据需要跳跃式阅读每个模式。在第 3 章和第 4 章中，前面的内容会较后面的内容更加基础，所以如果你之前并没有函数式经验的话，那么这些较为基础的内容值得优先阅读。

你可以在 1.2 节中找到我们为每个模式整理的快速摘要，供读者简单快速浏览。一旦你通读了这些介绍，便可以通过摘要来查找能解决特定问题的相应模式。

不过，如果你确实是一个从未接触过函数式编程的新手的话，你就应该从模式 1“替代函数式接口”、模式 2“替代承载状态的函数式接口”以及模式 12“尾递归模式”开始读起。

在线资源

在阅读本书的过程中，你可以从http://pragprog.com/titles/mbfpp/source_code下载本书所有的代码文件。在本书的主页http://pragprog.com/book/mbfpp上，你可以找到本书的论坛，并提交勘误。同样地，对于电子书读者，可以通过点击源代码链接来下载相应的代码。

致　谢

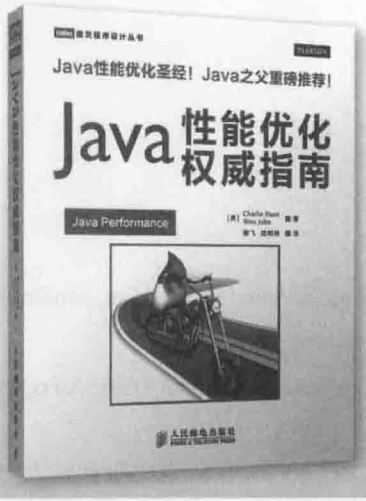
感谢我的父母，没有他们就没有我。

同样感谢我的绝世好女友。无数个夜晚和周末，她都在忍受着我对代码范例、时态不一致以及断句问题的嘟囔。

如果没有一个优秀的技术审校团队，本书内容将不会像现在这样精彩。在此要感谢一下这些人：Rod Hilton、Michajlo “Mishu” Matijkiw、Venkat Subramaniam、Justin James、Dave Cleaver、Ted Neward、Neal Ford、Richard Minerich、Dustin Campbell、Dave Copeland、Josh Carter、Fred Daoud 和 Chris Smith。

最后，我要感谢 Dave Thomas 和 Andy Hunt。他们的著作《程序员修炼之道：从小工到专家》是我职业生涯初期的启蒙读物之一。该书对我产生了深远的影响，至今我还保留着那本已经起了折角、布满指印、破旧不堪的书。在 Dave 和 Andy 的笃信力行下，Pragmatic Bookshelf 成为了一家真正致力于出版高品质技术图书的出版商，并给予了作者莫大的支持。

延展阅读



- **Java 性能优化圣经！**
- **Java 之父重磅推荐！**

《Java 性能优化权威指南》由曾任职于 Oracle/Sun 的性能优化专家编写，系统而详细地讲解了性能优化的各个方面，帮助你学习 Java 虚拟机的基本原理、掌握一些监控 Java 程序性能的工具，从而快速找到程序中的性能瓶颈，并有效改善程序的运行性能。

Java 性能优化的任何问题，都可以从本书中找到答案！

Java 性能优化权威指南
书号：978-7-115-34297-3
定价：109.00 元

七周七并发模型
书号：978-7-115-38606-9
定价：49.00 元

发布！软件的设计与部署
书号：978-7-115-38045-6
定价：49.00 元

正则表达式必知必会（修订版）
书号：978-7-115-37799-9
定价：29.00 元

程序员思维修炼（修订版）
书号：978-7-115-37493-6
定价：49.00 元

高效程序员的 45 个习惯：敏捷开发修炼之道（修订版）
书号：978-7-115-37036-5
定价：45.00 元

程序员健康指南
书号：978-7-115-36716-7
定价：39.00 元

目　录

第 1 章 模式和函数式编程

模式和函数式编程可以通过两种方式结合在一起。首先，对于很多面向对象的设计模式来说，采用函数式编程来实现会更加简单。之所以会造成这种差异，有很多原因。函数式语言为我们提供了更加简洁的方式来完成一些计算的传递，而无需创建新的类。同样地，使用表达式（expression）而非语句（statement）可以让我们消除那些额外的变量。并且，声明性是很多函数式解决方案所拥有的一个特质，这种特质可以让我们在一行代码中完成原本在命令式语言中需要五行代码才能完成的工作。我们甚至可以用函数式语言特性的简单应用来替代某些面向对象模式。

其次，函数式的世界也有一套它自己的有用的模式。这些模式专注于编写避免可变性且偏好声明性风格的代码，可以帮助我们编写出更加简单且更易维护的代码。而本书的两个主要部分将会涉及对这两组模式的讨论。

也许你会惊讶于第一组模式。难道我们所熟知和喜爱的模式无法扩展到各种别的语言？难道它们不应该为通用的问题提供通用的解决方案，而与你所使用的语言无关吗？只要你正在使用类似Java或其先祖C++之类的语言，那么对于这两个问题的答案都是肯定的。

然而，随着那些更具表现力的语言特性的出现，许多这样的模式开始失去了存在的意义。传统的Java自身就拥有一个由语言特性替代模式的实例：`foreach`。在Java 1.5中引入的`foreach`循环削弱了迭代器模式（参见《设计模式：可复用面向对象软件的基础》[GHJV95] 一书的描述）的作用，尽管`foreach`循环在其背后实现中仍然使用了这一模式。

这并不是说`foreach`循环与迭代器模式是完全等价的。`foreach`并不能在所有的场景中替代迭代器模式。它们所涉及的问题以一种更加简单的方式得到了解决。开发者之所以会选择内建的`foreach`循环是因为这种方式减轻了开发者实现功能的工作量，同时也降低了程序出现错误的可能性。

很多函数式语言的特性及技术会对项目产生类似效果。虽然它们可能与某个模式并不完全等同，但是它们通常为开发者提供了一个内建的可选项来解决同样的问题。与`foreach`替代迭代器模式实例类似，其他一些语言特性也提供了各种技术来为开发者减轻工作量，而且这些特性产出的代码会较原来的代码更加简洁且更容易理解。

这些函数式的特性和技术让我们的编程工具箱变得更加丰富，譬如Java 1.5中新增的`foreach`循环，不过它所涉及的范围和规模更加庞大。对于那些我们已经熟知并喜爱的来自面向对象世界的工具，这些新的函数式工具起到了很好的补充作用。

本书涉及的第二种模式则是原生的函数式模式，我们将会对这些从函数式风格演化而来的模式进行描述。这些模式在某些方面与你可能熟知的面向对象模式会有所不同。首先，最明显的区别是函数将成为程序最主要的组成单元，它在函数式世界的地位就好比在面向对象世界中对象的地位一样。

另一个关键的不同在于模式的粒度。《设计模式》（软件模式运动的原动力之一）中的模式通常是一些模板，这些模板定义了若干类，并指定了这些类如何组合在一起。它们中的绝大部分都是中等规模的。它们通常对自身的问题规模并不关心，无论是只包含几行代码的小问题，抑或是包含了全部程序的大问题。

而本书中的函数式模式所覆盖的范围更加广泛，其中的一些模式所涉及的问题只需一两行代码就可以实现。而另外一些模式则可以处理重大问题，例如创建新的小型编程语言。

通常来说，《建筑模式语言》[AIS77]一书被认为是开启了模式运动的标志性著作。这本有关建筑模式的著作开篇介绍了一个非常有名的模式“1：独立区域”，并一直讲到“248：软质面砖和软质砖”。前者概述了为什么这个星球应该被组织成一系列拥有上万人口的政治实体，而后者则说明了如何去制作你自己的砖块。

在深入学习本书的各种模式之前，让我们花一些时间来熟悉一下函数式编程本身。

1.1　什么是函数式编程

就其核心而言，函数式编程是关于不变性和函数（而非对象）组合的一种编程范式。而很多相关特征都源自这一风格。

函数式编程拥有如下特征。

拥有头等（first-class）函数：头等函数是指那些可以被传递、动态创建并可以存储于数据结构中的函数。你可以在语言中像处理任何其他头等对象那样对它进行处理。

偏好纯函数（pure function）：纯函数是指那些没有副作用的函数。副作用是指函数的某种行为，这种行为会对函数之外的状态进行修改。

组合函数：函数式编程支持通过将函数进行组合来自底向上地构建程序。

使用表达式：函数式编程偏爱表达式胜于语句。表达式会产生值，而语句则不然，它的存在仅仅是为了控制程序的执行流程。

使用不变性：因为函数式编程偏好纯函数，而纯函数不会修改数据，它同时又大量使用了不

可变数据。所以程序不会去修改一个已经存在的数据结构，而是有效地创建一份新数据。

转换数据而非修改数据：函数式编程使用函数来转换数据。将一个数据结构输入函数，然后输出一个新的不可变的数据结构。这与流行的面向对象模型产生了明显的对比，后者将对象视为对可变状态和行为的小型封装。

鉴于我们不能逐块去修改某个数据结构，对不可变数据的专注促使我们采用一种更具声明性的风格来编写程序。下面是一段以循环方式对列表进行奇数过滤的Java代码。请注意它是如何依赖对`filteredList`的变化从而向它一次添加一个奇数的。

JavaExamples/src/main/java/com/mblinn/mbfpp/intro/FilterOdds.java

```
public List<Integer> filterOdds(List<Integer> list) {
        List<Integer> filteredList = new ArrayList<Integer>();
        for (Integer current : list) {
                if (isOdd(current)) {
                        filteredList.add(current);
                }
        }
        return filteredList;
}
private boolean isOdd(Integer integer) {
        return 0 != integer % 2;
}
```

下面是一个采用Clojure编写的函数式版本。

```
(filter odd? list-of-ints)
```

该函数式版本相对于面向对象版本显得更加简短。正如前面所提到的，这是因为函数式编程是声明性的。也就是说，它指定了什么应该被完成而非如何去完成。就我们在编程中遇到的大多数问题而言，这种风格可以让我们以更高层次的抽象进行工作。

然而，有些问题是难以使用严格的函数式编程技术来解决的。编译器是一个纯函数。如果你输入一个程序，你会希望每次都得到相同的机器码。如果没有，这很可能是一个编译器的bug。而谷歌的搜索引擎则不是一个纯函数。如果你每次在谷歌搜索的查询中都得到相同的结果，那么我们将停留在20世纪90年代末的Web发展状态，那就太悲惨了。

出于这个原因，函数式编程语言在函数式纯度方面往往可以用一个严格性程度评估范围来衡量。某些函数式编程语言会较其他的语言纯度更高。而就我们在本书中所使用的两种语言而言，Clojure在函数式的严格性程度评估范围中纯度更高；至少在我们避免使用与Java的互操作性时是这样的。

举个例子，在惯用的Clojure中，我们不会像在Java中那样修改数据，而是依赖于一系列高效的不可变数据结构、一组引用类型以及一个软件的事务性内存系统来实现。这让我们既可以享受到可变性的好处，又无需承担任何风险。我们将会在2.4节中介绍这些技术。

相比Clojure，Scala对于可变数据有着更多的支持，但是它更推崇使用不可变数据。举一个例

子，Scala的集合库同时拥有可变和不可变两个版本的实现，但是不可变数据结构是被默认导入和使用的。

1.2　模式词汇表

下面我们将对本书中所涉及的所有模式进行介绍，并给出相应的概述。如果你希望以函数式方式来解决手头的具体问题，那这将是一个非常不错的可供快速浏览的列表。

替代面向对象模式

这一部分会向你展示如何采用函数式语言的特性来替代普通的面向对象模式。这样做通常可以减少我们需要编写的代码数量，从而让我们维护的代码变得更加简洁。

模式1　替代函数式接口

在这一模式中，我们采用原生的函数式特性来替代像`Runnable`或`Comparator`这样常见的函数式接口。

在这一部分中，我们引入了两个基本的函数式特性。第一个特性是高阶函数（higher-order function），它允许我们将函数作为头等数据进行传递。第二个特性是匿名函数，它允许我们编写快捷的一次性函数，而无需为其指定函数名。通过将这两个特性相结合，我们可以用一种非常简洁的方式来替代大多数的函数式接口实例。

模式2　替代承载状态的函数式接口

我们采用这种模式来替代那些需要承载某些状态信息的函数式接口实例，为此我们引入了一个新的特性：闭包。闭包可以将某个函数和某些状态进行打包传递。

模式3　替代命令模式

替代命令模式将行为封装于某个对象之中——在这一模式中，我们将了解如何使用在前两种模式中所介绍的技术来替代面向对象版本的命令模式。

模式4　替代生成器（Builder）模式来获得不可变对象

通常我们采用传统的Java约定（即一个具备`getter`和`setter`方法的类）来承载数据，但是这种方式却与可变性紧密相连。在这一模式中，我们将向你展示如何从得益于不变性的Java Bean中获取便利。

模式5　替代迭代器模式

替代迭代器模式让我们以顺序方式访问集合子项——我们将会看到如何采用高阶函数和序列推导（sequence comprehension）来解决我们原本采用迭代器模式所解决的诸多问题，这些技术给我们带来了更具声明性的解决方案。

模式6 替代模板方法（Template Method）模式

这一模式定义了超类中的算法轮廓，而该算法的具体细节留待子类来实现。在这一模式中，我们将看到如何采用高阶函数和函数组合来替代这一基于继承的模式。

模式7 替代策略（Strategy）模式

在这一模式中，我们定义了一组都实现了某一共同接口的算法。这让程序员可以简单地将一个算法实现替换成另一个算法实现。

模式8 替代空对象（Null Object）模式

在这一模式中，我们讨论了如何去替代空对象，同时也对null的其他处理方式进行了探讨。在Scala中，我们通过使用`Option`利用了Scala类型系统的优势。在Clojure中，我们依靠了`nil`和一些语言支持，从而使得对null的处理变得更加方便。

模式9 替代装饰器（Decorator）模式

替代的装饰器模式可以向对象添加新的行为而不改变其原有类。在这一模式中，我们将会看到如何采用函数组合来达到相同的效果。

模式10 替代访问者（Visitor）模式

替代访问者模式使得为某种数据类型添加操作变得更简单，却难以为类型添加新的实现。本章展示了采用Scala和Clojure的解决方案，从而使得为数据类型添加操作和实现都成为可能。

模式11 替代依赖注入

该模式可以向某个对象注入其依赖，而非内联地实例化这些依赖，这样做将允许我们对这些依赖的实现进行替换。我们将会探索Scala的Cake模式，它带给我们一种类似于依赖注入的模式。

函数式模式介绍

模式12 尾递归（Tail Recursion）模式

尾递归从功能上来说与循环等效，但是它提供了一种编写递归算法的特殊方式，这种方式避免了为每次递归调用都消耗栈桢。虽然我们在本书中更偏好声明式的解决方案，但有的时候解决问题最直接的方法就是更多地采用迭代。在本模式中，我们将会展示在这类场景中如何使用尾递归。

模式13 相互递归（Mutual Recursion）模式

相互递归是一种递归函数互相调用的模式。和尾递归一样，我们需要为相互递归找到一种切实有效的方式来避免栈桢的消耗。在本模式中，我们将展示如何使用一种被称为“蹦床”（trampolining）的特性来达成这一目标。

模式14　Filter-Map-Reduce模式

filter、map和reduce是我们最常使用的三个高阶函数。通过将它们组合在一起使用，我们便获得了一个非常强大的数据处理工具。而如今最流行的MapReduce数据处理范式的灵感便源自它们。在本模式中，我们将看到如何在一个更小的规模中使用它们。

模式15　操作链模式

函数式编程通常会尽可能避开可变性，所以为了避免对某个数据结构进行修改，我们通常会选择一个不可变的数据结构，并对该结构进行操作，然后产生一个新的数据结构。操作链模式对在Scala和Clojure中实现这一功能的不同方式进行了考察。

模式16　函数生成器模式

高阶函数可以使用函数生成器模式来创建其他函数。在这一部分中，我们将展示内置于多门函数式语言中的一些有关该模式的通用实例，同时对一些自定义的实例进行探讨。

模式17　记忆（Memoization）模式

该模式通过缓存纯函数调用结果来避免对昂贵计算的重复执行。

模式18　惰性序列（Lazy Sequence）模式

惰性序列是这样一个模式：序列中的元素仅在需要时才被逐个实例化。这允许我们创建一个无限长的序列，从而能简单地处理数据流。

模式19　集中的可变性

集中的可变性可以通过在程序中的一小段关键性代码中使用可变数据结构来优化性能。对该模式的需求远比你想象中的罕见。由JVM支持的Clojure和Scala为你提供了一套非常有效的机制来与不可变数据协同工作，所以不可变性几乎不会成为瓶颈。

模式20　自定义控制流

对于大多数语言而言，通常无法在不修改语言本身的情况下向语言添加新的控制流。然而，函数式语言通常为创建自定义控制抽象来满足特殊用途提供了方便之门。

模式21　领域特定语言

领域特定语言模式允许我们创建一种专用语言来解决特定的问题。使用一种经过精心设计的领域特定语言的实现，通常是处理问题的终极解决方案，因为这样做可以让我们更加贴近问题领域进行编程。这一模式减少了我们需要编写的代码总量，同时缓解了在将想法转化成代码过程中的困扰。

第2章

TinyWeb：让模式协同工作

2.1 TinyWeb 简介

在这趟探索之旅中，首先迎接我们的范例程序是一个叫作TinyWeb的小型Web框架，该框架大量使用了传统的面向对象模式。了解了TinyWeb之后，将介绍如何使用Scala这门混合了面向对象式和函数式风格的语言来重写这一框架。最后，将转而使用Clojure来进行重写，这是一门更具函数式风格的语言。

来关注一下该例中的几个目标。首先，在深入学习这些模式之前，先看看这些模式是如何在一个程序的代码里协同工作的。其次，为那些不熟悉Scala和Clojure这两门语言的朋友介绍一下这些语言的基本概念。如若完整介绍这些语言，则超出了本书的范围，但通过这一部分的基础性介绍，你将足以理解后续的大部分代码。最后要把现有的Java代码转换成Scala或Clojure代码，我们通过将Java版的TinyWeb转换成Scala和Clojure版来实现这一目标。

TinyWeb本身是一个小型的MVC（model-view-controller）Web框架。该框架并不完整，但是它应该会让那些曾经使用过任何像Spring MVC这样的流行框架的朋友倍感亲切。但是对TinyWeb来说，还是有一点小的转变：由于本书是有关函数式编程的，因此我们会尽可能地与不可变数据打交道，而这对Java来说颇具挑战性。

2.2 采用 Java 来编写 TinyWeb

Java版的TinyWeb是一个采用传统的面向对象风格编写的基本MVC Web框架。我们使用控制器来处理请求，该控制器是采用模板方法模式实现的，对于该模式，我们将在模式6“替代模板方法模式”中讨论。对于视图，我们采用了策略模式来实现，该模式将在模式7“替代策略模式”中讨论。

我们的框架围绕几个核心的数据对象而构建：`HttpRequest`和`HttpResponse`。我们希望这些数据对象都具有不可变性，以便我们易于使用。所以我们打算采用生成器模式来构建这些对象。该模式将在模式4“替代生成器模式来获得不可变对象”中讨论。生成器模式是Java中用以获取

不可变对象的标准方式。

最后，我们来看看请求过滤器。过滤器是在请求被处理之前运行的，它会对请求进行操作，比如修改请求。我们将采用Filter类来实现这些过滤器，这是模式1“替代函数式接口”中的一个简单例子。我们的过滤器同样展示了如何使用不可变对象来修改数据。

整个系统的概况如图2-1所示。

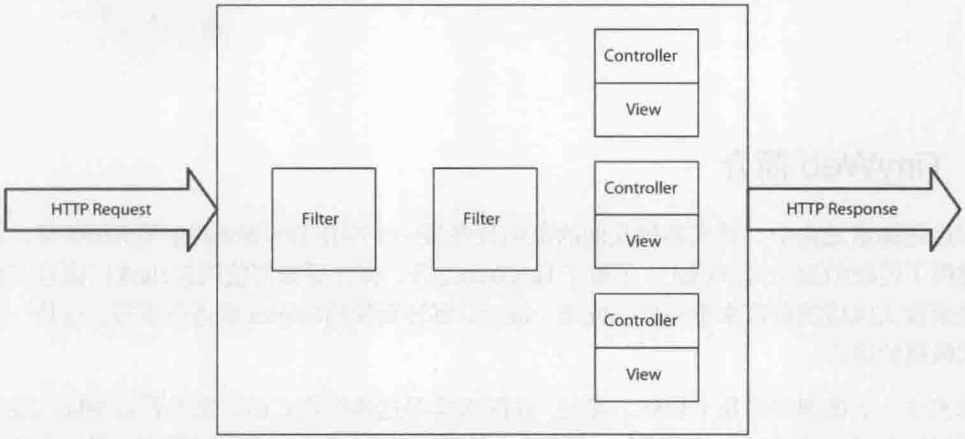


图2-1　TinyWeb概览图。这是TinyWeb的图形化概览

我们将从考察核心数据类型：HttpRequest和HttpResponse开始本次探索之旅。

HttpRequest和HttpResponse

让我们从HttpResponse开始深入查看一下代码。在本例中，我们的响应将只需要一个响应体和一个响应码，所以这些便是我们需要为响应添加的唯一属性集。以下代码块展示了我们是如何实现该类的。此处我们使用了连贯的生成器模式，该模式因在Java经典著作*Effective Java* [Blo08]中的描述而流行至今。

JavaExamples/src/main/java/com/mblinn/oo/tinyweb/HttpResponse.java

```
package com.mblinn.oo.tinyweb;

public class HttpResponse {
        private final String body;
        private final Integer responseCode;

        public String getBody() {
                return body;
        }
```

```
        public Integer getResponseCode() {
                return responseCode;
        }

        private HttpResponse(Builder builder) {
                body = builder.body;
                responseCode = builder.responseCode;
        }

        public static class Builder {
                private String body;
                private Integer responseCode;

                public Builder body(String body) {
                        this.body = body;
                        return this;
                }

                public Builder responseCode(Integer responseCode) {
                        this.responseCode = responseCode;
                        return this;
                }

                public HttpResponse build() {
                        return new HttpResponse(this);
                }

                public static Builder newBuilder() {
                        return new Builder();
                }
        }
}
```

这种方式将全部的可变性封装进了一个Builder对象，该对象将会用来创建不可变的HttpResponse。虽然这样做给我们带来了一种处理不可变数据的简洁方式，但是它相当详尽。举个例子，我们可以使用以下代码来创建一个简单的测试请求：

```
HttpResponse testResponse = HttpResponse.Builder.newBuilder()
        .responseCode(200)
        .body("responseBody")
        .build();
```

如果不使用Builder，我们便需要将所有的参数传递给构造器。这对我们这个小型的例子来说是没有问题的，但是这种做法在处理更大规模的类时将会变得非常笨拙。另一种方法是使用一个具备了getter和setter方法的Java Bean风格的类，但是那样将引入可变性。

让我们来快速看一下HttpRequest。由于该类与HttpResponse（虽然它需要我们设置一个请求体、请求头部以及一个请求路径）非常相似，我们将不再重复代码进行完整演示。不过，有一个特性值得一提。

由于我们的请求对象不是可变的，所以为了支持请求过滤器对传入请求的“修改”，需要创

建一个基于当前请求对象的新请求。我们将使用builderFrom()来实现。该方法将一个现有的HttpRequest作为入参，并使用它来设置新builder的初始值。builderFrom()方法的代码如下所示：

JavaExamples/src/main/java/com/mblinn/oo/tinyweb/HttpRequest.java

```
public static Builder builderFrom(HttpRequest request) {
        Builder builder = new Builder();
        builder.path(request.getPath());
        builder.body(request.getBody());

        Map<String, String> headers = request.getHeaders();
        for (String headerName : headers.keySet())
                builder.addHeader(headerName,
                                  headers.get(headerName));

        return builder;
}
```

这样做或许看上去有些浪费，但是JVM是当代软件工程的一个奇迹。它能够非常有效地对短命对象进行垃圾回收，因此这样的编程风格在大多数领域都表现不俗。

视图和策略模式

让我们继续TinyWeb之旅，来看看对视图的处理。在一个功能完整的框架中，我们很可能会囊括多种向视图插入模板引擎的方式，但是对于TinyWeb来说，我们只采取字符串处理这一种方式，并完全用代码来生成我们的响应体。

首先我们将需要一个View接口，它拥有唯一的方法render()。render()方法以类型为Map<String, List<String>>的model作为入参，该model代表了模型的属性和值。我们将使用List<String>来作为我们值的类型，这样单个属性就可以拥有多个值了。该方法将返回一个代表已渲染视图的字符串。

不可变性：不只是函数式程序员的专享

为了获得不可变对象，需要付出额外的努力。而经验丰富的面向对象程序员可能会对此产生抱怨，尤其当我们“仅仅是为了函数式”而这样做时。然而，不可变数据并不只是依附于函数式编程；它是可以帮助我们编写更整洁代码的一种非常好的实践。

有一大类的软件bug归根结底源自于某段代码以一种未曾预期的方式修改了位于另一段代码中的数据。这类bug在我们当前所处的多核世界中愈演愈烈。而一旦把数据变得不可变，就可以避免所有这一类的bug。

使用不可变数据也是Java世界中被反复提及的一个建议；该建议曾在《Effective Java中文版》的“第15条：使可变性最小化”中有所提及，但是却很少有人遵守该建议。很大程度

上是由于Java在设计上并没有过多地考虑不可变性，从而需要程序员花费大量的精力才能获得不可变性。

即便如此，一些流行且高质量的代码库，例如Joda-Time和Google的collection库都为实施不可变数据编程提供了很好的支持。事实上，这些流行的代码库为Java标准库的部分功能提供了对应的可选方案，从而有效地证明了不可变数据的作用。

值得庆幸的是，Scala和Clojure都具备了更多对不可变数据的头等支持，从某种程度上来说，可变数据通常比不可变数据更难使用。

View接口见如下代码：

JavaExamples/src/main/java/com/mblinn/oo/tinyweb/View.java

```
package com.mblinn.oo.tinyweb;

import java.util.List;
import java.util.Map;

public interface View {
        public String render(Map<String, List<String>> model);
}
```

接下去，我们需要两个类：StrategyView和RenderingStrategy。这两个类被设计用来共同实现策略模式。

RenderingStrategy负责完成实际的视图渲染工作，它将由框架的用户来进行实现。下面是策略模式中对应策略类的一个实例，代码如下所示：

JavaExamples/src/main/java/com/mblinn/oo/tinyweb/RenderingStrategy.java

```
package com.mblinn.oo.tinyweb;

import java.util.List;
import java.util.Map;

public interface RenderingStrategy {

        public String renderView(Map<String, List<String>> model);
}
```

现在来看看RenderingStrategy的代理类StrategyView。该类由框架实现，并负责对RenderingStrategy抛出的异常进行相应的处理。它的代码如下：

JavaExamples/src/main/java/com/mblinn/oo/tinyweb/StrategyView.java

```
package com.mblinn.oo.tinyweb;

import java.util.List;
```

```
import java.util.Map;

public class StrategyView implements View {

        private RenderingStrategy viewRenderer;

        public StrategyView(RenderingStrategy viewRenderer) {
                this.viewRenderer = viewRenderer;
        }

        @Override
        public String render(Map<String, List<String>> model) {
                try {
                        return viewRenderer.renderView(model);
                } catch (Exception e) {
                        throw new RenderingException(e);
                }
        }
}
```

为了实现视图，框架的用户创建了RenderingStrategy的子类。该子类拥有正确的视图渲染逻辑，而框架会将该子类的实例注入到StrategyView之中。

在这个简单的例子中，StrategyView没有起到很大的作用。它简单地吞掉RenderingStrategy子类抛出的异常，并使用RenderingException对它们进行包装，然后再次抛出包装后的异常，以便于它们能在更上层的代码中被正确地处理。而对于一个更加完整的框架来说，除此之外，它还会将StrategyView作为一个可供各种渲染引擎接入的集成点。但在此处，我们还是尽量让StrategyView保持简单。

控制器和模板方法模式

接下来是我们的控制器。Controller本身是只有一个方法handleRequest()的简单接口，该方法以HttpRequest作为入参，返回的是HttpResponse。该接口的代码如下所示：

JavaExamples/src/main/java/com/mblinn/oo/tinyweb/Controller.java

```
package com.mblinn.oo.tinyweb;

public interface Controller {
        public HttpResponse handleRequest(HttpRequest httpRequest);
}
```

我们将使用模板方法模式，从而让用户可以在他们自己的控制器中实现doRequest的相应逻辑。该实现的核心类是TemplateController，它具有一个抽象方法doRequest()，代码如下所示：

JavaExamples/src/main/java/com/mblinn/oo/tinyweb/TemplateController.java

```
package com.mblinn.oo.tinyweb;
```

```
import java.util.List;
import java.util.Map;

public abstract class TemplateController implements Controller {
        private View view;
        public TemplateController(View view) {
                this.view = view;
        }

        public HttpResponse handleRequest(HttpRequest request) {
                Integer responseCode = 200;
                String responseBody = "";

                try {
                        Map<String, List<String>> model = doRequest(request);
                        responseBody = view.render(model);
                } catch (ControllerException e) {
                        responseCode = e.getStatusCode();
                } catch (RenderingException e) {
                        responseCode = 500;
                        responseBody = "Exception while rendering.";
                } catch (Exception e) {
                        responseCode = 500;
                }

                return HttpResponse.Builder.newBuilder().body(responseBody)
                    .responseCode(responseCode).build();
        }
        protected abstract Map<String, List<String>> doRequest(HttpRequest request);
}
```

为了实现控制器，框架用户需要继承TemplateController并实现它的doRequest()方法。

我们在实现控制器中所使用的模板方法模式和在实现视图中所使用的策略模式都是为了完成相似的任务。它们允许我们拥有一些通用的代码，这些代码可能来自代码库或框架，并对另一些用以执行特定任务的代码起到了代理作用。其中模板方法模式采用继承来完成工作，而策略模式则是通过组合来完成工作的。

在函数式的世界里，我们将大量地依赖组合，这在面向对象的世界里同样也是一种很好的实践。然而，前者将会是对函数的组合，而后者则是对对象的组合。

过滤器和函数式接口

最后，让我们来看看过滤器。Filter类是一个函数式接口，它允许我们在处理HttpRequest之前可以对其执行一定的操作。举例来说，我们可能会在日志中记录一些有关请求的信息，甚至是向请求添加请求头部。Filter类只有一个方法：doFilter()。该方法以HttpRequest为入参，返回的是一个经过过滤的HttpRequest实例。

如果某个独立的过滤器需要对请求进行修改，它会基于现有的请求创建一个新的请求实例，然后返回该实例。这让我们既用上了不可变的HttpRequest，同时也给了我们一个“请求可以被修改”的错觉。

Filter的代码如下所示：

JavaExamples/src/main/java/com/mblinn/oo/tinyweb/Filter.java

```
package com.mblinn.oo.tinyweb;

public interface Filter {
        public HttpRequest doFilter(HttpRequest request);
}
```

现在，我们已经看过TinyWeb的每个组件，接下来看看这些组件是如何组合在一起的。

将所有组件拼装到一起

为了将所有组件拼装到一起，我们将会使用一个主类：TinyWeb。该类的构造器拥有两个参数。第一个参数是一个Map，该Map的键是用于表示请求路径的字符串，而值是对应的Controller对象。第二个参数是一个过滤器的列表，这些过滤器会在每个请求被传入合适的控制器之前得到运行。

TinyWeb类拥有一个公共方法：handleRequest()。该方法以HttpRequest作为入参，然后将每个传入的请求都经过过滤器的运行，完成过滤后为每个请求找到合适的控制器来对其进行处理，最后返回生成的HttpResponse。这部分代码如下所示：

JavaExamples/src/main/java/com/mblinn/oo/tinyweb/TinyWeb.java

```
package com.mblinn.oo.tinyweb;

import java.util.List;
import java.util.Map;

public class TinyWeb {
        private Map<String, Controller> controllers;
        private List<Filter> filters;

        public TinyWeb(Map<String, Controller> controllers, List<Filter> filters) {
                this.controllers = controllers;
                this.filters = filters;
        }

        public HttpResponse handleRequest(HttpRequest httpRequest) {

                HttpRequest currentRequest = httpRequest;
                for (Filter filter : filters) {
                        currentRequest = filter.doFilter(currentRequest);
                }
```

```
                Controller controller = controllers.get(currentRequest.getPath());

                if (null == controller)
                        return null;

                return controller.handleRequest(currentRequest);
        }
}
```

一个功能完备的Java Web框架不会像上面的代码那样直接将一个类暴露出来作为框架的管线装置，而是会使用一系列配置文件和注解将各个组件装配在一起。不管怎样，我们现在将停止向TinyWeb增加功能，接下来看一个使用TinyWeb框架的例子。

使用TinyWeb

让我们来实现一个例子程序，该程序将以一个HttpRequest作为入参，其中HttpRequest的请求体是一个以逗号分隔的名字列表，然后该程序会返回一段为这些名字生成的问候语。我们还将添加一个过滤器，该过滤器会以日志方式打印请求的路径信息。

我们从GreetingController开始看起。当控制器收到HttpRequest之后，它会从请求中获取请求体，然后以逗号为分隔符将请求体的内容进行切割，并将切割后生成的每个元素作为名字处理。接着，GreetingController会为每个名字都生成随机的问候语，并将这些问候语作为值，以键名“greetings”放入模型之中。具体代码如下所示：

JavaExamples/src/main/java/com/mblinn/oo/tinyweb/example/GreetingController.java

```
package com.mblinn.oo.tinyweb.example;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Random;

import com.mblinn.oo.tinyweb.HttpRequest;
import com.mblinn.oo.tinyweb.TemplateController;
import com.mblinn.oo.tinyweb.View;

public class GreetingController extends TemplateController {
        private Random random;
        public GreetingController(View view) {
                super(view);
                random = new Random();
        }

        @Override
        public Map<String, List<String>> doRequest(HttpRequest httpRequest) {
                Map<String, List<String>> helloModel =
                                new HashMap<String, List<String>>();
```

```
            helloModel.put("greetings",
                    generateGreetings(httpRequest.getBody()));
            return helloModel;
        }

        private List<String> generateGreetings(String namesCommaSeperated) {
            String[] names = namesCommaSeperated.split(",");
            List<String> greetings = new ArrayList<String>();
            for (String name : names) {
                greetings.add(makeGreeting(name));
            }
            return greetings;
        }

        private String makeGreeting(String name) {
            String[] greetings =
                    { "Hello", "Greetings", "Salutations", "Hola" };
            String greetingPrefix = greetings[random.nextInt(4)];
            return String.format("%s, %s", greetingPrefix, name);
        }
}
```

接下来看看GreetingRenderingStrategy。该类会对控制器生成的问候语列表进行逐个遍历，然后将每一段问候语都放入<h2>标记中。最后在这些<h2>标记前加上一个包含"Friendly Greetings:"的<h1>标记，如下面的代码所示：

JavaExamples/src/main/java/com/mblinn/oo/tinyweb/example/GreetingRenderingStrategy.java

```
package com.mblinn.oo.tinyweb.example;

import java.util.List;
import java.util.Map;

import com.mblinn.oo.tinyweb.RenderingStrategy;

public class GreetingRenderingStrategy implements RenderingStrategy {

        @Override
        public String renderView(Map<String, List<String>> model) {
            List<String> greetings = model.get("greetings");
            StringBuffer responseBody = new StringBuffer();
            responseBody.append("<h1>Friendly Greetings:</h1>\n");
            for (String greeting : greetings) {
                responseBody.append(
                        String.format("<h2>%s</h2>\n", greeting));

            }
            return responseBody.toString();
        }
}
```

最后，让我们来看看过滤器的例子。LoggingFilter类会将请求的路径打印出来，如下面的代码所示：

JavaExamples/src/main/java/com/mblinn/oo/tinyweb/example/LoggingFilter.java

```
package com.mblinn.oo.tinyweb.example;

import com.mblinn.oo.tinyweb.Filter;
import com.mblinn.oo.tinyweb.HttpRequest;

public class LoggingFilter implements Filter {

        @Override
        public HttpRequest doFilter(HttpRequest request) {
                System.out.println("In Logging Filter - request for path: "
                    + request.getPath());
                return request;
        }
}
```

我们装配起一个能将所有组件都与TinyWeb贯串起来的简单测试装置。然后抛给它一个HttpRequest，并将收到的响应打印在控制台，我们可以看到如下输出。该输出表明一切工作正常：

```
In Logging Filter - request for path: greeting/
responseCode: 200
responseBody:
<h1>Friendly Greetings:</h1>
<h2>Hola, Mike</h2>
<h2>Greetings, Joe</h2>
<h2>Hola, John</h2>
<h2>Salutations, Steve</h2>
```

我们了解了采用Java编写的TinyWeb框架，接下来看看如何采用函数式的一些特性来替代面向对象的模式。这将会给我们带来一个与当前框架功能上等效的新的TinyWeb框架，但是用于编写该框架的代码会更加简短，并且采用的是一种更具声明性且更容易阅读的风格。

2.3 采用 Scala 来编写 TinyWeb

让我们动手开始将Java版本的TinyWeb转换成Scala版本。我们将会通过逐步的微量修改来展示Scala代码是如何与现有的Java代码相互配合的。而框架的整体结构将与Java版本类似，但是我们会利用一些Scala的函数式特性来使代码变得更加简洁。

第一步：更换视图

我们将从视图的代码着手。在Java中，我们使用了传统的策略模式。而在Scala中，我们将继续采用策略模式，但是我们将使用高阶函数来实现策略。我们将会看到表达式相比控制流语句所带来的更多好处。

这里所做的最大的修改是视图渲染的代码。我们将放弃RenderingStrategy中原有的函数

式接口方式，转而使用一个高阶函数来实现相同功能。我们将在模式1“替代函数式接口”一节中对这一替代模式进行更加详细的讨论。

下面是我们以函数式之名修改后的视图代码：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tinyweb/stepone/View.scala

```
package com.mblinn.mbfpp.oo.tinyweb.stepone
import com.mblinn.oo.tinyweb.RenderingException

trait View {
  def render(model: Map[String, List[String]]): String
}
class FunctionView(viewRenderer: (Map[String, List[String]]) => String)
        extends View {
  def render(model: Map[String, List[String]]) =
    try
        viewRenderer(model)
    catch {
        case e: Exception => throw new RenderingException(e)
    }
}
```

我们从View 特质开始入手，该特质定义了一个render()方法，该方法以一个map作为入参来表示模型中的数据，并返回一段渲染后的字符串。

```
trait View {
  def render(model: Map[String, String]): String
}
```

接下来，让我们来看看FunctionView，下面的代码声明了一个类，该类拥有一个具有单个入参的构造器，参数名为viewRenderer，它会为该类设置一个同名的不可变字段。

```
class FunctionView(viewRenderer: (Map[String, String]) => String) extends View {
  «classBody»
}
```

viewRenderer这一函数参数拥有一个看上去相当奇怪的类型标注：(Map[String, String]) => String。这是一个函数类型，它说明viewRenderer是一个函数，它的入参是Map[String, String]类型，而返回的是一个String类型，就跟Java版RenderingStrategy中的renderView()一样。

接下来看看render()方法本身。如以下代码所示，该方法以model为入参，并将该model传入viewRender()函数中运行。

```
def render(model: Map[String, String]) =
  try
    viewRenderer(model)
  catch {
    case e: Exception => throw new RenderingException(e)
  }
```

请注意，上述代码片段中为什么没有看到return关键字？该场景也阐述了函数式编程的另一个重要方面。在函数式的世界里，我们主要是采用表达式来编程的。而该函数的值正好是函数中最后一个表达式的值。

在这个例子中，表达式恰好就是一个try代码块。如果没有异常抛出的话，该try代码块就会采用主分支的值；否则，它会选用catch分支中对应的case子句的值。

如果我们不再将异常打包进一个RenderException，而是想自己来为异常情况提供一个默认值，那么只需要让相应的case子句默认采用我们的值，如下面代码所示：

```
try
  viewRenderer(model)
catch {
  case e: Exception => ""
}
```

现在只要捕获到异常，try代码块就会采用一个空字符串来作为其值。

第二步：对控制器的第一次改造

现在来看看如何将我们的控制器代码转换成Scala。在Java中，我们使用了Controller接口和TemplateController类。每个控制器都通过继承TemplateController来实现。

使用Scala创建控制器时，我们依靠函数组成来实现，就像在实现视图时所做的，我们传入了一个doRequest()函数。

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tinyweb/steptwo/Controller.scala

```
package com.mblinn.mbfpp.oo.tinyweb.steptwo

import com.mblinn.oo.tinyweb.HttpRequest
import com.mblinn.oo.tinyweb.HttpResponse
import com.mblinn.oo.tinyweb.ControllerException
import com.mblinn.oo.tinyweb.RenderingException

trait Controller {
  def handleRequest(httpRequest: HttpRequest): HttpResponse
}

class FunctionController(view: View, doRequest: (HttpRequest) =>
  Map[String, List[String]] ) extends Controller {

  def handleRequest(request: HttpRequest): HttpResponse = {
    var responseCode = 200;
    var responseBody = "";

    try {
      val model = doRequest(request)
      responseBody = view.render(model)
    } catch {
```

```
        case e: ControllerException =>
          responseCode = e.getStatusCode()
        case e: RenderingException =>
          responseCode = 500
          responseBody = "Exception while rendering."
        case e: Exception =>
          responseCode = 500
      }

      HttpResponse.Builder.newBuilder()
                  .body(responseBody).responseCode(responseCode).build()
    }
  }
```

这份代码应该看上去与我们的视图代码非常相似。这简直就是从字面上将Java翻译成了Scala。但是这么做还不够函数式味，因为我们将try-catch块作为了语句来使用，并以这种方式来给responseCode和responseBody赋值。

我们同样还复用了Java的HttpRequest和HttpResponse。Scala提供了一种更加简明的方式来创建这些承载数据的类，我们称之为条件类（case class）。如果将以语句方式使用的try-catch代码块切换为表达式的方式，同时采用条件类，将帮助我们大大减少代码量。

我们将在下一步转换中应用这两种改造方式。

不可变的HttpRequest和HttpResponse

首先将使用生成器模式的方式切换为条件类的方式。代码非常简单，如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tinyweb/stepthree/HttpData.scala

```
package com.mblinn.mbfpp.oo.tinyweb.stepthree

case class HttpRequest(headers: Map[String, String], body: String, path: String)
case class HttpResponse(body: String, responseCode: Integer)
```

我们可以简单地创建新的HttpRequest和HttpResponse对象，如下面的REPL输出所示：

```
scala> val request = HttpRequest(Map("X-Test" -> "Value"), "requestBody", "/test")
request: com.mblinn.mbfpp.oo.tinyweb.stepfour.HttpRequest =
        HttpRequest(Map(X-Test -> Value),requestBody,/test)

scala> val response = HttpResponse("requestBody", 200)
response: com.mblinn.mbfpp.oo.tinyweb.stepfour.HttpResponse =
        HttpResponse(requestBody,200)
```

乍一看，除了不需要使用new关键词之外，上面的代码与使用带有构造器参数的Java类非常相似。不管怎样，我们将在模式4“替代生成器模式来获得不可变对象”中会进行更加深入的讨论，来了解Scala的一些能力和特性是如何满足生成器模式的原有意图的。这些能力和特性包括：Scala在构造器中提供默认参数的能力、条件类天然的不变性，以及从一个现有实例简单创建条件

类新实例的能力。

再来看看第二种改造方式。因为Scala中的try-catch代码块拥有一个值，我们将以表达式的方式来使用它。乍一看，这似乎有点奇怪，但重点是我们可以利用Scala的try-catch代码块是一个表达式的事实，来简单地以HttpResponse的值作为try-catch代码块的值。代码如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tinyweb/stepthree/Controller.scala

```
package com.mblinn.mbfpp.oo.tinyweb.stepthree
import com.mblinn.oo.tinyweb.ControllerException
import com.mblinn.oo.tinyweb.RenderingException

trait Controller {
  def handleRequest(httpRequest: HttpRequest): HttpResponse
}
class FunctionController(view: View, doRequest: (HttpRequest) =>
  Map[String, List[String]] ) extends Controller {
  def handleRequest(request: HttpRequest): HttpResponse =
    try {
      val model = doRequest(request)
      val responseBody = view.render(model)
      HttpResponse(responseBody, 200)
    } catch {
      case e: ControllerException =>
        HttpResponse("", e.getStatusCode)
      case e: RenderingException =>
        HttpResponse("Exception while rendering.", 500)
      case e: Exception =>
        HttpResponse("", 500)
    }
}
```

这种编程风格具有两个好处。首先，我们可以消除掉两个额外的变量：responseCode和responseBody。其次，返回响应的代码变得更加清晰。程序员原本需要通过通读整个方法来理解返回的请求响应，而现在他只需要查看一行代码就够了。

原本为了理解方法最后返回的HttpResponse，我们的代码阅读顺序需要从方法顶部try代码块中的responseCode和responseBody的值开始，一直追踪到最后的HttpResponse。而经过上面的改造之后，我们只需要看一下try代码块中的相应片段，就可以理解HttpResponse最后的值了。这些改造的结合，让我们的代码变得更具可读性，并且更加简洁。

将组件拼装到一起

现在让我们添加一个TinyWeb类来将所有的组件拼装到一起。跟它的Java版本一样，TinyWeb由一个控制器的map和一个过滤器的列表来初始化。与Java版本不同的是，我们没有为过滤器定义一个新的类，只是简单地使用了一组高阶函数。

还与Java版本相似的是Scala的TinyWeb也拥有一个单独的handleRequest()方法，该方法以

HttpRequest为入参。相比原来直接返回一个HttpResponse，我们现在返回的是一个Option[HttpResponse]，这种方式给我们带来了一个好处，即在我们无法为特定请求找到对应的控制器时，可以更加清晰地对问题进行处理。Scala版本的TinyWeb的代码如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tinyweb/stepfour/Tinyweb.scala

```
package com.mblinn.mbfpp.oo.tinyweb.stepfour
class TinyWeb(controllers: Map[String, Controller],
    filters: List[(HttpRequest) => HttpRequest]) {

  def handleRequest(httpRequest: HttpRequest): Option[HttpResponse] = {
    val composedFilter = filters.reverse.reduceLeft(
        (composed, next) => composed compose next)
    val filteredRequest = composedFilter(httpRequest)
    val controllerOption = controllers.get(filteredRequest.path)
    controllerOption map { controller => controller.handleRequest(filteredRequest) }
  }
}
```

让我们从类的定义开始深入查看一下它的细节。

```
class TinyWeb(controllers: Map[String, Controller],
    filters: List[(HttpRequest) => HttpRequest]) {
        «classBody»
}
```

这里我们定义了一个接收两个构造器参数的类，分别是一个控制器的map和一个过滤器的列表。请注意filters参数的类型：List[(HttpRequest) => HttpRequest]。这说明filters是一组函数，这些函数都以HttpRequest为入参，并且返回的也是HttpRequest。

接下来看看handleRequest()方法的签名：

```
def handleRequest(httpRequest: HttpRequest): Option[HttpResponse] = {
        «functionBody»
}
```

正如先前所提到的，我们返回Option[HttpResponse]类型替代原来的HttpResponse类型。这个Option类型是一个容器类型，它拥有两个子类型：Some和None。如果我们获取到了想要存入该类型的值，就将该值存储在一个Some的实例中；否则，使用None来表明我们没有获取到任何实际的值。我们将会在模式8“替代空对象”中对Option进行更加深入的讨论。

现在我们已经看过了TinyWeb框架，让我们来实战一下。我们将使用与Java章节相同的例子来返回一组问候语。然而，由于这次使用的是Scala，我们可以在REPL中来实践我们的例子。首先从视图的代码开始。

使用Scala版本的TinyWeb

让我们来看看如何使用Scala版本的TinyWeb框架。

我们先来创建一个FunctionView和一个用以组成该视图的渲染函数。下面的代码创建了该函数，我们将它命名为greetingViewRenderer()，同时还有跟它一起的FunctionView：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tinyweb/example/Example.scala

```
def greetingViewRenderer(model: Map[String, List[String]]) =
  "<h1>Friendly Greetings:%s".format(
      model
      getOrElse("greetings", List[String]())
      map(renderGreeting)
      mkString ", ")

private def renderGreeting(greeting: String) =
  "<h2>%s</h2>".format(greeting)

def greetingView = new FunctionView(greetingViewRenderer)
```

这里使用了一些本书中尚未介绍过的Scala新特性。首先，我们引入了map()方法，该方法可以让我们将传入的函数作用于某个序列中的所有元素，并返回一组映射后的新序列。其次，我们使用了一种Scala的语法糖，它允许我们将任何只有单个参数的方法作为一个中缀操作符来使用。该操作符左侧的对象被视为方法调用的接收者，而右侧的对象则被视为方法的入参。

这种语法意味着我们可以在使用Scala时省略掉我们所熟悉的点语法。举个例子，map()方法的以下两种用法是等价的：

```
scala> val greetings = List("Hi!", "Hola", "Aloha")
greetings: List[java.lang.String]

scala> greetings.map(renderGreeting)
res0: List[String] = List(<h2>Hi!</h2>, <h2>Hola</h2>, <h2>Aloha</h2>)

scala> greetings map renderGreeting
res1: List[String] = List(<h2>Hi!</h2>, <h2>Hola</h2>, <h2>Aloha</h2>)
```

现在让我们来看看控制器的代码。在下面的代码中，我们创建了handleGreetingRequest()函数，并将它传给了控制器。我们使用makeGreeting()来作为帮助方法，它接收一个名字并生成一段随机的问候语。

在方法handleGreetingRequest()中，我们创建了一个名字列表。在这个过程中，首先与Java版一样通过切割请求体返回了一个数组，然后将该数组转换成一个Scala list，并使用makeGreeting()函数来对该list中的每个元素进行映射处理。接着，便可以以"greetings"作为模型map的键，而以映射处理返回的list作为模型map的值：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tinyweb/example/Example.scala

```
def handleGreetingRequest(request: HttpRequest) =
  Map("greetings" -> request.body.split(",").toList.map(makeGreeting))

private def random = new Random()
private def greetings = Vector("Hello", "Greetings", "Salutations", "Hola")
```

```
private def makeGreeting(name: String) =
  "%s, %s".format(greetings(random.nextInt(greetings.size)), name)

def greetingController = new FunctionController(greetingView, handleGreetingRequest)
```

Scala函数和方法

由于Scala是一门“混血”的语言，它同时兼具了函数和方法。方法是使用def关键词来进行定义的，如以下代码片段所示：

```
scala> def addOneMethod(num: Int) = num + 1
addOneMethod: (num: Int)Int
```

可以通过使用Scala的匿名函数语法来创建函数并为其命名，同时将生成的函数分配给val变量，如以下代码所示：

```
scala> val addOneFunction = (num: Int) => num + 1
addOneFunction: Int => Int = <function1>
```

我们几乎总是能将方法作为高阶函数使用。举个例子，此处我们将方法和函数两个版本的addOne()都传给了map()。

```
scala> val someInts = List(1, 2, 3)
someInts: List[Int] = List(1, 2, 3)

scala> someInts map addOneMethod
res1: List[Int] = List(2, 3, 4)

scala> someInts map addOneFunction
res2: List[Int] = List(2, 3, 4)
```

由于方法的定义具有更加清晰的语法，所以每当我们需要定义函数时，都会使用定义方法的语法，而不是定义函数的语法。每当我们需要将一个方法手动转换成一个函数时，可以通过使用下划线操作符来达到目的，如下面的REPL会话所示：

```
scala> addOneMethod _
res3: Int => Int = <function1>
```

由于大部分情况下，Scala都能智能地做好自动转换，所以上述需求不是很常见。

最后，来看看我们的日志过滤器。该函数从传入的HttpRequest中获取到请求路径，然后将该路径打印到控制台，最后返回未经修改的请求：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tinyweb/example/Example.scala

```
private def loggingFilter(request: HttpRequest) = {
  println("In Logging Filter - request for path: %s".format(request.path))
  request
}
```

为了结束本示例，我们需要创建一个TinyWeb实例，而该实例必须配备了我们定义过的那些控制器、视图和过滤器。同时，还需要创建一个测试的HttpRequest：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tinyweb/example/Example.scala

```
def tinyweb = new TinyWeb(
    Map("/greeting" -> greetingController),
    List(loggingFilter))
def testHttpRequest = HttpRequest(
    body="Mike,Joe,John,Steve",
    path="/greeting")
```

现在可通过在REPL中运行TinyWeb的handleRequest()方法来测试请求，并可以看到相应的HttpResponse：

```
scala> tinyweb.handleRequest(testHttpRequest)
In Logging Filter - request for path: /greeting
res0: Option[com.mblinn.mbfpp.oo.tinyweb.stepfour.HttpResponse] =
Some(HttpResponse(<h1>Friendly Greetings:<h2>Mike</h2>, <h2>Nam</h2>, <h2>John</h2>,
200))
```

以上就是我们的Scala版TinyWeb。我们对原来在Java版本中所使用的风格做了一些改造。首先，将大部分循环代码都替换成了更具声明性的代码。其次，采用条件类来替换了笨重的生成器。最后，采用普通的老式函数替代了函数式接口。

总体看来，这些细小的改造帮我们节省了大量的代码，同时也让代码逻辑变得更加简短和易于阅读。下一步，我们将去看看Clojure版本的TinyWeb。

2.4 采用 Clojure 来编写 TinyWeb

现在让我们来将TinyWeb翻译成Clojure版本。相比于将TinyWeb从Java翻译成Scala，这将是一次更大的飞跃，所以我们会循序渐进地逐步深入。

Clojure和Java最明显的区别要数语法。Clojure与那些源自C语言语法的大多数现代编程语言有很大不同。这并非偶然：它的语法成就了Clojure最强大的特性之一：宏（macro）。我们将会在模式21“领域特定语言”中对其进行讨论。

Clojure简介

现在，先让我们来看一段简短的介绍。Clojure使用了前缀语法（prefix syntax），这意味着在一个函数调用中，函数名会出现在函数参数的前面。下面是一段示例代码，我们通过调用count函数来获得vector（Clojure的一种不可变数据结构）的大小：

```
=> (count [1 2 3 4])
4
```

与Scala一样，Clojure与现有Java代码有着很好的互操作性。调用一个Java类的方法几乎与调用一个Clojure函数是一样的；你只需要将带有点号的方法名置于类实例之前，而不是像Java中那样置于其后。举个例子，下面是我们如何在Java `String`类实例上调用`length()`方法的一个例子：

```
=> (.length "Clojure")
7
```

相比于在Java中将代码组织为对象和方法，以及在Scala中将代码组织为对象、方法和函数，Clojure将代码组织为函数与命名空间。我们Clojure版本的TinyWeb跟Java和Scala的版本一样，也基于模型、视图、控制器和过滤器等组件。然而，这些组件都将以一种不同的形式来呈现。

我们的视图、控制器和过滤器代码仅仅是一些函数，而我们的模型都是map。我们使用了一个名为TinyWeb的函数来将所有的组件装配到一起，该函数将以所有的组件为入参，并返回一个接收HTTP请求的函数。该函数首先将传入请求通过过滤器的运行过滤，然后再将请求路由到合适的控制器和视图。

Clojure中的控制器

首先从控制器入手来看一下Clojure代码。如下面的代码所示，我们实现了一个简单的控制器，该控制器将一个传入HTTP请求的请求体设置为模型的值，并在模型中为它设置了一个名字。在我们的首次迭代开发中，我们将使用与Java代码中相同的`HttpRequest`，并在后续过程中对其进行修改，使它更符合Clojure的惯用风格：

ClojureExamples/src/mbfpp/oo/tinyweb/stepone.clj

```
(ns mbfpp.oo.tinyweb.stepone
  (:import (com.mblinn.oo.tinyweb HttpRequest HttpRequest$Builder)))
(defn test-controller [http-request]
  {:name (.getBody http-request)})
(def test-builder (HttpRequest$Builder/newBuilder))
(def test-http-request (.. test-builder (body "Mike") (path "/say-hello") build))
(defn test-controller-with-map [http-request]
  {:name (http-request :body)})
```

让我们从命名空间开始，逐行地来看看代码。

ClojureExamples/src/mbfpp/oo/tinyweb/stepone.clj

```
(ns mbfpp.oo.tinyweb.stepone
  (:import (com.mblinn.oo.tinyweb HttpRequest HttpRequest$Builder)))
```

此处我们定义了一个名为`mbfpp.oo.tinyweb.stepone`的命名空间。一个命名空间就是指代码库中的一个函数集合，它可以在其他命名空间中被完全或部分地导入。

作为定义的一部分，我们导入了两个Java类：`HttpRequest`和`HttpRequest$Builder`。其中第二个类也许看上去会有点奇怪，但它只不过是我们在`HttpRequest`中作为其中一部分创建的一个静态内部类`Builder`的全名。Clojure并没有为静态内部类提供任何特殊的语法，所以我们只能

使用完整类名。

而关键词:import则是一个Clojure关键字的例子。一个关键字就是提供了非常快速的相等性检查的标识符，而且通常前面会带上一个冒号。这里我们使用了:import关键字来表明哪些类应该被导入到我们刚才声明的命名空间中去。但是除此之外，关键字还有很多其他的用途。例如，它们通常在map中被作为为键来使用。

现在来看看控制器，它接收一个来自原有Java版本的HttpRequest，然后产出一个Clojure map作为模型：

ClojureExamples/src/mbfpp/oo/tinyweb/stepone.clj

```
(defn test-controller [http-request]
  {:name (.getBody http-request)})
```

此处，我们调用HttpRequest的getBody()方法来获得请求体，并使用它来创建一个具有单对键值对的map。而map的键是关键字:name，值是HttpRequest的字符串请求体。

在我们继续前进之前，先进一步来看看Clojure map。在Clojure中，通常都使用map来传递数据，而在Clojure中创建map的语法便是在键值对外围加上一对花括号。举个例子，下面是我们创建的一个具有两对键值对的map。第一对的键是关键字:name，而值是字符串"Mike"；第二对的键是关键字:sex，而值是另一个关键字:male，如下所示：

```
=> {:name "Mike" :sex :male}
{:name "Mike" :sex :male}
```

map在Clojure中相当于它们键的函数。这意味着我们可以将map作为函数来调用，传入一个期望的键（前提是该键存在于该map之中），该map将会返回对应的值。如果传入的键不存在于map，将会返回nil，代码如下所示：

```
=> (def test-map {:name "Mike"})
#'mbfpp.oo.tinyweb.stepone/test-map
=> (test-map :name)
"Mike"
=> (test-map :orange)
nil
```

关键字在Clojure中也同样是函数。当我们将map传递给关键字时，它们将会查看自己是否存在于该map之中，正如下面的代码片段，它展示了从map中查找值的最通用的方式：

```
=> (def test-map {:name "Mike"})
#'mbfpp.oo.tinyweb.stepone/test-map
=> (:name test-map)
"Mike"
=> (:orange test-map)
nil
```

现在来创建一些测试数据。如下所示，我们创建了一个HttpRequest$Builder并使用它来创建新的HttpRequest：

ClojureExamples/src/mbfpp/oo/tinyweb/stepone.clj

```
(def test-builder (HttpRequest$Builder/newBuilder))
(def test-http-request (.. test-builder (body "Mike") (path "/say-hello") build))
```

这段代码使用了两个新的Clojure与Java之间的互操作特性。首先，方法前的斜杠让我们可以调用一个类的静态方法或引用一个静态变量。因此片段(HttpRequest$Builder/newBuilder)是指调用在类HttpRequest$Builder中的方法newBuilder()。在下面的例子中，我们也使用了这种语法，通过调用Integer类上的parseInt()方法将一个字符串解析成了整数。

```
=> (Integer/parseInt "42")
42
```

另一个特性是..宏，这是一个相当便捷的互操作特性，可以非常容易地在Java对象上调用一连串方法。它的工作方式是将第一个参数传递给..，然后将对后续参数的调用贯串起来。

在代码片段snippet (.. test-builder (body "Mike") (path "/say-hello") build)中，首先以"Mike"作为入参调用test-builder上的方法body()。获得结果之后，以入参"say-hello"调用该返回结果上的path()方法，最后在path()返回的结果上调用build()方法，并返回一个HttpResult的实例。

下面是使用..宏的另一个例子，这段代码将字符串"mike"转化成大写形式，然后取出了它的第一个字符：

```
=> (.. "mike" toUpperCase (substring 0 1))
"M"
```

数据map

我们了解了一些基本的Clojure和Clojure/Java的互操作性，接下来进入将TinyWeb转换成Clojure版本的下一环节。我们将对test-controller进行改造，它也将接收一个map作为入参，就像它所返回的模型一样。我们将同样引入一个view函数和一个render函数，主要负责对视图进行调用。下一个迭代版本的代码如下所示：

ClojureExamples/src/mbfpp/oo/tinyweb/steptwo.clj

```
(ns mbfpp.oo.tinyweb.steptwo
  (:import (com.mblinn.oo.tinyweb RenderingException)))

(defn test-controller [http-request]
  {:name (http-request :body)})

(defn test-view [model]
  (str "<h1>Hello, " (model :name) "</h1>"))

(defn- render [view model]
  (try
    (view model)
    (catch Exception e (throw (RenderingException. e)))))
```

让我们进一步来看看上面的代码，从新的test-controller开始。正如我们所看到的，我们期望http-request是一个以:body为键的map，它代表了HTTP请求的请求体。我们通过该键从map中获取到值之后，又将该值放入了一个代表模型的新map之中：

ClojureExamples/src/mbfpp/oo/tinyweb/steptwo.clj

```
(defn test-controller [http-request]
  {:name (http-request :body)})
```

通过REPL可以很容易探究test-controller是如何运作的。我们需要做的就是定义一个名为test-http-request的map，然后将它传给test-controller，下面就是我们在REPL运行的代码及其输出：

```
=> (def test-http-request {:body "Mike" :path "/say-hello" :headers {}})
#'mbfpp.oo.tinyweb.steptwo/test-http-request
=> (test-controller test-http-request)
{:name "Mike"}
```

Clojure版本的视图

现在我们的控制器已经差不多圆满完成了，接下来看看视图的代码。跟控制器一样，视图也都将会是函数。它们对代表模型的map进行处理，然后返回一个代表视图输出的字符串。

下面是一个名为test-view的简单视图，它采用<h1>标记来对名字信息进行了包装：

ClojureExamples/src/mbfpp/oo/tinyweb/steptwo.clj

```
(defn test-view [model]
  (str "<h1>Hello, " (model :name) "</h1>"))
```

跟前面一样，我们可以继续在REPL中进行测试。首先定义一个测试模型，然后将它传递给该函数：

```
=> (def test-model {:name "Mike"})
#'mbfpp.oo.tinyweb.steptwo/test-model
=> (test-view test-model)
"<h1>Hello, Mike</h1>"
```

离完成我们的视图处理代码还差一点点。在Scala中，我们使用了模式7“替代策略模式”来确保任何视图处理器中的异常都会被正确地包装在一个RenderingException之中。在Clojure中，我们将用高阶函数来完成相同的事情。正如下面的代码所示，我们需要做的就是将view函数传给render函数，后者将负责运行视图以及对任何异常的包装：

ClojureExamples/src/mbfpp/oo/tinyweb/steptwo.clj

```
(defn- render [view model]
  (try
    (view model)
    (catch Exception e (throw (RenderingException. e)))))
```

将所有组件拼装到一起

现在，我们已经搞定了Clojure视图和控制器，让我们再添加上过滤器，以及一些能将所有的组件拼装起来的胶水代码，从而来完成这个例子。我们将在名为core的命名空间中来完成这最后一步。这是当你创建新项目时，Clojure构建工具Leiningen创建的标准核心命名空间，所以这也成了Clojure项目中事实上的标准核心命名空间。

为了做到这一点，我们将添加一个execute-request函数，它的职责是执行http-request。该函数接收一个http-request和一个请求处理器作为入参。请求处理器就是一个map，该map包含了控制器和用于请求处理的视图。

我们也同样需要apply-filters，它以http-request为入参，可以将一系列过滤器应用于该请求，并返回一个新的http-request。最后，还需要一个tinyweb函数。

tinyweb函数是将所有事物都拼装到一起的粘合剂。它接收两个入参：一个请求处理器的map，该map以每个需要处理的请求路径作为key，以及一个过滤器的序列。该函数将返回一个以http-request为入参的函数，并将过滤器序列应用于入参请求，然后将它路由到合适的请求处理器，并最终返回结果。

以下是完整的Clojure TinyWeb库的代码：

ClojureExamples/src/mbfpp/oo/tinyweb/core.clj

```
(ns mbfpp.oo.tinyweb.core
  (:require [clojure.string :as str])
  (:import (com.mblinn.oo.tinyweb RenderingException ControllerException)))
(defn- render [view model]
  (try
    (view model)
    (catch Exception e (throw (RenderingException. e)))))
(defn- execute-request [http-request handler]
  (let [controller (handler :controller)
        view (handler :view)]
    (try
      {:status-code 200
       :body
       (render
         view
         (controller http-request))}
    (catch ControllerException e {:status-code (.getStatusCode e) :body ""})
    (catch RenderingException e {:status-code 500
                                 :body "Exception while rendering"})
    (catch Exception e (.printStackTrace e) {:status-code 500 :body ""}))))
(defn- apply-filters [filters http-request]
  (let [composed-filter (reduce comp (reverse filters))]
    (composed-filter http-request)))
(defn tinyweb [request-handlers filters]
  (fn [http-request]
    (let [filtered-request (apply-filters filters http-request)
```

```
        path (http-request :path)
        handler (request-handlers path)]
    (execute-request filtered-request handler))))
```

上述代码的render方法相比前一个迭代版本而言并没有什么变化，所以让我们先来看看execute-request函数。我们已经在完整的Clojure TinyWeb库中定义了该函数，在分析execute-request函数之前，先让我们在REPL中定义一些测试数据。我们还需要两个函数，即在上一个迭代版本中用于创建测试请求处理器的test-controller和test-view，请看下面的示例代码：

```
=> (defn test-controller [http-request]
{:name (http-request :body)})

(defn test-view [model]
(str "<h1>Hello, " (model :name) "</h1>"))
#'mbfpp.oo.tinyweb.core/test-controller
#'mbfpp.oo.tinyweb.core/test-view
=> (def test-request-handler {:controller test-controller
                              :view test-view})
#'mbfpp.oo.tinyweb.core/test-request-handler
```

现在只需要test-http-request，就可以对我们期望的结果进行验证，即execute-request采用传入的http-request作为入参来运行传入的request-handler所产生的结果：

```
=> (def test-http-request {:body "Mike" :path "/say-hello" :headers {}})
#'mbfpp.oo.tinyweb.steptwo/test-http-request
=> (execute-request test-http-request test-request-handler)
{:status-code 200, :body "<h1>Hello, Mike</h1>"}
```

让我们通过在REPL中对这些函数进行尝试来详细讨论execute-request的组成。首先从let语句开始，该语句将控制器和视图从request-handler中提取了出来，下面是这部分的大致框架：

```
(let [controller (handler :controller)
      view (handler :view)]
«let-body»)
```

let语句是Clojure中用于分配局部名称的一种方式，局部名称就如同Java中的局部变量。然而与变量不同，这些名称所引用的值并不意味着可以被改变。在上述let语句中，我们从request-handler map中提取出视图和控制器函数，并将它们命名为controller和view。然后便可以在let语句中通过这些名称来引用这些函数。

来看一个更加简单的let表达式的例子。如下所示，我们使用let将name与字符串"Mike"进行绑定，并将greeting与字符串"Hello"进行绑定。然后在let表达式内部，我们使用它们创建了一个greeting：

```
=> (let [name "Mike"
         greeting "Hello"]
     (str greeting ", " name))
"Hello, Mike"
```

现在我们已经了解了let，接下来瞧瞧try表达式，即我们在下述示例中所展示的片段。与Scala非常相似，try是一个拥有值的表达式。如果没有异常被抛出，那么try将以表达式本身的值作为返回值；否则将以catch子句的值作为返回值：

```
(try
  {:status-code 200
   :body
     (render
       view
       (controller http-request))}
(catch ControllerException e {:status-code (.getStatusCode e) :body ""})
(catch RenderingException e {:status-code  500
                             :body "Exception while rendering"})
(catch Exception e (.printStackTrace e) {:status-code 500 :body ""})
```

在上面的示例中，如果没有异常抛出，try表达式将以携有两对键值对的map的值作为返回值，该map代表了我们的HTTP响应。map中的第一个键是:status-code，对应的值是200；第二个键是:body，它对应的值由以下过程产生：首先controller函数对传入的http-request进行计算，然后controller将计算结果传递给render函数，最后由render函数将controller的计算结果与view一起渲染产生出该结果值。

可以使用下面的test-view和test-controller来进行实践，从而理解上述的流程：

```
 => (render test-view (test-controller test-http-request))
"<h1>Hello, Mike</h1>"
```

在进行下一步之前，先通过两个更加简单的例子来看看Clojure的异常处理。如下所示，我们看到了一个try表达式的例子，它的主体部分只是一个字符串"hello, world"，所以整个表达式的值便是"hello, world"：

```
=> (try
     "hello, world"
   (catch Exception e (.message e)))
"hello, world"
```

下面是一个关于在主体出现错误时，try表达式将如何运作的简单例子。在如下try表达式的主体中，我们抛出一个携带了消息"It's broke!"的RuntimeException。在catch分支中，我们捕获到了异常并从中提取了消息，而该消息便成为了catch分支的值，因此它也成为了整个try表达式的返回值。

```
=> (try
     (throw (RuntimeException. "It's broke!"))
   (catch Exception e (.getMessage e)))
"It's broke!"
```

接下来看看如何应用过滤器。我们使用了一个apply-filters函数，该函数以一个过滤器序列和一个HTTP请求作为入参，它将这些过滤器混合为一个单独的过滤器，然后将它应用于该HTTP请求。代码如下所示：

```
(defn- apply-filters [filters http-request]
  (let [composed-filter (reduce comp filters)]
       (composed-filter http-request)))
```

我们将在模式16“函数生成器模式”中对comp函数进行深入探讨。

为了完成Clojure TinyWeb实现，我们需要一个函数tinyweb来将所有的组件拼装到一起。该函数以一个请求处理器的map和一个过滤器序列为入参。它同样返回了一个函数，该函数以一个HTTP请求为入参，同时使用了apply-filters来将所有的过滤器应用于该请求。

接着，该函数从HTTP请求中获取请求路径，然后在请求处理器的map中定位到合适的处理器，并使用execute-request来执行该处理器。以下便是tinyweb函数的代码：

```
(defn tinyweb [request-handlers filters]
  (fn [http-request]
    (let [filtered-request (apply-filters filters http-request)
      path (:path http-request)
      handler (request-handlers path)]
        (execute-request filtered-request handler))))
```

使用TinyWeb

让我们来看看如何使用Clojure版本的TinyWeb。首先定义一个供测试用的HTTP请求：

```
=> (def request {:path "/greeting" :body "Mike,Joe,John,Steve"})
#'mbfpp.oo.tinyweb.core/request
```

现在再来看一下控制器的代码，它只不过是一个简单的函数，并且它的工作方式和Scala版本的控制器差不多：

ClojureExamples/src/mbfpp/oo/tinyweb/example.clj

```
(defn make-greeting [name]
  (let [greetings ["Hello" "Greetings" "Salutations" "Hola"]
        greeting-count (count greetings)]
    (str (greetings (rand-int greeting-count)) ", " name)))

(defn handle-greeting [http-request]
  {:greetings (map make-greeting (str/split (:body http-request) #","))})
```

我们通过它来运行我们的测试请求，并返回合适的模型map，如下所示：

```
=> (handle-greeting request)
{:greetings ("Greetings, Mike" "Hola, Joe" "Hola, John" "Hola, Steve")}
```

接下来是视图代码，该代码将模型渲染到HTML。它也是一个函数，以合适的模型map作为入参，并返回一个字符串：

ClojureExamples/src/mbfpp/oo/tinyweb/example.clj

```
(defn render-greeting [greeting]
  (str "<h2>"greeting"</h2>"))

(defn greeting-view [model]
  (let [rendered-greetings (str/join " " (map render-greeting (:greetings model)))]
    (str "<h1>Friendly Greetings</h1> " rendered-greetings)))
```

如果在handle-greeting的输出之上运行greeting-view，将会得到如下渲染后的HTML：

```
=> (greeting-view (handle-greeting request))
"<h1>Friendly Greetings</h1>
<h2>Hola, Mike</h2>
<h2>Hello, Joe</h2>
<h2>Greetings, John</h2>
<h2>Salutations, Steve</h2>"
```

下面来看看logging-filter。这是一个用于在请求返回前记录其访问路径的简单函数：

ClojureExamples/src/mbfpp/oo/tinyweb/example.clj

```
(defn logging-filter [http-request]
  (println (str "In Logging Filter - request for path: " (:path http-request)))
  http-request)
```

最后，将所有组件全都拼装到一个TinyWeb的实例中，代码如下所示：

ClojureExamples/src/mbfpp/oo/tinyweb/example.clj

```
(def request-handlers
  {"/greeting" {:controller handle-greeting :view greeting-view}})
(def filters [logging-filter])
(def tinyweb-instance (tinyweb request-handlers filters))
```

如果我们通过TinyWeb实例来运行测试请求，它将如期得到过滤和处理：

```
=> (tinyweb-instance request)
In Logging Filter - request for path: /greeting
{:status-code 200,
:body "<h1>Friendly Greetings</h1>
<h2>Greetings, Mike</h2>
<h2>Greetings, Joe</h2>
<h2>Hello, John</h2>
<h2>Hola, Steve</h2>"}
```

到此我们就完成了对TinyWeb的讨论！本章中的代码尽可能地保持了简单性，同时我们坚持只使用语言特性的最小集，并且忽略了大量的错误处理及很多有用的特性。不管怎样，通过这些例子，我们清楚地了解了本书将要讨论的一些模式是如何组合在一起的。

在本书的后续部分，我们将在继续函数式编程之旅的同时，进一步讨论我们所提及的这些模式及更多其他有意思的内容。

第3章

替代面向对象模式

3

简介

面向对象模式是当代软件工程的主题之一。在本章，我们将来了解那些最常见的面向对象模式，以及这些模式所解决的问题。随后，还将介绍用于解决上述同类问题的更具函数式特色的解决方案。

针对每一种模式，我们首先着眼于Java是如何实现的。接着，会看看Scala是如何解决这些模式所解决的相同问题的。最后，会以一个Clojure版本的解决方案来结束讨论。

有的时候，Scala版本和Clojure版本的替代方案会非常相似。举个例子，在模式1“替代函数式接口”和模式7“替代策略模式”中，Scala和Clojure的解决方案在很大程度上是相同的。而有些时候，我们使用这两种语言所探寻的解决方案则会大相径庭，但是它们始终都体现了相同的函数式概念。

在模式4“替代生成器模式来获得不可变对象”中，Scala和Clojure的解决方案有着很大的区别。但是不管怎样，这两种情况都展示了使用不可变数据的简单方式。

通过对Scala和Clojure这两种语言之间相似度和差异度的探索，你应该会对其中每种语言是如何进行函数式编程的，以及它们与你所熟悉的传统的命令式风格有何区别有一定的了解。

让我们从第一个模式“替代函数式接口”开始我们的旅程！

模式 1

替代函数式接口

目的

将一些程序逻辑进行封装，以支持对这些程序逻辑的传递，以及将其存储于数据结构中，通

常还可以将这段封装后的程序逻辑作为任何其他头等的程序构造元素来处理。

概述

函数式接口是一种基本的面向对象设计模式。它由一个只有单个方法的接口组成，该方法的名称是例如run、execute、perform、apply或一些其他常见动词。函数式接口的实现会像任何其他方法都应该遵循的那样，只执行单一明确的行为。

函数式接口让我们将对象作为函数调用，这使得我们传递给程序的参数由名词变成了动词。这颠覆了传统面向对象的世界观。以严格的面向对象观点来看，名词性质的对象才是王者。而动词或方法只属于二等公民，通常都只能附属于对象，并注定终身受到名词统治者们的奴役。

别名

函数对象（Function Object）
Functoid
函子（Functor）

函数式替换方案

严格的面向对象的观点使得很多问题的解决方案变得较为笨拙。为了将一行有用的代码包装到`Runnable`或`Callable`这两个Java中最流行的函数式实例中，你不得不编写五六行范例代码。我都已经忘了到底干过多少次这样的事了。

为了让事情简单化，我们可以采用普通的函数来替代函数式接口。竟然可以采用更加原始的函数来替换对象？这一事实看上去似乎有些令人惊讶，但事实上，函数式编程中的函数远比C语言中的函数或Java中的方法来得更加强大。

在函数式语言中，函数都是高阶的：它们可以作为其他函数的结果返回，也可以作为其他函数的入参。它们都是头等的程序构造元素，这意味着除了高阶之外，我们还可以将它们分配给变量、放进数据结构或者进行一般性地操作。它们可以是未经命名的，即匿名函数，匿名函数是非常有用的一次性代码片段。事实上，函数式接口（正如它的名字所暗示的）这一面向对象世界里的模式的行为近似于函数式世界中的函数。

我们将在本节讨论用于实现对函数式接口替代的两种不同方式。第一种方式是以一个匿名函数来替代较小的只有几行代码的模式实例。这与在Java中使用匿名内部类来实现函数式接口十分相似，这种方式将会在“范例代码：匿名函数”中进行讨论。

第二种方式覆盖到涉及代码行数较多的场景。在Java中，我们会使用具名类而非匿名类来实现这些场景；而在函数式的世界里，我们也会使用一个具名函数来实现，具体见“范例代码：具

名函数”一节。

范例代码：匿名函数

第一个例子描绘了匿名函数以及如何使用它们来替代函数式接口中的那些小型实例。一种常见的情形是我们需要对一个集合进行与其自然顺序不同的排序，而排序的方式通常是可指定的。

为了实现这一点，需要创建一个自定义的比较器，这样一来排序算法才知道哪个元素应该排在第一位。在传统的Java中，我们需要创建一个以匿名类方式实现的`Comparator`。而在Scala和Clojure中，只需要使用一个匿名函数就能达到相同的目的。来看一个简单的例子，该例子将对对象进行非自然顺序排序：根据名字（即西方人的第一个名字）而非姓氏进行排序（即西方人的最后一个名字）。

传统的Java实现

在传统的Java中，我们使用一个名叫`Comparator`的函数式接口来帮助我们进行排序。因为它只有少量的代码，所以我们将它作为匿名函数来进行实现，并且会将它传递给排序函数，该解决方案的核心代码如下所示：

JavaExamples/src/main/java/com/mblinn/mbfpp/oo/fi/PersonFirstNameSort.java

```
Collections.sort(people, new Comparator<Person>() {
        public int compare(Person p1, Person p2) {
                return p1.getFirstName().compareTo(p2.getFirstName());
        }
});
```

这种方式虽然能够正常地运行。但大部分的代码却是与核心逻辑无关的额外语法，这些额外语法将一行真正的逻辑代码封装进一个匿名类。让我们来看看匿名函数是如何清理这些额外语法的。

Scala实现

让我们来看看如何使用Scala来处理通过名字而非姓氏进行排序这一问题。我们将使用一个样本类（case class）来表示人员，而且我们的实现方式将不再使用函数式接口`Comparator`，而是采用普通的老式函数来替代它。

使用Scala来创建匿名函数的语法如下所示：

```
(arg1: Type1, arg2: Type2) => FunctionBody
```

举个例子，以下的REPL会话创建了一个匿名函数，该匿名函数接受两个整数类型的入参，然后将它们相加。

```
scala> (int1: Int, int2: Int) => int1 + int2
res0: (Int, Int) => Int = <function2>
```

了解了基本的语法，下面来看看如何使用匿名函数来解决我们的人员排序问题。为了解决这一问题，我们使用了集合库中的sortWith()方法。该方法接受一个比较器作为入参，然后使用它来协助集合进行排序，这跟以Comparator为入参的Collections.sort()非常相似。

让我们从Person样本类的代码入手：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/fi/PersonExample.scala

```
case class Person(firstName: String, lastName: String)
```

下面是一个装满了人员测试数据的vector：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/fi/PersonExample.scala

```
val p1 = Person("Michael", "Bevilacqua")
val p2 = Person("Pedro", "Vasquez")
val p3 = Person("Robert", "Aarons")

val people = Vector(p3, p2, p1)
```

sortWith()方法期望它的比较函数能返回一个布尔值来帮助它判断第一个参数是否大于第二个参数。Scala的比较操作符<和>适用于字符串类型，所以可以使用这两个操作符来达到这一目的。

以下代码对这一方式进行了展示。我们可以省略函数参数的类型标注。Scala可以从sortWith()方法推断出这些类型：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/fi/PersonExample.scala

```
people.sortWith((p1, p2) => p1.firstName < p2.firstName)
```

在Scala的REPL中运行以上代码将会得到如下输出：

```
res1: scala.collection.immutable.Vector[...] =
        Vector(
                Person(Michael,Bevilacqua),
                Person(Pedro,Vasquez),
                Person(Robert,Aarons))
```

这比使用一个等效的函数式接口更加简短也更加简单。

Clojure实现

我们在Clojure中使用fn这一特殊形式来定义一个匿名函数，如以下的代码结构所示：

```
(fn [arg1 arg2] function-body)
```

先创建一些供测试使用的人员数据。在Clojure中，我们不会去定义一个类来承载数据，而是使用一个简单的map：

ClojureExamples/src/mbfpp/rso/person.clj

```
(def p1 {:first-name "Michael" :last-name "Bevilacqua"})
```

```
(def p2 {:first-name "Pedro" :last-name "Vasquez"})
(def p3 {:first-name "Robert" :last-name "Aarons"})

(def people [p3 p2 p1])
```

现在我们创建了一个匿名的排序函数，并将它与想要排序的人员数据一起传递给sort函数，如下面的代码所示：

```
=> (sort (fn [p1 p2] (compare (p1 :first-name) (p2 :first-name))) people)
({:last-name "Bevilacqua", :first-name "Michael"}
{:last-name "Vasquez", :first-name "Pedro"}
{:last-name "Aarons", :first-name "Robert"})
```

通过消除那些在Java中为了将排序函数封装进比较器而不得不编写的额外代码，我们只需编写那些关键的核心代码。

范例代码：具名函数

让我们对人员排序问题进行一下扩展。我们将会为每个人员增加一个中间名字，并对我们与众不同的排序算法进行修改。该排序算法首先会根据第一个名字进行排序，如果第一个名字相同则会按照姓氏排序，如果姓氏也相同，则会按照中间名字进行排序。

这样一来使得比较逻辑的代码变得过长，因此我们不应该继续将它以内嵌的方式保留在使用它的代码中。在Java中，我们会将代码从匿名内部类中提取出来，并迁移到一个具名类中。而在Clojure和Scala中，我们会将代码迁移至一个具名函数中。

传统的Java实现

当我们需要封装的逻辑较小时，匿名类和函数是非常不错的选择。但是一旦该逻辑变得更加庞大，这种嵌入的逻辑将会显得比较混乱。在传统的Java中，我们会转而使用一个具名类，它的代码结构框架如下所示：

```
public class ComplicatedNameComparator implements Comparator<Person> {
        public int compare(Person p1, Person p2) {
                «complicatedSortLogic»
        }
}
```

对于高阶函数而言，我们将使用一个具名函数。

Scala实现

首先对Scala样本类进行扩展，为其增加中间名字，同时定义一些测试数据：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/fi/PersonExpanded.scala

```
case class Person(firstName: String, middleName: String, lastName: String)
val p1 = Person("Aaron", "Jeffrey", "Smith")
val p2 = Person("Aaron", "Bailey", "Zanthar")
```

```
val p3 = Person("Brian", "Adams", "Smith")
val people = Vector(p1, p2, p3)
```

现在来创建一个具名的比较函数，并将它传递给sortWith()，如下面的代码所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/fi/PersonExpanded.scala

```
def complicatedSort(p1: Person, p2: Person) =
  if (p1.firstName != p2.firstName)
    p1.firstName < p2.firstName
  else if (p1.lastName != p2.lastName)
    p1.lastName < p2.lastName
  else
    p1.middleName < p2.middleName
```

瞧！可以使用一个任意命名的函数来对人员进行简单排序了。

```
scala> people.sortWith(complicatedSort)
res0: scala.collection.immutable.Vector[...] =
        Vector(
                Person(Aaron,Jeffrey,Smith),
                Person(Aaron,Bailey,Zanthar),
                Person(Brian,Adams,Smith))
```

Clojure实现

Clojure的解决方案与Scala版非常相似。我们需要一个能够根据更加复杂的规则集来进行人员比较的具名函数，同时需要为人们添加一个中间名字。

下面便是我们复杂的排序算法：

ClojureExamples/src/mbfpp/rso/person_expanded.clj

```
(defn complicated-sort [p1 p2]
  (let [first-name-compare (compare (p1 :first-name) (p2 :first-name))
        middle-name-compare (compare (p1 :middle-name) (p2 :middle-name))
        last-name-compare (compare (p1 :last-name) (p2 :last-name))]
    (cond
      (not (= 0 first-name-compare)) first-name-compare
      (not (= 0 last-name-compare)) last-name-compare
      :else middle-name-compare)))
```

现在可以像前面那样调用sort函数了，但是入参不再是一个匿名函数，我们传递的是一个具名函数complicated-sort：

ClojureExamples/src/mbfpp/rso/person_expanded.clj

```
(def p1 {:first-name "Aaron" :middle-name "Jeffrey" :last-name "Smith"})
(def p2 {:first-name "Aaron" :middle-name "Bailey" :last-name "Zanthar"})
(def p3 {:first-name "Brian" :middle-name "Adams" :last-name "Smith"})
(def people [p1 p2 p3])

=> (sort complicated-sort people)
({:middle-name "Jeffrey", :last-name "Smith", :first-name "Aaron"}
```

```
{:middle-name "Bailey", :last-name "Zanthar", :first-name "Aaron"}
{:middle-name "Adams", :last-name "Smith", :first-name "Brian"})
```

以上便是Clojure方案所有的内容了。

讨论

函数式接口有一点奇怪。它源于Java将每一个事物都转换成对象（这一类名词）的一贯作风。这就好比为了实现run这个动词，就必须使用ShoePutterOnner、DoorOpener和Runner这样的名词！采用高阶函数来替代这些模式可以在很多方面对我们有所帮助。首先，这样做可以降低很多通用任务的语法开销，即那些除了你真正想要编写的代码之外还需要额外编写的繁琐内容。

举个例子，我们在本节中所遇到的第一个`Comparator`就需要通过五行（格式正确的）Java代码来传达仅仅一行的实际计算：

```
new Comparator<Person>() {
        public int compare(Person left, Person right) {
                return left.getFirstName().compareTo(right.getFirstName());
        }
}
```

将它与单行的Clojure代码进行比较。

```
(fn [left right] (compare (left :first-name) (right :first-name)))
```

更重要的是，使用高阶函数为我们在传递小型计算方面带来了一致的方式。对于函数式接口，你需要为每一个希望解决的小问题都找到合适的接口，并且要理解如何去使用它。我们已经看过本章中的`Comparator`，并提到了该模式的其他一些常见用途。数以百计这样的接口存在于Java标准库和其他流行的代码库中，每个接口都像雪花一样独一无二，但是这些接口间烦人的区别已经掩盖了其应有的美感。

本章的函数式接口与它的替代方案之间存在着一些区别，但是这些区别并不触及它们所要解决的问题核心。由于函数式接口是由类来实现的，它定义了类型，并且可以使用像子类化和多态这样的面向对象特性。而高阶函数则不然。这实际上是高阶函数相对于函数式接口的一个优势：你不需要为每一种函数式接口都定义新的类型，因为现有的函数类型就可以满足你的需要。

延展阅读

《Effective Java中文版》第21条：用函数对象表示策略

JSR 335: Lambda Expressions for the Java Programming Language [Goe12][①]

① http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-4.html

相关模式

模式3　替代命令模式

模式6　替代模板方法模式

模式7　替代策略模式

模式16　函数生成器模式

模式2

替代承载状态的函数式接口

目的

将一些状态与程序逻辑封装到一起，以支持对这些程序逻辑的传递，以及将其存储于数据结构之中，通常还可以将这段封装后的程序逻辑作为任何其他头等的程序构造元素来处理。

概述

在模式1中，我们看到了如何采用高阶函数来替代函数式接口，但是我们所看到的实例并没有承载任何程序的状态。在本模式中，我们将会去看看闭包这一强大程序构造元素，以及它将如何替代需要状态的函数式接口实现。

别名

函数对象（Function Object）

Functoid

函子（Functor）

函数式替代方案

函数式世界中的函数是被称为“闭包”的这一强大的程序构造元素的一部分。一个闭包将函数与该函数创建时的可见状态进行了打包。这意味着一个函数在被调用时，可以引用它被创建时所在作用域中的任意变量。程序员无须为创建闭包做任何特殊的事情，编译器和运行时会替你完成闭包的创建。而闭包可以简单地自动捕获所有它所需的状态。

在传统的Java中，为了达到承载数据的目的，需要在类中创建字段，并为这些字段提供setters

方法，或通过构造器来设置它们。而在函数式的世界里，我们可以利用闭包来承载数据，无须任何额外装置。闭包的确有一些不可思议，所以在我们继续前进之前有必要对它们进行更加深入的讨论。

一个闭包由一个函数及该函数创建时的可用状态组成。让我们来看看它的样子，如图3-1所示。

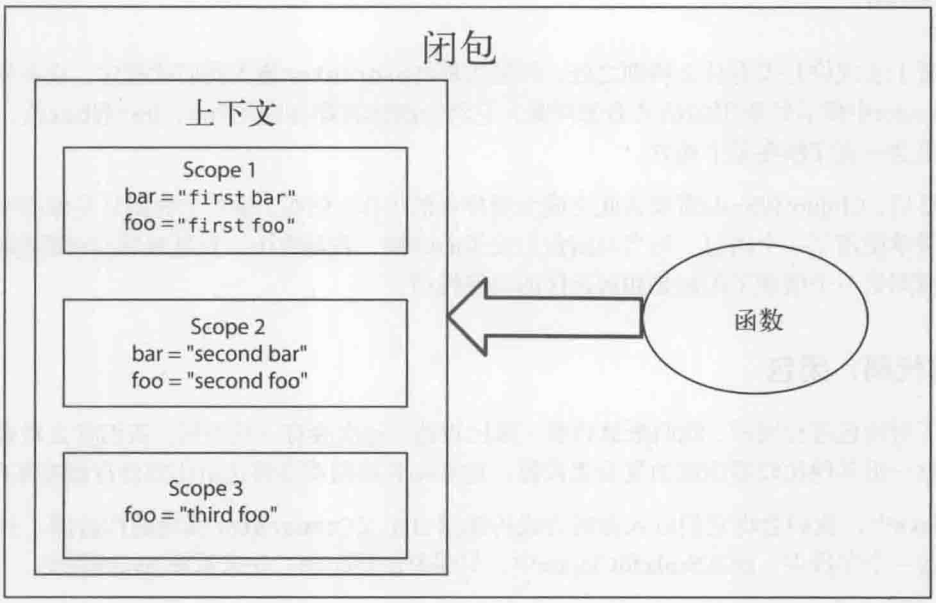


图3-1　闭包的结构。闭包是由一个函数和它被创建时的上下文所组成的一个数据结构

下面我们可以看到，闭包拥有一个函数和一个供其在执行工作时查找任意所需变量的作用域链。我们将它翻译成Clojure实现之后如下所示：

ClojureExamples/src/mbfpp/rso/closure_example.clj

```
(ns mbfpp.rso.closure-example)

; Scope 1 - Top Level
(def foo "first foo")
(def bar "first bar")
(def baz "first baz")

(defn make-printer [foo bar] ; Scope 2 - Function Arguments
  (fn []
    (let [foo "third foo"] ; Scope 3 - Let Statement
      (println foo)
      (println bar)
      (println baz))))
```

如果使用这段代码来生成打印函数并运行该函数，正如任何有经验的开发者所预料的那样，它将会打印最内层作用域所定义的foo、bar和baz：

```
=> (def a-printer (make-printer "second foo" "second bar"))
#'closure-example/a-printer
=> (a-printer)
third foo
second bar
first baz
nil
```

这看上去或许并没有什么特别之处，但是如果将a-printer传入我们的程序，或者将它存储在一个vector中供后续取用的话又会怎样呢？它将会继续打印相同的foo、bar和baz值，这意味着这些值会一直守候在某个地方。

在幕后，Clojure和Scala需要为此完成大量神奇的工作。然而，事实上我们只是像声明一个函数一样简单使用了一个闭包。每当与闭包打交道的时候，我脑海中一直显现着前面的那幅插图，因为这幅图是一个展现了闭包是如何运作的良好模型。

简单的代码：闭包

为了对闭包进行展示，我们来最后看一眼比较器，这次会有一些不同。我们将去看看如何创建一个由一组其他比较器组成的复合比较器，这意味着我们需要将这组比较器存储在某些地方。

在Java中，我们会将它们以入参的方式传递给自定义Comparator实现的构造器，然后将它们存储在一个字段中。而在Scala和Clojure中，只需要使用闭包。先来看看Java的例子。

传统的Java实现

在Java中，我们创建了一个名为ComposedComparator的Comparator自定义实现，该实现拥有一个使用了可变长参数的构造器，该参数接收一个比较器的数组，并将它们存储在一个字段中。

当ComposedComparator的compare()方法被调用时，它会逐个运行数组中的比较器，并返回第一个非零的结果。如果所有的结果都是零，它将会返回零。该解决方案的大致结构如下所示：

```
public class ComposedComparator<T> implements Comparator<T> {
        private Comparator<T>[] comparators;
        public ComposedComparator(Comparator<T>... comparators) {
                this.comparators = comparators;
        }
        @Override
        public int compare(T o1, T o2) {
          //遍历比较器并逐个调用每个比较器的比较方法

        }
}
```

在函数式的世界里，我们可以使用闭包来替代创建新类。让我们来看看Scala是如何做到这一点的。

Scala实现

在Scala中，可以利用闭包来避免显式地对复合比较器中的比较器列表进行跟踪。我们的Scala解决方案围绕着一个核心的高阶函数`makeComposedComparison()`。该函数使用可变长参数作为入参来接收一个比较函数的数组，并返回一个能逐个执行这些函数的函数。

Java和Scala解决方案的另一个差别是如何返回最终的结果。在Java中，我们对比较器列表逐个进行迭代，一旦发现有非零的比较结果，就会将它作为结果返回。

而在Scala中，我们使用`map()`来对输入运行比较器。然后查找第一个非零的结果。如果没有找到一个非零的值，即所有比较都是一样大的，那么我们将会返回零。

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/fi/personexpanded/ClosureExample.scala

```
def makeComposedComparison(comparisons: (Person, Person) => Int*) =
  (p1: Person, p2: Person) =>
    comparisons.map(cmp => cmp(p1, p2)).find(_ != 0).getOrElse(0)
```

现在我们将选取两个比较函数，并将它们组合到一起。在下面的代码中，我们定义了`firstNameComparison()`和`lastNameComparison()`，然后将它们组装进了`firstAndLastNameComparison()`：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/fi/personexpanded/ClosureExample.scala

```
def firstNameComparison(p1: Person, p2: Person) =
  p1.firstName.compareTo(p2.firstName)

def lastNameComparison(p1: Person, p2: Person) =
  p1.lastName.compareTo(p2.lastName)

val firstAndLastNameComparison = makeComposedComparison(
    firstNameComparison, lastNameComparison
)
```

让我们定义一组人员，并对它们进行比较来看看我们的复合比较函数：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/fi/personexpanded/ClosureExample.scala

```
val p1 = Person("John", "", "Adams")
val p2 = Person("John", "Quincy", "Adams")

scala> firstAndLastNameComparison(p1, p2)
res0: Int = 0
```

有一个地方可以进行优化，即创建一个短路版本的复合比较器，它可以在发现首个非零结果的时候停止运行比较器。为了做到这一点，我们可以使用模式12 “尾递归模式”中的递归函数，而不再使用此处为了方便大家理解所用的这种方式。

Clojure实现

我们将查看如何在Clojure中创建复合比较器，从而结束本模式的代码示例。我们将会依靠一个名为make-composed的比较器，但是它的工作方式与Scala版本存在着一些区别。

在Scala中，可以使用find()方法来查找首个非零的结果；而在Clojure中，我们可以使用some函数。这与Scala的Some类型有着非常大的区别！

在Clojure中，some函数接受一个判定条件和一个序列，它将返回序列中首个使得判定条件为真的值。在下面的代码中，我们使用了Clojure的some和for来运行所有的比较器，并选择出正确的最终值：

ClojureExamples/src/mbfpp/rso/closure_comparison.clj

```
(defn make-composed-comparison [& comparisons]
  (fn [p1 p2]
    (let [results (for [comparison comparisons] (comparison p1 p2))
          first-non-zero-result
          (some (fn [result] (if (not (= 0 result)) result nil)) results)]
      (if (nil? first-non-zero-result)
        0
        first-non-zero-result))))
```

现在可以创建first-name-comparison和last-name-comparison函数，并将它们组装到一起：

ClojureExamples/src/mbfpp/rso/closure_comparison.clj

```
(defn first-name-comparison [p1, p2]
  (compare (:first-name p1) (:first-name p2)))

(defn last-name-comparison [p1 p2]
  (compare (:last-name p1) (:last-name p2)))

(def first-and-last-name-comparison
  (make-composed-comparison
    first-name-comparison last-name-comparison))
```

我们将使用它们对两个人员进行比较：

ClojureExamples/src/mbfpp/rso/closure_comparison.clj

```
(def p1 {:first-name "John" :middle-name "" :last-name "Adams"})
(def p2 {:first-name "John" :middle-name "Quincy" :last-name "Adams"})

=> (first-and-last-name-comparison p1 p2)
0
```

以上便是我们对使用闭包来替代承载状态的函数式接口实现的讨论。在继续前行之前，让我们详细地讨论一下闭包和类之间的关系。

讨论

有一个关于闭包和类的笑话：类是穷人的闭包，而闭包则是穷人的类。这个笑话除了证明了函数式程序员可能不适合进入单口相声这一行业之外，也阐明了类和闭包之间关系的一些有意思的事情。

在某些方面，闭包和类非常相似。它们都可以承载状态和行为。而在别的方面，它们却又大相径庭。围绕着类有着一整串面向对象的机制，它们定义类型，也可以是类型层次体系中的一部分，等等。而闭包则更加简单——它们仅仅是由一个函数和该函数创建时所在的上下文组成的。

拥有闭包使得解决许多常见的编程任务变得更加简单，正如我们在本节中所看到的，没有闭包的语言（比如Java）的使用者只能通过使用新建类来完成闭包的工作，所以说类是穷人的闭包。然而，类拥有很多闭包所没有的编程特性，所以说闭包是穷人的类。Scala同时给予了我们类和闭包来解决这一问题，而Clojure通过解构例如多态和类型层次体系这些来自类中的好东西，从而为程序员解决该问题另辟蹊径。

拥有闭包和高阶函数可以简化常见的模式（命令模式、模板方法模式和策略模式等），以至于这些模式几乎销声匿迹。闭包和高阶函数非常有用，以至于凭借着JSR 335（即Lambda表达式）的旗号，闭包和高阶函数成为了即将到来的Java 8中的主要功能特性之一。

对于一门需要绝对向后兼容的成熟语言来说，这是一个重大的变化，所以这不是一项能轻易完成的任务。对于Java的设计者们来说，这一变化举足轻重。但是高阶函数是如此的成功，以至于设计者们认为对该功能的包含势在必行。对于这一改变的设计和实现花费了数年的努力，但是它们终于要来了。

延展阅读

《Effective Java中文版》第21条：用函数对象表示策略

JSR 335: Lambda Expressions for the Java Programming Language [Goe12][①]

相关模式

模式3　替代命令模式

模式6　替代模板方法模式

模式7　替代策略模式

模式16　函数生成器模式

① http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-4.html

模式 3

替代命令模式

目的

将方法调用转变成一个对象，并在集中的地方运行该对象，以保持对调用的跟踪，从而方便我们对调用进行撤销、记录以及重做。

概述

命令模式对某种行为及其执行时所需的信息进行了封装。虽然它看上去很简单，但是该模式具有不少移动的部分。除了Command接口及其实现之外，还包括客户端、调用者（invoker）和接收者（receiver）。客户端负责创建命令，调用者负责运行命令，而接收者则是命令中行为的执行者。

调用者经常被误解，所以值得我们进行讨论。它在发起调用的客户端和方法调用之间起到了解耦的作用，并为各种调用提供了集中的地方。再加上调用是由一个对象表示的，这就方便我们完成一些像记录方法调用这样的事情，以便于对这些执行进行撤销或将它们序列化到磁盘上。

图3-2展示了命令模式是如何组织的。

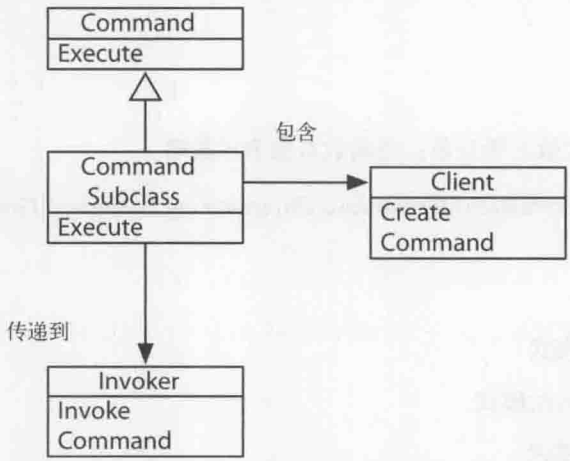


图3-2　命令模式概览，描述了命令模式的概况

以日志记录为例。在这里，客户端是需要进行日志记录的任意类，而接收者是一个Logger实例。调用者是客户端调用的一个类，它替代了Logger实例供客户端调用，这样一来客户端就

不再直接调用Logger了。

别名

行为（Action）

函数式替代方案

命令模式具有一些移动的部分，在我们的函数式替代方案中也是如此。Command类本身是一个通常承载了状态的函数式接口，所以我们将会用模式2中介绍过的闭包来替代该Command类。

接着我们将用一个简单的函数来替代调用者执行命令，我将称它为执行函数。跟调用者差不多，执行函数给了我们一个集中的地方来控制命令的执行，让我们根据实际需要对它们进行存储或其他操作。

最后，将会创建一个函数生成器，它负责创建我们的命令，让创建过程变得简单且一致。

范例代码：现金出纳机

让我们看看如何采用命令模式来实现一个简单的现金出纳机。我们的现金出纳机功能非常简单：它只能处理整额的美元，同时包含了一定总量的现金，并且只允许将现金增加到出纳机。

我们将会保持一份事务日志，以方便对操作进行重放。在开始采用以Scala和Clojure为代表的函数式替代方案之前，先来看看如何采用传统的命令模式来实现这一功能。

传统的Java实现

Java实现从定义一个标准的Command接口开始。这是模式1中的一个例子。我们通过Purchase类来实现该接口。

该模式中的接收者是CashRegister类。而Purchase类中将包含一个指向CashRegister类的引用，而CashRegister类正是Purchase类执行的对象。为了保持模式的完整性，我们还需要一个调用者，即PurchaseInvoker类来实际地执行我们的购买行为。

该模式的类图如下所示，而完整的源代码可以从本书的代码示例中找到。

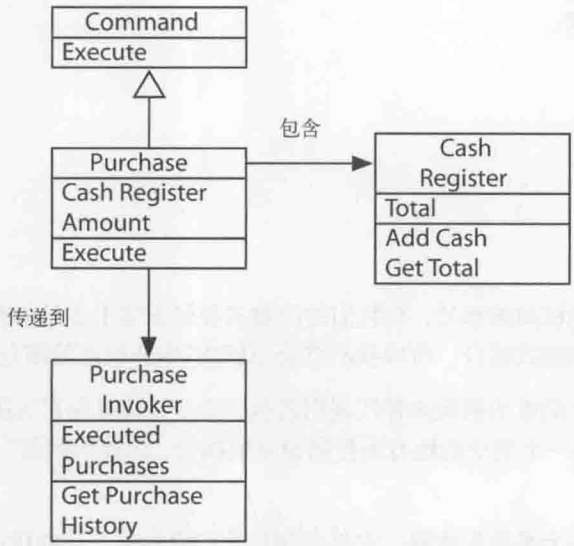


图3-3　现金出纳机命令模式。作为Java版命令模式实例的现金出纳机结构展示

现在我们已经勾勒出命令模式的一个Java实现，下面来看看如何使用函数式编程来简化这一模式。

Scala实现

使用Scala来替代命令模式最彻底的方式便是利用Scala的混合特性。我们将会保留Java版中CashRegister类；不过，我们将采用高阶函数来替代创建Command接口及其实现。同时不再创建独立的类来扮演调用者的角色，而是创建一个执行函数。让我们从CashRegister类本身开始来看看代码：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/command/register/Register.scala

```
class CashRegister(var total: Int) {
  def addCash(toAdd: Int) {
    total += toAdd
  }
}
```

接下来将创建函数makePurchase来生成购买函数。makePurchase函数接受amount和register作为入参，返回最终会完成实际工作的函数，代码如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/command/register/Register.scala

```
def makePurchase(register: CashRegister, amount: Int) = {
  () => {
    println("Purchase in amount: " + amount)
    register.addCash(amount)
  }
}
```

最后，让我们来看看执行函数：executePurchase。它在执行传入的purchase函数前将该函数添加到了一个Vector来保持对购买行为的跟踪。代码如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/command/register/Register.scala

```
var purchases: Vector[() => Unit] = Vector()
def executePurchase(purchase: () => Unit) = {
  purchases = purchases :+ purchase
  purchase()
}
```

为什么这里使用的是var?

上述代码中的重要位置出现了一个可变的引用。

```
var purchases: Vector[() => Unit] = Vector()
```

在一本关于函数式编程的书中看到这样的代码会让人觉得有些奇怪。难道不应该所有的一切都是不可变的吗？但事实证明，虽然不是不可能，但对每一件事物都以纯函数式的方式来建模是非常困难的。跟踪状态变化这一场景就变得尤为棘手。

不过别太担心，此处所做的一切仅仅是去移动一个指向了一些不可变数据的引用。这给我们带来了不可变性的诸多好处。举个例子，我们可以安全地创建足够多的指向原始Vector的引用，而无需担心对原始数据的意外修改，如下面的代码所示：

```
scala> var v1 = Vector("foo", "bar")
v1: scala.collection.immutable.Vector[String] = Vector(foo, bar)

scala> val v1Copy = v1
v1Copy: scala.collection.immutable.Vector[String] = Vector(foo, bar)

scala> v1 = v1 :+ "baz"
v1: scala.collection.immutable.Vector[String] = Vector(foo, bar, baz)

scala> v1Copy
res0: scala.collection.immutable.Vector[String] = Vector(foo, bar)
```

也可以通过使用优秀的Scalaz库[1]来完成纯函数式方式的编程，但是本书更专注于一些更加实用的函数式编程方式。

1. https://code.google.com/p/scalaz/

下面是实际的解决方案：

```
scala> val register = new CashRegister(0)
register: CashRegister = CashRegister@53f7eb48

scala> val purchaseOne = makePurchase(register, 100)
```

```
purchaseOne: () => Unit = <function0>

scala> val purchaseTwo = makePurchase(register, 50)
purchaseTwo: () => Unit = <function0>

scala> executePurchase(purchaseOne)
Purchase in amount: 100

scala> executePurchase(purchaseTwo)
Purchase in amount: 50
```

如你所见，出纳机现在得到了正确的总金额：

```
scala> register.total
res2: Int = 150
```

如果将出纳机重置为0，我们可以通过存储在名为purchases的vector中的函数来对购买行为进行重放：

```
scala> register.total = 0
register.total: Int = 0

scala> for(purchase <- purchases){ purchase.apply() }
Purchase in amount: 100
Purchase in amount: 50

scala> register.total
res4: Int = 150
```

相比于Java版本，Scala版本显得非常直截了当。一旦拥有了高阶函数，你将不再需要Command类、Purchase类以及独立的调用类。

Clojure实现

Clojure版解决方案的总体结构与Scala版非常相似。我们将使用高阶函数来承担命令的角色，并采用一个执行函数去执行这些命令。Scala版与Clojure版解决方案最大的区别在于现金出纳机本身。因为Clojure并不具备面向对象特性，我们无法创建一个CashRegister类。

相反，我们简单地采用了一个Clojure的atom类型来对出纳机中的现金保持跟踪。为了做到这一点，我们创建了一个make-cash-register函数和一个add-cash函数，前者将会返回一个代表新出纳机的新atom类型，而后者以register和金额作为入参。我们同样还会创建一个reset函数来将register重置为0。

以下是Clojure版现金出纳机的代码：

ClojureExamples/src/mbfpp/oo/command/cash_register.clj

```
(defn make-cash-register []
  (let [register (atom 0)]
    (set-validator! register (fn [new-total] (>= new-total 0)))
    register))
```

```
(defn add-cash [register to-add]
  (swap! register + to-add))

(defn reset [register]
  (swap! register (fn [oldval] 0)))
```

可以创建一台空的出纳机：

```
=> (def register (make-cash-register))
#'mblinn.oo.command.ex1.version-one/register
```

然后向它添加一些现金：

```
=> (add-cash register 100)
100
```

现在我们拥有了一台现金出纳机，再来看看如何创建命令。在Java中我们必须为此实现一个Command接口。而在Clojure中，我们只需要使用一个函数来表示购买。

为了创建它们，我们将使用make-purchase函数，该函数以register和amount作为入参，并返回一个可以向出纳机添加金额的函数，代码如下所示：

ClojureExamples/src/mbfpp/oo/command/cash_register.clj

```
(defn make-purchase [register amount]
  (fn []
    (println (str "Purchase in amount: " amount))
    (add-cash register amount)))
```

我们用它来创建两个购买命令：

```
=> (def register (make-cash-register))
#'mblinn.oo.command.ex1.version-one/register
=> @register
0
=> (def purchase-1 (make-purchase register 100))
#'mblinn.oo.command.ex1.version-one/purchase-1
=> (def purchase-2 (make-purchase register 50))
#'mblinn.oo.command.ex1.version-one/purchase-2
```

然后运行它们：

```
=> (purchase-1)
Purchase in amount: 100
100
=> (purchase-2)
Purchase in amount: 50
150
```

在结束这个例子之前，还需要一个执行函数execute-purchase。该函数在执行购买之前存储了这些购买命令。我们将会使用一个名为purchases的atom类型，它包装了一个用于存储这些命令的vector。代码如下所示：

ClojureExamples/src/mbfpp/oo/command/cash_register.clj

```
(def purchases (atom []))
(defn execute-purchase [purchase]
  (swap! purchases conj purchase)
  (purchase))
```

现在可以使用execute-purchase来执行我们刚才定义的购买命令了。因此，这次我们将从购买历史中取出这些购买命令。首先会重置出纳机：

```
=> (execute-purchase purchase-1)
Purchase in amount: 100
100
=> (execute-purchase purchase-2)
Purchase in amount: 50
150
```

即使现在再次重置出纳机，我们仍可以通过运行购买历史来重新运行购买命令：

```
=> (reset register)
0
=> (doseq [purchase @purchases] (purchase))
Purchase in amount: 100
Purchase in amount: 50
nil
=> @register
150
```

以上便是我们的Clojure解决方案！在整个过程中，有意思的是我们没有使用对象来对现金出纳机进行建模，只是简单地使用了一些可以在其上进行操作的数据和函数。当然，这在函数式世界里是极其常见的，这一特性可以帮助我们创造更加简单的代码和更加小型的系统。

乍一看，这似乎限制了那些经验丰富的面向对象程序员；假设你需要多态或类型层次的话怎么办？不要害怕，Clojure为程序员提供了所有来自面向对象世界的好东西，只不过是采用了一种截然不同且更加解耦的方式。举个例子，Clojure拥有一套自己的方式来创建特别层级，而它的多重方法和协议也给我们带来了多态的特性。我们将会在模式10“替代访问者模式”中对其中一些工具进行更加详细的讨论。

讨论

虽然到处都在使用命令模式，但我认为它算的上是《设计模式》一书中最常被人们误解的几个模式之一。人们通常将Command接口和命令模式混为一谈。但事实上Command接口只是整个命令模式中一个很小的部分，而它本身也是模式1中的一个例子。这并不是说我们通常使用命令模式的方式是错的，但是通常我们所使用的方式与“四人组”（Gang of Four）所描述的命令模式有所不同，这为我们讨论该模式带来了一些困惑。

本章节中的例子为命令模式中的调用者、接收者和客户端的替代方式都提供了实现，但是我

们可以很容易地从中剔除不再需要的部分。举个例子，如果我们的命令不再需要与出纳机协同工作，我们将无须再向makePurchase传递出纳机。

延展阅读

《设计模式：可复用面向对象软件的基础》中有关命令的内容

相关模式

模式1　替代函数式接口

模式4

替代生成器模式来获得不可变对象

目的

创建不可变对象时，我们通常会采用一种友好的语法来为对象设置属性——因为我们无法在创建完成之后再对其进行修改。同时，我们也需要一种简单的方式基于现有对象来创建新的对象，并为新对象的一些属性设置新值。

概述

在本节内容中，我们将讨论连贯生成器（Fluent Builder）模式，它可以用于生成不可变对象。这是一个通用的模式，Java标准库在StringBuilder和StringBuffer中使用了这一模式。当然，还有很多其他的通用代码库也都使用了这种模式，比如Google的Protocol Buffers。

使用不可变对象是一种经常在Java中被忽略的良好实践，而在Java中，最常见的用于承载数据的方式就是采用一个拥有一堆getter和setter方法的类。这就迫使数据对象必须是可变的，而可变性是很多常见问题的罪魁祸首。

在Java中，获得不可变对象的最简单的方式就是创建一个将所有所需数据作为构造器入参的类。不过，《Effective Java中文版》的第2条指出，当涉及处理大量属性的时候，这种方式将会导致两个问题。

首先，拥有太多构造器参数的Java类非常难以使用。一个程序员需要记住每个参数所在的位置，而不能只通过它们的名字来引用它们。其次，由于我们需要将所有属性的值传递给构造器，导致我们无法简单地为属性创建默认值。

解决此类问题的一个方法是创建多个只接受部分属性值的构造器，而针对那些没有传入值的属性则赋以默认值。但是一旦碰到很大的对象，这将导致重叠构造器问题。在这种情况下，一个类需要实现很多不同的构造器，并需要将来自较小的构造器中的值再传递给更大的构造器。

本节所讨论的生成器模式，在《Effective Java中文版》中也有相关描述，它可以用于解决以上这些问题，但是需要付出一定的代价，即编写相当多的代码。

函数式替代方案

在Scala和Clojure中用以替代这两种模式的技术完全不同，但是它们都享用了同一个非常重要的性质，即它们都可以非常简单地进行不可变对象的创建。先来看看Scala。

Scala实现

在Scala中，我们在创建不可变数据结构时将会涉及三种不同的技术，每一种技术都有其优势和不足。

首先我们将讨论完全由不可变值组成的Scala类。我们将会展示如何使用已命名参数和默认值来达到这一目的，这种方式跟在Java中采用连贯生成器生成不可变对象非常相似，但是会有一小部分额外开销。

接下来看看Scala的样本类。样本类天生就是用于承载数据的，所以它们拥有一些已经实现的用于处理数据的便利方法，比如`equals()`和`hashCode()`。而样本类还可以跟Scala的模式匹配一起使用，从而简单地对多个样本类进行区分处理。这使得样本类成为很多数据承载场景中理想的默认选择。

在这两个实例中，我们都将使用Scala的构造器来创建对象。Scala的构造器没有刚才讨论的Java构造器的那些缺点，因为我们可以指定参数名并为它们提供默认值。这帮助我们避免了重叠构造器问题以及在涉及传递多个未命名参数时需要尝试记住每一个参数位置的问题。

最后，讨论Scala的元组（tuple），这是一种用来传递小型复合数据结构而无需为此创建新的类的便利方式。

Clojure实现

Clojure也支持新类的创建，但这一特性通常只限于在与Java的互操作中使用。因此，在Clojure中通常使用普通的不可变map来构造聚合数据。

以Java世界的观点来看，这或许并不是一个好主意。但是因为Clojure拥有对与map协同工作的绝佳支持，所以这样做变得非常方便。使用map来构造数据能让我们完整地发挥Clojure序列库在操纵数据时的能力，这种能力是非常强大的。

很多代码库依赖数据对象审查对它们的数据执行操作，例如XStream和Hibernate，前者将数

据对象序列化成XML，而后者则可以生成SQL查询。如果要在Java中完成这一类的工作，需要用到与反射相关的代码库。而在Clojure中，你将只需要使用简单的map操作。

在Clojure中构造数据的另一种方式是使用记录。记录暴露了一个类似map的接口，因此你仍然可以在此之上发挥Clojure序列库的全部威力，而且记录有着一些超越map的优势。

首先，记录通常拥有更好的性能。另外，记录定义了一个可以参与Clojure多态的类型。举一个古老的面向对象中的例子，它允许我们定义一个make-noise函数，当我们传入一个狗的实例时，它会发出“汪汪”的叫声，而当传入的是一个猫的实例时，它将发出“喵喵”的叫声。除此之外，记录还允许我们对可以放入某个数据结构的属性进行约束。

通常来说，使用Clojure完成工作的最佳实践是先采用map对数据进行建模，而当你需要更快的速度或需要使用多态，抑或只是想要对处理的属性名称进行限制时，可以转而使用记录。

范例代码：不可变数据

在这一节中，我们先来看看在Java中如何使用生成器模式来获取不可变对象，从而完成数据表示。然后了解Scala中用以替换生成器模式的三种方式：拥有不可变属性的普通类、样本类以及元组。最后，来学习Clojure的两种替代方案：普通的map和记录。

传统的Java实现

在传统的Java中，我们可以使用良好的语法及连贯生成器来创建不可变对象。为了解决问题，我们创建了一个只有属性getter方法的ImmutablePerson类。而在该类的内部，我们创建了一个生成器Builder类，通过它可以构造一个ImmutablePerson实例。

当我们想要创建一个ImmutablePerson实例时，我们不会直接去构建；我们会创建一个新的Builder，并对想要设置的属性进行设置，然后调用build()方法来得到一个ImmutablePerson实例。代码框架如下所示：

```
public class ImmutablePerson {

        private final String firstName;
        // 省略更多其他属性

        public String getFirstName() {
                return firstName;
        }
        // 省略更多其他属性的getter方法

        private ImmutablePerson(Builder builder) {
                firstName = builder.firstName;
                // 省略更多对其他属性的设置
        }

        public static class Builder {
                private String firstName;
```

```
            // 省略更多其他属性

            public Builder firstName(String firstName) {
                this.firstName = firstName;
                return this;
            }
            // 省略更多其他属性的setter方法
            public ImmutablePerson build() {
                return new ImmutablePerson(this);
            }
        }
        public static Builder newBuilder() {
            return new Builder();
        }
}
```

这种做法的不足之处在于为了完成这样一个基本的任务我们需要编写大量的代码。传递聚合数据是我们作为程序员需要完成的最基本的工作之一，因此，编程语言应该给予我们一种更好的方式来完成。值得庆幸的是，Scala和Clojure都做到了这一点。让我们从Scala开始一探究竟。

Scala实现

我们将来了解Scala中用于表示不可变数据的三种不同的方式：不可变类、样本类和元组。不可变类是仅包含不可变属性的普通类；样本类则是一种用于和Scala模式匹配一起工作的特殊的类；而元组是一种不可变的数据结构，它允许我们在无需定义新类的情况下对数据进行分组。

- 不可变类

先来看看如何以Scala的方式来产生不可变对象。我们要做的就是定义一个类，并将一些val类型的变量作为其构造器的入参。这将导致传入的值会最终分配给那些公共的val变量。以下是该解决方案的代码：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/javabean/Person.scala

```
class Person(
    val firstName: String,
    val middleName: String,
    val lastName: String)
```

现在我们可以根据参数的位置来设置这些构造器参数，从而创建一个Person实例：

```
scala> val p1 = new Person("John", "Quincy", "Adams")
p1: Person = Person@83d2eb1
```

也可以按照已命名参数的方式来使用这些构造器参数：

```
scala> val p2 = new Person(firstName="John", middleName="Quincy", lastName="Adams")
p2: Person = Person@33d6798
```

我们可以为参数添加默认值，以便在使用已命名参数的形式时可以省略这些拥有默认值的构造器参数。在下面的代码中，我们将中间名默认为空：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/javabean/Person.scala

```
class PersonWithDefault(
    val firstName: String,
    val middleName: String = "",
    val lastName: String)
```

这样一来，就可以方便地处理那些没有中间名的人了：

```
scala> val p3 = new PersonWithDefault(firstName="John", lastName="Adams")
p3: PersonWithDefault = PersonWithDefault@6d0984e0
```

这种做法为我们在Scala中创建不可变对象提供了一种简单的方式，但是这种方式存在着一些不足。如果我们想要了解对象的相等性、hash值以及在打印时尽可能地保持美观，就需要自己去实现这些功能。样本类赋予了我们所有这些开箱即用的功能，同时也可以与Scala的模式匹配协同工作。然而样本类无法进行很好地扩展，所以它们并不能满足所有的使用场景。

- 样本类

我们使用case class来定义样本类，如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/javabean/Person.scala

```
case class PersonCaseClass(
    firstName: String,
    middleName: String = "",
    lastName: String)
```

现在我们可以像创建普通类一样来创建一个PersonCaseClass，唯一不同的是我们无需使用new操作符了。在下面的代码中，我们创建了一个使用了已命名参数的类，并省略了其中的中间名字：

```
scala> val p = PersonCaseClass(firstName="John", lastName="Adams")
p: PersonCaseClass = PersonCaseClass(John,,Adams)
```

请注意样本类是如何以PersonCaseClass(John,,Adams)那样的方式进行打印的，我们并不需要去实现toString()方法。我们同样从样本类免费得到了equals()和hashCode()方法。下面来测试一下它们的相等性：

```
scala> val p2 = PersonCaseClass(firstName="John", lastName="Adams")
p2: PersonCaseClass = PersonCaseClass(John,,Adams)

scala> p.equals(p2)
res1: Boolean = true

scala> val p3 = PersonCaseClass(
        firstName="John",
        middleName="Quincy",
        lastName="Adams")
p3: PersonCaseClass = PersonCaseClass(John,Quincy,Adams)

scala> p2.equals(p3)
res2: Boolean = false
```

样本类是不可变的，因此我们无法对它们进行修改，但是可以通过使用copy()方法基于现有的样本类实例创建一个新的样本类实例，从而达到相同的效果，正如我们在如下REPL会话中所做的：

```
scala> val p2 = p.copy(middleName="Quincy")
p2: com.mblinn.mbfpp.oo.javabean.PersonCaseClass =
       PersonCaseClass(John,Quincy,Adams)
```

最后，样本类可以与Scala的模式匹配一起使用。在下面的代码中，我们使用了模式匹配来挑选出第6任美国总统：

```
scala> p3 match {
     | case PersonCaseClass(firstName, middleName, lastName) => {
     |     "First: %s - Middle: %s - Last: %s".format(
                              firstName, middleName, lastName)
     | }}
res0: String = First: John - Middle: Quincy - Last: Adams
```

还有最后一种用于在Scala中表示数据的通用方式：元组。元组可以用于表示一条固定规模的记录，但是它们并不像类和样本类那样需要创建新的类型。它们可以很方便地在探索性开发中使用；你可以在对数据结构尚不明朗的早期阶段使用它们来构建数据，并在后续过程中逐步切换到对类和样本类的使用。让我们来看看这种方式是如何工作的。

- *元组*

可以通过将值内嵌在一对圆括号中来创建元组，就像下面这样：

```
scala> def p = ("John", "Adams")
p: (java.lang.String, java.lang.String)
```

并可以通过对它们位置的引用来获取相应的值，如下所示：

```
scala> p._1
  res0: java.lang.String = John

scala> p._2
res1: java.lang.String = Adams
```

最后，元组也可以简单地在模式匹配中使用，就像样本类那样：

```
scala> p match {
     | case (firstName, lastName) => {
     |     println("First name is: " + firstName)
     |     println("Last name is: " + lastName)
     | }}
First name is: John
Last name is: Adams
```

以上内容覆盖了在Scala中与不可变数据进行协同工作的三种主要方式。

当你需要处理超过样本类可以处理的二十二个属性的不可变数据时，采用普通的不可变类将会是不错的选择。但是如果你的数据对象需要一些现成的方法的话，你就有必要去精简你的数据

模型了。

当你需要一些对象内建的equals()、hashCode()和toString方法或需要与模式匹配一起工作时，样本类会非常有用。最后，元组对于探索性开发来说很有帮助，你可以在切换到使用类和样本类之前先采用元组来对数据进行简单地建模。

Clojure实现

下面来学习Clojure中用于表示不可变数据的两种方式。第一种是简单地将数据存储于map之中，第二种则是使用记录。map是一种我们大家都熟知且喜爱的简单数据结构，记录则有一些不同。记录允许我们定义数据类型，并对它们所包含的属性进行约束，但是记录同样给我们提供了类似map的接口。

- map

先来看看这两种方式中较为简单的一种：使用一个不可变的map。我们要做的就是创建一个以关键词作为键，以数据作为值的map，如下所示：

ClojureExamples/src/mbfpp/oo/javabean/person.clj

```
(def p
  {:first-name "John"
   :middle-name "Quincy"
   :last-name "Adams"})
```

我们可以像操作任意的map那样从该map中获取到属性：

```
=> (p :first-name)
"John"
=> (get p :first-name)
"John"
```

一个不太明显的优势就是我们可以充分利用map所支持的全套操作，包括将map作为序列处理。举个例子，如果我们想要将名字的所有部分都转型为大写字母，可以按如下代码进行操作：

```
=> (into {} (for [[k, v] p] [k (.toUpperCase v)]))
{:middle-name "QUINCY", :last-name "ADAMS", :first-name "JOHN"}
```

如果使用对象和getter方法来完成相同的工作，就需要调用所有不同类型相对应的getter方法。这意味着我们最终采用的解决方案所能处理的是一个通用的问题——将全部由字符串组成的数据结构中的全部属性转型为大写字母，而且这个问题的解决方案本身就降低了它的通用性，即它仅能将一种特定类型的属性转型为大写，这就意味着我们需要为每一种对象的类型都重新实现这一解决方案。

将使用不可变map作为承载及传递数据的主要方式之一还有一些别的好处。创建它们所采用的语法非常简单，因此当你向它们添加属性时不会有什么约束。这使得它们非常有益于探索性编程。

这种灵活性也具有一些缺点，Clojure的map并不像Java类那样简单高效。一旦你采用了像Java类那样更加充实的数据模型，它将会帮助你对所要处理的属性进行约束。

最重要的是，不管怎样，使用普通的map使得我们无法将map与多态机制一起使用，因为在使用map的过程中并没有定义新的类型。下面来看看Clojure的另一个特性，它不仅可以解决以上这些问题，而且仍然提供了类似map的接口。

- 记录

为了将记录描述清楚，我们借用一个古老的面向对象的例子：来创建猫和狗两种类型。为了创建我们的猫和狗的类型，我们使用了如下代码：

ClojureExamples/src/mbfpp/oo/javabean/catsanddogslivingtogether.clj

```
(defrecord Cat [color name])

(defrecord Dog [color name])
```

我们可以将它们作为map来对待，这样便获得了上文中所提到的所有功能：

```
=> (def cat (Cat. "Calico" "Fuzzy McBootings"))
#'mbfpp.oo.javabean.catsanddogslivingtogether/cat
=> (def dog (Dog. "Brown" "Brown Dog"))
#'mbfpp.oo.javabean.catsanddogslivingtogether/dog
=> (:name cat)
"Fuzzy McBootings"
=> (:name dog)
"Brown Dog"
```

不合适，而非不可能

此前我曾说过，当你想要获得基于类型的多态机制时，使用map将显得不合时宜。这没错，但是Clojure非常灵活，以至于这仅仅是不太合适，而非不可能。我们可以对map中的类型进行编码，然后使用Clojure的多重方法，代码如下所示：

ClojureExamples/src/mbfpp/oo/javabean/sidebar.clj

```
(def cat {:type :cat
          :color "Calico"
          :name "Fuzzy McBootings"})

(def dog {:type :dog
          :color "Brown"
          :name "Brown Dog"})

(defmulti make-noise (fn [animal] (:type animal)))
(defmethod make-noise :cat [cat] (println (str (:name cat)) "meows!"))
(defmethod make-noise :dog [dog] (println (str (:name dog)) "barks!"))
```

通常情况下，如果你想要获得类型上的多态，最好的选择就是使用Clojure的协议。而当你需要足够强大的能力来定义你自己的分派函数时，多重方法将给你带来更加专业的多态机制。

它们也可以很容易地通过使用Clojure的协议来使用多态。在下面的代码中我们定义了一个只有单个make-noise函数的协议，并创建了NoisyCat和NoisyDog来启用该协议：

ClojureExamples/src/mbfpp/oo/javabean/catsanddogslivingtogether.clj

```
(defprotocol NoiseMaker
  (make-noise [this]))

(defrecord NoisyCat [color name]
  NoiseMaker
  (make-noise [this] (str (:name this) "meows!")))

(defrecord NoisyDog [color name]
  NoiseMaker
  (make-noise [this] (str (:name this) "barks!")))

=> (def noisy-cat (NoisyCat. "Calico" "Fuzzy McBootings"))
#'mbfpp.oo.javabean.catsanddogslivingtogether/noisy-cat
=> (def noisy-dog (NoisyDog. "Brown" "Brown Dog"))
#'mbfpp.oo.javabean.catsanddogslivingtogether/noisy-dog
=> (make-noise noisy-cat)
"Fuzzy McBootingsmeows!"
=> (make-noise noisy-dog)
"Brown Dogbarks!"
```

以上便是Clojure中用于承载和传递数据的两种主要方式。首先，普通的map是你开始启动项目时的最佳选择。一旦你的数据模型变得更加确定，或是你想要利用Clojure协议的多态机制时，便可以切换到使用记录来完成工作。

讨论

对于你来说，在锁定数据结构和保持数据结构的灵活性之间存在着一定的权衡。在开发期间，当你的数据模型还不稳定时，可以让数据结构尽可能保持灵活，但是锁定数据结构却可以尽早暴露出代码的问题。一旦代码进入生产阶段，锁定数据结构将会变得非常重要。这在更宽泛的技术世界里关于传统的关系型数据库的诸多争论中可见一斑。比如强制要求严格模式的传统关系型数据库和那些提倡无模式或更宽松模式的非关系型数据库之间的争论，它们都各自宣称自己的方式更加出色。

在现实生活中，就不同的情况而言，这两种方式都非常有用。Clojure和Scala使我们鱼和熊掌得以兼得，让我们在一开始可以保持数据结构的灵活性（使用Clojure中的map和Scala中的元组），并允许我们在进一步了解了数据结构之后将它们进行锁定（使用Clojure中的记录和Scala中的类及样本类）。

延展阅读

《Effective Java中文版》第2条：遇到多个构造器参数时要考虑使用生成器

《Effective Java中文版》第15条：使可变性最小化

相关模式

模式19　集中的可变性

模式5

替代迭代器模式

目的

在无需对序列中的元素进行索引的情况下，有序地对元素进行迭代访问。

概述

迭代器是一个可以让我们对序列中的所有元素进行迭代访问的对象。它通过维持内部状态来保持对当前序列中迭代器所在位置的跟踪。就最简单的迭代器实现来说，它只需要一个可以返回序列中下一个元素的方法，并在序列中没有更多元素时能返回起到“哨兵”作用的值。

大部分的迭代器实现都有一个独立的方法来检查迭代器中是否还存在着更多元素，而无需通过“哨兵”值来进行检测。还有一些迭代器实现允许修改底层的容器来删除当前项。

别名

游标（Cursors）

枚举器（Enumerator）

函数式替换方案

在本节中，我们将专注于如何通过组合高阶函数和序列推导（sequence comprehension）来替代迭代器模式。序列推导是一种巧妙的技术，能让我们以某些精巧的方式将一组序列转型为另一组序列。它们就好比是“打了激素”的`map`函数。

许多基本的迭代器模式的使用场景都可以由简单的高阶函数来替代。举个例子，对一个序列中元素的求和操作可以采用Clojure中的`reduce`高阶函数来完成。

而其他一些更复杂的使用场景可以由序列推导来处理。序列推导为我们根据老的序列来创建新序列提供了一种简洁的方式，它还可以对我们不想要的值进行过滤。

在本节中，我们将会坚持使用迭代器模式。在Java中，可以以foreach循环的方式来表达该模式。另外，一些不太常见的使用场景可以由一些函数式的模式来替代。比如模式12“尾递归模式”和模式13“相互递归模式”。

范例代码：高阶函数

先来看一组可被高阶函数替代的简单的迭代器使用场景。首先，来看如何从字符串中识别出元音字母，然后了解一个为一组人名加上"Hello, "前缀的例子。最后，对一个序列进行求和。

我们首先将会以Java的命令式风格来实现这些例子，然后将这些例子重构成Scala和Clojure版本的更具声明式风格的实现。

传统的Java实现

为了识别出一个单词中的元音字母集，我们需要遍历每个字符并逐个对照元音字母表来进行检查。如果该字符存在于元音字母表之中，便将它添加到vowelsInWorld中并最终返回该Set实例。下面的代码依赖于一个名为isVowel()的帮助方法，从而勾勒出了整个解决方案：

JavaExamples/src/main/java/com/mblinn/oo/iterator/HigherOrderFunctions.java

```
public static Set<Character> vowelsInWord(String word) {

        Set<Character> vowelsInWord = new HashSet<Character>();

        for (Character character : word.toLowerCase().toCharArray()) {
                if (isVowel(character)) {
                        vowelsInWord.add(character);
                }
        }

        return vowelsInWord;
}
```

这里涉及一个抽象层次更高的模式：我们从一个序列中过滤出某些类型的元素。这里是过滤出一个字符串中的所有元音字母，但是我们可能希望过滤出所有的奇数、名叫“Michael”的人或者其他东西。我们将在函数式替代方案中启用更高阶的模式，这一方案将会用到filter函数。

接下来讨论如何为一组人名添加"Hello, "这一前缀。下面，我们将选取一组人名列表，然后遍历它们，为每个人名添加"Hello, "前缀，并将附加了前缀的元素添加到一个新的列表中。最后，返回该列表。

下面的代码对这一方式进行了展示：

JavaExamples/src/main/java/com/mblinn/oo/iterator/HigherOrderFunctions.java

```
public static List<String> prependHello(List<String> names) {
        List<String> prepended = new ArrayList<String>();
        for (String name : names) {
```

```
                prepended.add("Hello, " + name);
        }
        return prepended;
}
```

同样地，此处也潜藏着一个更高抽象层次的模式。我们将某个操作映射到序列中的每个元素，此处是为每个单词添加"Hello, "字符串的前缀。后面将会看到如何使用高阶的map函数来完成此类任务。

让我们来看看最后一个问题：对一个数字序列求和。在传统的Java中，我们通过迭代整个列表，并将每个数字累加到一个sum变量来完成求和计算：

JavaExamples/src/main/java/com/mblinn/oo/iterator/HigherOrderFunctions.java

```
public static Integer sumSequence(List<Integer> sequence) {
        Integer sum = 0;
        for (Integer num : sequence) {
                sum += num;
        }
        return sum;
}
```

这一迭代类型属于另一个模式的一个实例，即在一组序列上执行操作并将这组序列规约为一个单独的值。我们将在函数式替代方案中通过采用reduce函数和一个相近的fold函数来展示对该模式的使用。

Scala实现

来看第一个例子，返回一个单词中的元音字母集。在函数式的世界里，这可以通过两个步骤来完成：首先使用filter()函数过滤出单词中的所有元音字母，然后将得到的序列转换成一个set，从而去除掉那些字母中重复的部分。为了完成过滤操作，我们可以利用Scala的一项特性，即set可以作为判定函数来进行调用。如果set包含传入的参数，它将返回true；否则返回false，代码如下所示：

```
scala> val isVowel = Set('a', 'e', 'i', 'o', 'u')
isVowel: scala.collection.immutable.Set[Char] = Set(e, u, a, i, o)

scala> isVowel('a')
res0: Boolean = true

scala> isVowel('z')
res1: Boolean = false
```

现在可以将上面的isVowel()函数与filter()及toSet()一起使用，从而从一个字符串中获得一个元音字母的集合：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/iterator/HigherOrderFunctions.scala

```
val isVowel = Set('a', 'e', 'i', 'o', 'u')
def vowelsInWord(word: String) = word.filter(isVowel).toSet
```

下面通过实战来看看这个函数的功效，从字符串中过滤出元音字母：

```
scala> vowelsInWord("onomotopeia")
res4: scala.collection.immutable.Set[Char] = Set(o, e, i, a)

scala> vowelsInWord("yak")
res5: scala.collection.immutable.Set[Char] = Set(a)
```

来看下一个例子：为人名列表添加"Hello, "前缀，这可以通过在字符串序列上映射一个添加前缀的函数来实现。此处的映射是指函数可以应用于序列中的每个元素，然后返回一个携带了期望结果的新序列。在下面的代码中，我们为序列映射了一个函数，该函数可以为序列中的每个人名添加"Hello, "字符串前缀：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/iterator/HigherOrderFunctions.scala

```
def prependHello(names : Seq[String]) =
  names.map((name) => "Hello, " + name)
```

如下面的代码所示，该函数完成了我们期望的工作。Scala的REPL会在序列中的每两个元素之间插入一个逗号，因此我们看到在每一组问候语之间都被加上了一个额外的逗号。

```
scala> prependHello(Vector("Mike", "John", "Joe"))
res0: Seq[java.lang.String] = Vector(Hello, Mike, Hello, John, Hello, Joe)
```

接下来，轮到最后一个例子了——对序列求和。我们会对序列实施一个操作，即相加，从而将该序列规约为一个单独的值。在Scala中，完成这一工作的最简单的方式就是使用一个被称为reduce的函数。该函数名副其实，它接受一个参数，即一个规约函数。

下面我们创建了一个规约函数，该函数会将它的参数相加到一起，然后我们使用它对序列求和：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/iterator/HigherOrderFunctions.scala

```
def sumSequence(sequence : Seq[Int]) =
  if(sequence.isEmpty) 0 else sequence.reduce((acc, curr) => acc + curr)
```

让通过实践来看看这一函数：

```
scala> sumSequence(Vector(1, 2, 3, 4, 5))
res0: Int = 15
```

就是它了——没有迭代，没有可变量，只有一个简单的高阶函数！

Clojure实现

我们的第一个例子充分利用了Scala方案中所使用的一个技巧，即set可以作为判定函数来使用。如果传入的元素存在于set之中，那么集合将返回该元素（请牢记，在Clojure中，除了false和nil之外的所有值都被视为true）；否则将返回nil。

此处，我们通过充分利用Clojure set的这一属性定义了vowel?判定函数，使用该函数以及

filter函数可以从序列中过滤出元音字母。然后，使用Clojure的set函数从一个现有的序列构建新的set。整合后的代码如下所示：

ClojureExamples/src/mbfpp/oo/iterator/higher_order_functions.clj

```
(def vowel? #{\a \e \i \o \u})
(defn vowels-in-word [word]
  (set (filter vowel? word)))
```

现在可以使用它从一个单词中过滤出元音字母集了：

```
=> (vowels-in-word "onomotopeia")
#{\a \e \i \o}
=> (vowels-in-word "yak")
#{\a}
```

接下来是我们的hello前缀器函数：prepend-hello。跟Scala的例子类似，我们简单地采用了一个将"Hello, "添加为字符串前缀的函数，并将该函数映射到序列中的每一个人名之上。代码如下所示：

ClojureExamples/src/mbfpp/oo/iterator/higher_order_functions.clj

```
(defn prepend-hello [names]
  (map (fn [name] (str "Hello, " name)) names))
```

我们可以使用它来生成一系列问候语：

```
=> (prepend-hello ["Mike" "John" "Joe"])
("Hello, Mike" "Hello, John" "Hello, Joe")
```

最后，来看看如何采用Clojure对一组序列求和。跟Scala一样，我们可以继续使用reduce函数，但我们不需要像在Scala中那样亲自创建对整数加和的函数：只需使用Clojure的+函数。代码如下所示：

ClojureExamples/src/mbfpp/oo/iterator/higher_order_functions.clj

```
(defn sum-sequence [s]
  {:pre [(not (empty? s))]}
  (reduce + s))
```

下面，我们用它对一个序列进行求和：

```
=> (sum-sequence [1 2 3 4 5])
15
```

不熟悉Clojure的人可能会发现一个有点奇怪的部分：+被作为函数传入了reduce函数。通常而言，这正是Clojure和Lisp的优势之一。很多事物在其他的语言中会被视为特殊的操作符，而在Clojure中，它们却都只是函数。这允许我们将它们作为参数传递给像reduce这样的高阶函数。

对于Clojure和Scala中的reduce来说，有一点需要格外注意：虽然在这里可以以相同的方式来使用它们，它们实际上却存在着一些区别。Scala的reduce()是针对同一类型序列上的操

作，它会返回这一类型的单个值。举个例子，对Int类型的List进行规约后得到的是一个单独的Int值。

而在Clojure中却并非如此。它允许你在reduce函数中返回任意的内容，包括另一个某种类型的集合！这会更加通用（通常也非常方便），而在Scala中，支持更加通用的规约思想的函数有着另一个名字：foldLeft()。

在Scala中，当你确实在规约一组相同类型实例的序列时，通常使用reduce()会更加简单和清晰。如若不然，就使用foldLeft()。

范例代码：序列推导

Scala和Clojure都支持一个非常方便的特性，我们称之为序列推导。序列推导给我们带来了一套方便的语法，使我们可以将不同的事情放在一起完成。与map函数非常相似，序列推导让我们可以将一个序列转换成另一个。同时也允许我们包含一个过滤的步骤，为我们从聚合数据中获得不同部分提供了便捷的方式，我们称之为解构（destructuring）。

让我们来看看如何使用序列推导来解决一个关于“美味”的小问题。现在我们的手上有着一个人名列表，当我们的新餐厅“The Lambda Bar and Grille”开业的时候，我们需要通知这些人，邀请他们参加这一盛大的开业派对。

我们已经拥有了人名和地址，然后会识别出那些住所离Lambda最近的人，因为他们更有可能过来参加派对，所以应该首先向他们发出邀请。最后，过滤掉那些住所离餐厅过远以至于我们几乎可以确信他们不会来参加的人。

我们打算按如下方式来解决这一问题：基于邮编对客户进行分组，然后向离餐厅最近的邮编所在分组的人群首先发出邀请。另外，我们会将自己限制在一小组较近的邮编之中。

来看看如何解决这个问题。我们将和往常一样，从Java的迭代方案开始，然后转向通过使用Scala和Clojure中的序列推导来完成函数式方案。

传统的Java实现

在Java中，我们采用了习以为常的JavaBean格式来创建了一个Person类和一个Address类，以及一个peopleByZip()方法，该方法接受一组客户人员的信息为入参，并过滤掉那些住得不够近的人员，然后返回一个以邮编为键的map，该map中的每个邮编都对应一个客户人员信息列表。

为了完成这一工作，我们采用了一个标准的迭代方案，同时用到了两个辅助方法。第一个方法是addPerson()，它将一个人员添加到列表中，如果列表不存在，该方法还会创建列表，方便我们处理某个邮编遇到第一个人员的情况。

第二个方法是isCloseZip()，如果人员住所所在区域的邮编离“the Lambda Bar and Grille”餐厅足够近，从而满足我们发送派对邀请的条件时，它将会返回true，否则返回false。为了保持

范例代码的规模足够小，我们在这里对一对邮编进行了硬编码。但是由于我们将邮编的检验逻辑提取到了方法中，所以如果需要将它改造为从一些动态数据源中获取我们所关心的邮编数据也将会变得非常容易。

为了解决这个问题，我们只需对人员信息的列表进行迭代。对于每一个人员，检查他是否拥有一个足够近的邮编，如果足够近就将他们添加到一个名为closePeople的map的相应列表中，该map由列表组成，并以邮编为键。完成迭代之后，我们将返回一个map。该解决方案如下：

JavaExamples/src/main/java/com/mblinn/oo/iterator/TheLambdaBarAndGrille.java

```
public class TheLambdaBarAndGrille {

        public Map<Integer, List<String>> peopleByZip(List<Person> people) {
                Map<Integer, List<String>> closePeople =
                                new HashMap<Integer, List<String>>();

                for (Person person : people) {
                        Integer zipCode = person.getAddress().getZipCode();
                        if (isCloseZip(zipCode)){
                                List<String> peopleForZip =
                                                closePeople.get(zipCode);
                                closePeople.put(zipCode,
                                                addPerson(peopleForZip, person));
                        }
                }

                return closePeople;
        }
        private List<String> addPerson(List<String> people, Person person) {
                if (null == people)
                        people = new ArrayList<String>();
                people.add(person.getName());
                return people;
        }
        private Boolean isCloseZip(Integer zipCode) {
                return zipCode == 19123 || zipCode == 19103;
        }
}
```

这是一个相当简单的数据转换，但是以命令式的风格来完成这一任务需要做不少的工作，因为要在向新的列表添加元素上花费很多的时间，这一切都源于我们并不具备从现有列表中过滤元素的头等方式。更具声明性的序列推导将帮助我们提升抽象的层次。现在来看看Scala的版本。

Scala实现

在Scala中，我们可以使用Scala的语法来进行序列推导，for推导表达式可以以一种更加清晰的方式来为我们生成问候语。我们将使用样本类来生成Person类和Address类，同时编写一个for推导表达式，该推导接受一个Person的序列，并生成一个问候语序列。

出于几点原因，for推导表达式在这一过程中显得格外便捷。首先我们可以在其中使用Scala

的模式匹配语法，这一特性赋予我们一种简洁的方式从一个Person中分离出姓名与地址。

其次，作为一种保护机制，for推导表达式让我们直接将过滤器包含在表达式中，因此无需为过滤错误邮编的人员编写单独的if语句。最后，for推导表达式可用于创建新的序列，所以我们也不需要为积累新值而创建一个临时的列表；只需要返回该表达式的值即可。

除了for推导表达式之外，我们仍会使用一个辅助的isCloseZip()方法，但只将它作为for推导表达式自身保护机制的一部分，同时将彻底废除Java解决方案中的问候语可变列表，因为我们所需要的结果只是for推导表达式自身的值。

整个解决方案的代码如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/iterator/TheLambdaBarAndGrille.scala

```
case class Person(name: String, address: Address)
case class Address(zip: Int)
def generateGreetings(people: Seq[Person]) =
  for (Person(name, address) <- people if isCloseZip(address.zip))
    yield "Hello, %s, and welcome to the Lambda Bar And Grille!".format(name)
def isCloseZip(zipCode: Int) = zipCode == 19123 || zipCode == 19103
```

使用for推导表达式的时候，有一点可能并不明显，即如何处理仅需要for推导表达式的副作用的情况。因为我们使用函数式风格进行编程，所以这种情况并不常见。正如在上面的内容中所看到的，我们并不需要一个可变的列表来生成我们的问候语列表。在控制台打印信息是在函数式世界里简单运用副作用的一个例子，同时也是我们所需要的一种场景。

下面是我们重写的一个例子，它只是将问候语打印在控制台，而不是将它们收集到一个序列中：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/iterator/TheLambdaBarAndGrille.scala

```
def printGreetings(people: Seq[Person]) =
  for (Person(name, address) <- people if isCloseZip(address.zip))
    println("Hello, %s, and welcome to the Lambda Bar And Grille!".format(name))
```

我们在此处所接触到的仅仅是Scala for推导表达式的基础功能；事实上，它们的功能是非常强大的。它们可以配合多种其他的特性，并与多个序列和多重保护机制一起使用。但此处所涉及的案例，是我们以前会选择迭代器模式来处理的最为通用的场景。

Clojure实现

Clojure同样可以通过采用宏for来获得内建的序列推导。跟Scala一样，Clojure序列推导的根本点在于配合内建的过滤机制将一个序列转换成另一个序列。Clojure的序列推导也为解构和分离聚合数据提供了方便之门。

由于Clojure和Scala的序列推导非常相似，至少从此处的基础使用场景来看是这样的，而它们的解决方案的结构看上去也格外类似。我们有一个close-zip?函数，该函数利用了Clojure方便

的set即函数的特性，而generate-greetings函数仅由一行for语句组成。

下面的for语句使用了close-zip?过滤出我们关注范围之外的人员，并为过滤后剩下的人员生成问候语。代码如下所示：

ClojureExamples/src/mbfpp/oo/iterator/lambda_bar_and_grille.clj

```
(def close-zip? #{19123 19103})

(defn generate-greetings [people]
  (for [{:keys [name address]} people :when (close-zip? (address :zip-code))]
  (str "Hello, " name ", and welcome to the Lambda Bar And Grille!")))
```

Clojure也支持使用类似序列推导的语法来获得副作用，尽管Clojure将它提取到了宏doseq中。下面我们使用了doseq来打印问候语列表而不是将它们收集起来：

ClojureExamples/src/mbfpp/oo/iterator/lambda_bar_and_grille.clj

```
(defn print-greetings [people]
  (for [{:keys [name address]} people :when (close-zip? (address :zip-code))]
    (println (str "Hello, " name ", and welcome to the Lambda Bar And Grille!"))))
```

Scala和Clojure的序列推导虽然并不完全相同，但在某些方面格外相似。Scala的for语句在使用上通常更加普遍，并且其使用方式往往会令门外汉大吃一惊。举个例子，for语句可以结合Scala的option类型来使用，从而为原本在Java中需要做大量null检查的问题提供优雅的解决方案。这一部分的内容将在模式8“替代空对象”中进行讨论。

同样地，Scala的模式匹配和clojure的解构也有一些相似的地方，它们都允许我们从聚合数据结构中分离出数据；Scala中的模式匹配较Clojure中的解构来说灵活性稍差一些。解构允许我们对任意的map和vector进行数据分离，而Scala的模式匹配则局限于样本类和其他一些在编译时进行静态定义的构件。

讨论

在迭代器模式和我们在本章中所讨论的解决方案之间存在着一个并不显著的区别。这个区别在于迭代器模式从根本上来说是命令式的，因为它依赖于可变的状态。每个迭代器都在其内部拥有一些状态，这些状态用于保持迭代器对当前位置的跟踪。一旦你将迭代器进行传递，并且你程序的一部分意外地移动了迭代器的指针，从而影响了程序的另一部分，那么这将会为你带来麻烦。

相比之下，本章所采用的解决方案依赖于将一个不可变序列转换成另一个序列。事实上，我们在前面讨论的序列推导同时也是单子变换（monadic transformation）技术的几个实例，这一技术经由Haskell这门高度函数式语言的推广而普及，它依赖于一个源自范畴论的概念，称之为单子（monads）。

对于单子的解释，在函数式程序员之间都有着各自的理解与认识，从而涌现出了一大批的博客文章。这些文章尝试通过与墨西哥卷饼、大象、写字台和木偶剧等其他事物进行类比来解释单

子。我们不会在这里给你灌输更多的关于单子的说明；对于使用序列推导来说，并不需要我们去理解单子的概念。而且Scala和Clojure也都没有对它们各自推导的单子本质进行特别地强调。

然而在一个很高的层次上来看，单子的使命之一就是为我们提供了一种新的方式，通过在管道中转换不可变数据让我们获得了颇具函数式风格的程序，让我们不再依赖于那些可变的状态。如果你对单子感到好奇，可以参考阅读《Haskell趣学指南》[Lip11]。

延展阅读

《设计模式：可复用面向对象软件的基础》中Iterator相关章节

Java标准库[①]

相关模式

模式12　尾递归模式

模式13　相互递归模式

模式14　Filter-Map-Reduce模式

模式6

替代模板方法模式

目的

指定算法的大致轮廓，并让调用者完成对某些细节的插入

概述

模板方法模式由一个抽象类组成，该抽象类定义了一些操作或根据抽象子操作定义了一组操作。模板方法模式的用户通过实现该抽象模板类来为每个子步骤提供实现。一个模板类的大致结构如下面的代码片段所示：

```
public abstract class TemplateExample{

  public void anOperation(){
    subOperationOne();
    subOperationTwo();
```

① http://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html

```
    }

    protected abstract void subOperationOne();

    protected abstract void subOperationTwo();
}
```

为了使用该模板，我们必须继承TemplateExample类，并实现其中的几个抽象子方法。

例如，使用模板方法模式来创建一个棋盘游戏。我们需要创建一个定义了可以进行棋盘游戏的一系列抽象步骤的游戏模板（这些步骤包括setUpBoard()、makeMove()和declareWinner()等）。为了实现任何特定的棋盘游戏，我们需要继承抽象的Game类，并为某一游戏实现对应的子步骤。

函数式替代方案

模板方法模式的函数式替代方案完全能满足我们的目的，即为某些算法创建骨架，然后让调用者插入一些细节。我们不再使用类来实现子步骤方法，转而采用高阶函数；同时，也不再依赖于子类继承的方式，而是依赖于函数的组合。我们将把所有的子操作传入一个函数生成器（Function Builder），该生成器会返回一个执行所有操作的新函数。

这种方式以Scala来实现的大致框架如下：

```
def makeAnOperation(
  subOperationOne: () => Unit,
  subOperationTwo: () => Unit) =
  () => {
    subOperationOne()
    subOperationTwo()
  }
```

这样一来就可以更加直接地进行程序编写，因为不再需要定义子操作和子类。

范例代码：评分报表

来看一个用于打印评分报表的模板方法模式的实例。它通过两个步骤来实现。第一步，获取一组数字形式的列表并将它转换成字母的形式；第二步，对报表进行格式化并打印。

由于这两个步骤可以由很多不同的方式来完成，我们只对用于创建报表的基本骨架进行指定，即首先转换评分的形式，然后对其格式化并打印报表。关于评分如何转换，以及报表如何打印等每个独立环节的实现就留给后续的调用者。

我们将会完成如下两种实现。第一种将评分转换成完整的字母评分，例如：A、B、C、D和F，并打印出简单的直方图。第二种会为部分字母添加加号或减号，并打印完整的评分列表。

传统的Java实现

在传统Java中采用模板方法模式来解决这一问题的框架如下：一个GradeReporterTemplate类，它拥有一个已经完全实现的方法reportGrades()，以及两个抽象方法numToLetter()和printGradeReport()。

方法numToLetter()指定了如何将单个数字评分转换成一个字母评分，而printGradeReport()指定了如何对评分报表进行格式化并打印。这两个方法都需要由模板的用户来实现。下面的类图展示了该框架的轮廓：

为了获得不同行为的模板实现，模板类的用户需要创建两个对numToLetter()和printGradeReport()方法具有不同实现的子类。

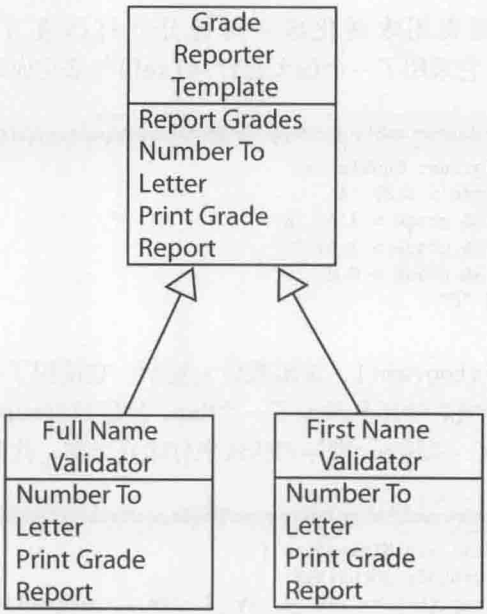


图3-4 评分报表模板。使用模板方法模式来报告评分

Scala实现

Scala不再依赖于继承，它的模板方法模式的替代方案选择采用函数生成器来对子操作进行组合。

该解决方案的核心是函数makeGradeReporter()，该函数接受一个numToLetter()函数来将数字评分转换成字母评分，并接受一个printGradeReport()函数来打印报表。最终，makeGradeReporter()函数会返回一个新的函数，这个函数会将传入的函数组合在一起。

我们将需要两组不同实现的numToLetter()和printGradeReport()函数，可以通过实战来

看看这一解决方案。

首先来看makeGradeReporter()。它接受numToLetter()和printGradeReport()函数作为入参，并产生一个以代表评分列表的Seq[Double]类型的参数作为入参的新函数。它使用map()函数将每一个数字评分转换成字母评分，并将转换后的新的评分列表传递给printGradeReport()。代码如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tm/GradeReporter.scala

```
def makeGradeReporter(
  numToLetter: (Double) => String,
  printGradeReport: (Seq[String]) => Unit) = (grades: Seq[Double]) => {
    printGradeReport(grades.map(numToLetter))
}
```

现在来看那些我们需要用来转化成字母评分并打印直方图的函数。首先是函数fullGradeConverter()，它采用了一个很大的if-else语句来完成评分的转换：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tm/GradeReporter.scala

```
def fullGradeConverter(grade: Double) =
  if(grade <= 5.0 && grade > 4.0) "A"
  else if(grade <= 4.0 && grade > 3.0) "B"
  else if(grade <= 3.0 && grade > 2.0) "C"
  else if(grade <= 2.0 && grade > 0.0) "D"
  else if(grade == 0.0) "F"
  else "N/A"
```

下一个函数是printHistogram()，该函数更为复杂。它使用了一个名为groupBy()的方法来对评分进行分组，并将分组后的评分放入了一个Map，然后通过map()方法将该Map转换成一个评分计数的元组列表。最后，采用for推导表达式来打印直方图，代码如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tm/GradeReporter.scala

```
def printHistogram(grades: Seq[String]) = {
  val grouped = grades.groupBy(identity)
  val counts = grouped.map((kv) => (kv._1, kv._2.size)).toSeq.sorted
  for(count <- counts) {
    val stars = "*" * count._2
    println("%s: %s".format(count._1, stars))
  }
}
```

下面来逐行看一下这个示例，从printHistogram()方法体的第一行开始看起：

```
Val grouped = grades.groupBy(identity)
```

方法groupBy()以一个函数作为入参，并使用该函数对序列中的所有元素进行分组，该函数会返回与当前元素相同的值。我们在此处传入identify函数，它将会返回我们传入的任何内容，因此我们可以对所有相同的评分进行分组。下面的REPL输出展示了如何对一个vector中的评分进行分组：

```
scala> val grades = Vector("A", "B", "A", "B", "B")
grades: scala.collection.immutable.Vector[java.lang.String] = Vector(A, B, A, B, B)

scala> val grouped = grades.groupBy(identity)
grouped: scala.collection.immutable.Map[...] =
        Map(A -> Vector(A, A), B -> Vector(B, B, B))
```

接下来我们得到了分组后的评分map，并采用map()和toSeq()来将该map转换成一个元组的序列。该元组的第一个元素是评分，而第二个元素是评分的数量。然后对序列进行排序。默认情况下，Scala对元组序列的排序是根据元组第一个元素的大小来进行的，这样一来我们便得到了一个已排序的评论数序列。

```
val counts = grouped.map((kv) => (kv._1, kv._2.size)).toSeq.sorted
```

REPL的输出向我们展示了该代码片段是如何帮我们获取评分数序列的：

```
scala> val counts = grouped.map((kv) => (kv._1, kv._2.size)).toSeq.sorted
counts: Seq[(java.lang.String, Int)] = ArrayBuffer((A,2), (B,3))
```

最后，对元组序列执行for推导表达式，并打印出评分的直方图，代码片段如下所示：

```
for(count <- counts) {
  val stars = "*" * count._2

  println("%s: %s".format(count._1, stars))
}
```

这部分代码凸显了Scala的*操作符的一个有趣的用法。它可以用于对字符串的重复处理，如下面REPL的输出所示：

```
scala> "*" * 5
res0: String = *****
```

现在我们只需要使用makeGradeReporter()来对两个函数进行组合，从而构造出一个fullGradeReporter()函数，代码如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tm/GradeReporter.scala

```
val fullGradeReporter = makeGradeReporter(fullGradeConverter, printHistogram)
```

接下来可以定义一些样例数据，并运行fullGradeReporter()打印一个直方图：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tm/GradeReporter.scala

```
val sampleGrades = Vector(5.0, 4.0, 4.4, 2.2, 3.3, 3.5)

scala> fullGradeReporter(sampleGrades)
A: **
B: ***
C: *
```

如果现在我们想要改变评分转换和报表打印的方式，只需要创建额外的评分转换和报表函数。可以使用makeGradeReporter()来将这两个函数重新组合起来。

来看看如何重写模板方法模式的这个实例，我们要将其改造为可转换成带有加减号的评分方式，并能打印出完整的评分列表。跟以前一样，我们需要两个函数。第一个函数是plusMinusGradeConverter()，用于我们的评分转换。第二个是printAllGrades()方法，负责打印出转换后的评分列表。

下面是plusMinusGradeConverter()函数的代码：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tm/GradeReporter.scala

```
def plusMinusGradeConverter(grade: Double) =
  if(grade <= 5.0 && grade > 4.7) "A"
    else if(grade <= 4.7 && grade > 4.3) "A-"
        else if(grade <= 4.3 && grade > 4.0) "B+"
        else if(grade <= 4.0 && grade > 3.7) "B"
        else if(grade <= 3.7 && grade > 3.3) "B-"
        else if(grade <= 3.3 && grade > 3.0) "C+"
        else if(grade <= 3.0 && grade > 2.7) "C"
        else if(grade <= 2.7 && grade > 2.3) "C-"
        else if(grade <= 2.3 && grade > 0.0) "D"
        else if(grade == 0.0) "F"
        else "N/A"
```

接着是printAllGrades()的代码：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tm/GradeReporter.scala

```
def printAllGrades(grades: Seq[String]) =
  for(grade <- grades) println("Grade is: " + grade)
```

现在我们只需要使用makeGradeReporter()来将它们组合起来，然后便可以使用它来创建完整的评分报表了，代码如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tm/GradeReporter.scala

```
val plusMinusGradeReporter =
  makeGradeReporter(plusMinusGradeConverter, printAllGrades)

scala> plusMinusGradeReporter(sampleGrades)
Grade is: A
Grade is: B
Grade is: A-
Grade is: D
Grade is: C+
Grade is: B-
```

以上就是模板方法模式在Scala中的替代方案。下面来看一下Clojure中的替代方案。

Clojure实现

模板方法模式在Clojure中的替代方案与Scala版非常相似。跟Scala版一样，我们将会用到模式16“函数生成器模式”，采用一个名为make-grade-reporter的函数来将一个用于将数字评分转换成字母评分的函数和另一个用于打印报表的函数组合起来。make-grade-reporter将返回

一个函数，该函数将num-to-letter映射到数字评分的序列上。先来看它的代码：

ClojureExamples/src/mbfpp/oo/tm/grade_reporter.clj

```
(defn make-grade-reporter [num-to-letter print-grade-report]
  (fn [grades]
    (print-grade-report (map num-to-letter grades))))
```

在将数字评分转换成完整字母评分中，我们采用了cond表达式，如下所示：

ClojureExamples/src/mbfpp/oo/tm/grade_reporter.clj

```
(defn full-grade-converter [grade]
  (cond
    (and (<= grade 5.0) (> grade 4.0)) "A"
    (and (<= grade 4.0) (> grade 3.0)) "B"
    (and (<= grade 3.0) (> grade 2.0)) "C"
    (and (<= grade 2.0) (> grade 0)) "D"
    (= grade 0) "F"
    :else "N/A"))
```

我们可以按照Scala的方式来打印直方图，首先使用group-by对评分进行分组，然后通过在分组后的评分上映射一个函数来获得评分数，最后再用序列推导将最终的直方图打印出来。下面是打印直方图的代码：

ClojureExamples/src/mbfpp/oo/tm/grade_reporter.clj

```
(defn print-histogram [grades]
  (let [grouped (group-by identity grades)
        counts (sort (map
                      (fn [[grade grades]] [grade (count grades)])
                      grouped))]
    (doseq [[grade num] counts]
      (println (str grade ":" (apply str (repeat num "*")))))))
```

现在使用make-grade-reporter将full-grade-converter和print-histogram组装成一个新的函数：full-grade-reporter。代码如下所示：

ClojureExamples/src/mbfpp/oo/tm/grade_reporter.clj

```
(def full-grade-reporter (make-grade-reporter full-grade-converter print-histogram))
```

接下来使用它运行一些样例数据：

ClojureExamples/src/mbfpp/oo/tm/grade_reporter.clj

```
(def sample-grades [5.0 4.0 4.4 2.2 3.3 3.5])

=> (full-grade-reporter sample-grades)
A:**
B:***
C:*
```

为了改变转换评分和打印报表的方式，我们只需要创建新的函数来配合make-grade-reporter

组成新的函数。让我们重新创建plus-minus-grade-converter和print-all-grades，并将它们组装成新的plus-minus-grade-reporter函数。

函数plus-minus-grade-reporter简洁明了，它就是一个简单的cond表达式：

ClojureExamples/src/mbfpp/oo/tm/grade_reporter.clj

```
(defn plus-minus-grade-converter [grade]
  (cond
    (and (<= grade 5.0) (> grade 4.7)) "A"
    (and (<= grade 4.7) (> grade 4.3)) "A-"
    (and (<= grade 4.3) (> grade 4.0)) "B+"
    (and (<= grade 4.0) (> grade 3.7)) "B"
    (and (<= grade 3.7) (> grade 3.3)) "B-"
    (and (<= grade 3.3) (> grade 3.0)) "C+"
    (and (<= grade 3.0) (> grade 2.7)) "C"
    (and (<= grade 2.7) (> grade 2.3)) "C"
    (and (<= grade 2.3) (> grade 0)) "D"
    (= grade 0) "F"
    :else "N/A"))
```

该print-all-grades函数简单地使用了一个序列推导来打印出每一个评分：

ClojureExamples/src/mbfpp/oo/tm/grade_reporter.clj

```
(defn print-all-grades [grades]
  (doseq [grade grades]
    (println "Grade is:" grade)))
```

现在我们可以通过make-grade-reporter将它们组合起来，并在样例数据上运行它们，从而打印出评分报表：

ClojureExamples/src/mbfpp/oo/tm/grade_reporter.clj

```
(def plus-minus-grade-reporter
  (make-grade-reporter plus-minus-grade-converter print-all-grades))

=> (plus-minus-grade-reporter sample-grades)
Grade is: A
Grade is: B
Grade is: A-
Grade is: D
Grade is: C+
Grade is: B-
```

以上就是Clojure版本的模板方法模式替代方案。最后将对模板方法模式与它的函数式替代方案进行总结比较。

讨论

模板方法模式的函数式替代方案实现了与原有模式一样的目的，但是它们在操作上有些不

同。我们不再使用子类型来实现特定的子操作，而是选择使用函数式的组合和高阶函数。

这也反映出旧有的面向对象是偏好组合胜于继承的。即使在面向对象的世界里，我也更偏爱使用模式11“替代依赖注入”中所描述的模式，将子操作注入到一个类中，而不是使用模板方法模式和子类化。

其中一个主要原因是这样做可以帮助我们防止代码的重复。举个例子，在本章所使用的例子中，如果我们想要一个能打印带有加减号评分的直方图的类，就需要创建一个更深层次的继承结构，或者从现有的实现中复制黏贴代码。在一个真实的系统中，这样做很快就会出现问题。

组合也可以使得API变得更加明确，模板方法类可能会将原本只能供框架代码使用的protected级别的辅助方法暴露出来，而这些方法本不应该被客户端所调用。唯一能说明这一点的方式就是在API的文档中进行注释。

延展阅读

《设计模式：可复用面向对象软件的基础》中Template Method（模板方法）一节

相关模式

模式1　替代函数式接口

模式7　替代策略模式

模式16　函数生成器模式

模式7

替代策略模式

目的

以抽象的形式来定义算法，使其可以由不同的方式来实现。同时允许将算法注入到客户端，便于被多个不同的客户端所使用。

概述

策略模式由多个部分组成。首先是一个代表某个算法的接口，例如一些校验逻辑或排序程序。第二部分是该接口的一个或多个实现类，这些便是策略类本身。最后是使用这些策略对象的一个或多个客户端。

举个例子，处理来自某个网站上的表单的数据集时，我们可能想采用多种不同的方式来对其进行验证，同时也希望这些验证代码能复用在多个不同的地方。可以创建一个拥有validate()方法的Validator接口来作为我们的策略对象，以及该接口的多套实现，而这些实现可以注入到代码中合适的地方。

别名

Policy

函数式替代方案

策略模式与替代函数式接口（模式1）中的模式非常相近。策略对象本身通常就是一个简单的函数式接口，但是策略模式比函数式接口包含更多变化的部分。不过，模式1中的模式仍将是策略模式在函数式世界中的一种相当直接的替代方案。

为了替代策略类，我们使用了高阶函数来实现所需要的算法。这样可以避免专门为不同的策略实现去创建和定义接口。这样一来，我们就可以直接将策略函数进行传递，并在需要的地方使用它们。

范例代码：人名校验

策略模式一个常见用途是创建多个用于校验同一数据集的不同算法。让我们来看一个策略模式的实例。

我们将为人名（西方人名，包含第一个名字、中间名和姓）的校验实现两种不同的校验策略。第一种校验策略考量人名是否有效的依据是他或她是否具有第一个名字，而第二种策略的考量依据是三个名字是否都已设置齐备。最重要的是，我们还会看到一些简单的客户端代码，这些客户端代码会将有效的人名收集到一起。

Java实现

在Java中，我们需要一个PersonValidator接口，这是我们的两个校验策略类——FirstNameValidator和FullNameValidator都将会实现的接口。校验器本身非常简单，它们会在认为人名有效时返回true，否则返回false。

校验器可以在PersonCollector类中被组装起来，该类将会把通过校验的人名收集起来。下面的类图概括了整个解决方案：

该方案可以很好地完成工作，但是它无端地将我们的逻辑分散到了多个类中。让我们来看函数式技术的运用可以对策略模式进行怎样的简化。

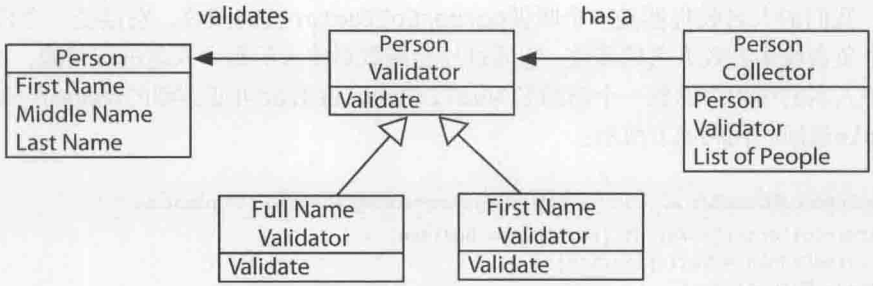


图3-5　人名校验策略。使用策略模式来检验人名

Scala实现

在Scala中，无需像Java那样创建PersonValidator接口。我们将会使用普通的老式函数来完成校验。为了处理好人名，我们将依赖一个将人名各个部分作为其属性的样本类。最后，使用一个高阶函数而非一个完整的类来对人名进行收集。该高阶函数会返回另一个负责收集人名的函数。

让我们从Person样本类开始。这是一个相当标准的样本类，但是要注意我们是如何使用Option[String]替代字符串表示人名的。之所以这样是因为样本类表示的人名中可能会有部分名字是缺失的。

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/strategy/PeopleExample.scala

```
case class Person(
    firstName: Option[String],
    middleName: Option[String],
    lastName: Option[String])
```

现在，来看第一个名字的校验器，这是一个叫作isFirstNameValid()的函数。如下面的代码所示，我们在isDefined()方法中使用了Scala的Option，该方法会在Option包含Some时返回true，否则返回false，从而判断人名中是否包含了第一个名字：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/strategy/PeopleExample.scala

```
def isFirstNameValid(person: Person) = person.firstName.isDefined
```

我们的全名校验器是一个名为isFullNameValid()的函数。在这里，我们使用了Scala的match语句来将人名的各个部分分离出来，然后使用isDefined()来确保每个部分的名字都是存在的。代码如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/strategy/PeopleExample.scala

```
def isFullNameValid(person: Person) = person match {
  case Person(firstName, middleName, lastName) =>
    firstName.isDefined && middleName.isDefined && lastName.isDefined
}
```

最后，我们的人名收集器是一个叫做personCollector()的函数，它接受一个校验函数，并产出一个负责收集有效人名的函数。它通过校验函数对传入的每个人名进行校验，并在人名通过校验后将人名的引用存储到一个新的名为validPeople的var可变类型的vector中。最后它会将validPeople返回，代码如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/strategy/PeopleExample.scala

```
def personCollector(isValid: (Person) => Boolean) = {
  var validPeople = Vector[Person]()
  (person: Person) => {
    if(isValid(person)) validPeople = validPeople :+ person
    validPeople
  }
}
```

让我们通过实践来看看校验器和人名收集器的效果，先创建一个只考虑单个部分名字有效性的人名收集器，然后创建一个考察全名是否有效的收集器：

```
scala> val singleNameValidCollector = personCollector(isFirstNameValid)
singleNameValidCollector: ...

scala> val fullNameValidCollector = personCollector(isFullNameValid)
fullNameValidCollector: ...
```

定义一些供测试使用的人名：

```
scala> val p1 = Person(Some("John"), Some("Quincy"), Some("Adams"))
p1: com.mblinn.mbfpp.oo.strategy.PeopleExample.Person = ...

scala> val p2 = Person(Some("Mike"), None, Some("Linn"))
p2: com.mblinn.mbfpp.oo.strategy.PeopleExample.Person = ...

scala> val p3 = Person(None, None, None)
p3: com.mblinn.mbfpp.oo.strategy.PeopleExample.Person = ...
```

接下来运行这两个人名收集器，从singleNameValidCollector()开始：

```
scala> singleNameValidCollector(p1)
res0: scala.collection.immutable.Vector[...] =
       Vector(Person(Some(John),Some(Quincy),Some(Adams)))

scala> singleNameValidCollector(p2)
res1: scala.collection.immutable.Vector[...] =
       Vector(
              Person(Some(John),Some(Quincy),Some(Adams)),
              Person(Some(Mike),None,Some(Linn)))

scala> singleNameValidCollector(p3)
res2: scala.collection.immutable.Vector[...] =
       Vector(
              Person(Some(John),Some(Quincy),Some(Adams)),
              Person(Some(Mike),None,Some(Linn)))
```

然后以fullNameValidCollector()作为该部分的结尾：

```
scala> fullNameValidCollector(p1)
res3: scala.collection.immutable.Vector[...] =
        Vector(Person(Some(John),Some(Quincy),Some(Adams)))

scala> fullNameValidCollector(p2)
res4: scala.collection.immutable.Vector[...] =
        Vector(Person(Some(John),Some(Quincy),Some(Adams)))

scala> fullNameValidCollector(p3)
res5: scala.collection.immutable.Vector[...] =
        Vector(Person(Some(John),Some(Quincy),Some(Adams)))
```

正如我们所见，两个收集器都按预期的方式工作着，它们将校验的工作都委派给了创建时传入的校验函数。

Clojure实现

在Clojure中，我们将会采用跟Scala中相似的方式来解决人名收集问题，即使用函数来实现校验器，并采用一个接受校验器的高阶函数，该高阶函数会产生一个用于收集人名的函数。为了表示这些人名，我们将使用我们熟知的Clojure map。由于Clojure是一门动态语言，它并不具备Scala中的Option类型，因此我们将使用nil来表示名字的缺失部分。

先来看看first-name-valid?，该函数会检查人名的:first-name是不是nil，如果不是nil，它将会返回true，否则返回false。

ClojureExamples/src/mbfpp/oo/strategy/people_example.clj

```
(defn first-name-valid? [person]
  (not (nil? (:first-name person))))
```

而函数full-name-valid?会检查人名中的所有部分，只有在三个部分全都不是nil的情况下，它才会返回true：

ClojureExamples/src/mbfpp/oo/strategy/people_example.clj

```
(defn full-name-valid? [person]
  (and
    (not (nil? (:first-name person)))
    (not (nil? (:middle-name person)))
    (not (nil? (:last-name person)))))
```

最后，来看看person-collector，它接受一个校验函数，并产生一个收集函数。这种工作方式跟Scala的版本几乎一模一样，最大的区别是我们需要使用一个atom来存储指向不可变vector的引用。

ClojureExamples/src/mbfpp/oo/strategy/people_example.clj

```
(defn person-collector [valid?]
  (let [valid-people (atom [])]
```

```
    (fn [person]
      (if (valid? person)
        (swap! valid-people conj person))
      @valid-people)))
```

在结束之前，让我们通过实践来看看Clojure的人名收集方案，从定义收集函数开始，如下所示：

```
=> (def first-name-valid-collector (person-collector first-name-valid?))
#'mbfpp.oo.strategy.people-example/first-name-valid-collector
=> (def full-name-valid-collector (person-collector full-name-valid?))
#'mbfpp.oo.strategy.people-example/full-name-valid-collector
```

接下来需要一些测试数据：

```
=> (def p1 {:first-name "john" :middle-name "quincy" :last-name "adams"})
#'mbfpp.oo.strategy.people-example/p1
=> (def p2 {:first-name "mike" :middle-name nil :last-name "adams"})
#'mbfpp.oo.strategy.people-example/p2
=> (def p3 {:first-name nil :middle-name nil :last-name nil})
#'mbfpp.oo.strategy.people-example/p3
```

我们可以使用收集器来运行这些数据，从只需要检查第一个名字有效性的收集器开始：

```
=> (first-name-valid-collector p1)
[{:middle-name "quincy", :last-name "adams", :first-name "john"}]
=> (first-name-valid-collector p2)
[{:middle-name "quincy", :last-name "adams", :first-name "john"}
{:middle-name nil, :last-name "adams", :first-name "mike"}]
=> (first-name-valid-collector p3)
[{:middle-name "quincy", :last-name "adams", :first-name "john"}
{:middle-name nil, :last-name "adams", :first-name "mike"}]
```

最后，来看看检查全部名字有效性的收集器：

```
=> (full-name-valid-collector p1)
[{:middle-name "quincy", :last-name "adams", :first-name "john"}]
=> (full-name-valid-collector p2)
[{:middle-name "quincy", :last-name "adams", :first-name "john"}]
=> (full-name-valid-collector p3)
[{:middle-name "quincy", :last-name "adams", :first-name "john"}]
```

这两个收集器的工作效果都符合我们的预期，它们对传入的人名进行校验，并存储通过校验的有效人名，最后返回完整的有效人名的名单。

讨论

策略模式和模板方法模式都服务于相似的目的。它们都可以向一个较大规模的框架或算法注入一些自定义代码。不同的是策略模式采用了组合，而模板方法模式采用的是继承。而我们都是基于函数式组合来代替了这两种模式。

虽然Clojure和Scala都具有允许我们构建继承结构的语言特性，但我们选择了函数式组合来替换策略模式及模板方法模式。之所以这样做，是因为函数式组合为我们带来了更简单的能用于处理通用问题的解决方案，同时也反映了在传统的面向对象思想中，比起继承，我们更偏好于组合。

延展阅读

《设计模式：可复用面向对象软件的基础》中Strategy相关章节

相关模式

模式1　替代函数式接口

模式6　替代模板方法模式

模式 8

替代空对象

目的

为了避免将空检查的逻辑散落在代码中，我们将专门用于处理null引用的措施封装进了一个起代理作用的空对象中。

概述

在Java中，我们通常会采用一个null引用来表示值的缺失。这样做催生了大量的代码：

```
if(null == someObject){
  //默认的空值处理行为
}else{
  someObject.someMethod()
}
```

这种风格的代码将会造成空值处理逻辑散落在所有的代码中，并且不断地重复。即使有合适的默认行为可以用于处理值的缺失，但如果我们忘记了对null的检查，则可能遭遇空指针异常，进而导致程序崩溃。

一种常见的解决方案是创建一个单例的空对象，该对象拥有与实际对象一样的接口，只不过它实现了我们处理空值的默认行为。可以使用该对象来替代null引用。

这样做主要有以下两点好处。

(1) 避免将空检查散落在代码中，使代码保持整洁，且易于阅读。

(2) 集中维护所有处理值的缺失的逻辑。

然而，是否要使用空对象也是值得斟酌的。如果普遍采用这一模式，就意味着你的程序很可能不会快速失败。你或许会因为某个bug而在程序中生成一个空对象，但是你很可能会因此无法察觉到bug的存在。直到程序运行了很久之后，问题才会暴露出来，这让我们对bug来源的追踪变得更困难。

在Java中，我通常只在确信有着足够的理由时才会使用空对象。我的判断依据是为什么某些内容会没有值以及为什么我不在别的地方进行空检查。这两种情况有着微妙的区别。

举个例子，假设我们正在编写一个系统的某个部分，该部分的功能是根据ID查找人员。而该ID是以某种方式生成的，且保持唯一。如果该ID与我们所编写的系统紧密相关（即系统拥有一致可用的ID数据，且不存在延时），并且我们确信每次查询都会成功，并且返回一个人员的实例，那么我会坚持使用null引用。在这种方式下，如果系统出了什么状况导致我们查询的人员不存在，那么系统将会快速失败，而我们也可以立即阻断问题的传播。

然而，如果ID并不与我们的程序紧密相关，那我很可能会采用空对象来处理。比如，ID是由一些别的系统生成，然后通过批处理导入到我们的系统中，这就意味着从ID创建到在我们的系统中可用这两个时间点之间存在着一些延时。在这种情况下，处理缺失的ID将会成为我们程序标准操作的一部分。而我所采用的空对象一方面可以保持代码的整洁，另一方面也避免了外部的空检查。

接下来的函数式替代方案将会对这些权衡点进行更深入的探讨。

函数式替代方案

在本小节的内容中，我们将会考察多种不同的方式。在Scala中，我们将会利用其静态类型和Option类型来替换空对象引用。在Clojure中，将主要集中在对nil的处理上，但同样会涉及Clojure可选的静态类型系统，该系统可以为我们提供类似Scala的Option类型。

Scala实现

在Scala中，我们具有像Java中一样的null引用；然而，我们并不会经常使用它们，而是利用Scala的类型系统来替代null引用和空对象。我们将会看到两个容器类型：Option和Either。先来看Option，它让我们以一种类型安全的方式来表明我们的变量可能会没有值。而Either可以让我们在获得值的时候提供所获得的值，而在没有获得值的时候提供一个默认值或错误值。

首先，进一步来看看Option。Option类型是一个容器，与Map和Vector非常相似，唯一的差别是它一次最多只能容纳一个元素。Option拥有两个重要的子类型：Some和None。前者承载了一个值，而后者则是一个单例的对象，它没有承载任何值。在下面的代码中，我们创建了一个

承载了值"foo"的Some[String]以及一个指向None的引用：

```
scala> def aSome = Some("foo")
aSome: Some[java.lang.String]

scala> def aNone = None
aNone: None.type
```

现在可以以多种方式来使用Option实例了。其中最简单的方式或许是getOrElse()方法。方法getOrElse()调用只接受一个入参，即一个默认的值。当我们在Some实例上调用该方法时，其承载的值将会返回；而如果在None上调用该方法，我们传入的默认值将会返回，代码如下所示：

```
scala> aSome.getOrElse("default value")
res0: java.lang.String = foo

scala> aNone.getOrElse("default value")
res1: java.lang.String = default value
```

使用Option的时候，最干净利落的方式就是将它视为另一个容器类型。举个例子，如果我们需要对Option中的值做某些处理，可以使用老朋友map()，代码如下所示：

```
scala> aSome.map((s) => s.toUpperCase)
res2: Option[java.lang.String] = Some(FOO)
```

我们将会在代码样例中对Option的一些更为复杂的使用方式进行考察。

对于Option，还有一点需要我们注意：虽然Option还有很多更加强大的方式供我们利用，但是其最简单的形式跟我们在Java中的空检查用法差不多。然而，即使采用这种最简单的方式，它们之间也存在着一个主要的区别。

Option是类型系统的一部分，如果我们一贯地使用它，便可以确切地知道代码中的哪些部分需要我们对值的缺失或默认值进行处理。正因为有了这样的前提，才能保证我们必将有值，进而可以在任何其他地方安全地进行代码编写。

Clojure实现

在Clojure中，我们并不具备Scala静态类型系统中所提供的Option类型。不过，我们拥有nil，它在字节码层面等同于Java的null。然而，Clojure提供了许多方便的特性，使得采用nil来处理值的缺失变得相当干净利落，同时也为我们带来了很多在采用空对象时才能获得的好处。

首先，在Clojure中，nil会被作为false来处理。结合表达式的普遍使用，这使得Clojure中对nil的检查远比Java中的null检查简单，如下面的代码所示：

```
=> (if nil "default value" "real value")
"real value"
```

其次，我们用来从Clojure复合数据结构中获取值的函数，可以在我们尝试获取的值不存在的情况下为我们提供默认值。下面使用get方法尝试从一个空的map中获取:foo所对应的值，而我们将取回传入的默认值：

```
=> (get {} :foo "default value")
"default value"
```

然而，对应某个键的值究竟是缺失的还是值为nil两者之间是有区别的，如下面的代码所示：

```
=> (get {:foo nil} :foo "default value")
nil
```

接下来看一些代码示例。

范例代码：默认值

先来看一个例子，该例子展示了当我们在一次map查找中并未获得值时，将如何使用空对象来作为默认值。在这个例子中，我们将会拥有一个装满了人员数据的map，而该map的数据以ID作为键。如果根据指定的ID未能找到人员数据，那么我们需要返回一个名为“John Doe”的默认人员。

传统的Java实现

在传统的Java中，我们将创建一个Person接口，该接口拥有两个子类：RealPerson和NullPerson。其中RealPerson允许我们来设置一个名字和姓，而NullPerson的名字和姓则被硬编码为"John"和"Doe"。

当我们尝试根据某个ID查询人员，而获得null的时候，我们将会返回一个NullPerson的实例；否则，返回从map中取出来的RealPerson实例。下面的代码框架对这一方式进行了概述：

JavaExamples/src/main/java/com/mblinn/oo/nullobject/PersonExample.java

```
public class PersonExample {
        private Map<Integer, Person> people;

        public PersonExample() {
                people = new HashMap<Integer, Person>();
        }

        public Person fetchPerson(Integer id) {
                Person person = people.get(id);
                if (null != person)
                        return person;
                else
                        return new NullPerson();
        }
        //添加或删除人员的代码

        public Person buildPerson(String firstName, String lastName){
                if(null != firstName && null != lastName)
                        return new RealPerson(firstName, lastName);
                else
                        return new NullPerson();
        }
}
```

让我们看看如何采用Scala的Option来消除Java中需要显式进行的空检查。

Scala实现

在Scala中，Map的get()方法并不直接返回值。如果键存在，那么返回的值将会包装在一个Some中，否则会返回None。

例如，在下面的代码中，我们创建了一个map，该map拥有两个整数的键，1和2，它们对应的值分别是一个问候语字符串。当我们尝试用两个键中的任意一个去调用get()时，将取回一个包装在Some中的问候语字符串。而如果传入任意别的键，我们将取回None。

```
scala> def aMap = Map(1->"Hello", 2->"Aloha")
 aMap: scala.collection.immutable.Map[Int,java.lang.String]

scala> aMap.get(1)
res0: Option[java.lang.String] = Some(Hello)

scala> aMap.get(3)
res1: Option[java.lang.String] = None
```

我们可以直接对Option类型进行操作，但是Scala为我们提供了一个很好用的速记方法：getOrElse()。通过该方法我们可以在值不存在的时候从map中直接取得默认值。在下面的REPL输出中，我们使用它从map中尝试获取以3为键的值。由于该值并不存在，我们将取回传入的默认值。

```
scala> aMap.getOrElse(3, "Default Greeting")
res3: java.lang.String = Default Greeting
```

现在来看看如何使用这一特性来实现person-fetching的例子。下面我们使用一个特质来作为人员的基本类型，并采用了样本类来实现RealPerson和NullPerson。可以使用一个NullPerson实例作为查找失败的默认值。下面的代码展示了这一方式：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/nullobject/Examples.scala

```
case class Person(firstName: String="John", lastName: String="Doe")
val nullPerson = Person()

def fetchPerson(people: Map[Int, Person], id: Int) =
  people.getOrElse(id, nullPerson)
```

让我们来定义一些测试数据，从而通过实际的运行来看看这一方式的效果：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/nullobject/Examples.scala

```
val joe = Person("Joe", "Smith")
val jack = Person("Jack", "Brown")
val somePeople = Map(1 -> joe, 2 -> jack)
```

现在，如果我们采用一个存在的key来调用fetchPerson()，那么对应的人员数据将会返回；否则，将返回默认的人员数据：

```
scala> fetchPerson(somePeople, 1)
res0: com.mblinn.mbfpp.oo.nullobject.Examples.Person = Person(Joe,Smith)

scala> fetchPerson(somePeople, 3)
res1: com.mblinn.mbfpp.oo.nullobject.Examples.Person = Person(John,Doe)
```

现在让我们来看看如何采用Clojure来完成这一工作。

Clojure实现

在Clojure中，当我们尝试从map中查找一个并不存在的键时，将会返回nil。

```
=> ({} :foo)
nil
```

Clojure为我们提供了另一种方式从map中查找键，即get函数。该函数允许我们提供一个可选的默认值。下面的REPL片段展示了get用法的一个简单例子。

```
=> (get :foo {} "default")
"default"
```

为了使用Clojure来编写我们的人员查找实例，我们所要做的便是定义一个默认的null-person。在尝试查找时，将它作为默认值传递给get函数，如下面的代码及REPL输出所示：

ClojureExamples/src/mbfpp/oo/nullobject/examples.clj

```
(def null-person {:first-name "John" :last-name "Doe"})
(defn fetch-person [people id]
  (get id people null-person))

=> (def people {42 {:first-name "Jack" :last-name "Bauer"}})
#'mbfpp.oo.nullobject.examples/people
=> (fetch-person 42 people)
{:last-name "Bauer", :first-name "Jack"}
=> (fetch-person 4 people)
{:last-name "Doe", :first-name "John"}
```

本实例中的代码展示了空对象的基本用途，即在查询时如何将空对象作为默认值来使用。而接下来，让我们再去看看在修改时如何利用好空对象及其替代方案。

范例代码：从无到有

让我们换个角度来看这个人员实例。这一次，我们不再查找那些并不存在的人员数据，我们想在仅当拥有有效的姓和名的时候去创建一个人员数据。否则，使用默认的人员数据。

传统的Java实现

在Java中，我们将使用与“范例代码：默认值”一节中“传统Java实现”部分一样的空对象。如果同时拥有可用的姓和名，我们将使用RealPerson；否则使用NullPerson。

为了实现这一需求，我们编写了buildPerson()方法，该方法接受一个firstName和一个

lastName作为入参。如果这两个参数中的任意一个是null，我们将返回NullPerson；否则，返回根据传入姓名构建的RealPerson。下面的代码对这一解决方案进行了概括：

JavaExamples/src/main/java/com/mblinn/oo/nullobject/PersonExample.java

```
public Person buildPerson(String firstName, String lastName){
        if(null != firstName && null != lastName)
                return new RealPerson(firstName, lastName);
        else
                return new NullPerson();
}
```

这一方式允许我们最小化了用以处理null的代码量，帮助我们减少了意外的空指针。现在来看看在不引入额外的空对象的情况下，如何采用Scala来实现相同的需求。

Scala实现

在Scala中，用以处理该问题的方式是利用Option来取代创建特殊的空对象类型。我们传入buildPerson()的firstName和lastName参数都是Option[String]，而返回的是Option[Person]。

如果传入的firstName和lastName都是Some[String]，那么我们将返回Some[Person]；否则，返回None。在Scala中，这样做的正确方式是将Option像任何其他容器一样来处理，譬如Map或Vector。

早些时候，我们见过一个在Some上使用map()方法的简单例子。让我们来看看应该如何使用Scala最强大的序列处理工具来操作Option类型，该处理工具便是我们在模式5中介绍过的序列推导。

首先，在REPL中输入一些测试数据。在下面的片段中，我们定义了一个简单的Vector和一些Option类型：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/nullobject/Examples.scala

```
def vecFoo = Vector("foo")
def someFoo = Some("foo")
def someBar = Some("bar")
def aNone = None
```

正如我们在下面的代码中所看到的，操作Some与操作仅含有单个值的Vector非常相似：

```
scala> for(theFoo <- vecFoo) yield theFoo
res0: scala.collection.immutable.Vector[java.lang.String] = Vector(foo)
scala> for(theFoo <- someFoo) yield theFoo
res1: Option[java.lang.String] = Some(foo)
```

使用for推导表达式来处理Option的真正威力在我们一次处理多个Option时发挥得淋漓尽致。我们可以使用多重生成器，每个Option对应一个生成器，用来获取对应的值。在下面的代码中，我们使用这一技术来从someFoo和someBar中抽取字符串，并将它们放入一个元组中，我们会将该元组yield出来：

```
scala> for(theFoo <- someFoo; theBar <- someBar) yield (theFoo, theBar)
res2: Option[(java.lang.String, java.lang.String)] = Some((foo,bar))
```

当我们以这种方式来处理Option的时候，如果任何一个生成器产生了一个None，那么整个表达式的值便是None。这种方式给我们提供了一种简洁的语法来处理Some和None：

```
scala> for(theFoo <- someFoo; theNone <- aNone) yield (theFoo, theNone)
res3: Option[(java.lang.String, Nothing)] = None
```

我们可以相当简单地将这种方式应用到我们的person-building中。我们在for推导表达式中使用了两个生成器，一个用于firstName，而另一个用于lastName。然后对Person进行yield操作。该for推导表达式将人员数据包装在一个Option中，然后我们使用getOrElse()来获得该人员数据或默认数据。下面的代码对此进行了展示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/nullobject/Examples.scala

```
def buildPerson(firstNameOption: Option[String], lastNameOption: Option[String]) =
  (for(
      firstName <- firstNameOption;
      lastName <- lastNameOption)
    yield Person(firstName, lastName)).getOrElse(Person("John", "Doe"))
```

下面通过实践来看看这个例子：

```
scala> buildPerson(Some("Mike"), Some("Linn"))
res4: com.mblinn.mbfpp.oo.nullobject.Examples.Person = Person(Mike,Linn)

scala> buildPerson(Some("Mike"), None)
res5: com.mblinn.mbfpp.oo.nullobject.Examples.Person = Person(John,Doe)
```

下面来看如何采用Clojure来处理person-building这个例子，从而结束整个代码样例。

Clojure实现

在Clojure中，person-building例子可以归结为对nil的简单检查。我们将first-name和last-name传入我们的build-person函数。如果它们都是非nil的，我们将使用它们来创建相应的人员数据；否则，创建默认的人员数据。

Clojure将nil作为“falsey”处理这一机制让我们的工作变得非常方便，但是除此之外，它与我们在Java中的处理方式几乎没什么区别。代码如下所示：

ClojureExamples/src/mbfpp/oo/nullobject/examples.clj

```
(defn build-person [first-name last-name]
  (if (and first-name last-name)
    {:first-name first-name :last-name last-name}
    {:first-name "John" :last-name "Doe"}))
```

它产生了一条实际的人员数据和一条默认的人员数据：

```
=> (build-person "Mike" "Linn")
{:first-name "Mike", :last-name "Linn"}
```

```
=> (build-person "Mike" nil)
{:first-name "John", :last-name "Doe"}
```

让我们最后再来看一眼这种处理空值的方式，如果代码中有很多部分想要对值的缺失进行处理，那么这种处理方式与Java的方式是一样的。

讨论

在Clojure中，我们惯用的处理值的缺失的方式与在Scala中的方式大相径庭。差别主要源自于Scala采用的是静态类型系统，而Clojure是动态的。Scala的静态类型系统和类型参数使得`Option`类型的方式成为可能。

对于此处Scala和Clojure方案之间的权衡，一定程度上也反映了我们通常在静态与动态类型之间的权衡。如果采用Scala的方式，那么编译器将帮助我们确保在编译期合理地处理空值。虽然如此，我们还是需要小心不要将Java中的`null`渗透到Scala的代码中。

如果采用Clojure的方式，我们可能会在几乎任何地方遭遇到空指针，就跟在Java中一样。我们需要合理地处理它们，并且要更加小心，否则将面临运行时错误的风险。

不管我使用的是Scala的`Option`类型还是Java与Clojure共有的`null`/`nil`，我都倾向于在代码的最外层做好空值的处理。举个例子，如果我向数据库查询一个有可能存在的人员，那么当我们尝试从数据库取回数据时我往往会对他或她的存在性仅做一次检查。如果需要，我将会使用本模式提到的这些技术来创建一条默认的人员数据。这允许我们在其余的代码中无需再进行null的检查或对`Option`类型进行处理。我发现Scala的`Option`类型可以让我们更容易地以这种风格来编写程序，因为它迫使我们在可能无法获得到值的情况下明确地对值的缺失进行处理，从而确保我们在任何其他地方都将能获得到值。

延展阅读

Pattern Languages of Program Design 3 [MRB97]：Null Object

《重构：改善既有代码的设计》[FBBO99]：引入Null对象

模式 9

替代装饰器模式

目的

将行为添加到单独的对象上，而不是对象所属的整个类上，让我们可以对一个既有的类的行为进行更改。

概述

当我们需要向一个既有的类添加一些行为，但无法对该类进行修改时，装饰器模式将会派上用场。我们可能希望为某个类引入一些重大的变更，但是无法修改系统中已经使用了该类的其他部分。又或者该类是某个代码库的一部分，而我们不能或不想对该代码库进行修改。

装饰器模式结合使用了继承和组合。它从一个至少有一个具体实现的接口开始。该实现便是那个我们不能或不想修改的类。

我们使用一个抽象的装饰类来实现这个接口，该装饰类通过组合的方式获得了一个既有的具体实现类的实例。我们的抽象装饰类自身可以拥有多个实现，它们采用组合的方式来调整既有类的行为，如图3-6所示：

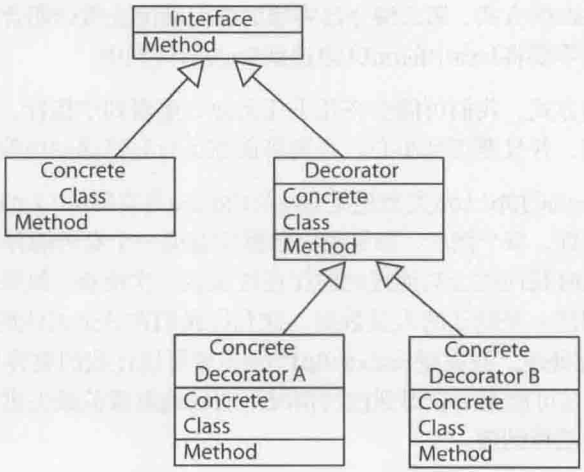


图3-6　装饰器模式图。装饰器模式的类图

该模式有助于我们为既有的类添加或修改行为，但是我们通常仅限于对该类进行非常小的调整，因为我们仍然依赖于该组成类的基础行为。

别名

包装器（Wrapper）

函数式替代方案

装饰器模式的本质是采用一个新的类来对既有类进行包装，以便该新的类可以对既有类的行为进行调整。在函数式的世界里，一种简单的替代方案就是创建一个高阶函数，该高阶函数以既有的函数作为入参，然后返回一个新的经过包装的函数。

该包装函数会执行它的工作，并将一些工作委托给既有的函数。举个例子，我们可以创建一个wrapWithLogger()函数来对一个既有的函数进行包装，包装函数会在原来的基础上提供一些日志的功能，然后返回一个新的包装函数。

范例代码：日志计算器

让我们使用装饰器模式来处理一个四则运算计算器。该计算器拥有四个操作：add()、subtract()、multiply()和divide()。为了展示装饰器模式，我们在这个基本的计算器的基础上加上了日志功能，它可以将我们的计算过程展示在控制台。

传统的Java实现

在Java中，我们的解决方案由一个接口和两个具体的类组成。其中CalculatorImpl和LoggingCalculator两个具体类都实现了Calculator接口。其中LoggingCalculator类便是我们的装饰类，它需要将CalculatorImpl的实例组合进它的实现中，并以此来完成工作。该方案的概要可以从图3-7中一探究竟：

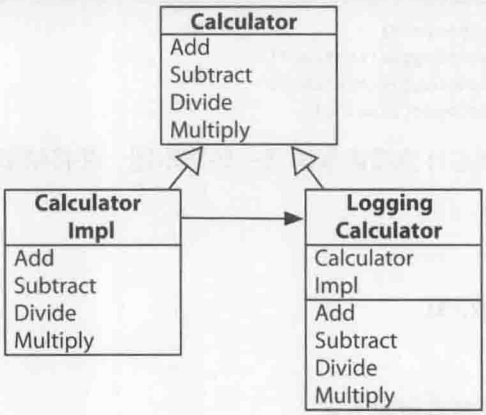


图　3-7

其中LoggingCalculator类会将计算工作委托给组合的CalculatorImpl实例，而它自己将会把计算过程记录到控制台。

Scala实现

在Scala中，我们的计算器仅仅是四个函数的一个集合。为了保持简单，我们设定了一个限制，即只支持整数的操作。这是因为在Scala中实现通用的数字计算函数有一点复杂。我们的Scala计算器代码如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/decorator/Examples.scala

```
def add(a: Int, b: Int) = a + b
```

```
def subtract(a: Int, b: Int) = a - b
def multiply(a: Int, b: Int) = a * b
def divide(a: Int, b: Int) = a / b
```

为了将计算器函数包装进我们的日志代码，我们使用了makeLogger()函数。这是一个高阶函数，它以计算器函数作为入参，并返回一个新的函数，该函数会运行原有的计算器函数，并在返回前将结果打印在控制台。

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/decorator/Examples.scala

```
def makeLogger(calcFn: (Int, Int) => Int) =
  (a: Int, b: Int) => {
    val result = calcFn(a, b)
    println("Result is: " + result)
    result
  }
```

为了使用makeLogger()，我们通过它来运行原有的计算器函数，并将结果赋值给新的val变量，如下面的代码所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/decorator/Examples.scala

```
val loggingAdd = makeLogger(add)
val loggingSubtract = makeLogger(subtract)
val loggingMultiply = makeLogger(multiply)
val loggingDivide = makeLogger(divide)
```

现在可以使用我们的日志计算器函数来做一些算术题，并将结果打印出来：

```
scala> loggingAdd(2, 3)
Result is: 5
res0: Int = 5

scala> loggingSubtract(2, 3)
Result is: -1
res1: Int = -1
```

再来看看Clojure中的计算器解决方案。

Clojure实现

Clojure解决方案的结构与Scala版非常相似，但是由于Clojure是动态类型的，所以它们的主要区别是Clojure解决方案并不只限于处理整数。下面的代码定义了我们的计算器函数：

ClojureExamples/src/mbfpp/oo/decorator/examples.clj

```
(defn add [a b] (+ a b))
(defn subtract [a b] (- a b))
(defn multiply [a b] (* a b))
(defn divide [a b] (/ a b))
```

接下来需要一个高阶函数make-logger将我们的计算器函数与日志代码包装起来：

ClojureExamples/src/mbfpp/oo/decorator/examples.clj

```
(defn make-logger [calc-fn]
  (fn [a b]
    (let [result (calc-fn a b)]
      (println (str "Result is: " result))
      result)))
```

最后，我们可以创建一些日志计算器函数，并使用它们来做一些具备日志功能的数学计算：

ClojureExamples/src/mbfpp/oo/decorator/examples.clj

```
(def logging-add (make-logger add))
(def logging-subtract (make-logger subtract))
(def logging-multiply (make-logger multiply))
(def logging-divide (make-logger divide))

=> (logging-add 2 3)
Result is: 5
5
=> (logging-subtract 2 3)
Result is: -1
-1
```

Scala和Clojure对于计算器问题的解决方案如此相似并非偶然：它们都只依赖于高阶函数，这也是这两门语言的相似之处。

延展阅读

《设计模式：可复用面向对象软件的基础》：装饰器模式

相关模式

模式7　替代策略模式

模式16　函数生成器模式

模式 10

替代访问者模式

目的

以某种方式对在某个数据结构上执行的行为进行封装，进而在不修改原有数据结构的情况下为该数据结构添加新的操作。

概述

对于所有大型的且历时长久的程序而言，一个共同的关注点是如何对数据类型进行扩展。我们想要扩展的维度有两个：首先，为数据类型的既有实现添加新的操作；其次，为数据类型添加新的实现。

我们希望能在不重新编译源代码的情况下完成这一任务，如果可能的话，甚至无需访问源代码。这是一个自从有编程以来就已经存在的问题，我们称之为“表达式问题”（expression problem）。

举个例子，我们将Java的`Collection`作为样例数据类型。该`Collection`接口定义了很多方法或操作，同时它也具有很多实现。在一个完美的世界里，我们可以轻松地向`Collection`添加新的操作和实现。

然而，在面向对象的语言中，只有后者实现起来较为容易。你可以通过实现接口来为`Collection`添加新的实现。如果要为`Collection`添加新的操作，并且要在所有既有的`Collection`实现中生效，我们就没有这么幸运了。

在Java中，我们通常通过创建某个全都是静态工具方法的类来应对这一问题，而非直接向数据类型添加操作。符合这种形式的一个类库就是专门为`Collection`创建的Apache软件基金会的`CollectionUtils`。

访问者模式是用以解决部分此类问题的另一个方案。它让我们可以为既有的数据类型添加新的操作。同时，该模式通常与树形结构的数据一起使用。访问者模式让我们可以相当简单地向既有的数据类型添加新的操作，但它同时又让为数据类型添加新的实现变得非常困难。

访问者模式

访问者模式的类图（如图3-8所示）展示了该模式的主要组成部分。此处的数据类型是`DataElement`类，它拥有两个实现。我们并不直接在`DataElement`的子类型上实现操作，而是创建了一个`accept()`方法，该方法接受一个`Visitor`实例作为入参，并调用该实例的`visit()`方法，同时将指向自己的引用传入该方法。

该模式打破了原有的面向对象的约束，即容易为数据类型添加新的实现，却难以为其添加新的操作。现在，如果想要添加新的操作，只需创建新的访问者，并编写代码来指定如何访问既有的具体元素。

然而，这样一来我们又难以为`DataElement`添加新的实现。因为如果要这么做，需要修改既有的所有访问者实现，以便让它们知道如何访问新的`DataElement`实现。如果这些`Visitor`类并不在我们的掌控之中，那么这将变成不可能完成的任务。

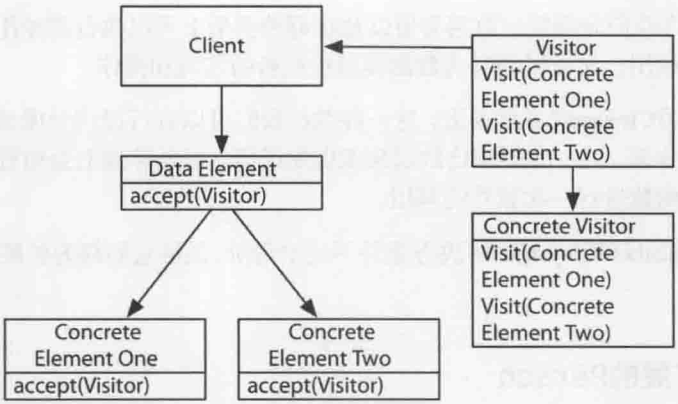


图3-8　访问者模式类图。访问者模式框架图

函数式替代方案

访问者模式使得我们为面向对象的数据类型添加新的操作成为可能，却让我们难以或甚至无法为该类型添加新的实现。在函数式的世界里，这也是一种常态。可以通过编写新的函数来操作某些数据类型从而简单地为其添加新的操作，但是很难让既有的操作支持新的类型。

在我们的替代方案中，我们将会通过使用Scala和Clojure来考察用以处理这一扩展性问题的不同方式。这两门语言的解决方案完全不同。在某种程度上，这是由它们的类型系统造成的，其中Scala是静态类型的，而Clojure却是动态类型的。这意味着Scala要在保证静态类型安全的同时尝试执行扩展是一个较难解决的问题。

另一个区别在于，Scala所采用的多态机制是对传统面向对象模型的一个扩展，它使用了子类的继承结构。而Clojure采纳了一种较为新颖的观点，它采用一种更加特殊的方式来提供多态机制。因为多态是与可扩展性密切相关的，这将对整个解决方案的整体架构产生影响。

Scala实现

Scala是一门混血语言，而对既有代码的扩展正好需要我们启用它的面向对象特性，尤其是它的类型系统。

首先，我们来看一个用于扩展既有类库操作的方法，该方法使用了Scala的隐式转换系统。它让我们可以向既有的类库添加新的操作。

其次，了解如何利用Scala的混入继承（mix-in inheritance）和特质（trait）这两个特性。利用它们，我们可以简单地为数据类型同时添加新的操作和新的实现。

Clojure实现

在Clojure中，我们将了解该语言独特的多态机制。首先，我们将会看到Clojure的数据类型和

协议。这些特性允许我们分别指定数据类型以及在每种类型上可以执行的操作。通过对JVM高度优化的方法分派的利用，又可以同时为数据类型扩展新的实现和操作。

接下来将会学习Clojure的多重方法。这一特性让我们可以自行提供分派函数，从而随心所欲地对方法调用进行分派。这一方案相比协议来说更加灵活，但是性能上会稍慢一些，因为它需要对用户提供的分派函数进行一次额外的调用。

我们所讨论的Scala和Clojure的解决方案并不完全等价，但是它们都为扩展既有代码提供了灵活的方式。

范例代码：可扩展的Person

在这个例子中，我们将看到Person类型。然后考虑如何为该类型同时扩展新的实现和操作。这并不能完全替代访问者模式，但能体现访问者模式所涉及的相关问题。

Java实现

下面是一个用于扩展既有类库的基础实例，同时它并没有对原有对象进行包装。在Java中，只要原有类库的作者为Person定义了接口，那么创建Person类型的新的实现将会是小菜一碟。

而难点在于如何为Person添加新的操作。我们不能仅仅为Person创建一个具有新方法的子接口，因为该子接口无法在使用普通Person的地方使用。基于相同的原因，将Person包装进一个新的类也是无济于事的。

Java在对一个既有的类扩展新的操作方面并不在行，所以我们通常会通过创建某个用于操作该类型的全部是静态工具方法的类来避开这一问题。然而，Scala和Clojure给我们提供了更加灵活的方式，让我们能同时在两个维度上对类型进行扩展。

Scala实现

在Scala中，我们的Person是通过特质来定义的。该特质指定了用于获取人的姓、名、门牌号和街道的方法。除此之外，还有一个用于获取人的全名的方法，如下面的代码所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/visitor/Examples.scala

```
trait Person {
  def fullName: String
  def firstName: String
  def lastName: String
  def houseNum: Int
  def street: String
}
```

现在来创建一个Person类型的实现：SimplePerson。我们将会利用Scala的一个特性，即自动创建用于暴露传入构造器属性的getter方法。唯一需要我们手动实现的方法是fullName()，如下面的代码片段所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/visitor/Examples.scala

```
class SimplePerson(val firstName: String, val lastName: String,
                   val houseNum: Int, val street: String) extends Person {
  def fullName = firstName + " " + lastName
}
```

现在可以创建一个SimplePerson实例，并调用它的fullName()方法：

```
scala> val simplePerson = new SimplePerson("Mike", "Linn", 123, "Fake. St.")
simplePerson: com.mblinn.mbfpp.oo.visitor.Examples.SimplePerson = ...
scala> simplePerson.fullName
res0: String = Mike Linn
```

如果我们想要扩展Person类型从而为其添加另一个操作fullAddress()又该怎么办呢？一种方式是直接创建一个具有新操作的新的子类型，但这样一来我们又无法将该新类型用在需要Person的地方。

在Scala中有一种更好的方式，就是通过定义一个隐式转换来将Person转换成一个具有fullAddress()方法的新类。隐式转换会根据上下文将一种类型改变为另一种类型。

大多数语言都具有某些内建的显式转换或投射的功能。举个例子，在Java中，你可以在一个int变量和一个String变量之间使用+操作符。而该int变量将会被转换成String，同时这两个字符串会被连接起来。

Scala允许程序员自己来定义隐式转换。其中一种方式就是使用隐式类。一个隐式类会将其构造器暴露为隐式转换的候选构造器。下面的代码片段创建了一个隐式类，将Person实例转换成具有方法fullAddress()的ExtendedPerson实例：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/visitor/Examples.scala

```
implicit class ExtendedPerson(person: Person) {
  def fullAddress = person.houseNum + " " + person.street
}
```

当我们尝试调用Person实例上的fullAddress()方法时，Scala编译器会发现Person类型并没有这个方法。它将开始查找所有来自Person的隐式转换类型，直到找到ExtendedPerson类。

编译器会通过将Person实例传入ExtendedPerson的主构造器来构造一个ExtendedPerson实例，然后调用该实例上的fullAddress()方法，如下面的REPL输出所示：

```
scala> simplePerson.fullAddress
res1: String = 123 Fake. St.
```

现在，我们已经学会了一个可以“模拟地”为既有类型添加新方法的技巧，那么最困难的部分已经被我们攻克了。为类型添加新的实现就跟创建一个原有Person特质的新实现一样简单。

来看一个Person的实现：ComplexPerson。它使用了不同的对象来表示Person的名字和地址。

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/visitor/Examples.scala

```
class ComplexPerson(name: Name, address: Address) extends Person {
  def fullName = name.firstName + " " + name.lastName

  def firstName = name.firstName
  def lastName = name.lastName
  def houseNum = address.houseNum
  def street = address.street
}
class Address(val houseNum: Int, val street: String)
class Name(val firstName: String, val lastName: String)
```

现在来创建一个新的ComplexPerson：

```
scala> val name = new Name("Mike", "Linn")
name: com.mblinn.mbfpp.oo.visitor.Examples.Name = ..

scala> val address = new Address(123, "Fake St.")
address: com.mblinn.mbfpp.oo.visitor.Examples.Address = ..

scala> val complexPerson = new ComplexPerson(name, address)
complexPerson: com.mblinn.mbfpp.oo.visitor.Examples.ComplexPerson = ...
```

我们既有的隐式转换仍然会有效！

```
scala> complexPerson.fullName
res2: String = Mike Linn

scala> complexPerson.fullAddress
res3: String = 123 Fake St.
```

这意味着我们能为数据类型同时扩展新的操作和新的实现了。

Clojure实现

来看看可扩展Person实例的Clojure方案。我们将会从定义协议开始，该协议只有一个单独的操作：extract-name。该操作的目的是从一个Person实例中提取其全名，代码如下所示：

ClojureExamples/src/mbfpp/oo/visitor/examples.clj

```
(defprotocol NameExtractor
  (extract-name [this] "Extracts a name from a person."))
```

现在我们可以采用defrecord来创建一个Clojure的记录类型：SimplePerson。该数据类型拥有多个列：

ClojureExamples/src/mbfpp/oo/visitor/examples.clj

```
(defrecord SimplePerson [first-name last-name house-num street])
```

可以使用->SimplePerson工厂函数来创建一个SimplePerson的新实例，正如我们在如下代码片段中所做的：

```
=> (def simple-person (->SimplePerson "Mike" "Linn" 123 "Fake St."))
```

```
#'mbfpp.oo.visitor.examples/simple-person
```

一旦创建成功，便可以将该数据类型视为以关键词为键的map来使用，并从中获取字段的值。在下面的代码片段中，我们从SimplePerson实例中得到了人的名字：

```
=> (:first-name simple-person)
"Mike"
```

注意到我们是如何分别定义数据类型和操作集的了吗？可以通过extend-type使SimplePerson实现NameExtractor协议，如以下代码所示：

ClojureExamples/src/mbfpp/oo/visitor/examples.clj

```
(extend-type SimplePerson
  NameExtractor
  (extract-name [this]
    (str (:first-name this) " " (:last-name this))))
```

现在来调用SimplePerson实例上的extract-name方法，并将该Person实例的全名提取出来：

```
=> (extract-name simple-person)
"Mike Linn"
```

现在来看如何创建一个新的类型：ComplexPerson。该类型采用了内嵌的map来表示它的名字和地址。我们将通过defrecord版本的内联方式在创建记录的同时实现协议。这只是一种便利的方式；而我们所创建的记录和协议仍然跟原来一样：

ClojureExamples/src/mbfpp/oo/visitor/examples.clj

```
(defrecord ComplexPerson [name address]
  NameExtractor
  (extract-name [this]
    (str (-> this :name :first) " " (-> this :name :last))))
```

现在来创建ComplexPerson，然后提取出它的全名：

```
=> (def complex-person (->ComplexPerson {:first "Mike" :last "Linn"}
                                        {:house-num 123 :street "Fake St."}))
#'mbfpp.oo.visitor.examples/complex-person
=> (extract-name complex-person)
"Mike Linn"
```

要向既有的类型添加新的操作或操作集，我们仅需要创建新的协议，然后采用该协议来扩展该类型。在下面的代码片段中，我们创建了一个协议，通过该协议我们可以从一个person的实例中提取出地址信息：

ClojureExamples/src/mbfpp/oo/visitor/examples.clj

```
(defprotocol
  AddressExtractor
  (extract-address [this] "Extracts and address from a person."))
```

现在可以按照新的协议来扩展我们的类型了，如下面的代码所示：

ClojureExamples/src/mbfpp/oo/visitor/examples.clj

```
(extend-type SimplePerson
  AddressExtractor
  (extract-address [this]
                   (str (:house-num this) " " (:street this))))

(extend-type ComplexPerson
  AddressExtractor
  (extract-address [this]
                   (str (-> this :address :house-num)
                        " "
                        (-> this :address :street))))
```

如以下REPL输出所示，现在的数据类型都已遵循新的协议：

```
=> (extract-address complex-person)
"123 Fake St."
=> (extract-address simple-person)
"123 Fake St."
```

虽然使用Scala的隐式转换和Clojure协议都可以达到相同的效果，但是它们却并不相同。在Scala中，我们所看到的操作是定义在类中的方法，它们是类型的一部分。而Scala的隐式转换技术允许我们将一种类型隐式转换成另一种类型，这种方式看上去就好比我们可以向既有的类型添加操作一样。

另一方面，Clojure通过协议和记录分别完整地定义了操作集和类型。我们可以对任意记录扩展任意数量的协议，这就使得我们可以为某个既有的解决方案轻松扩展新的操作和类型。

范例代码：可扩展的几何形状

来看一个更加复杂的例子。首先定义两个形状：圆形和矩形，以及计算它们周长的操作。

接下来将展示如何为既有的周长计算操作添加新的可支持的形状，以及如何为既有的形状添加新的操作。最后，将这两种类型的扩展合并起来。

Java实现

在Java中，这是一个不太容易解决的问题。为形状类型扩展新的实现非常简单。我们可以创建一个Shape接口以及多个该接口的实现。

如果想扩展一个已经拥有实现的Shape，就有点犯难了，但是我们可以使用如下访问者类图所展示的访问者模式来解决这一问题。

然而，如果按这个路线走下去，我们现在将很难扩展新的实现，因为我们不得不为此修改所有既有的访问者实现。如果访问者是由第三方的代码来实现的话，那么在不引入向后不兼容变更

的情况下将无法在该维度上进行扩展。

在Java中，需要在一开始就决定我们是想要为Shape添加新的操作还是添加新的实现。

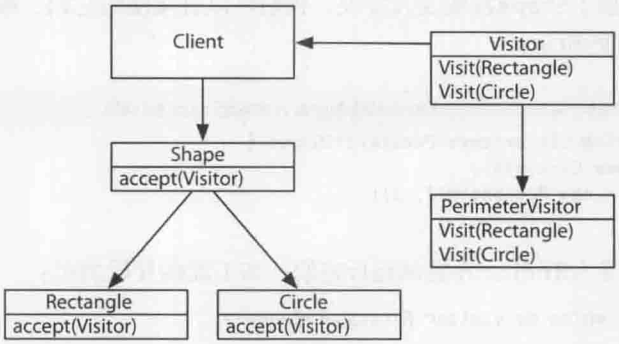


图3-9　几何形状的访问者。访问者模式实现

Scala实现

在Scala中，我们使用某个技术的简化版本。该技术由Scala的设计者Martin Odersky在他的一篇论文中首次提出。

我们将会创建一个名为Shape的特质，并将它作为所有形状的基础。我们一开始会为它添加一个perimeter()方法以及Circle和Rectangle两个实现。

为了展示扩展的魔力，我们将会使用Scala类型系统的一些高级特性。首先，对一个事实善加利用，即我们可以将Scala的特质作为模块使用。在接下去的每一步，我们会将代码打包到一个顶级的特质中，而不是我们正在使用并用以表示Shape的那个特质。

这允许我们将一组数据类型与操作进行捆绑，并在后面使用Scala的混入继承来扩展该捆绑组合。这样我们便可以为新类型扩展多个不同的特质，进而将独立扩展结合起来。

让我们来仔细看一下代码，从最开始的Shape特质以及最早的两个实现开始：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/visitor/Shapes.scala

```
trait PerimeterShapes {
  trait Shape {
    def perimeter: Double
  }

  class Circle(radius: Double) extends Shape {
    def perimeter = 2 * Math.PI * radius
  }

  class Rectangle(width: Double, height: Double) extends Shape {
    def perimeter = 2 * width + 2 * height
  }
}
```

除了最顶层的PerimeterShapes特质之外，这就是一个关于Shape特质及其两组实现的简单声明。为了使用形状代码，我们可以采用该顶层特质来扩展一个object。

这样就为对象添加了Shape特质及其实现。我们可以直接使用它们，或者简单地将它们导入到REPL，如下面的代码所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/visitor/Shapes.scala

```
object FirstShapeExample extends PerimeterShapes {
  val aCircle = new Circle(4);
  val aRectangle = new Rectangle(2, 2);
}
```

现在将这些形状导入REPL，并尝试运行它们，如下面的片段所示：

```
import com.mblinn.mbfpp.oo.visitor.FirstShapeExample._

scala> aCircle.perimeter
res1: Double = 25.132741228718345

scala> aRectangle.perimeter
res2: Double = 8.0
```

在大多数纯面向对象的语言中，为形状扩展新的操作是比较困难的，所以让我们首先解决这个问题。为了扩展初始形状的集合，我们创建了一个叫做AreaShapes的新的顶层特质，该特质扩展于PerimeterShapes。

在AreaShapes内，我们为Shape类扩展了一个area()方法，然后创建了一个新的Circle和新的Rectangle特质，它们都实现了area()方法。扩展代码如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/visitor/Shapes.scala

```
trait AreaShapes extends PerimeterShapes {
  trait Shape extends super.Shape {
    def area: Double
  }

  class Circle(radius: Double) extends super.Circle(radius) with Shape {
    def area = Math.PI * radius * radius
  }

  class Rectangle(width: Double, height: Double)
        extends super.Rectangle(width, height) with Shape {
    def area = width * height
  }
}
```

来深入分析一下该扩展代码。首先，我们创建了顶层的特质AreaShapes，该特质扩展于PerimeterShapes。这让我们可以简单地对AreaShapes内部的类和特质进行引用和扩展：

```
trait AreaShapes extends PerimeterShapes {
       «area-shapes»
}
```

接下来，我们在AreaShapes内部创建了一个新的Shape特质，并让它扩展了PerimeterShapes中那个老的Shape特质：

```
trait Shape extends super.Shape {
        def area: Double
}
```

为了将PerimeterShapes中的Shape类与我们刚刚在AreaShapes中创建的Shape类进行区分，需要通过super.Shape来引用PerimeterShapes中的Shape类。

现在我们准备对area()方法进行实现。为此我们首先扩展了老的Circle和Rectangle类，然后将它们混入新的Shape特质，该特质拥有新的area()方法。

最后，我们实现了新的Circle和Rectangle类的area()方法，如下面的片段所示：

```
class Circle(radius: Double) extends super.Circle(radius) with Shape {
        def area = Math.PI * radius * radius
}

class Rectangle(width: Double, height: Double)
                extends super.Rectangle(width, height) with Shape {
        def area = width * height
}
```

现在可以创建一些样例形状，通过实践的方式来看看perimeter()和area()方法：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/visitor/Shapes.scala

```
object SecondShapeExample extends AreaShapes {
    val someShapes = Vector(new Circle(4), new Rectangle(2, 2));
}

scala> for(shape <- someShapes) yield shape.perimeter
res0: scala.collection.immutable.Vector[Double] = Vector(25.132741228718345, 8.0)

scala> for(shape <- someShapes) yield shape.area
res1: scala.collection.immutable.Vector[Double] = Vector(50.26548245743669, 4.0)
```

以上内容已经覆盖了我们认为最为困难的部分，即为Shape扩展新的操作。现在来看看较为简单的部分。我们将通过创建Square类来为Shape扩展新的实现。

在后续扩展的最初部分，我们创建了顶层特质MorePerimeterShapes，该特质扩展了原来的PerimeterShapes。在该特质内部，我们为原来的Shape特质类创建了新的Square实现。如下面的代码所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/visitor/Shapes.scala

```
trait MorePerimeterShapes extends PerimeterShapes {
  class Square(side: Double) extends Shape {
    def perimeter = 4 * side;
  }
}
```

现在创建另一个新的顶层特质MoreAreaShapes，该特质扩展了原来的AreaShapes，并混入了我们刚创建的MorePerimeterShapes特质。在该特质内部，我们对刚才创建的拥有area()方法的Square进行了扩展：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/visitor/Shapes.scala

```
trait MoreAreaShapes extends AreaShapes with MorePerimeterShapes {
  class Square(side: Double) extends super.Square(side) with Shape {
    def area = side * side
  }
}
```

现在将Square也添加到我们的测试形状集合中，然后通过实践来看看所有的形状及其操作，正如我们在如下代码中所做的：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/visitor/Shapes.scala

```
object ThirdShapeExample extends MoreAreaShapes {
    val someMoreShapes = Vector(new Circle(4), new Rectangle(2, 2), new Square(4));
}

scala> for(shape <- someMoreShapes) yield shape.perimeter
res2: scala.collection.immutable.Vector[Double] =
       Vector(25.132741228718345, 8.0, 16.0)

scala> for(shape <- someMoreShapes) yield shape.area
res3: scala.collection.immutable.Vector[Double] =
       Vector(50.26548245743669, 4.0, 16.0)
```

现在我们已经成功地为Shape添加了新的实现和新的操作，而且我们所采用的是一种类型安全的方式。

Clojure实现

Clojure解决方案依赖于多重方法，该特性允许我们自己来指定任意的分派函数。来看一个简单的例子。

首先，通过使用defmulti来创建多重方法。该操作并没有指定任何方法的实现；但是它包含了一个分派函数。在下面的片段中，我们创建了一个名为test-multimethod的多重方法。分派函数是一个只接受一个入参的方法，并且它会将参数原封不动地返回。然而，它也可以是任意的代码块：

ClojureExamples/src/mbfpp/oo/visitor/examples.clj

```
(defmulti test-multimethod (fn [keyword] keyword))
```

多重方法是通过使用defmethod来实现的。方法的定义看上去跟函数的定义差不多，唯一的差别是它们还包含了一个分派值，该值与分派函数返回的值相对应。

在下面的片段中，我们定义了test-multimethod的两个实现。其中第一个实现所期望的分

派值是:foo，第二个则是:bar。

ClojureExamples/src/mbfpp/oo/visitor/examples.clj

```
(defmethod test-multimethod :foo [a-map]
  "foo-method was called")
(defmethod test-multimethod :bar [a-map]
  "bar-method was called")
```

当多重方法被调用的时候，分派函数会首先被调用，然后Clojure会将当前调用分派给分派值与之匹配的的方法。因为分派函数会将输入的参数返回，所以我们将以所需的分派值来调用它：

```
=> (test-multimethod :foo)
"foo-method was called"
=> (test-multimethod :bar)
"bar-method was called"
```

现在我们已经通过实战经历了一个多重方法的基础实例，下面会更深入一些。将我们的周长计算操作定义为一个多重方法。该分派函数所期望的入参是一个用以代表形状的map。该map的其中一个键就是:shape-name，分派函数会从中提取出该键对应的值来作为我们的分派值。

perimeter多重方法以及圆形和矩形的周长计算实现定义如下：

ClojureExamples/src/mbfpp/oo/visitor/examples.clj

```
(defmulti perimeter (fn [shape] (:shape-name shape)))
(defmethod perimeter :circle [circle]
  (* 2 Math/PI (:radius circle)))
(defmethod perimeter :rectangle [rectangle]
  (+ (* 2 (:width rectangle)) (* 2 (:height rectangle))))
```

现在定义一些测试形状：

ClojureExamples/src/mbfpp/oo/visitor/examples.clj

```
(def some-shapes [{:shape-name :circle :radius 4}
                  {:shape-name :rectangle :width 2 :height 2}])
```

接着调用它们的周长计算方法：

```
=> (for [shape some-shapes] (perimeter shape))
(25.132741228718345 8)
```

为了添加新的操作，我们创建了新的多重方法来处理现有的分派值。在下面的代码片段中，我们添加了对面积计算操作的支持：

ClojureExamples/src/mbfpp/oo/visitor/examples.clj

```
(defmulti area (fn [shape] (:shape-name shape)))
(defmethod area :circle [circle]
  (* Math/PI (:radius circle) (:radius circle)))
(defmethod area :rectangle [rectangle]
  (* (:width rectangle) (:height rectangle)))
```

现在也可以计算形状的面积了：

```
=> (for [shape some-shapes] (area shape))
(50.26548245743669 4)
```

对于现有的形状集合而言，我们可以同时处理它们的周长计算和面积计算。为了向该集合添加新的形状，我们为多重方法添加了新的实现，该实现可以处理相对应的分派值。在下面的代码中，我们增加了对正方形的支持：

ClojureExamples/src/mbfpp/oo/visitor/examples.clj

```
(defmethod perimeter :square [square]
  (* 4 (:side square)))
(defmethod area :square [square]
  (* (:side square) (:side square)))
```

向测试形状的容器添加一个正方形：

ClojureExamples/src/mbfpp/oo/visitor/examples.clj

```
(def more-shapes (conj some-shapes
                       {:shape-name :square :side 4}))
```

我们可以对正方形操作的正常运作进行确认：

```
=> (for [shape more-shapes] (perimeter shape))
(25.132741228718345 8 16)
=> (for [shape more-shapes] (area shape))
(50.26548245743669 4 16)
```

到目前为止我们只触及了多重方法的一些表层功能。由于我们可以指定任意的分派函数，所以可以基于任何事物来进行分派。Clojure还提供了一种让多重方法与用户自定义层级结构一起工作的方式，自定义层级结构与面向对象语言中的类层级结构非常相似。然而，即便是我们目前所看到的对多重方法的简单利用也足以替代访问者模式中的那些有意思的方面。

讨论

在本章的内容中，Scala所面临的问题解决起来要困难得多，因为它在允许对数据类型的实现和其上的操作进行扩展的同时，还要维持静态类型的安全性。而Clojure是动态类型的，它没有这样的需求。

是选择像Scala这样极具表现力的静态类型语言，还是选择像Clojure这样的动态类型语言是需要权衡的，而我们的访问者模式替代方案就是一个很好的例子。在Scala中，我们需要花费更多的努力，而Scala的解决方案也并不像Clojure解决方案那样简单。然而，如果我们尝试在某类型上执行一些在Clojure中无法处理的操作，那么这将是一个运行时而非编译时的问题了。

延展阅读

《设计模式：可复用面向对象软件的基础》：访问者模式

相关模式

模式9　替代装饰器模式

模式 11

替代依赖注入

目的

采用外部的配置或代码来组合对象，而不是让对象自行初始化其依赖——这样做让我们为对象注入不同的依赖实现变得非常简单，并为我们理解给定对象有哪些依赖提供了一个集中的管理场所。

概述

对象是面向对象世界中的基本组成单元。依赖注入将对象组合在一起。以最简单的方式来看，依赖注入所参与的工作就是通过构造器或setter方法将依赖注入对象。

举个例子，下面的类粗略地描述了一个电影服务，该服务可以返回用户收藏的电影。它依赖于一个收藏服务和一个电影的DAO（数据访问对象），前者用来获取收藏的电影列表，后者可以获取每部电影的详情：

JavaExamples/src/main/java/com/mblinn/mbfpp/oo/di/MovieService.java

```
package com.mblinn.mbfpp.oo.di;
public class MovieService {

        private MovieDao movieDao;
        private FavoritesService favoritesService;
        public MovieService(MovieDao movieDao, FavoritesService favoritesService){
                this.movieDao = movieDao;
                this.favoritesService = favoritesService;
        }
}
```

此处使用的是传统的基于构造器的依赖注入。当该服务被创建时，MovieService类必须将它依赖的内容以入参的方式传入构造器。这项工作可以手动完成，但是通常我们会选择一个依赖注入的框架来替我们完成工作。

依赖注入拥有多个好处。它使得替换给定依赖的实现变得更简单，尤其在单元测试中，当我们使用“测试桩”（stub）来替换实际的依赖时会显得格外方便。

在有适当容器的支持下，依赖注入还可以让我们以声明性的方式来更简单地指定系统的整体

结构。我们会将系统每个组件的依赖都抽取到配置文件或配置代码中，并在需要的时候由容器完成注入。

函数式替代方案

当我们以更具函数式的风格进行编程时，对此类依赖注入模式的需求将不会这么强烈。正如我们将在模式16“函数生成器模式”中所看到的，函数式编程本身就包含对函数的组合能力。由于这里所涉及的函数组合能力与依赖注入对类的组合能力非常类似，所以我们无需任何代价就可以直接从函数组合中获得与依赖注入相似的好处。

然而，简单的函数组合并不能解决所有依赖注入所能处理的问题。由于Scala是一门混血语言，所以这一点尤为突出。较大的代码段通常会被组织成对象。

Scala实现

传统的依赖注入也可以在Scala中使用，甚至可以使用像Spring或Guice这样为业界所熟知的Java框架。然而，在不使用任何框架的情况下也能达到许多同样的目标。

我们将会看到一个Scala的模式，人们称之为Cake模式。该模式使用了Scala的特质和自身类型标注（self-type annotation），从而在无需容器的情况下实现了与依赖注入提供的相同的组成和结构。

Clojure实现

由于Clojure并不是面向对象的，所以在Clojure中，注入的单元变成了函数。在大多数情况下，这意味着在面向对象语言中原本采用依赖注入所解决的问题，在Clojure中并不存在，因为我们天然就可以对函数进行组合。

然而在Clojure中，有一个依赖注入的用途需要我们做一些特殊的处理。基于测试的目的，我们需要为函数生成测试桩。为此，可以使用一个名为`with-redefs`的宏，该宏让我们可以暂时采用测试桩来替换函数。

范例代码：收藏的视频

让我们进一步来看看“概述”中所提到的问题。我们创建了一个电影服务，该电影服务可以完成一些与电影相关的行为。每个视频都与一部电影关联，且都需要关联电影的详情内容来点缀，譬如电影的标题。

为了完成该工作，我们创建了一个顶层的电影服务。该服务依赖于一个电影Dao来获得电影的详情，并通过一个收藏服务来获取给定用户的收藏。

传统的Java实现

在Java中，顶层`MovieService`的设计如下面的类所示。可以使用依赖注入通过构造器向

MovieService注入FavoritesService和MovieDao。

JavaExamples/src/main/java/com/mblinn/mbfpp/oo/di/MovieService.java

```
package com.mblinn.mbfpp.oo.di;
public class MovieService {

        private MovieDao movieDao;
        private FavoritesService favoritesService;
        public MovieService(MovieDao movieDao, FavoritesService favoritesService){
                this.movieDao = movieDao;
                this.favoritesService = favoritesService;
        }
}
```

在一个完整的程序中，通常会使用框架来装配MovieService的依赖。该实现有多种方式，从XML配置文件到Java配置类以及注解，都可以自动化地对依赖进行装配。

所有这些方式都拥有一个共同的特征：它们都需要一个外部的框架来起作用。接下来，我们将去看看Scala和Clojure的可选方案，这些方案没有上述的限制。

Scala实现

现在来看一个Scala Cake模式的例子。大致的思想就是将需要注入到顶层特质的依赖进行封装，它们代表了可注入的组件。我们不会在特质的内部直接初始化依赖，而是在对它们进行装配的时候，创建抽象的val字段来保持对它们的引用。

然后，我们将会利用Scala的自身类型标注和混入继承，以一种类型安全的方式来指定装配。最后，采用一个简单的Scala对象作为我们的组件注册表。我们会将所有的依赖混入到该容器对象中并对它们进行初始化，然后采用前面提到的抽象val字段来保持对它们的引用。

这种方式具备了一些不错的特性。正如前面所提到的，它并不需要使用外部的容器。此外，在将事物装配起来的同时还维持了静态类型的安全。

让我们看看即将处理的数据。我们拥有三个样本类：Movie、Video和DecoratedMovie。其中DecoratedMovie表示配备了视频的电影。

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/di/ex1/Services.scala

```
case class Movie(movieId: String, title: String)
case class Video(movieId: String)
case class DecoratedMovie(movie: Movie, video: Video)
```

现在定义一些特质作为接口，并将它们作为依赖：FavoritesService和MovieDao。我们会将这些特质嵌入到另一组用于表示可注入组件的特质之中。稍后就会在本例中看到这样做的必要性。

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/di/ex1/Services.scala

```
trait MovieDaoComponent {
```

```
  trait MovieDao {
    def getMovie(id: String): Movie
  }
}

trait FavoritesServiceComponent {
  trait FavoritesService {
    def getFavoriteVideos(id: String): Vector[Video]
  }
}
```

接下来，完成对刚才所引入的组件的实现。我们将通过实现这些接口来为MovieDao和FavoritesService生成测试桩，从而返回静态的响应。请注意，此外还需要对这些包装进来的组件特质进行扩展。

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/di/ex1/Services.scala

```
trait MovieDaoComponentImpl extends MovieDaoComponent {
  class MovieDaoImpl extends MovieDao {
    def getMovie(id: String): Movie = new Movie("42", "A Movie")
  }
}

trait FavoritesServiceComponentImpl extends FavoritesServiceComponent {
  class FavoritesServiceImpl extends FavoritesService {
    def getFavoriteVideos(id: String): Vector[Video] = Vector(new Video("1"))
  }
}
```

现在来看看MovieServiceImpl，它依赖于前面定义的FavoritesService和MovieDao。该类就实现了一个方法：getFavoriteDecoratedMovies()，该方法以用户ID为入参，并返回用户收藏的电影，这些电影都配备了相关联的视频。

MovieServiceImpl的全部代码包装在一个顶层的MovieServiceComponentImpl特质中，如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/di/ex1/Services.scala

```
trait MovieServiceComponentImpl {
  this: MovieDaoComponent with FavoritesServiceComponent =>

  val favoritesService: FavoritesService
  val movieDao: MovieDao

  class MovieServiceImpl {
    def getFavoriteDecoratedMovies(userId: String): Vector[DecoratedMovie] =
      for (
        favoriteVideo <- favoritesService.getFavoriteVideos(userId);
        val movie = movieDao.getMovie(favoriteVideo.movieId)
      ) yield DecoratedMovie(movie, favoriteVideo)
  }
}
```

让我们一步步深入其中。首先，我们在顶层的MovieServiceComponentImpl特质中使用了自身类型标注。这正是Scala魔法的神奇所在，它可以确保Cake模式的类型安全。

```
this: MovieDaoComponent with FavoritesServiceComponent =>
```

该自身类型标注确保了不管在何时，只要某个对象或类混入了MovieServiceComponentImpl，那么该对象的引用都会拥有类型MovieDaoComponent with FavoritesServiceComponent。换句话说，它确保了每当某些对象或类混入MovieServiceComponentImpl时，MovieDaoComponent和FavoritesServiceComponent抑或是它们的子类也都将会混入该对象或类。

接下来是显式的val字段，我们将会把依赖的引用存储在这些字段中：

```
val favoritesService: FavoritesService
val movieDao: MovieDao
```

以上这些能确保每当我们将MovieServiceComponentImpl混入容器对象时，都需要将它们分配给抽象的val字段。

最后，来创建作为组件注册表的对象：ComponentRegistry。该注册表扩展了我们所有的依赖实现，并对它们进行了初始化，然后将它们的引用存储在前面定义的val字段中：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/di/ex1/Services.scala

```
object ComponentRegistry extends MovieServiceComponentImpl
  with FavoritesServiceComponentImpl with MovieDaoComponentImpl {
  val favoritesService = new FavoritesServiceImpl
  val movieDao = new MovieDaoImpl

  val movieService = new MovieServiceImpl
}
```

现在我们便可以在需要的时候从注册表中获取装配完整的MovieService了：

```
scala> val movieService = ComponentRegistry.movieService
movieService: ...
```

前面说过，这种装配方式可以维持静态类型的安全。让我们更详细地探讨一下这句话的含义。首先来看，如果只是在对象注册表中扩展MovieServiceComponentImpl的话会发生什么，如下面的代码所示：

```
object BrokenComponentRegistry extends MovieServiceComponentImpl {

}
```

这么做将会导致编译器错误，错误信息大致如下：

illegal inheritance; self-type com.mblinn.mbfpp.oo.di.ex1.Example. BrokenComponentRegistry. type *does not conform to com.mblinn.mbfpp.oo.di.ex1.Example.MovieServiceComponentImpl's selftype...*

在上面的信息中，编译器已经告诉我们，BrokenComponentRegistry并不符合我们为

MovieServiceComponentImpl所声明的自身类型，即我们并没有为BrokenComponentRegistry混入MovieDaoComponent和FavoritesServiceComponent。

可以通过扩展FavoritesServiceComponentImpl和MovieDaoComponentImpl来修复该错误，如在以下代码中所做的：

```
object BrokenComponentRegistry extends MovieServiceComponentImpl
        with FavoritesServiceComponentImpl with MovieDaoComponentImpl {
}
```

然而，这又会向我们提示另一个编译器错误，该错误信息以如下的提示内容开头：

```
object creation impossible, since: it has 2 unimplemented members...
```

该错误指出，我们还没有实现favoritesService和movieDao，而它们是MovieServiceComponentImpl需要我们去实现的成员。

Clojure实现

Clojure并没有与依赖注入类似的东西。相反，我们在需要的时候会将函数直接传递给其他函数。举个例子，下面声明了get-movie和get-favorite-videos两个函数：

ClojureExamples/src/mbfpp/functional/di/examples.clj

```
(defn get-movie [movie-id]
  {:id "42" :title "A Movie"})

(defn get-favorite-videos [user-id]
  [{:id "1"}])
```

下面将它们传递给函数get-favorite-decorated-videos，该函数使用了这两个函数：

ClojureExamples/src/mbfpp/functional/di/examples.clj

```
(defn get-favorite-decorated-videos [user-id get-movie get-favorite-videos]
  (for [video (get-favorite-videos user-id)]
    {:movie (get-movie (:id video))
     :video video}))
```

还有一种可能是采用函数生成器模式，该模式会将依赖的函数打包进一个闭包。

然而在Clojure中，我们通常只在希望函数的用户能掌控传入依赖时才会执行此类直接注入。我们往往不需要使用它来定义程序的整体结构。

采用Clojure和其他Lisp语言编写的程序通常会被组织成一系列分层的领域特定语言。我们将会在模式21“领域特定语言”中看到这样的一个实例。

范例代码：测试桩

虽然，依赖注入主要的关注点是程序的整体组织，但是它在测试这个特定的领域中却起着非

常重要的作用，尤其有助于为测试注入依赖的测试桩。

传统的Java实现

在Java中，我们只需采用构造器注入的方式来手动地为MovieService注入测试桩或mock对象。另一个选择是使用依赖注入容器来初始化一组测试依赖。

最好的方式还是取决于我们当前编写的测试种类。对于单元测试而言通常比较简单，只需手动注入个别mock对象即可。而对于较大型的集成类型的测试，我更偏好于选择功能齐全的容器方式。

Scala实现

通过使用Scala的Cake模式，可以很容易地创建依赖的mock版本。在如下的代码片段中便是这样做的：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/di/ex1/Services.scala

```
trait MovieDaoComponentTestImpl extends MovieDaoComponent {
  class MovieDaoTestImpl extends MovieDao {
    def getMovie(id: String): Movie = new Movie("43", "A Test Movie")
  }
}

trait FavoritesServiceComponentTestImpl extends FavoritesServiceComponent {
  class FavoritesServiceTestImpl extends FavoritesService {
    def getFavoriteVideos(id: String): Vector[Video] = Vector(new Video("2"))
  }
}
```

现在只需混入并初始化测试桩组件（而不是真实的组件），供测试的电影服务就可以使用了：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/di/ex1/Services.scala

```
object TestComponentRegistery extends MovieServiceComponentImpl
  with FavoritesServiceComponentTestImpl with MovieDaoComponentTestImpl {
  val favoritesService = new FavoritesServiceTestImpl
  val movieDao = new MovieDaoTestImpl

  val movieService = new MovieServiceImpl
}
```

Clojure实现

结合前面例子中所编写的样例Clojure代码，我们只需要创建其依赖函数的测试版本并将它们传入get-favorite-decorated-videos便完成了测试桩的注入。下面的代码片段对这一方式进行了展示：

ClojureExamples/src/mbfpp/functional/di/examples.clj

```
(defn get-test-movie [movie-id]
  {:id "43" :title "A Test Movie"})
```

```
(defn get-test-favorite-videos [user-id]
  [{:id "2"}])

=> (get-favorite-decorated-videos "2" get-test-movie get-test-favorite-videos)
({:movie {:title "A Test Movie", :id "43"}, :video {:id "2"}})
```

然而，由于我们不会总是通过“将每个依赖以高阶函数的方式传入”的方式来组织整个Clojure程序，所以通常还需要另一种方法来为测试依赖生成测试桩。来看另一个版本的get-favorite-decorated-videos函数，该函数直接依赖于它的依赖项，而不再采用将依赖作为高阶函数传入的方式：

ClojureExamples/src/mbfpp/functional/di/examples.clj

```
(defn get-favorite-decorated-videos-2 [user-id]
  (for [video (get-favorite-videos user-id)]
    {:movie (get-movie (:id video))
     :video video}))
```

如果调用get-favorite-decorated-videos-2，它将会使用硬编码的依赖：

```
=> (get-favorite-decorated-videos-2 "1")
({:movie {:title "A Movie", :id "42"}, :video {:id "1"}})
```

我们可以使用with-redefs暂时对依赖进行重新定义，如下所示：

```
=> (with-redefs
     [get-favorite-videos get-test-favorite-videos
      get-movie get-test-movie]
     (doall (get-favorite-decorated-videos-2 "2")))
({:movie {:title "A Test Movie", :id "43"}, :video {:id "2"}})
```

请注意，我们将对get-favorite-decorated-videos-2的调用封装在对doall的调用中。采用doall的方式会强制由get-favorite-decorated-videos-2产生的懒序列提前完成计算。

之所以在此处需要使用它是因为惰性（laziness）和with-redefs之间具有一种微妙的相互作用，这可能会让人感到困惑。如果不强制序列提前完成计算，那么它在REPL尝试输入它之前是不会完成完整计算的。而到那时，重新绑定的函数就已经还原为原来的函数了。

Clojure的with-redefs是一把双刃剑。你可能已经猜到了，动态替换函数定义是相当危险的，所以最好只在测试代码中使用它。

相关模式

模式16　函数生成器模式

模式21　领域特定语言

第 4 章

函数式模式

4

简介

函数式编程也拥有一套属于它自己的模式，这些模式是从函数式风格演进而来的。

这些模式很大程度上依赖不可变性。举个例子，尾递归模式（模式12）中展示了一个用于替代迭代的通用方案，该方案并不依赖于可变计数器。而操作链模式（模式15）展示了如何通过在不可变数据结构上执行链式转换从而完成对不可变数据的处理。

在这些模式中普遍存在的另一个主题是“将高阶函数作为组合的基本单元”。该主题与第一个主题十分吻合，即不可变性和对不可变数据的转换。通过使用高阶函数，我们可以很容易地完成这些转换，正如在Filter-Map-Reduce模式（模式14）中所展示的。

我们将要探索的最后一个主题是“采用函数式语言来创建用于解决特定问题域的小型语言的能力”。这一类编程的传播发展已不止于函数式风格，但它是自Clojure所传承的Lisp传统发源而来的。我们将在模式12和模式21中看到这一主题的相关内容。

首先来看第一个模式：尾递归模式。

模式 12

尾递归模式

目的

在不使用可变状态且没有栈溢出的情况下完成对某个计算的重复执行。

概述

迭代是一个需要可变状态的命令式风格的技术。举个例子，让我们来看一个简单的问题：编

写一个函数来计算从1到任意数字之间所有数字的总和（包含最后的任意数字）。代码如下所示，但是它需要i和sum两个变量都是可变的：

JavaExamples/src/main/java/com/mblinn/functional/tailrecursion/Sum.java

```
public static int sum(int upTo) {
        int sum = 0;
        for (int i = 0; i <= upTo; i++)
                sum += i;
        return sum;
}
```

由于函数式世界强调的是不可变性，所以迭代就只能出局了。我们可以使用递归来替代迭代，这是一种不需要不可变性的技术。但是递归也有它自身的问题；特别重要的一点是，每一次递归调用都会导致在程序的调用栈上增加一个新的栈帧。

为了规避该问题，可以使用递归的一种特殊形式，我们称之为“尾递归”（tail recursion）。这种方式可以对递归进行优化，从而在整个递归过程中只占用一个栈帧。该过程被称之为“尾部调用优化”（tail call optimization）或简称TCO。

考虑一下该如何将sum()编写成一个递归函数：sumRecursive()。首先，需要确定递归应该在何时结束以及何时开始。因为我们会对所有的数字求和，直到遇到某个任意的数字为止，所以我们可以从该任意数字开始递归计算，直到遇到0便结束递归。这一停止递归的位置被称为基线条件（base case）。

接下来我们需要想清楚在实际计算中应该做些什么。在这个例子中，我们会获取当前的数字，并将它与调用tailRecursive()返回的结果相加，而在每次的tailRecursive()调用前，需要将当前的数字减1之后作为入参传入该函数。最后，我们抵达了0这一基线条件，到此为止，程序的调用栈将不再继续展开。然后，每个栈帧依次返回当前部分的求和结果，直到到达栈顶，然后将最终的求和值返回。下面的代码展示了这一解决方案：

JavaExamples/src/main/java/com/mblinn/functional/tailrecursion/Sum.java

```
public static int sumRecursive(int upTo) {
        if (upTo == 0)
                return 0;
        else
                return upTo + sumRecursive(upTo - 1);
}
```

然而，这里存在着一个问题。每一次递归调用都会向调用栈添加一个栈帧，这意味着该解决方案的内存占用量是与所计算序列的规模成正比的，如图4-1所示。

显然这是不现实的，但是我们可以做得更好。而撑爆调用栈的根本原因在于，在每一次递归调用中，我们需要该次调用的结果来完成当前调用的计算。这意味着在运行时我们别无选择，只能将所有的中间结果存储在调用栈上。

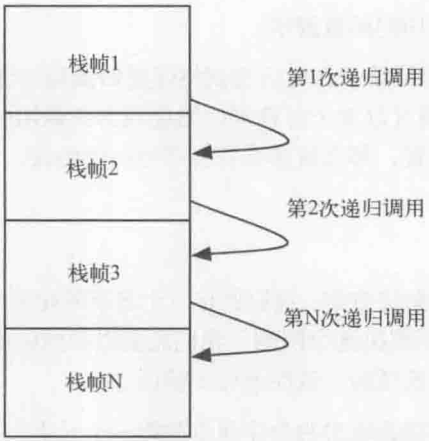


图4-1　简单的调用栈。图解正常递归调用期间的调用栈——每一次递归调用都会向调用栈新增一个栈帧；这些栈帧所代表的内存在递归完成之前是无法被回收的

如果我们可以确定递归调用是函数所有代码分支中最后发生的事情，即尾递归，那么情况将不会这么糟糕。这样做需要我们提前将中间值存储到栈上，并通过调用链来传递它们。下面的代码描述了这一方式：

JavaExamples/src/main/java/com/mblinn/functional/tailrecursion/Sum.java

```
public static int sumTailRecursive(int upTo, int currentSum) {
        if (upTo == 0)
                return currentSum;
        else
                return sumTailRecursive(upTo - 1, currentSum + upTo);
}
```

一旦我们将函数重写成尾递归的形式，就可以使用TCO来运行该函数，而本次调用只会占用一个栈帧，如图4-2所示。

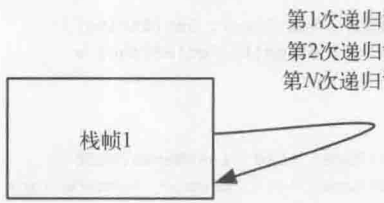


图4-2　采用了TCO的调用栈。通过TCO，在尾部的递归调用将不会产生新的栈帧。相反，每一次调用会使用现有的栈帧，并在复用该栈帧前清除其中与前一次调用相关的所有数据

遗憾的是，JVM并不能直接支持TCO，所以Scala和Clojure需要采用一些技巧来将尾递归调用编译降级为与迭代相同的字节码。在Clojure的例子中，是通过提供`loop`和`recur`这两种特殊的形

式实现的，从而取代了采用通用的函数调用。

在Scala的例子中，Scala编译器会在背后尝试将尾递归调用转译成迭代，同时Scala还提供了一个注解：@tailrec，该注解可以置于打算采用尾递归方式调用的函数上。如果该函数的递归调用并没有发生在一个尾部位置，那么编译器将会产生一个错误。

范例代码：递归的“人”

来看一个简单问题的递归解决方案。我们拥有一个名字的序列和一个姓氏的序列，然后想要将它们组合成一个“人”。为了解决这个问题，我们需要以相同的步调来遍历两个序列。假设程序的其他部分已经对两个序列长度的一致性进行了验证。

在递归的每一步中，我们将会取出两个序列中的第一个元素，并将它们组合成全名的形式。然后再将每个序列的剩余部分以及当前已经完成组合的people序列传递给下一次递归调用。来看看Scala是如何完成这一工作的。

Scala实现

首先需要采用Scala的样本类来表示我们的“人”。下面便是一个拥有姓和名的人：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/tr/Names.scala

```
case class Person(firstNames: String, lastNames: String)
```

接下来是递归函数本身的实现。实际上我们需要将该递归函数拆分成两个函数。第一个是名为makePeople的函数，它以firstNames和lastNames两个序列作为入参。而第二个则是一个帮助函数，该函数内嵌于makePeople，它比makePeople函数多了一个参数，该参数用来在递归调用中传递已经完成组合的人名列表。在我们将该函数拆分之前先来看看它的全貌：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/tr/Names.scala

```
def makePeople(firstNames: Seq[String], lastNames: Seq[String]) = {
  @tailrec
  def helper(firstNames: Seq[String], lastNames: Seq[String],
                    people: Vector[Person]): Seq[Person] =
    if (firstNames.isEmpty)
      people
    else {
      val newPerson = Person(firstNames.head, lastNames.head)
      helper(firstNames.tail, lastNames.tail, people :+ newPerson)
    }
  helper(firstNames, lastNames, Vector[Person]())
}
```

首先，来看看makePeople函数的签名：

```
def makePeople(firstNames: Seq[String], lastNames: Seq[String]) = {
      «function-body»
}
```

该签名只说明makePeople函数接受两个String类型的序列作为入参。由于我们并没有指定函数的返回类型，编译器将会从函数体中推断出返回类型。

接下来，让我们再去看看helper函数的签名。该函数负责实际的尾递归调用。函数helper由一个@tailrec注解标注，该注解会促使编译器在发现递归调用不是尾递归的情况下产生错误。该函数的签名中添加了一个额外的参数：一个名为people的向量，该参数会在递归调用中累计返回的结果。

请注意，虽然我们通常会在Scala的范例代码中省略对函数返回类型的定义，但是在该函数中指定了返回类型。这是因为编译器无法从递归调用的函数中推断出返回类型。

```
def helper(firstNames: Seq[String], lastNames: Seq[String],
                        people: Vector[Person]): Seq[Person] =
                        «function-body»
```

现在来实现helper的函数体。如果firstNames序列是空的，我们将返回已经构建好的people列表。否则，我们会从两个序列中各自取出第一个姓和第一个名，并将它们组织成一个Person。然后将两组序列剩余的部分以及添加了新“人”的列表作为入参，再次调用helper函数：

```
if (firstNames.isEmpty)
        people
else {
        val newPerson = Person(firstNames.head, lastNames.head)
        helper(firstNames.tail, lastNames.tail, people :+ newPerson)
}
```

最后，以两组名字的序列和一个空的用以存储“人”的Vector作为入参调用helper函数：

```
helper(firstNames, lastNames, Vector[Person]())
```

最后有一个关于语法的注意点：使用“方法”这一Scala面向对象的特性，可以减少由于递归函数定义而产生的一些冗长代码。上述函数倘若换成方法的签名将会如下所示[①]：

```
def makePeopleMethod(firstNames: Seq[String], lastNames: Seq[String]) = {
    «method-body»

}
```

因为本书主要专注于Scala的函数式部分，所以我们在大部分例子中所使用的都是函数而不是方法。在Scala中，方法通常可以作为高阶函数来使用，但在有些场景下却未必合适。

Clojure实现

在Clojure中，尾递归调用是无法得到优化的，因此每一次尾递归调用最终还是会消耗一个栈帧。Clojure没有提供TCO，但是它给了我们两种形式的操作：loop和recur。其中loop形式定义了一个递归的发生位置，而关键字recur会让程序跳转回该位置，并将新的值传递给它。

① 本着“减少冗长代码”的原则，此处代码略有改动。——译者注

实际上，这样做几乎跟定义个一个私有的helper函数是一样的，所以Clojure解决方案在形式上与Scala的解决方案非常相似。我们将会使用一个简单的map来存储people，这在Clojure中是一种标准的做法，代码如下所示：

ClojureExamples/src/mbfpp/functional/tr/names.clj

```
(defn make-people [first-names last-names]
  (loop [first-names first-names last-names last-names people []]
    (if (seq first-names)
      (recur
        (rest first-names)
        (rest last-names)
        (conj
          people
          {:first (first first-names) :last (first last-names)}))
      people)))
```

这段代码第一个有意思的地方在于对loop的声明。在这里，我们定义了递归的发生位置以及开始递归时所需要的值：作为入参传入的姓、名的序列以及一个在recur时用于累加“人”的空向量。

```
(loop [first-names first-names last-names last-names people []]
       «loop-body»
)
```

代码片段first-names first-names last-names last-names people []或许看上去有点搞笑，但是它完成了对loop中定义的first-names和last-names以及people向量的初始化。该片段会将first-names和last-names初始化为传入序列的值，而将people初始化为一个空的向量。

该例子大部分的代码位于if表达式中。如果名字序列还有子项，那么我们会将两个序列中各自的第一项取出来，然后用它们来创建一个用于表示“人”的map，并采用conj将该map添加到累加器people上。

一旦将新的“人”添加到people向量，我们便会使用recur跳转回采用loop定义的递归的发生位置。这与我们在Scala的样例中所采用的递归调用是一样的。

如果没有跳转回递归的发生位置，就表明我们已经遍历完整个名字的序列，接着便可以将已经构建完成的people返回。

```
(if (seq first-names)
       (recur
              (rest first-names)
              (rest last-names)
              (conj
                     people
                     {:first (first first-names) :last (first last-names)}))
       people)
```

或许对于你来说，为什么上面if表达式中的条件判断会生效并不是非常明显。这是因为对空集合的seq操作会返回nil，其求值后的结果是false。而seq对其他任意集合的操作会返回一个非空的序列。如下面的代码所示：

```
=> (seq [])
nil
=> (seq [:hi])
(:hi)
```

在Clojure中处理序列时，我们通常会采用nil作为递归的基线条件。

讨论

尾递归与迭代是等效的。事实上，Scala和Clojure的编译器会各自以它们自己的方式将尾递归降级处理成Java世界中与迭代同一类型的字节码。尾递归相对迭代的主要优势就在于它消除了语言中的可变性来源，这也正是为什么递归在函数式世界中如此流行的原因。

我个人之所以偏好尾递归胜于迭代也有两个次要的原因。首先，尾递归可以节省一个额外的索引变量。其次，尾递归使得在计算中需要操作什么数据结构以及会生成什么数据结构变得更加明确，因为它们都会作为参数在整个调用链中传递。

在一个迭代的解决方案中，如果我们尝试以一致的步调来操作两个序列并生成另一个数据结构，这些序列势必都会被混入一个原本用于处理其他逻辑的函数之中。我发现，使用尾递归相比迭代可以对组织良好的函数结构起到更好的强制作用，因为我们所操作的所有数据都必须通过调用链传递。可是，如果需要处理的数据不止那么几块的话，这将变得难以处理。

虽然尾递归与迭代是等效的，但它确实是一个相当底层的操作。还有一些相对尾递归来说更高层次且更具声明性的方式可以用于解决迭代的问题。举个例子，下面是一个用于解决person-making问题的更简短的版本，该版本利用了Clojure中的一些高阶函数：

ClojureExamples/src/mbfpp/functional/tr/names.clj

```
(defn shorter-make-people [first-names last-names]
  (for [[first last] (partition 2 (interleave first-names last-names))]
    {:first first :last last}))
```

选择哪种解决方案只是一个个人喜好问题，但是经验丰富的函数式程序员往往倾向于更简短且更具声明性的解决方案。因为这样的方案更易于阅读，他们只需扫上一眼便已了然于心。但是这些方案也有一些缺点，对于新手来说，它们并不易于理解，因为它们需要你掌握很多高阶库函数的知识。

每当我打算编写需要尾递归的解决方案时，都会先梳理一下高阶函数的API文档，或是某组高阶函数的组合，看看是否有方案可以匹配我所需要完成的工作。如果无法找到可以使用的高阶函数，或者我所需要的解决方案涉及高阶函数之间过于错综复杂的结合，那么我会返回继续使用

尾递归来解决手头的问题。

相关模式

模式5　替代迭代器模式

模式13　相互递归模式

模式14　Filter-Map-Reduce模式

模式 13

相互递归模式

目的

采用相互递归函数来表达具体的算法，包括对树形数据结构的遍历、递归下降分析和状态机操作等。

概述

在上一个模式中，我们看到了如何使用尾递归来对数据序列进行遍历。同时，由于JVM并没有提供对尾递归的直接支持，所以Clojure和Scala各自采用了一些技巧来避免递归时的栈帧消耗。

对于大多数场景来说，模式12中的尾递归已足以满足要求了，所谓尾递归就是指所有的递归调用都属于自己调用自己的自递归（self-recursive）。然而，在应对一些更加复杂的问题的解决方案中，我们需要函数能递归地调用其他函数。

举个例子，有限状态机是一种可以对多种不同类型的问题进行建模的有效方式，而相互递归更是一种用于有限状态机编程的绝佳方式。网络协议、很多物理系统（如自动售货机和电梯），以及对半结构化文本的解析都可以由状态机来完成建模和实现。

在本模式中，我们将会看到一些可以通过相互递归模式来解决的问题。由于JVM并不支持对尾递归的优化，所以Scala和Clojure需要使用一些巧妙的手段才能完成对实际相互递归的支持。这与它们在处理普通尾递归时的做法差不多，以避免将栈空间消耗殆尽。

在相互递归中，我们所采用的优化技巧被称为蹦床（trampoline）。相比于直接产生相互递归调用，蹦床会返回一个用于发起所需调用的函数，而后续的工作则留待编译器或在运行时来完成。

Scala对蹦床的支持隐藏了大量底层的细节，它为我们提供了`tailcall()`方法和`done()`方法，前者会产生相互递归的调用，而后者供我们在完成递归时调用。

这听起来很玄乎，看上去却很简单。为了证明该模式的作用，让我们在深入更多实例之前，快速地来看一个相互递归的入门示例。这是一个用来判断数字奇偶的例子，该示例所采用的方式虽颇具数学之美效率却极其低下。

下面是对该示例工作方式的描述：我们需要两个函数：isEven()和isOdd()。每个函数都以一个Long型的n作为入参。其中isEven()函数会检查n是否为0，如果是的话，它将会返回true；否则将n减1，然后调用isOdd方法。而isOdd()方法也会检查传入的n是否为0，如果是，就返回false；否则将n减1，然后继续调用isEven方法。

有代码有真相，请看下面的代码：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/mr/EvenOdd.scala

```
def isOdd(n: Long): Boolean = if (n == 0) false else isEven(n - 1)

def isEven(n: Long): Boolean = if (n == 0) true else isOdd(n - 1)

scala> isEven(0)
res0: Boolean = true

scala> isOdd(1)
res1: Boolean = true

scala> isEven(1000)
res2: Boolean = true

scala> isOdd(1001)
res3: Boolean = true
```

该方法在处理较小的数字时十分有效，但是如果我们尝试用它来处理更大的数字时会怎样呢？

```
scala> isOdd(100001)
java.lang.StackOverflowError
...
```

正如你所看到的，每一次相互的递归调用都会消耗一个栈帧，从而导致栈溢出错误。看看如何通过使用Scala的蹦床来修复该问题。

在Scala中，对蹦床的支持位于scala.util.control.TailCalls中。该类的功能分为两个部分。首先是done()函数，用来从递归调用中返回最终的结果。第二部分是tailcall()函数，它可以用来产生递归调用。

除此之外，尾递归函数的结果并不是直接返回的，而是被包装在一个TailRec类型中。为了在最后取到结果，我们可以在最后的TailRec实例上调用result()函数。

下面便是奇偶的判断代码，我们利用Scala的蹦床技巧重写了原来的实现：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/mr/EvenOdd.scala

```
def isOddTrampoline(n: Long): TailRec[Boolean] =
  if (n == 0) done(false) else tailcall(isEvenTrampoline(n - 1))

def isEvenTrampoline(n: Long): TailRec[Boolean] =
  if (n == 0) done(true) else tailcall(isOddTrampoline(n - 1))

scala> isEvenTrampoline(0).result
res0: Boolean = true

scala> isEvenTrampoline(1).result
res1: Boolean = false

scala> isEvenTrampoline(1000).result
res2: Boolean = true

scala> isEvenTrampoline(1001).result
res3: Boolean = false
```

尝试运行它：

```
scala> isOddTrampoline(100001).result
res4: Boolean = true
```

这一次，在处理大型数字的时候没有再出现栈溢出的错误。但是如果你尝试的是一个足够大的数字，就会需要等待一段很长的时间，因为这一算法的时间复杂度是与数字大小成正比的。

别名

间接递归（Indirect Recursion）

范例代码：物质的状态

在本实例中，我们将会使用相互递归模式来构建一个简单的状态机，该状态机接受一组转换的序列，该序列是一个在物质的不同状态之间进行的转换过程的序列，其中包括`liquid`（液态）、`solid`（固态）、`vapor`（蒸汽态）和`plasma`（等离子态）。同时，状态机会对序列的有效性进行验证。举个例子，物质的转换可以从固态变为液态，但是无法从固态直接变为等离子态。

状态机中的每一个状态都由函数表示，而转换则由一组转换过程的名字序列表示，比如`condensation`（凝结）和`vaporization`（汽化）。状态函数会从序列中取出第一个转换过程，然后判断它是否有效，如果有效，则调用可以将它转换到对应状态的函数，并将该转换过程和剩余的转换过程序列传递给该函数。如果转换是无效的，我们将停止运行并返回`false`。

举个例子，如果我们目前处于`solid`（固态）状态，并看到接下去的转换是`melting`（熔化），那么我们将会调用`liquid`（液态）函数。如果接下去的转换是`condensation`（凝结），那么它对固态来说并不是一个有效的转换，因此我们会立即返回`false`。

在查看代码之前，先来看一幅描述物质状态的状态机简图。

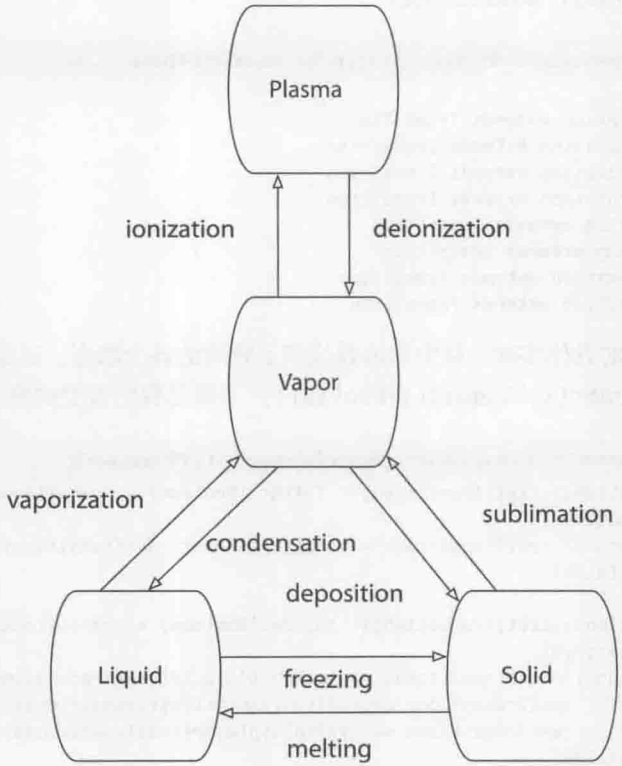


图4-3　物质的状态。物质的各个状态以及各状态之间的转换过程

图中的节点代表了各个函数，我们在使用相互递归模式的时候需要这些函数来对状态机建模，而图中的各条边代表了这些函数之间可以执行的转换。让我们来看看代码，先从Scala的实现开始。

Scala实现

我们的Scala解决方案依赖于四个函数，物质的每个状态各对应一个函数，它们分别是：`plasma()`、`vapor()`、`liquid()`和`solid()`。除此之外，还需要一系列样本对象来表示所有的转换过程，例如：`Ionization`、`Deinonizaton`和`Vaporization`等。

每个函数都接受一个单独的参数，即一个转换的列表，它们都采用Scala的模式匹配来解构入参。如果传入的列表是`Nil`，就表明转换已经成功结束，我们便可以调用`done()`来结束递归，同时传入`true`。

否则，我们会检查列表中的第一个转换过程是否有效。如果有效，就会转换到其指定的状态，并将剩下的转换过程传递下去。如果第一个转换是无效的，我们将调用`done()`函数，同时传入`false`。

让我们来看看代码，从代表转换的样本对象开始。它们相当简单，每个转换都是一个独立的对象，而且它们都继承自Transition类。

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/mr/Phases.scala

```
class Transition
case object Ionization extends Transition
case object Deionization extends Transition
case object Vaporization extends Transition
case object Condensation extends Transition
case object Freezing extends Transition
case object Melting extends Transition
case object Sublimation extends Transition
case object Deposition extends Transition
```

现在来看该实例的具体实现，其中的函数代表了物质的各个状态。正如我们约定的，它们分别是：plasma()、vapor()、liquid()和solid()。下面是我们需要的所有函数集：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/mr/Phases.scala

```
def plasma(transitions: List[Transition]): TailRec[Boolean] = transitions match {
  case Nil => done(true)
  case Deionization :: restTransitions => tailcall(vapor(restTransitions))
  case _ => done(false)
}
def vapor(transitions: List[Transition]): TailRec[Boolean] = transitions match {
  case Nil => done(true)
  case Condensation :: restTransitions => tailcall(liquid(restTransitions))
  case Deposition :: restTransitions => tailcall(solid(restTransitions))
  case Ionization :: restTransitions => tailcall(plasma(restTransitions))
  case _ => done(false)
}

def liquid(transitions: List[Transition]): TailRec[Boolean] = transitions match {
  case Nil => done(true)
  case Vaporization :: restTransitions => tailcall(vapor(restTransitions))
  case Freezing :: restTransitions => tailcall(solid(restTransitions))
  case _ => done(false)
}

def solid(transitions: List[Transition]): TailRec[Boolean] = transitions match {
  case Nil => done(true)
  case Melting :: restTransitions => tailcall(liquid(restTransitions))
  case Sublimation :: restTransitions => tailcall(vapor(restTransitions))
  case _ => done(false)
}
```

我们已经在一个较高的层面对这些函数的工作原理进行过描述，所以让我们从中挑选一个函数：vapor()，来详细了解它的工作方式，从它的签名开始：

```
def vapor(transitions: List[Transition]): TailRec[Boolean] = transitions match {
      «function-body»
}
```

正如我们所看到的，它接受一个名为transitions的Transition列表，然后采用模式匹配来对它进行解构。该函数并没有直接返回一个Boolean类型的值，它返回的是一个Boolean的TailRec实例，这样才能让我们充分利用Scala对蹦床技巧的支持。

再去看看match表达式中的第一个case子句，我们看到：该子句会在列表为空或Nil的时候调用done来返回true。这便是递归的基线条件；如果程序执行到这里，就说明我们已经成功执行了最初序列中的所有转换。

```
case Nil => done(true)
```

接下来是三个位于中间的子句。它们会在序列的首个元素有效的情况下使用模式匹配来获取该元素，并调用函数来转换到合适的状态，同时将剩下的转换过程作为入参传入该函数。

```
case Condensation :: restTransitions => tailcall(liquid(restTransitions))
case Deposition :: restTransitions => tailcall(solid(restTransitions))
case Ionization :: restTransitions => tailcall(plasma(restTransitions))
```

最后来看看最后一个子句，我们用它来处理所有其他的情况。如果程序不幸走到这里，那就说明我们未能处理完所有的转换，也就是说刚才所看到的转换并不是有效的，因此我们会调用done()，同时传入false。

```
case _ => done(false)
```

让我们通过实战来看看这个例子，首先从solid（固态）开始，同时传入一个有效的转换列表：

```
scala>   val validSequence = List(Melting, Vaporization, Ionization, Deionization)
validSequence: List[com.mblinn.mbfpp.functional.mr.Phases.Transition] =
        List(Melting, Vaporization, Ionization, Deionization)

scala> solid(validSequence).result
res0: Boolean = true
```

接下来再从liquid（液态）开始，同时传入一个无效的转换列表：

```
scala>   val invalidSequence = List(Vaporization, Freezing)
invalidSequence: List[com.mblinn.mbfpp.functional.mr.Phases.Transition] =
        List(Vaporization, Freezing)

scala> liquid(invalidSequence).result
res1: Boolean = false
```

到此我们结束了Scala中相互递归模式的演示，让我们去看看该实例在Clojure中是如何实现的。

Clojure实现

Clojure的代码与Scala非常相似，至少都是基于一个较高的层次来编写的。我们拥有plasma、vapor、liquid和solid函数，每个函数都接受一个转换过程的序列。

我们通过使用Clojure的解构从序列中抽取出当前的转换过程，并将它绑定到transition，同时将剩余的转换过程绑定到rest- transitions。

如果transition是nil，表明我们已经成功完成所有的转换，那么就应该返回true。否则检查当前的转换是否是一个有效的转换，如果是，就转换到合适的状态。如果不是，就返回false。下面是完整的代码：

ClojureExamples/src/mbfpp/functional/mr/phases.clj

```
(declare plasma vapor liquid solid)

(defn plasma [[transition & rest-transitions]]
  #(case transition
     nil true
     :deionization (vapor rest-transitions)
     :false))

(defn vapor [[transition & rest-transitions]]
  #(case transition
     nil true
     :condensation (liquid rest-transitions)
     :deposition (solid rest-transitions)
     :ionization (plasma rest-transitions)
     false))

(defn liquid [[transition & rest-transitions]]
  #(case transition
     nil true
     :vaporization (vapor rest-transitions)
     :freezing (solid rest-transitions)
     false))

(defn solid [[transition & rest-transitions]]
  #(case transition
     nil true
     :melting (liquid rest-transitions)
     :sublimation (vapor rest-transitions)
     false))
```

请注意，在上面的代码中为什么没有出现类似Scala版本中的done或tailcall呢？那是因为我们并没有使用tailcall，而只是返回了用于产生所需调用的函数。在这个例子中，我们使用了Clojure对匿名函数的简写方式来完成了同样的工作。

当我们真的想要启动相互递归的调用链时，需要将想要调用的函数及其参数传递给trampoline：

```
=> (def valid-sequence [:melting :vaporization :ionization :deionization])
#'mbfpp.functional.mr.phases/valid-sequence
=> (trampoline solid valid-sequence)
true
=> (def invalid-sequence [:vaporization :freezing])
```

```
#'mbfpp.functional.mr.phases/invalid-sequence
=> (trampoline liquid invalid-sequence)
false
```

正如我们所预期的，对于有效的序列，上述的代码返回了true；而对于无效的序列，则返回了false。

在结束该例子前，我想再聊聊Scala中的Nil和Clojure中的nil。我们所编写的代码看上去非常相似，但是它们之间却有一个微妙的区别值得一提。

在Scala中，Nil是空列表的同义词，我们可以在Scala的REPL中得到验证：

```
scala> Nil
res0: scala.collection.immutable.Nil.type = List()
```

在Clojure中，nil意味着“什么都没有”：它的意思是没有值，这与空列表是有区别的。多年来，各种各样的函数式语言对nil的处理都各有不同，所以当你接触到一门新的函数式语言时，建议你花一些时间来理解nil在该语言中的意义。

讨论

相互递归可以给你带来非常大的便利性，但是它的用武之地通常仅限于一些特定的场景。而状态机就是这样的一个场景；状态机的确是非常有用的克敌利器。然而，大部分开发者对它们的印象还停留在本科的计算机科学课程中，当时的他们不得不对有限状态机和正则表达式之间的等价性进行证明，这些内容虽然很有趣却无法给大部分的开发者带来任何实用的价值。

尽管如此，状态机在近些年却又开始变得流行起来，它们是actor模型的重要组成部分。actor模型是一个为并发和分布式编程而服务的模型，Scala的Akka框架和另一门函数式语言Erlang都在使用actor模型。Ruby也有一个用于创建状态机的巧妙的gem，这个gem有着一个名副其实的名字：state_machine。

另一点值得我们注意的是在Scala和Clojure中都看到过的trampoline，它只是用来实现相互递归模式的一种方式。之所以需要它是因为JVM并不直接支持对尾递归调用的优化。

相关模式

模式5　替代迭代器模式

模式14　Filter-Map-Reduce模式

模式 14

Filter-Map-Reduce 模式

目的

采用filter、map和reduce函数，以声明性的方式来操作某个序列（列表、向量等），并最终产生一个新的序列——这是一种可在较高层次完成多种序列操作的强大方式。如果不采用这种方式，那么你需要编写相当冗长的代码来完成同样的工作。

概述

在过程式语言中，我们用于操作序列的方式通常更接近于计算机工作的方式，而不是人类思考的方式。迭代相比可怕的goto语句虽有所改善，但是它的设计更偏重于代码是否能更容易地被翻译成机器码，而不是其易用性。

在过程式语言中，如果我们想要对序列进行重排序或修改序列的元素，就必须编写代码来对序列进行迭代，并逐个地遍历这些元素。而Filter-Map-Reduce模式为我们完成各种序列操作带来了更具声明性的方式，从而无需再编写如此冗长的代码。我们可以通过使用filter函数在一个较高的层次筛选出关心的元素，接着用map对每个元素进行转换，最后用reduce（通常也可以使用fold）将结果合并起来。

Filter-Map-Reduce模式可以将大部分面向对象程序员所使用的迭代算法替换成声明性的代码。

Filter-Map-Reduce模式相比于迭代的主要优势在于代码的清晰度。采用Filter-Map-Reduce模式所编写的代码量相比等效的迭代方式的代码量有着很大的精简。对于经验丰富的从业者来说，Filter-Map-Reduce模式编写的代码就好比普通的行文，看一眼便能了然于心。而通常迭代的解决方案至少需要一轮的循环以及一次条件的判断。

该模式有一个缺点，即并非所有的迭代都可由Filter-Map-Reduce来替代。另一个问题在于，有些时候，如何来创建可执行Filter-Map-Reduce的序列会显得比较困难或者不是很清晰。针对这些情况，“相关模式”列表中的某个模式或许会更合适。

范例代码：折扣计算

Filter-Map-Reduce模式的实现结合了filter、map和reduce，但并不总是按这一顺序来执行。让我们来看一个实例，该实例用于计算一个价格序列的总折扣，计算方式如下：任何达到或超过

20美金的价格将按照10%的折扣计算，而低于20美金的价格不打折扣。

Scala实现

Scala中的Filter-Map-Reduce实现与Clojure非常相似。我们从一个filter函数看起，该函数用于筛选出大于20美金的价格：

```
scala> Vector(20.0, 4.5, 50.0, 15.75, 30.0, 3.5) filter (price => price >= 20)
res0: scala.collection.immutable.Vector[Double] = Vector(20.0, 50.0, 30.0)
```

我们使用map来计算这些价格的10%：

```
scala> Vector(20.0, 50.0, 30.0) map (price => price * 0.10)
res1: scala.collection.immutable.Vector[Double] = Vector(2.0, 5.0, 3.0)
```

接着，使用reduce来计算这些折扣的总和：

```
scala> Vector(2.0, 5.0, 3.0) reduce ((total, price) => total + price)
res2: Double = 10.0
```

将它们整合在一起便有了calculateDiscount函数：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/mfr/Discount.scala

```
def calculateDiscount(prices : Seq[Double]) : Double = {
  prices filter(price => price >= 20.0) map
      (price => price * 0.10) reduce
              ((total, price) => total + price)
}

scala> calculateDiscount(Vector(20.0, 4.5, 50.0, 15.75, 30.0, 3.5))
res1: Double = 10.0
```

虽然我更偏爱上面匿名函数的版本。但是如果你喜欢，也可以使用具名函数：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/mfr/Discount.scala

```
def calculateDiscountNamedFn(prices : Seq[Double]) : Double = {
  def isGreaterThan20(price : Double) = price >= 20.0
  def tenPercent(price : Double) = price * 0.10
  def sumPrices(total: Double, price : Double) = total + price

  prices filter isGreaterThan20 map tenPercent reduce sumPrices
}
```

Clojure实现

让我们来创建一个函数：calculate-discount。该函数使用了Filter-Map-Reduce模式来计算总折扣。出于对实例需要的考虑，我们将使用一个浮点数的向量来表示价格。首先，对价格进行filter，以便只留下大于等于20美金的价格，如下所示：

```
=> (filter (fn [price] (>= price 20)) [20.0 4.5 50.0 15.75 30.0 3.50])
(20.0 50.0 30.0)
```

然后，通过map来为每个过滤后的价格乘以0.10，从而获得每个价格的折扣：

```
=> (map (fn [price] (* price 0.10)) [20.0 50.0 30.0])
(2.0 5.0 3.0)
```

最后，采用reduce来将所有的结果合并求和：

```
=> (reduce + [2.0 5.0 3.0])
10.0
```

将以上内容整合后，我们便得到了calculate-discount函数：

ClojureExamples/src/mbfpp/functional/mfr/discount.clj

```
(defn calculate-discount [prices]
  (reduce +
          (map (fn [price] (* price 0.10))
               (filter (fn [price] (>= price 20.0)) prices))))
```

阅读代码时有一个技巧，这个技巧可以让经验丰富的Lisp爱好者只需一瞥就能了然于心，也可以让没有掌握这一技巧的新手颇具挫败感。为了能更容易地阅读Lisp代码，你需要由内而外逐层阅读上面的代码。即从filter函数开始阅读，然后是map，最后查看reduce。

如果以行文的方式来描述这一过程，就是“先对价格进行过滤，保留那些大于20美金的价格，然后为所剩的价格乘以10%，并将它们相加到一起。”通过稍加实践，你就会发现阅读这样的代码不仅更加自然，而且因为这些代码是基于一个较高的层次编写的，且更接近于自然语言，所以它比等效的迭代方案的代码阅读起来更加快捷。

可以通过为map和filter函数命名从而对这一模式的实现稍加修改，如下面的代码所示：

ClojureExamples/src/mbfpp/functional/mfr/discount.clj

```
(defn calculate-discount-namedfn [prices]
  (letfn [(twenty-or-greater? [price] (>= price 20.0))
          (ten-percent [price] (* price 0.10))]
         (reduce + 0.0 (map ten-percent (filter twenty-or-greater? prices)))))
```

通过牺牲一些额外代码编写的工作量，从而让模式的实现读上去更接近于普通的行文。如果map和filter函数像上述实例中那样只使用一次，我将会更偏向于原来的匿名函数版本，但是这两种风格都很常见。具体使用哪一种取决于你的品味。

讨论

Filter-Map-Reduce模式依赖于声明性的数据操作，这种方式的抽象层次较迭代方案更高，而且通常比显式的递归方式也更高。做一个类比，这跟“采用SQL来将关系型数据库中的数据生成报表”与“对拥有相同数据的普通文件进行逐行迭代”之间的差别非常相似。通常编写良好的SQL版本会更加简短和清晰，因为它所使用的语言是专门为操作数据而创建的。采用Map-Reduce-Filter赋予了我们大量声明性的力量。

另一个需要注意的点是，该如何自下而上的构建我们的解决方案，即从在REPL中创建map、reduce和filter函数开始，然后将它们结合起来。这种自下而上的工作流在函数式编程中是极为普遍的。在REPL中不断实验，并通过探索将程序组合起来的能力是非常强大的，我们将会在函数式模式中看到更多这样的例子。

相关模式

模式5　替代迭代器模式

模式12　尾递归模式

模式13　相互尾递归模式

模式 15
操作链模式

目的

将一个计算序列串联成链式调用，让我们在无需存储大量临时结果的情况下，得以干净利落地处理不可变数据。

概述

让某些数据贯穿一系列操作是一项非常有用的技术，尤其是在处理不可变数据的时候。鉴于不能修改不可变数据的结构，当我们需要对不可变数据执行不止一次的改变时，会选择让该数据贯穿一系列的转换操作。

选择将操作串连起来的另一个原因是这样做可以让代码变得更加简洁。举个例子，我们在模式4中看到过通过将操作串连起来从而保持代码精益的做法，如下面的片段所示：

JavaExamples/src/main/java/com/mblinn/oo/javabean/PersonHarness.java

```
ImmutablePerson.Builder b = ImmutablePerson.newBuilder();
ImmutablePerson p = b.firstName("Peter").lastName("Jones").build();
```

有些时候，我们会将方法调用串连起来，从而避免由创建烦琐的临时值所带来的困扰。在下面的代码中，我们从一个List中获取一个String值，同时将它转型成了大写的形式：

JavaExamples/src/main/java/com/mblinn/mbfpp/functional/coo/Examples.java

```
List<String> names = new ArrayList<String>();
names.add("Michael Bevilacqua Linn");
names.get(0).toUpperCase();
```

这种编程风格在函数式世界中会显得越发强大，因为我们拥有高阶函数。例如，下面的Scala代码片段为某个名字创建了全体首字母的缩写：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/coo/Examples.scala

```
val name = "michael bevilacqua linn"
val initials = name.split(" ") map (_.toUpperCase) map (_.charAt(0)) mkString
```

上述代码通过调用split()将名字转化成了字符串数组，然后在该数组上映射函数，从而将所有字符串转换成了大写的形式，接着从每个数组元素的字符串中摘取出第一个字母。最后，我们将首字母组成的数组重新拼成了一个字符串。

这种形式的代码简洁且颇具声明性，因此读起来非常友好。

范例代码：函数调用链

让我们来看一个例子，该例子包含了多个链式的函数调用。我们的目标是编写出符合以下条件的代码——可以在阅读代码的同时很容易地对数据流进行逐步跟踪。

我们将采用一个存放视频的向量来表示某人的视频播放历史，然后计算出这个人花费在观看与猫相关的视频上的时间总和。为此，我们需要从向量中挑选出仅仅与猫相关的视频，然后获得这些视频的片长，最后对它们进行求和。

Scala实现

在Scala的解决方案中，我们将会以一个样本类来代表视频，该视频具有标题、视频类型以及片长三个属性。定义该样本类的代码如下所示，同时我们还生成了一些测试数据：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/coo/Examples.scala

```
case class Video(title: String, video_type: String, length: Int)

val v1 = Video("Pianocat Plays Carnegie Hall", "cat", 300)
val v2 = Video("Paint Drying", "home-improvement", 600)
val v3 = Video("Fuzzy McMittens Live At The Apollo", "cat", 200)

val videos = Vector(v1, v2, v3)
```

为了计算花费在观看与猫相关的视频上的时间总和，我们过滤出video_type等于"cat"的视频，然后从剩下的视频中提取出length字段，并将这些片长相加。完成该功能的代码如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/coo/Examples.scala

```
def catTime(videos: Vector[Video]) =
  videos.
  filter((video) => video.video_type == "cat").
  map((video) => video.length).
  sum
```

现在我们便可以在测试数据上应用catTime()函数了，并获得花费在观看猫的视频上的时间总和：

```
scala> catTime(videos)
res0: Int = 500
```

该解决方案的代码从上至下阅读起来非常顺畅，就好比阅读普通的行文一般。它在完成需求的同时无需额外的变量与任何变化，这便是函数式世界中的理想状态。

Clojure实现

让我们来看看Clojure对于该“猫视频浏览”问题的解决方案。我们将使用map来表示视频。在下面的代码片段中创建了一些测试数据：

ClojureExamples/src/mbfpp/functional/coo/examples.clj

```
(def v1
  {:title "Pianocat Plays Carnegie Hall"
   :type :cat
   :length 300})

(def v2
  {:title "Paint Drying"
   :type :home-improvement
   :length 600})

(def v3
  {:title "Fuzzy McMittens Live At The Apollo"
   :type :cat
   :length 200})

(def videos [v1 v2 v3])
```

我们尝试在Clojure中编写cat-time函数。和之前一样，我们将从向量中过滤出视频，并提取出它们的片长。为了对片长的序列进行求和，我们将使用apply和+函数。该方案的代码如下所示：

ClojureExamples/src/mbfpp/functional/coo/examples.clj

```
(defn cat-time [videos]
  (apply +
         (map :length
              (filter (fn [video] (= :cat (:type video))) videos))))
```

为了理解这段代码，我们将从filter函数读起，接着是map，然后才是apply。对于很长的操作序列来说，这样做会比较棘手。一个方案是使用let来为中间结果命名，让代码更易于理解。

另一个方案是使用Clojure的->和->>宏。这些宏可用于让某份数据在一系列函数之间串连调用。

宏->可以让某个表达式在一组形式（form）之间串连执行，即每一步产生的表达式的结果会插入到下一个形式的第二个操作数的位置上。例如，在下面的代码中，我们使用->让一个整数经历了两次减法：

```
=> (-> 4 (- 2) (- 2))
0
```

宏->首先将4穿插到(- 2)中的第二个位置，即 (- 4 2) ，从而构成了4减去2的计算，并得到了结果2。接着，该结果又穿插到后面的(- 2)的第二个空位，即(- 2 2)，从而计算得到最终的值0。

如果我们使用->>，那么将会得到一个不同的结果，如下面的代码所示：

```
=> (->> 4 (- 2) (- 2))
4
```

此处，->>将4穿插到第一个(- 2)的最后一个空位，即(- 2 4)，所以构成计算2减去4，得到结果–2。接着该结果又被穿插到第二个(-2)的最后一个位置，即(- 2 -2)，从而得到最后的结果4。

现在我们已经了解了串行操作符，通过使用->>，我们可以自上而下地来阅读catTime函数。在下面的代码片段中，我们完成了上述工作：

ClojureExamples/src/mbfpp/functional/coo/examples.clj

```
(defn more-cat-time [videos]
  (->> videos
    (filter (fn [video] (= :cat (:type video))))
    (map :length)
    (apply +)))
```

该函数的运行结果与原来的函数保持一致：

```
=> (more-cat-time videos)
500
```

该“串行宏”存在一个限制，如果我们想要使用它们来串接函数调用，那么传入函数调用链中的那一份数据必须一直作为函数的第一个入参或最后一个入参。

范例代码：采用序列推导来完成链式操作

操作链模式有一个常见的用途，即对某些容器类型内的值执行多次操作，在像Scala这样的静态语言中尤其常见。

举个例子，有一系列Option值需要整合成一个单独的值，但如果这些值中有任意一个是None的话，就会返回None。有很多方式可以用来完成这一工作，但是最简洁的一种方式依赖于使用for推导表达式来取出值并生成结果。

Scala实现

我们在模式5中遇到过序列推导，当时我们使用了该技术来作为迭代器模式的替代方案。在这里，我们可以利用它来一次操作多个序列，这对实现操作链模式非常有用。

来看一个在两个向量上执行的序列推导，其中每个向量都拥有一个整数。我们将会通过序列推导来对两个向量中的值进行相加。下面的代码对测试向量进行了定义：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/coo/Examples.scala

```
val vec1 = Vector(42)
val vec2 = Vector(8)
```

下面是我们所使用的for推导表达式。从第一个向量中取出i1，从第二个向量中取出i2，然后使用yield对它们求和：

```
scala> for { i1 <- vec1; i2 <- vec2 } yield(i1 + i2)
res0: scala.collection.immutable.Vector[Int] = Vector(50)
```

上面的例子距离for推导表达式来配合Option的使用已经非常接近了。在下面的代码中，我们定义了一对可选值：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/coo/Examples.scala

```
val o1 = Some(42)
val o2 = Some(8)
```

现在可以像之前那样将它们从向量中取出来相加了。

```
scala> for { v1 <- o1; v2 <- o2 } yield(v1 + v2)
res1: Option[Int] = Some(50)
```

这样做的好处是无需再调用get()或使用模式匹配来从Option中取出所需的值。当我们向Option中混入None的时候，这种方式的威力将会愈加明显：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/coo/Examples.scala

```
val o3: Option[Int] = None

scala> for { v1 <- o1; v3 <- o3 } yield(v1 + v3)
res2: Option[Int] = None
```

现在，for推导表达式产生了一个None。

操作链中的每一个环节都可能产生None，这在Scala中是很常见的。让我们来看一个例子，在该例子中，我们通过一系列的操作获得了某个用户在某个电影网站上的收藏视频列表。

为了获得该视频列表，首先需要通过ID来查找该用户，然后通过该用户来查找他收藏的电影。最后，我们需要找到与该电影相关的视频列表，例如演员专访、片花，甚至该电影的完整视频。

我们将从创建一对用于表示用户和电影的类开始：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/coo/Examples.scala

```
case class User(name: String, id: String)
case class Movie(name: String, id: String)
```

现在定义一组方法来分别获取用户、收藏的电影以及该电影的视频列表。每个函数都会在无法找到对应输入的响应时返回None。针对本实例，我们将会通过硬编码的值来完成上述工作，而在实际操作中，通常会由某个数据库或服务来配合完成这些工作：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/coo/Examples.scala

```
def getUserById(id: String) = id match {
  case "1" => Some(User("Mike", "1"))
  case _ => None
}

def getFavoriteMovieForUser(user: User) = user match {
  case User(_, "1") => Some(Movie("Gigli", "101"))
  case _ => None
}

def getVideosForMovie(movie: Movie) = movie match {
  case Movie(_, "101") =>
    Some(Vector(
        Video("Interview With Cast", "interview", 480),
        Video("Gigli", "feature", 7260)))
  case _ => None
}
```

现在可以编写函数来获取用户的收藏视频了。在该函数中，我们会在for语句中串连起对刚才定义的函数的调用：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/coo/Examples.scala

```
def getFavoriteVideos(userId: String) =
  for {
    user <- getUserById(userId)
    favoriteMovie <- getFavoriteMovieForUser(user)
    favoriteVideos <- getVideosForMovie(favoriteMovie)
  } yield favoriteVideos
```

如果以一个正确的用户ID来调用getFavoriteVideos()，它将会返回一个收藏视频的列表。

```
scala> getFavoriteVideos("1")
res3: Option[scala.collection.immutable.Vector[...] =
      Some(Vector(Video(Interview With Cast,interview,480),
                  Video(Gigli,feature,7260)))
```

如果以一个并不存在的用户来调用该函数，整个调用链将会返回None：

```
scala> getFavoriteVideos("42")
res4: Option[scala.collection.immutable.Vector[...]] = None
```

Clojure实现

由于Clojure并不是静态类型的语言，所以在它语言的核心部分中并没有像Scala中的Option这样的东西。

然而，Clojure的序列推导的工作方式在处理其他容器类型方面与Scala非常相似。举个例子，我们可以使用for从容器中取出内容，并将它们相加，正如我们在Scala的实例中所做的。下面的代码片段完成了这一工作：

ClojureExamples/src/mbfpp/functional/coo/examples.clj

```
(def v1 [42])
(def v2 [8])

=> (for [i1 v1 i2 v2] (+ i1 i2))
(50)
```

如果向量中有一个是空的，那么for推导表达式将会产生一个空的序列。如下面的代码所示：

ClojureExamples/src/mbfpp/functional/coo/examples.clj

```
(def v3 [])

=> (for [i1 v1 i3 v3] (+ i1 i3))
()
```

尽管Clojure的序列推导与Scala的工作方式非常相似，但是静态类型和Option类型的缺失意味着我们在Scala中见过的链式操作并不符合Clojure的语言习惯。所以我们通常会依赖于对函数的链式调用，并辅以显式的空检查。

Lisp的灵活性让我们甚至可以以代码库的方式向语言添加类似静态类型检查器这样的基本语言构造。core.typed库[①]中尚在开发阶段的代码库就是这样的一个例子，它为我们提供了可选的静态类型。

随着该代码库不断发展成熟，我们在Scala实例中所见过的链式调用类型将会在Clojure中变得愈发常见。

讨论

我们在“范例代码：采用序列推导来完成链式操作”中所看到的内容属于序列或列表单子的范例。虽然我们并没有对单子进行精确的定义，却展示了一个它们可以解决的问题类型的基本例子。它们让我们在对容器内的数据进行操作时可以很自然地将实施在容器类型上的操作串接起来。

① https://github.com/clojure/core.typed

在编程世界中，单子是我们用来将IO和其他非纯特性添加到一门纯函数式语言中的最常见的方式。不过仅从上面的例子来看，对于单子如何在一门纯函数式语言中处理IO或许并不显而易见。

由于Scala和Clojure并不以这种方式来使用单子，所以此处将不作深入讨论。然而，通常使用单子的原因是单子容器类型可以携带一些额外的信息来贯穿整个调用链。举个例子，一个处理IO的单子需要在贯穿整个操作链的过程中搜集所有的IO，并在操作链完成后将搜集到的IO提交给运行时。而运行时将会负责执行这些IO。

这一编程风格由Haskell首创。感兴趣的读者可以在《Haskell趣学指南》[Lip11]一书中找到对这一风格的精彩介绍。

延展阅读

《Haskell趣学指南》[Lip11]

相关模式

模式4　替代生成器模式来获得不可变对象

模式5　替代迭代器模式

模式14　Filter-Map-Reduce模式

模式16　函数生成器模式

模式16

函数生成器模式

目的

创建可以生成函数的函数，从而让我们可以动态地合成行为。

概述

有的时候，我们手头拥有一个可执行某个有用行为的函数，但实际上所需要的却是另一个可执行其他相关行为的函数。举个例子，我们可能拥有一个叫作`vowel?`的判断函数，该函数会在你传入元音字母的时候返回`true`，而我们实际需要的却是一个能够判断辅音字母的`consonant?`函数。

还有一些时候，我们手头拥有一些数据，并需要将其转换成某个行为。比如，我们拥有一个折扣率，但需要一个函数来为商品集计算折扣。

通过函数生成器，可以编写一个以现有数据或函数（正如我们所见，函数和数据之间的区别并不明显）作为入参的函数，并使用它来创建一个新的函数。

为了使用函数生成器模式，我们需要编写一个可以返回函数的高阶函数。函数生成器模式的实现对我们已经见过的一些模式进行了封装。

举个例子，为了通过vowel?判断函数来创建一个consonant?判断函数，我们会创建一个新的函数，该函数会调用vowel?并对返回的结果取反计算。为了通过even?函数来创建odd?函数，我们会创建一个函数来调用even?，并对返回结果取反。为了通过alive?函数来创建dead?函数，我们会创建一个函数来调用dead?，并对返回的结果取反。

在以上的内容中可以看到一个很明显的模式。我们会通过函数生成器来对该模式进行封装，并命名为negate。函数negate以一个函数为入参，并会返回一个新的函数，该函数会调用传入的函数，同时对返回的结果取反。

函数生成器模式的另一个常见用途是，将静态数据作为某个行为的基础。举个例子，我们可以通过编写一个函数来将一个静态的百分比转换成一个计算百分比的函数，该函数以百分比作为入参，返回的是一个拥有一个入参的函数。返回的函数接受一个用以计算百分比的数字，并将该数字存储在它的闭包之中，最后会使用它来完成百分比计算。

稍后，我们将会看到函数生成器模式在以上两个用途方面的多个实例。

范例代码：通过静态数据生成函数

函数生成器模式的使用方式之一是通过静态数据来产生函数。这种方式允许我们将数据，即名词，转换成行为，即动词。来看两个例子，首先是一个以百分比作为入参的函数，它会创建一个函数，新创建的函数可以基于这些百分比计算打折后的价格。

折扣计算器生成器

函数生成器discount()以一个0~100之间的百分比作为入参，同时会返回一个可基于该百分比计算折后价格的函数。如果传入50作为入参，那么会返回一个以50%进行折扣计算的函数，如果以25作为入参，将会返回一个以25%进行折扣计算的函数，以此类推。来看看Scala的实现版本。

- Scala实现

我们的Scala代码定义了一个discount()函数，它接受一个名为percent的Double类型的入参，然后检查该参数是否在0到100之间。接着，它创建了一个函数，该函数使用了percent来计算折扣。代码如下：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/fb/DiscountBuilder.scala

```
def discount(percent: Double) = {
  if(percent < 0.0 || percent > 100.0)
    throw new IllegalArgumentException("Discounts must be between 0.0 and 100.0.")
  (originalPrice: Double) =>
    originalPrice - (originalPrice * percent * 0.01)
}
```

来看看它是如何工作的。最简单的方式是采用discount()[①]来创建一个可以直接调用的匿名函数。如下所示，我们使用它来计算200元的50%折扣价：

```
scala> discount(50)(200)
res0: Double = 100.0
```

接下来使用它来分别计算200元的0%折扣价（全价）和100%折扣价（全免）：

```
scala> discount(0)(200)
res1: Double = 200.0

scala> discount(100)(200)
res2: Double = 0.0
```

如果需要不止一次地使用折扣函数，我们可以为该折扣函数命名。下面的例子便是按这种方式进行的，并对两个商品向量的折后总价进行了计算：

```
scala> val twentyFivePercentOff = discountedPrice (25)
twentyFivePercentOff: Double => Double = <function1>

scala> Vector(100.0, 25.0, 50.0, 25.0) map twentyFivePercentOff sum
res3: Double = 150.0

scala> Vector(75.0, 25.0) map twentyFivePercentOff sum
res4: Double = 75.0
```

- Clojure实现

在Clojure中，该实例的工作方式与Scala非常类似。唯一有趣的差别在于我们可以使用Clojure的先决条件（precondition）来确保折扣处于一个有效的范围之中。让我们来看看代码：

ClojureExamples/src/mbfpp/functional/fb/discount_builder.clj

```
(defn discount [percentage]
  {:pre [(and (>= percentage 0) (<= percentage 100))]}
  (fn [price] (- price (* price percentage 0.01))))
```

创建一个固定折扣的价格函数，并将它作为匿名函数调用：

```
=> ((discount 50) 200)
100.0
```

① 原书正文中为discountedPrice()，而在代码片段中混用了discount()，为保持一致，相关处统一采用discount()。——译者注

正如我们所提到过的，尝试创建一个位于可接受范围之外的折扣将会抛出异常：

```
=> (discount 101)
AssertionError Assert failed: ...
```

如果我们想要为折扣函数命名，并多次使用该函数，可以按如下的方式进行：

```
=> (def twenty-five-percent-off (discount 25))
=> (apply + (map twenty-five-percent-off [100.0 25.0 50.0 25.0]))
150.0
=> (apply + (map twenty-five-percent-off [75.0, 25.0]))
75.0
```

该折扣计算器是一个相当简单的例子；在下一节中，我们将会看到一个更加复杂的例子。

map的键选择器

来看一个更加复杂的函数生成器实现。我们要尝试解决的问题如下：我们拥有一个由互相嵌套的map组成的数据结构，想要创建出可以帮助我们从该数据结构中取出值的函数，并且能支持对深层嵌套部分的取值。

有一种方法就是编写简单的声明性语言来从深度嵌套的map中取出数据。这跟XPath非常相似，后者可以从深度嵌套的XML结构中筛选出任意的元素，这与CSS选择器对HTML的筛选功能也很相似。

我们首先创建了一个函数：`selector`，它以目标数据的查找路径作为入参。举个例子，如果我们以map来代表某个人的实例，并以`name`作为键，而该键对应的值则是另一个map。另一个map拥有一个名为`first`的键，`first`对应的值则是这个人的第一个名字，而我们想要为第一个名字创建一个选择器，例如：`selector('name, 'first)`。可以在下面的代码中看到这部分内容的实现：

```
scala> val person = Map('name -> Map('first -> "Rob"))
person: ...

scala> val firstName = selector('name, 'first)
firstName: scala.collection.immutable.Map[Symbol,Any] => Option[Any] = <function1>

scala> firstName(person)
res0: Option[Any] = Some(Rob)
```

此类结构在处理像XML或JSON结构化数据的时候会非常方便。这些结构化的数据可以被解析成某种嵌套的结构，而这一类的函数生成器则可以帮助我们从这些结构中筛选出所需的数据。

- Scala实现

选择器实例的Scala实现与上文中的描述保持一致，创建了可以从深度嵌套的map中挑选出数据的函数。该实现创建的`selector`函数会在找到嵌套值时返回`Some(Any)`；否则返回`None`。

为了创建选择器，需要传入多个与我们想要选择的路径相匹配的符号（`Symbol`）。由于这些

都是需要传入selector的参数，所以我们可以利用Scala对可变长参数列表（vararg）的支持来替代原来那种传入明确参数列表的方式。也就是说，如果要创建一个从某人的地址中选取街道名称的选择器，代码将如下所示：

```
scala> selector('address, 'street, 'name)
res0: scala.collection.immutable.Map[Symbol,Any] => Option[Any] = <function1>
```

一旦该选择器创建完毕，将会传入一个map，该选择器会递归地遍历该map，并基于给定的路径查找指定的值。这段代码稍稍有些复杂，所以在总览所有内容之后，我们会对这段代码进行拆解：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/fb/Selector.scala

```
def selector(path: Symbol*): (Map[Symbol, Any] => Option[Any]) = {

  if(path.size <= 0) throw new IllegalArgumentException("path must not be empty")

  @tailrec
  def selectorHelper(path: Seq[Symbol], ds: Map[Symbol, Any]): Option[Any] =
    if(path.size == 1) {
      ds.get(path(0))
    }else{
      val currentPiece = ds.get(path.head)
      currentPiece match {
        case Some(currentMap: Map[Symbol, Any]) =>
          selectorHelper(path.tail, currentMap)
        case None => None
        case _ => None
      }
    }
  (ds: Map[Symbol, Any]) => selectorHelper(path.toSeq, ds)
}
```

先来看看selector函数的签名：

```
def selector(path: Symbol*): (Map[Symbol, Any] => Option[Any]) = {
       «selector-body»
}
```

这说明selector以一组数量不定的符号参数作为入参，并会返回一个函数。该函数以一个键类型是符号，值类型是Any的map为入参，返回的则是Option[Any]类型。

代码的第一行对传入的路径进行了检查，判断它是否至少拥有一个元素，如果没有元素的话会抛出一个异常：

```
if(path.size <= 0) throw new IllegalArgumentException("path must not be empty")
```

函数的主干是一个嵌套的递归帮助函数。让我们来看看它的签名：

```
@tailrec
def selectorHelper(path: Seq[Symbol], ds: Map[Symbol, Any]): Option[Any] =
       «selector-helper-body»
}
```

这说明selectorHelper函数以一个作为路径的符号序列及一个由map组成的数据结构作为入参，其中map的键是符号类型，值是Any类型。该函数返回一个Option[Any]，该返回值代表了我们尝试从选择器中查找出的最终值。在上面的例子中，该值将会是某个人所在街道的名称。

接下去，我们来看看递归的基线条件。该条件会在我们达到路径末尾的时候触发。如果我们找到了想要查找的值便会将该值返回。如果值不存在，get()方法会返回None：

```
if(path.size == 1) {
        ds.get(path(0))
}
```

篇幅最大的代码块包含了尾递归调用。在下面的代码中，我们从数据结构中取出当前部分。如果当前部分存在，我们就会以路径的剩余部分和已摘取的数据结构来递归调用帮助函数。如果不存在，或者没有匹配到合适的类型，就会返回None：

```
else{
        val currentPiece = ds.get(path.first)
        currentPiece match {
                case Some(currentMap: Map[Symbol, Any]) =>
                        selectorHelper(path.tail, currentMap)
                case None => None
                case _ => None
                }
    }
```

最后，来看代码的最后一行，它返回了一个函数，该函数以适当的参数调用了selectorHelper：

```
(ds: Map[Symbol, Any]) => selectorHelper(path.toSeq, ds)
```

让我们进一步来看如何使用selector，从一个简单的例子开始，这是一个拥有单个键值对的map：

```
scala> val simplePerson = Map('name -> "Michael Bevilacqua-Linn")
simplePerson: scala.collection.immutable.Map[Symbol,java.lang.String] =
        Map('name -> Michael Bevilacqua-Linn)

scala> val name = selector('name)
name: scala.collection.immutable.Map[Symbol,Any] => Option[Any] = <function1>

scala> name(simplePerson)
res0: Option[Any] = Some(Michael Bevilacqua-Linn)
```

当然，它真正的强大之处在我们处理嵌套数据结构的时候显得尤为突出，比如下面的场景：

```
scala> val moreComplexPerson =
        Map('name -> Map('first -> "Michael", 'last -> "Bevilacqua-Linn"))
moreComplexPerson: scala.collection.immutable.Map[...] =
        Map('name -> Map('first -> Michael, 'last -> Bevilacqua-Linn))

scala> val firstName = selector('name, 'first)
firstName: scala.collection.immutable.Map[Symbol,Any] => Option[Any] = <function1>
```

```
scala> firstName(moreComplexPerson)
res1: Option[Any] = Some(Michael)
```

如果选择器并没有匹配到任何东西，将会返回None：

```
scala> val middleName = selector('name, 'middle)
middleName: scala.collection.immutable.Map[Symbol,Any] => Option[Any] = <function1>

scala> middleName(moreComplexPerson)
res2: Option[Any] = None
```

- Clojure实现

选择器的Clojure版本比Scala的简单得多。在某种程度上来说，这是因为Clojure是动态类型的，所以我们无需像在Scala中那样担心类型系统。除此之外，Clojure还拥有一个非常方便的叫做get-in的函数，该函数是为从深度嵌套的map中选取指定值量身定制的。

让我们在深入代码之前快速了解一下get-in。该函数以一个嵌套的map作为它的第一个参数，而后续参数中的序列代表了我们正在查找的值的路径。下面是一个用它来从一个嵌套map中查找街道名称的例子：

```
=> (def person {:address {:street {:name "Fake St."}}})
#'mbfpp.functional.fb.selector/person
=> (get-in person [:address :street :name])
"Fake St."
```

基于get-in来构建选择器是非常简单的。我们添加了一个验证器来确保路径不能为空，同时使用了可变长参数来表示路径。代码如下：

ClojureExamples/src/mbfpp/functional/fb/selector.clj

```
(defn selector [& path]
  {:pre [(not (empty? path))]}
  (fn [ds] (get-in ds path)))
```

接着就可以像Scala版本那样轻松使用该函数了。下面的代码展示了如何从一个简单的map中选取出某个人的名字：

```
=> (def person {:name "Michael Bevilacqua-Linn"})
#'mbfpp.functional.fb.selector/person
=> (def personName (selector :name))
#'mbfpp.functional.fb.selector/personName
=> (personName person)
"Michael Bevilacqua-Linn"
```

下面我们从一个嵌套层次更深的map中选取出了一个街道的名字：

```
=> (def person {:address {:street {:name "Fake St."}}})
#'mbfpp.functional.fb.selector/person
=> (def streetName (selector :address :street :name))
#'mbfpp.functional.fb.selector/streetName
=> (streetName person)
"Fake St."
```

在继续深入之前，先来快速看看该实例中Scala和Clojure两个版本的相对复杂性。现在的情况是Clojure拥有get-in，它几乎完成了我们需要完成的全部工作，这使得Clojure版本更加简洁。另一个原因则是Clojure是一门动态类型语言。由于嵌套的map可以持有任意类型的值，这使得像Scala这样的静态类型语言需要完成一些类型系统的操作。

在Clojure中，像上面那样使用map来持有数据是一种非常惯用的做法。而在Scala中，更为常见的做法是使用类或样本类。然而，在处理此类非常动态的问题时，我更偏向于采用map来持有数据。选择采用map意味着我们可以使用所有为map和集合建立的内建工具来处理数据结构了。

通过其他函数来生成函数

在函数式的世界里，由于函数本身也是可以被处理的数据块，所以采用函数生成器模式来将一个函数转换成另一个函数是很常见的。这很容易实现，只需要创建一个处理另一个函数返回值的新函数即可。举个例子，如果我们拥有一个isVowel函数，并想要一个isNotVowel函数，我们只需将调用委托给isVowel，然后对它返回的结果取反即可。这样做便创建了一个互补的函数，如下面的Scala代码所示：

```
scala> def isNotVowel(c: Char) = !isVowel(c)
isNotVowel: (c: Char)Boolean

scala> isNotVowel('b')
res0: Boolean = true
```

在这个例子中，我们将会看到另外两种从既有函数创建新函数的方式：函数组成和部分函数应用。函数组成让我们可以将多个函数串连在一起。而部分函数应用让我们可以通过一个函数和该函数的部分参数来生成一个参数数量更少的新函数。这两种方式是从函数创建函数最为通用的有效手段。

函数组成

函数组成是串连起多个函数调用的一种方式。将一组函数组合起来，然后返回一个新的函数。当我们调用该函数时，首先会调用原来函数列表中的第一个函数，接着将第一个函数的输出传入第二个函数，而该函数也会继续将输出传递给它的下一个函数，直到返回最终的结果。

在很多方面，函数组成与替代装饰器模式（模式9）中所使用的方式非常相似。在装饰器模式中，各自完成某个任务的一部分的多个装饰器被串连在了一起。而在这里，多个函数被串连在了一起。

可以通过手动将函数串连在一起来实现函数组成，但是由于这是一个如此常见的任务，所以函数式语言为函数组成提供了头等的支持。Clojure和Scala也不例外，下面就让我们来看看它们是如何实现的。

● Scala实现

在Scala中，我们可以使用compose操作符来将函数组合到一起。作为一个简单的实例，我们定义了三个函数：appendA、appendB和appendC，它们会分别为字符串添加字母a、b和c，如下面的代码所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/fb/CompositionExamples.scala

```
val appendA = (s: String) => s + "a"
val appendB = (s: String) => s + "b"
val appendC = (s: String) => s + "c"
```

现在，如果想要添加所有这三个字母，可以通过如下定义来使用函数组成：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/fb/CompositionExamples.scala

```
val appendCBA = appendA compose appendB compose appendC
```

正如函数名称所示，它会按c、b、a的顺序为字符串添加字母。这等同于编写一个这样的函数：首先将入参传递给appendC()，然后将返回的值传递给appendB()，最后再将appendB()返回的值传递给appendA()：

```
scala> appendCBA("z")
res0: java.lang.String = zcba
```

虽然这只是一个很简单的例子，却阐明了函数组成中一件很重要的事情：参与组成的函数的调用顺序。

组成函数链中的最后一个函数会被最先调用，而第一个函数则会最后一个调用，这就是为什么c会是添加到字符串的第一个字母。

来看一个更加复杂的例子。一个较为常见的情况是：在Web应用框架中，需要将HTTP请求穿过一组用户定义的代码块。J2EE的servlet过滤器[①]便是此类代码块的一个常见实例，我们会在请求被处理之前让它先穿过一个过滤器链。

过滤器链允许应用代码在请求被处理之前完成任何需要完成的事情，比如对请求进行解码和解压缩、检查身份认证凭证以及记录请求日志，等等。让我们来看看如何使用函数组成。首先，需要找到一种用以表示HTTP请求的方式。对于本例，我们将尽可能保持简单，并仍然以map来作为请求头，同时以一个字符串来表示请求体：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/fb/CompositionExamples.scala

```
case class HttpRequest(
    headers: Map[String, String],
    payload: String,
    principal: Option[String] = None)
```

① http://www.oracle.com/technetwork/java/filters-137243.html

接下来定义一些过滤器。每个过滤器都是一个以HttpRequest作为入参的函数，它们完成一些事情，然后返回一个HttpRequest。对于这个简单的示例，我们将返回相同的HttpRequest；如果过滤器需要对请求进行修改或添加一些内容，可以创建新HttpRequest，而该HttpRequest会包含所需的变更。

下面是一对范例过滤器。第一个过滤器模拟了对Authorization头部的检查，如果该头部合法将会向请求添加一个用户主体。第二个过滤器模拟了对请求指纹的日志记录，该日志可用于后续的问题排查：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/fb/CompositionExamples.scala

```
def checkAuthorization(request: HttpRequest) = {
  val authHeader = request.headers.get("Authorization")
  val mockPrincipal = authHeader match {
    case Some(headerValue) => Some("AUser")
    case _ => None
  }
  request.copy(principal = mockPrincipal)
}

def logFingerprint(request: HttpRequest) = {
  val fingerprint = request.headers.getOrElse("X-RequestFingerprint", "")
  println("FINGERPRINT=" + fingerprint)
  request
}
```

最后，我们需要一个以这组过滤器序列作为入参的函数，并将这些过滤器组合起来。可以通过在这组序列上进行reduce操作来完成对函数的组合：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/fb/CompositionExamples.scala

```
def composeFilters(filters: Seq[Function1[HttpRequest, HttpRequest]]) =
  filters.reduce {
    (allFilters, currentFilter) => allFilters compose currentFilter
  }
```

为了了解该函数的工作方式，让我们将一组范例过滤器组合成一个过滤器链。然后再运行一个测试的HttpRequest，并让它通过这个过滤器链：

```
scala> val filters = Vector(checkAuthorization, logFingerprint)
filters: ...

scala> val filterChain = composeFilters(filters)
filterChain: ...

scala> val requestHeaders =
        Map("Authorization" -> "Auth", "X-RequestFingerprint" -> "fingerprint")
requestHeaders: ...

scala> val request = HttpRequest(requestHeaders, "body")
request: ...
```

```
scala> filterChain(request)
FINGERPRINT=fingerprint
res0: com.mblinn.mbfpp.functional.fb.ScalaExamples.HttpRequest =
        HttpRequest(
                Map(Authorization -> Auth, X-RequestFingerprint -> fingerprint),
                body,
                Some(AUser))
```

正如我们所看到的，过滤器链正确地运行着HttpRequest，而HttpRequest顺利通过了链中的每一个过滤器。这些过滤器向请求添加了用户主体，并将请求指纹的日志记录在了控制台。

- Clojure实现

在Clojure中，完成函数组成最简单的方式就是使用comp。在下面的代码中，我们使用它将字符串追加函数组合在了一起：

ClojureExamples/src/mbfpp/functional/fb/composition_examples.clj

```
(defn append-a [s] (str s "a"))
(defn append-b [s] (str s "b"))
(defn append-c [s] (str s "c"))

(def append-cba (comp append-a append-b append-c))
```

这与Scala版本的工作方式非常相似：

```
=> (append-cba "z")
"zcba"
```

在Clojure中，我们会将HTTP请求与头部都建模成map。下面是一个范例请求：

ClojureExamples/src/mbfpp/functional/fb/composition_examples.clj

```
(def request
  {:headers
   {"Authorization" "auth"
    "X-RequestFingerprint" "fingerprint"}
   :body "body"})
```

我们的范例过滤器从map中取出键，并使用nil来替代None表示不存在的值。在这里，通过函数生成器compose-filters，可以将它们组合成一个过滤器链：

ClojureExamples/src/mbfpp/functional/fb/composition_examples.clj

```
(defn check-authorization [request]
  (let [auth-header (get-in request [:headers "Authorization"])]
    (assoc
      request
      :principal
      (if-not (nil? auth-header)
        "AUser"))))

(defn log-fingerprint [request]
```

```
  (let [fingerprint (get-in request [:headers "X-RequestFingerprint"])]
    (println (str "FINGERPRINT=" fingerprint))
    request))

(defn compose-filters [filters]
  (reduce
    (fn [all-filters, current-filter] (comp all-filters current-filter))
    filters))
```

下面是过滤器链的实践代码，通过让请求通过这些过滤器，从而执行这些过滤器，最后会返回一个HTTP请求：

```
=> (def filter-chain (compose-filters [check-authorization log-fingerprint]))
#'mbfpp.functional.fb.composition-examples/filter-chain
=> (filter-chain request)
FINGERPRINT=fingerprint
{:principal "AUser",
:headers {"X-RequestFingerprint" "fingerprint", "Authorization" "auth"},
:body "body"}
```

函数组成是一个非常通用的操作，在本节的内容中，我们只接触了它的一部分用途。任何时候，如果你发现需要按相同的顺序多次调用相同的函数集合，或者当你有一个动态生成的函数列表，并需要将它们串连起来时，函数组成将是一个不错的选择。

部分应用函数

函数组成接受多个函数并将它们串连到一起，而部分应用函数则是接受一个函数和该函数入参的一个子集，然后返回一个新的函数。新函数的入参个数相比原来的函数更少，在部分应用函数被创建的时候，会对已传入的参数子集保持记录，并在后续参数传入时再使用这些已传入的参数。

让我们来看看它在Scala中是如何运作的。

- Scala实现

部分函数应用是另一个函数式特性，它在Scala中对函数的头等支持起着举足轻重的作用。它的工作方式是这样的：你调用一个函数，然后采用下划线来替代你当前尚未拥有值的变量。举个例子，如果现在有一个将两个整数相加的函数，而我们希望有一个能为任意整数加上42的函数，那么可以像下面这样进行创建：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/fb/PartialExamples.scala

```
def addTwoInts(intOne: Int, intTwo: Int) = intOne + intTwo

val addFortyTwo = addTwoInts(42, _: Int)
```

如下面的代码所示，addFortyTwo是一个只有一个入参的函数，并且该函数会为传入的参数值加上42。

```
scala> addFortyTwo(100)
res0: Int = 142
```

创建部分应用函数很简单，但是确定何时使用它们就有点困难了。下面有一个它们可以派上用场的例子：我们有一个用于根据状态来计算所得税的函数，与此同时我们想要创建一些可以为某个特定状态计算所得税的函数。可以使用部分应用函数来完成这一工作，就像下面这样：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/fb/PartialExamples.scala

```
def taxForState(amount: Double, state: Symbol) = state match {
  //简单的税收逻辑，仅供示例之用
  case ('NY) => amount * 0.0645
  case ('PA) => amount * 0.045
  // 省略剩余的其他状态...
}
val nyTax = taxForState(_: Double, 'NY)
val paTax = taxForState(_: Double, 'PA)
```

该函数为不同的状态都正确地计算出了税费：

```
scala> nyTax(100)
res0: Double = 6.45

scala> paTax(100)
res1: Double = 4.5
```

- Clojure实现

在Clojure中，部分应用函数的做法与Scala中的有些相似，但是存在着一些区别。为了保持语法的简单，Clojure仅允许函数参数列表中的第一个参数被部分应用。举个例子，我们仍然可以编写`add-forty-two`函数，正如我们在Scala中所做的，如下面的代码所示：

ClojureExamples/src/mbfpp/functional/fb/partial_examples.clj

```
(defn add-two-ints [int-one int-two] (+ int-one int-two))

(def add-fourty-two (partial add-two-ints 42))

=> (add-forty-two 100)
142
```

但是为了编写`ny-tax`和`pa-tax`，我们需要对`tax-for-state`进行参数的切换，如下所示：

ClojureExamples/src/mbfpp/functional/fb/partial_examples.clj

```
(defn tax-for-state [state amount]
  (cond
    (= :ny state) (* amount 0.0645)
    (= :pa state) (* amount 0.045)))

(def ny-tax (partial tax-for-state :ny))
(def pa-tax (partial tax-for-state :pa))

=> (ny-tax 100)
```

```
6.45
=> (pa-tax 100)
4.5
```

部分应用函数使用起来非常简单，但是我通常都认为何时使用它们才是值得思考的事情。我常常发现自己反复地调用着相同的函数，而且连使用的参数子集都保持不变。此时你会突然意识到，通过使用部分应用函数就可以清理掉这些反复的调用。

讨论

本节我们讨论了一些使用函数生成器模式的通用方式，但这并不表示这些就是唯一的方法。由于这是一个在函数式世界里极其常见的模式，所以在Clojure和Scala的代码库中还有很多其他的例子。

虽然大部分Clojure和Scala实例的实现都非常相似，但是“map的键选择器”这一节中的实例却大不相同。尤其是Scala版本的实现，显得非常冗长。从某种程度上来说，这是因为Clojure拥有着一个极其方便的`get-in`函数，该函数几乎完成了我们需要完成的所有工作；然而，这些区别在很大程度上是由Scala的类型系统造成的。

由于Scala是静态类型的，所以我们需要为待处理map的内容指定类型。内部节点是map本身，而叶子节点则可以是任何的东西。这导致我们不得不在Scala中完成少量与类型系统相关的工作。

这是一个在动态和静态类型之间的权衡。就算像Scala一样拥有一个强大的类型系统，由静态类型系统所带来的复杂性成本，以及对类型系统运作原理的理解成本仍然是不可忽视的。权衡点在于，对于静态类型系统来说，可以在编译时提前捕捉到大量的错误，而在一个动态类型系统中，这些问题很可能就会成为运行时错误。

相关模式

模式1　替代函数式接口

模式9　替代装饰器模式

模式 17

记忆模式

目的

对纯函数的调用结果进行缓存，从而避免重复执行相同的计算。

概述

由于在给定参数不变的情况下，纯函数始终会返回相同的值，所以我们可以采用缓存的调用结果来替代重复的纯函数调用。

我们可以手动编写这样一个函数，它可以对历次入参及结果保持跟踪。每当该函数被调用的时候，首先会检查缓存中是否存在该入参对应的结果。如果存在，它会返回缓存的值。否则，便会执行计算。

某些语言采用高阶函数对记忆模式提供了头等支持。举个例子，Clojure拥有一个名叫memoize的函数，该函数以一个函数作为入参，并返回一个可以缓存结果的函数。Scala并不具有内建的记忆函数，所以我们将手动来完成一个简单的实现。

范例代码：简单缓存

记忆模式的一个用途是可以作为资源消耗较为昂贵或较为耗时的函数的简单缓存，尤其是在以相同入参多次调用该函数的场景下。在这个例子中，我们将会通过线程睡眠来模拟时间消耗较久的操作。

Scala实现

先来看看模拟的代价昂贵的函数调用。在该例子中，我们通过ID从某个（假定较为慢速的）数据存储中查找内容。为了模拟这个慢速的操作，我们会在从静态map中返回值之前让线程睡上一秒钟。同时，也会在控制台打印出用于查找的ID，从而表明函数被执行了：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/memoization/Examples.scala

```
def expensiveLookup(id: Int) = {
  Thread.sleep(1000)
  println(s"Doing expensive lookup for $id")
  Map(42 -> "foo", 12 -> "bar", 1 -> "baz").get(id)
}
```

正如我们所预期的，这个冗长的函数在每次调用时都会被执行，我们可以从控制台输出中看到这一现象：

```
scala> expensiveLookup(42)
Doing expensive lookup for 42
res0: Option[String] = Some(foo)

scala> expensiveLookup(42)
Doing expensive lookup for 42
res1: Option[String] = Some(foo)
```

现在来看一个expensiveLookup()函数的简单记忆版本。为了创建该函数，我们将会使用memoizeExpensiveLookup()，该函数会初始化缓存，并返回一个封装了memoizeExpensive-Function()函数调用的新函数。

该新函数首先会检查缓存中是否已存在曾经调用执行的结果。如果存在，它将会返回该缓存的结果。否则，它会调用代价昂贵的查询操作，并在返回结果前将它们缓存起来。

最后，调用memoizeExpensiveFunction()函数，并将返回结果中的函数的引用存储在一个新的var变量中。整个解决方案的代码如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/memoization/Examples.scala

```
def memoizeExpensiveLookup() = {
  var cache = Map[Int, Option[String]]()
  (id: Int) =>
    cache.get(id) match {
      case Some(result: Option[String]) => result
      case None => {
        val result = expensiveLookup(id)
        cache += id -> result
        result
      }
    }
}
val memoizedExpensiveLookup = memoizeExpensiveLookup
```

正如我们从下面的REPL输出中所看到的，该代价昂贵的函数仅在给定入参首次调用时被执行。在此之后，它将只会返回已缓存的结果：

```
scala> memoizedExpensiveLookup(42)
Doing expensive lookup for 42
res2: Option[String] = Some(foo)

scala> memoizedExpensiveLookup(42)
res3: Option[String] = Some(foo)
```

该例子的一个特殊之处在于最后一行：

```
val memoizedExpensiveLookup = memoizeExpensiveLookup
```

在这里，我们调用memoizeExpensiveLookup()函数并返回了一个新的函数，接着对该函数的引用进行了保存。这样做可以让我们将缓存包装在一个闭包之中，并且只有这个返回的函数才拥有对缓存的引用。如果我们需要另一个缓存，可以像下面这样再次创建一个：

```
scala> val memoizedExpensiveLookup2 = memoizeExpensiveLookup
memoizedExpensiveLookup2: Int => Option[String] = <function1>

scala> memoizedExpensiveLookup2(42)
Doing expensive lookup for 42
res4: Option[String] = Some(foo)
```

我们的Scala解决方案显得有些笨拙，主要是因为该方案是通过手动方式来为特定案例所定制的，但是它为展示记忆模式的幕后运作方式提供了一个很好的模型。让我们来看看如何使用Clojure中的memoize函数来解决相同的问题。

Clojure实现

在Clojure中，我们首先会模拟一个类似的代价昂贵的函数。然而，我们不再需要手动保持对它的结果记忆。我们将使用Clojure的memoize函数自动地返回一个记忆版本的函数，如下面的代码所示：

ClojureExamples/src/mbfpp/functional/memoization/examples.clj

```
(defn expensive-lookup [id]
  (Thread/sleep 1000)
  (println (str "Lookup for " id))
  ({42 "foo" 12 "bar" 1 "baz"} id))

(def memoized-expensive-lookup
  (memoize expensive-lookup))
```

我们可以从下面的REPL输出中看到，该函数与Scala版本的表现行为非常相似，对于代价昂贵的操作仅会执行一次：

```
=> (memoized-expensive-lookup 42)
Lookup for 42
"foo"
=> (memoized-expensive-lookup 42)
"foo"
```

在这段代码的背后，memoize函数创建了一个新的函数。该函数与在Scala中采用map作为缓存的手工实现的例子非常相似。

讨论

本章没有涉及记忆模式的另一个用途，即解决动态规划问题，而这也正是该模式最初的用途。动态规划问题是指那些可以被递归拆解成更加简单的子问题的问题。一个通俗易懂的经典示例就是计算斐波那契数。

计算第n个斐波那契数的公式是将该序列中的前两个数字进行相加。一个简单的用于计算斐波那契数的Clojure函数的定义如下所示：

```
ClojureExamples/src/mbfpp/functional/memoization/examples.clj
(def slow-fib
  (fn [n]
    (cond
      (<= n 0) 0
      (< n 2) 1
      :else (+ (slow-fib (- n 1)) (slow-fib (- n 2))))))
```

该函数的一个好处是它真实地反映了斐波那契数在数学上的定义。然而，它需要重复地递归计算每个组成部分，所以对于达到一定规模的数字计算来说它的性能就变得非常差。如果我们对函数进行记忆化，例如在下面的代码中所做的，每个组成部分将会被缓存，函数的执行性能将会

相当不错：

```
ClojureExamples/src/mbfpp/functional/memoization/examples.clj
(def mem-fib
  (memorize
    (fn [n]
      (cond
        (<= n 0) 0
        (< n 2) 1
        :else (+ (mem-fib (- n 1)) (mem-fib (- n 2)))))))
```

我们通过运行这两个函数，来展现一下它们在性能上的巨大差别：

```
=> (time (slow-fib 40))
"Elapsed time: 6689.204 msecs"
102334155
=> (time (mem-fib 40))
"Elapsed time: 0.402 msecs"
102334155
```

动态规划问题丰富多彩，让人着迷。然而，它们仅出现在数量有限的领域之中。我通常看到的记忆模式的使用场景是在代价昂贵或耗时较久的操作中作为缓存使用，而不是作为动态规划的工具使用。

模式 18

惰性序列模式

目的

创建一个序列，该序列的成员仅在需要时才会得到计算——这让我们可以简单地将计算结果转化为流的形式，从而处理无限长的序列。

概述

我们通常一次只会处理序列中的一个元素。就因为这样，我们往往不需要在开始处理其他元素之前就将整个序列中的元素都实例化出来。举个例子，我们可能希望对磁盘上的文件进行一行行的流式读取，从而无需将整个文件读到内存中就可以逐行对它的内容进行处理。也可以使用尾递归模式（模式12）来对文件进行遍历查找，不过惰性序列为此类流式计算提供了更加整洁的抽象。

惰性序列只在被要求时才会在序列中创建一个元素。在文件读取的例子中，文件的每一行只在被要求时才会从磁盘上读出来，并且会在我们完成对它们的处理之后被垃圾回收掉，尽管我们还是需要花一点心思来确保它们已经被回收。

当我们创建一个元素的时候，我们称之为“实例化”（realizing）了该元素。一旦元素被实例化之后，我们便会采用记忆模式将它放入缓存中。这意味着，我们一次只需要在序列中实例化一个元素。图4-4描绘了这一过程。

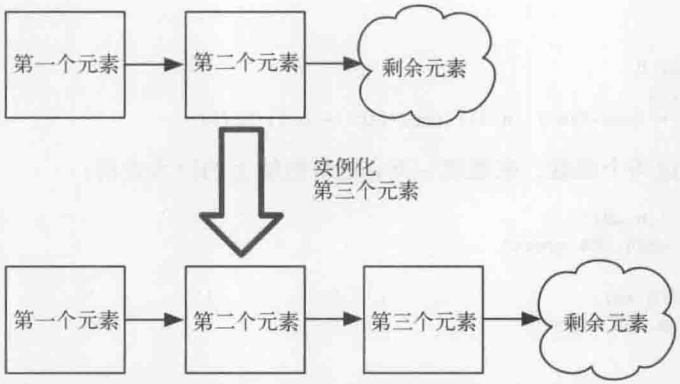


图4-4　惰性序列。惰性序列在第三个元素被实例化前后的实例状态

惰性序列也让我们创建出另一个极其有用的抽象：无限长序列。乍一看，这似乎没什么用处，但是由于整个序列并不是一次就实例化出来的，所以我们可以先处理序列的开头部分，并延迟对剩余部分的创建。这样可以让我们创建一个名义上由伪随机数字组成的无限长字符串，而事实上只需要实例化其中的一部分。

范例代码：内建的惰性序列

先来看两个来自内建代码库的简单示例。在第一个例子中，我们将会展示如何处理无限长的整数列表。

在第二个例子中，将会展示如何使用惰性序列模式来产生一组随机的测试数据。

首先来看Scala的代码。

Scala实现

Scala在它的Stream库中提供了对惰性序列的内建支持。或许我们可以采用惰性序列来完成的最简单的事情就是创建一个包含了所有整数的无穷序列。Scala的Stream库拥有一个可以完成这一工作的方法——`from()`。根据Scala的文档描述，该方法将“创建一个以start作为开始项的无穷流，并以step作为步长进行递增”。

下面，我们使用`from()`创建了一个包含所有整数的序列，从0开始：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/ls/LazySequence.scala

```
val integers = Stream.from(0)
```

这似乎看上去是一件新奇的事情，但是我们还可以使用另一个方法take()来对序列中的前几个数字进行处理。在下面的代码中，我们使用该方法从无限长列表中取出了前五个整数，并将它们打印出来：

```
scala> val someints = integers take 5
someints: scala.collection.immutable.Stream[Int] = Stream(0, ?)

scala> someints foreach println
0
1
2
3
4
```

让我们再来看一个关于惰性序列的更加奇特的实例，它使用了Scala Sequence库中的另一个方法。方法continually()通过对传入表达式进行重复计算从而创建了一个无限长的序列。

我们使用它来创建一个伪随机数的无限长序列。为此，我们创建了一个新的随机数生成器，并将它赋给了val变量generate，然后将generate.nextInt传入continually()方法，如下面的代码所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/ls/LazySequence.scala

```
val generate = new Random()
val randoms = Stream.continually(generate.nextInt)
```

现在我们可以从无穷列表中取出一些随机数：

```
scala> val aFewRandoms = randoms take 5
aFewRandoms: scala.collection.immutable.Stream[Int] = Stream(326862669, ?)

scala> aFewRandoms foreach println
326862669
-473217479
-1619928859
785666088
1642217833
```

如果我们想要获取更多的随机数，可以再次调用take()，同时传入更大的数字作为入参：

```
scala> val aFewMoreRandoms = randoms take 6
aFewMoreRandoms: scala.collection.immutable.Stream[Int] = Stream(326862669, ?)

scala> aFewMoreRandoms foreach println
326862669
-473217479
-1619928859
785666088
1642217833
1819425161
```

请注意前五个数字是如何重复出现的。这是因为Stream库依赖于记忆模式，它会将你看到的

每个数字进行缓存。前五个数字的值在我们首次打印aFewRandoms时就已经实例化了，而第六个数字仅在我们打印aFewMoreRandoms时实例化了一次。

Clojure实现

惰性序列在Clojure中也拥有内建的支持，但是它并不集中在一个单一的库中。我们甚至可以说，大部分Clojure的核心序列处理函数都是以惰性的方式进行工作的。举个例子，Clojure的标准range函数就是处理惰性序列的。下面的代码产生了一个所有正整数的列表：

ClojureExamples/src/mbfpp/functional/ls/examples.clj

```
(def integers (range Integer/MAX_VALUE))
```

我们可以采用take函数从该长列表的头部取出一些整数：

```
=> (take 5 integers)
(0 1 2 3 4)
```

为了产生随机整数列表，我们可以使用Clojure的repeatedly函数。这种方式以一个函数作为入参，并可以重复对它执行无限次，如下面的代码所示：

ClojureExamples/src/mbfpp/functional/ls/examples.clj

```
(def randoms (repeatedly (fn [] (rand-int Integer/MAX_VALUE))))
```

为了获取一些数字，我们可以再次使用take函数：

```
=> (take 5 randoms)
(2147483647 2147483647 2147483647 2147483647 2147483647)
```

如果想要更多的数字，我们可以以更大的数字作为入参来调用take。再次证明，前五个随机数字不会再被重新计算，它们将会从一个记忆缓存中被提取出来：

```
=> (take 6 randoms)
(2147483647 2147483647 2147483647 2147483647 2147483647 2147483647)
```

Scala和Clojure对于惰性序列的处理有着一些关键的不同之处。大部分Clojure的序列处理函数都是惰性的，但是它们在处理序列的时候大都是以32个元素为单位来成块处理的。如果我们仅从一个整数惰性序列中取出一个数字，Clojure也将会实例化出序列的前32个整数，即使我们只要求获取一个数字。

可以通过向惰性序列产生的方法体加入具有副作用的函数来看到这一现象。在下面的代码中，我们看到惰性序列实例化了32个整数，即便它只返回了第一个数字：

```
=> (defn print-num [num] (print (str num " ")))
#'mbfpp.functional.ls.examples/print-num
=> (take 1 (map print-num (range 100)))
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
 19 20 21 22 23 24 25 26 27 28 29 30 31 nil)
```

另一个更加细微的差别来自于在REPL中使用惰性序列。当Scala的REPL遇到一个以流形式存

在的惰性序列实例时，它并不会尝试对整个序列进行实例化。

当我们拥有明显的副作用时，这一现象是很容易看到的。在下面的Scala代码中，我们使用了continually()来向控制台打印"hello"，并将指向产生流的引用存储在printHellos中。正如我们所看到的，第一个"hello"在我们调用continually的时候被打印了出来，这表明该方法实例化了流中的第一个元素：

```
scala>   val printHellos = Stream.continually(println("hello"))
hello
printHellos: scala.collection.immutable.Stream[Unit] = Stream((), ?)
```

现在，如果我们在printHellos上调用take()函数，将不会在控制台上再看到任何更多的"hello"打印信息，这意味着REPL并不会尝试去实例化返回的流。

```
scala> printHellos take 5
res0: scala.collection.immutable.Stream[Unit] = Stream((), ?)
```

如果我们想要强制实例化剩余的"hello"，可以采用任何能对流进行迭代的方法，或者使用force()：

```
scala> printHellos take 5 force
hello
hello
hello
hello
res1: scala.collection.immutable.Stream[Unit] = Stream((), (), (), (), ())
```

这并不是你通常需要去做的事情，但是这对我们理解惰性序列中的元素是在何时被实例化的来说是非常重要的。

与此形成鲜明对比的是，Clojure的REPL将会尝试对整个惰性序列实例进行实例化；然而，它在定义惰性序列实例的过程中并不会对序列的首个元素进行实例化！下面我们定义了一个print-hellos函数，它与Scala的版本非常相似。请注意，控制台并没有打印出"hello"。

```
(def print-hellos (repeatedly (fn [] (println "hello"))))
```

然而，如果我们要取出五个元素，REPL在对惰性序列的结果实例进行求值时将会强制将它们打印到控制台上。

```
=> (take 5 print-hellos)
(hello
hello
nil hello
nil hello
nil hello
nil nil)
```

上面的例子反应了Scala和Clojure在REPL对惰性序列求值时的不同之处。它同样也强调了使用惰性序列时需要注意的一些地方。虽然你可以创建无穷序列，但是我们需要确保不会尝试去一次性实例化整个无穷序列。举个例子，如果我们忘记在Clojure的例子中使用take，而是直接求值

(repeatedly (fn [] (println "hello"))，那么我们将对打印"hello"的无限长序列进行实例化！

范例代码：分页的数据响应

在第一个例子中，我们看到了两个用于创建惰性序列实例的高阶函数。现在让我们看看如何自己动手来从头创建一个。

本节中将会使用的这个例子是一个可以用于遍历一组分页数据的惰性序列。在这个简单的例子中，我们将会以一个本地的函数调用来模拟分页数据，尽管在一个实际的程序中，这些分页数据可能来自一个例如Web服务的外部数据源。让我们从Scala的代码开始看起。

Scala实现

我们的Scala解决方案由两个部分组成：序列本身pagedSequence，以及一个用于生成范例分页数据的方法getPage()。

我们需要递归地为问题定义解决方案，正如在模式12中所做的一样。然而，我们不再在一个调用栈中传递序列，而是通过使用#::操作符来将序列添加到每一次递归调用中。

下面的代码便是分页数据问题的整个解决方案：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/ls/LazySequence.scala

```
def pagedSequence(pageNum: Int): Stream[String] =
  getPage(pageNum) match {
    case Some(page: String) => page #:: pagedSequence(pageNum + 1)
    case None => Stream.Empty
  }

def getPage(page: Int) =
  page match {
    case 1 => Some("Page1")
    case 2 => Some("Page2")
    case 3 => Some("Page3")
    case _ => None
}
```

让我们再更加深入地来看看pagedSequence，首先是操作符#::，它可以让我们向流的头部添加值。下面我们使用它向一个新的流的头部添加字符串"foo"和"bar"：

```
scala> val aStream = "foo" #:: "bar" #:: Stream[String]()
aStream: scala.collection.immutable.Stream[String] = Stream(foo, ?)
```

我们可以从这个流中取出头部和尾部的元素，就跟处理其他的序列一样：

```
scala> aStream.head
res0: String = foo

scala> aStream.tail
res1: scala.collection.immutable.Stream[String] = Stream(bar, ?)
```

让我们通过下面的代码片段来看一下这个解决方案的核心内容：

```
getPage(pageNum) match {
  case Some(page: String) => page #:: pagedSequence(pageNum + 1)
  case None => Stream.Empty
}
```

我们调用getPage()并对返回结果进行了模式匹配。如果匹配到的是一个Some，我们便知道获取到的是一个有效的页面。我们会将它添加到序列的头部，然后继续递归调用方法来生成序列，并将后续要尝试获取的下一页页码传递给pagedSequence。

如果获取到的是None，我们便知道已经遍历完所有的页面，我们随即会向惰性序列追加一个空的流：Stream.Empty。这表明序列已经到达尾部。

现在我们可以像处理前面例子中的序列一样处理pagedSequence了。下面从序列中取出从第一个元素开始的两个页面：

```
scala> pagedSequence(1) take 2 force
res2: scala.collection.immutable.Stream[String] = Stream(Page1, Page2)
```

这里我们强制实例化了整个序列，这对该序列来说是安全的，因为它虽然是惰性的，但是它并不是无穷的：

```
scala> pagedSequence(1) force
res3: scala.collection.immutable.Stream[String] = Stream(Page1, Page2, Page3)
```

到此Scala的分页序列实例便结束了。让我们再去看看在Clojure中是如何实现的。

Clojure实现

在Clojure中，我们可以使用lazy-seq从头构建惰性序列的实例，然后采用cons向序列添加元素，如下面的代码片段所示：

```
=> (cons 1 (lazy-seq [2]))
(1 2)
```

接下来可以使用递归函数调用来构建有用的序列了。为了使用Clojure来完成分页序列实例，首先需要定义一个get-page函数来模拟分页数据的获取。而我们的解决方案的核心代码则位于paged-sequence函数中。

函数paged-sequence从起始页面开始被调用，它会在你获取页面的时候递归构建出一个惰性序列，并将它添加到原有序列的后面，然后再以下一页页码作为入参来调用其自身。整个解决方案如下所示：

ClojureExamples/src/mbfpp/functional/ls/examples.clj

```
(defn get-page [page-num]
  (cond
    (= page-num 1) "Page1"
    (= page-num 2) "Page2"
```

```
    (= page-num 3) "Page3"
    :default nil))

(defn paged-sequence [page-num]
  (let [page (get-page page-num)]
    (when page
      (cons page (lazy-seq (paged-sequence (inc page-num)))))))
```

现在我们可以像处理其他任何序列一样来处理这个惰性序列了。如果在REPL中调用paged-sequence，我们将会得到整个序列：

```
=> (paged-sequence 1)
("Page1" "Page2" "Page3")
```

如果使用take，我们便可以只获取它的一个部分：

```
=> (take 2 (paged-sequence 1))
("Page1" "Page2")
```

该实例为我们处理流式数据给出了一种非常清晰的方式。

讨论

当你使用惰性序列的时候，有一件事值得注意：在你并非刻意的情况下，可能会凑巧保持了对该序列头部的引用，如图4-5所示。

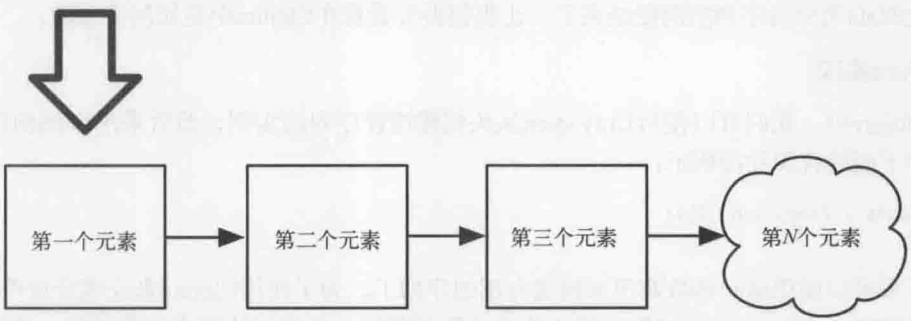


图4-5　保持对序列头部的引用。保持对一个惰性序列的头部引用将会使整个序列一直保持在内存之中

在Scala中，很容易一不小心就遇上这种情况，比如将一个惰性序列赋给了一个val变量，如下面的代码所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/ls/LazySequence.scala

```
val holdsHead = {
  def pagedSequence(pageNum: Int): Stream[String] =
    getPage(pageNum) match {
      case Some(page: String) => {
```

```
        println("Realizing " + page)
        page #:: pagedSequence(pageNum + 1)
      }
      case None => Stream.Empty
    }
  pagedSequence(1)
}
```

如果我们尝试多次强制序列实例化，会看到在第二次时序列使用了缓存的内容，如下面的REPL输出所示：

```
scala> holdsHead force
Realizing Page1
hello
Realizing Page2
Realizing Page3
res0: scala.collection.immutable.Stream[String] = Stream(Page1, Page2, Page3)
scala> holdsHead force
res1: scala.collection.immutable.Stream[String] = Stream(Page1, Page2, Page3)
```

如果我们不想保持对序列头部的引用，可以使用def来替代val，如下面的代码所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/ls/LazySequence.scala

```
def doesntHoldHead = {
  def pagedSequence(pageNum: Int): Stream[String] =
    getPage(pageNum) match {
      case Some(page: String) => {
        println("Realizing " + page)
        page #:: pagedSequence(pageNum + 1)
      }
      case None => Stream.Empty
    }
  pagedSequence(1)
}
```

这样做会强制序列在每一次被强制实例化时都重新实例化，并且不会保持对序列头部的引用：

```
scala> doesntHoldHead force
Realizing Page1
Realizing Page2
Realizing Page3
res2: scala.collection.immutable.Stream[String] = Stream(Page1, Page2, Page3)

scala> doesntHoldHead force
Realizing Page1
Realizing Page2
Realizing Page3
res3: scala.collection.immutable.Stream[String] = Stream(Page1, Page2, Page3)
```

其实，不小心保持了对序列头部的引用并不会比无意中持有了任何对象的引用显得更加神秘，但是如果你没有注意到这个问题的话，很可能会给你带来出乎意料的结果。

模式 19

集中的可变性

目的

在一个程序中，我们会在那些对性能敏感的小部分代码中使用可变的数据结构，同时会将这些部分隐藏在一个函数之中，而在大多数代码中，仍将使用不可变数据。

概述

计算机是构建于像主存和硬盘这样的可变组件之上的，而我们却在计算机上采用高性能的不可变数据进行编程，这着实是一件神奇的事情。目前我们所拥有的一整套技术体系让这一神奇的事情成为了可能，尤其是在JVM上。日益增长的处理器能力和内存资源使得我们无需再为榨干计算机的最后一滴性能而费尽心力。

创建或销毁那些小而短命的对象变得相当廉价，这多亏了JVM卓越的分代垃圾收集器。Scala和Clojure都使用了极其巧妙的数据结构，这些数据结构让不可变集合之间能够共享状态。这样一来，我们就无需在局部数据发生改变时对整个集合进行复制了，也就是说集合拥有着一个合理的内存占用量，并且可以非常快速地处理修改。

尽管如此，使用不可变数据仍有一些性能开销。就算是Clojure和Scala使用的巧妙的数据结构也比相应的可变版本占用更多的内存，而它们的执行性能却更糟糕。不可变数据的好处不仅仅在于大大简化了并发编程，它同样简化了一般大型系统的开发，在这层意义上，它的收益是大于开销的。然而，有的时候你确实需要更多额外的性能，这种情况往往出现在程序中一个被频繁调用的紧凑的循环计算中。

集中的可变性展示了如何通过创建函数在某些场景下使用可变数据，这些函数往往以一些不可变的数据结构作为入参，并在函数中完成对可变数据的操作，然后返回另一个不可变的数据结构。这样不仅避免了可变性对程序的干扰，同时又获得了更好的性能，这缘于我们将可变性封闭进了一个函数之中，而这恰恰是一个除了函数本身再无他者能看到内部可变性的区域。

在使用集中的可变性这一模式时有一个因素值得我们思考，即将一个可变的数据结构转型成不可变数据结构的过程中需要花费多少开销。Clojure提供了对此的头等支持，这一特性被称为暂态（transient）。暂态让我们将一个不可变的数据结构在常量时间内转换成一个可变的数据结构，并在我们完成对它的处理之后同样在常量时间内将它转换回不可变的数据结构。

在Scala中，由于没有对类似Clojure中的暂态这样的头等支持，所以这会变得有些棘手。我们不得不使用Scala中那些数据结构的可变版本，然后采用集合库中的转换方法来将这些可变的数据

结构转换成不可变的结构。谢天谢地，Scala完成这一转换还是相当高效的。

范例代码：添加元素到索引序列

让我们先来看一个非常简单的例子，即向一个索引序列添加一组数字。这在实践中未必非常有用，但这确实是一个非常简单的例子，通过它我们可以完成一些基本的性能分析。

在这个例子中，我们将会统计向一个可变的索引序列添加100万个元素并随后将它转换成一个不可变结构所耗费的时间，以及直接构建不可变序列所耗费的时间，然后会对这两个时间进行对比。这里涉及了一些微基准测试，所以我们将会对每个测试进行多次试验性的运行，从而尽可能地发现一些由垃圾回收、缓存问题等造成的异常值。

这当然不是一种用于执行微基准测试的最佳方式，但是对于我们用以感受哪一种解决方案更快以及有多快而言已经足够了。

Scala实现

在Scala中，我们将会对“直接向一个不可变的Vector添加元素”与“向一个可变的ArrayBuffer添加元素然后将它转换成一个不可变的Vector”进行对比。除了向Vector和ArrayBuffer添加元素的测试函数之外，我们还会需要一些用来辅助计时和测试运行的基础设施代码。

首先来看不可变数据结构的部分。下面的代码定义了一个函数：testImmutable()，该函数负责向一个不可变的Vector追加count元素，并且在每次追加完新元素后更新引用，使其指向一个新的Vector：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/fm/FocusedMutation.scala

```
def testImmutable(count: Int): IndexedSeq[Int] = {
  var v = Vector[Int]()
  for (c <- Range(0, count))
    v = v :+ c
  v
}
```

现在再来看看testMutable()，它与上面的代码非常相似，不同之处在于它将元素追加到一个可变的ArrayBuffer，这是一个有点类似于Java中的ArrayList的数据结构。代码如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/fm/FocusedMutation.scala

```
def testMutable(count: Int): IndexedSeq[Int] = {
  val s = ArrayBuffer[Int](count)
  for (c <- Range(0, count))
    s.append(c)
  s.toIndexedSeq
}
```

现在我们需要找到对测试函数的运行时间信息进行统计的方法。我们会在测试运行前后记录

下系统时间，从而完成对运行时间的统计。我们不会选择在测试函数中内嵌时间统计的代码，而是创建一个可以完成时间统计的高阶函数time()，以及另一个函数timeRuns()，后者可以帮助我们一次运行多个测试。代码如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/fm/FocusedMutation.scala

```
def time[R](block: => R): R = {
  val start = System.nanoTime
  val result = block
  val end = System.nanoTime
  val elapsedTimeMs = (end - start) * 0.000001
  println("Elapsed time: %.3f msecs".format(elapsedTimeMs))
  result
}

def timeRuns[R](block: => R, count: Int) =
  for (_ <- Range(0, count)) time { block }
```

一切就绪后，我们便可以开始运行一些测试了。首先来进行五次不可变版本的测试，其中count的赋值为100万：

```
scala> val oneMillion = 1000000
scala> timeRuns(testImmutable(oneMillion), 5)
Elapsed time: 127.499 msecs
Elapsed time: 127.479 msecs
Elapsed time: 130.501 msecs
Elapsed time: 142.875 msecs
Elapsed time: 123.623 msecs
```

正如我们所看到的，统计的时间范围差不多在123毫秒到142毫秒之间。现在让我们来试试可变数据结构的版本，该版本只会在完成所有的数据修改之后才将原来可变的数据结构转换成一个不可变的数据结构：

```
scala> timeRuns(testMutable(oneMillion), 5)
Elapsed time: 98.339 msecs
Elapsed time: 105.240 msecs
Elapsed time: 88.800 msecs
Elapsed time: 65.997 msecs
Elapsed time: 54.918 msecs
```

这一次，统计时间的范围差不多在54毫秒到105毫秒之间。相比于不可变数据结构版本最快的123毫秒，可变数据结构版本最快的成绩是54毫秒，差不多有230%的性能提升。而对比最长的时间消耗，分别是142毫秒对105毫秒，差不多也有140%的性能提升。

虽然测量结果会随着机器配置、JVM版本以及对垃圾回收算法的调优等因素的不同而不同，但是这些基本的微基准测试已经表明：这与我们所预期的结果相符，可变数据结构的版本通常快于不可变的版本。

Clojure实现

Clojure通过名为暂态的特性从而拥有了对集中的可变性模式的内建支持。暂态让我们有如魔法般地将一个不可变的数据结构转换成一个可变的数据结构。为了使用该特性，我们会将不可变数据结构传入transient!的形式之中。举个例子，通过(def t (transient []))，我们将获得一个暂态的可变vector。

顾名思义，暂态被认为是一种暂时的状态，这与Java中的transient关键词的意义有着很大的差别。暂态在Clojure中的意义是你可以在某个函数中暂时地将它们作为可变结构处理，然后在继续传递它们之前将它们转型回不可变的数据结构。

暂态可以通过conj的特殊版本conj!来完成追加操作。使用惊叹号来标注在可变数据上的操作是一种旧式的Lisp约定，这意味着它正在向你传达：你即将做的一些事情是刺激且危险的！

让我们再来看看上文中那个集中的可变性的基本实例，现在我们已经采用Clojure的暂态来完成了对它的重写。首先，我们需要可变的函数。在Clojure中，我们将通过一个递归函数来完成数字序列的构建，该函数以一个vector作为入参，然后将它贯穿于整个调用链中，并在每一次调用中采用conj来将单个数字追加到vector中。代码如下所示：

ClojureExamples/src/mbfpp/functional/fm/examples.clj

```
(defn test-immutable [count]
  (loop [i 0 s []]
    (if (< i count)
      (recur (inc i) (conj s i))
      s)))
```

下面的可变版本看上去几乎相同；唯一的不同之处在于，我们使用transient创建了一个暂态的vector，它可以在函数内部完成修改操作。当我们处理完修改之后，需要采用persistent!来将它转换回不可变的数据结构，如下面的代码所示：

ClojureExamples/src/mbfpp/functional/fm/examples.clj

```
(defn test-mutable [count]
  (loop [i 0 s (transient [])]
    (if (< i count)
      (recur (inc i) (conj! s i))
      (persistent! s))))
```

最后，我们需要为实例统计时间。Clojure拥有内建的time函数，这与我们在Scala中编写的格外相似，但是我们仍然需要一种能一次运行多个测试的方式。有一些看上去比较神秘的宏能胜任这一场景。如果你对Lisp的宏还不是很熟悉的话，我们将会在模式21中讨论它们。

ClojureExamples/src/mbfpp/functional/fm/examples.clj

```
(defmacro time-runs [fn count]
  `(dotimes [_# ~count]
    (time ~fn)))
```

现在可以对Clojure的解决方案进行测试了。首先是不可变数据结构的版本：

```
=> (time-runs (test-immutable one-million) 5)
"Elapsed time: 112.03 msecs"
"Elapsed time: 114.174 msecs"
"Elapsed time: 117.223 msecs"
"Elapsed time: 114.976 msecs"
"Elapsed time: 300.29 msecs"
```

接下来是可变数据结构部分：

```
=> (time-runs (test-mutable one-million) 5)
"Elapsed time: 84.752 msecs"
"Elapsed time: 73.398 msecs"
"Elapsed time: 196.601 msecs"
"Elapsed time: 70.859 msecs"
"Elapsed time: 70.402 msecs"
```

这些统计时间与Scala的非常相似，但是这并不奇怪，因为Scala的不可变数据结构与Clojure的不可变数据结构都是基于相同的技术实现的。对比两个版本的最短运行时间和最长运行时间，其中可变数据结构的版本差不多有1.5倍的性能提升，这一结果并不算很差。

该例子中值得注意的另一件有趣的事情是其中的两个异常值，不可变版本运行了300.29毫秒，而可变版本运行了196.601毫秒，这与它们各自版本的最快时间相比，都慢了两倍。

如果采用性能分析工具来对这些例子进行一些深入的剖析，你就会发现原来罪魁祸首主要在于垃圾回收器，它正好在这两个样本数据所在的测试过程中有过运行，而在其他的测试中则并没有发生过垃圾回收。像该例子中垃圾回收器所带来的负面作用是可以通过调优来减少的，但是就调优这一部分内容而言，已经够再写一本书了。

范例代码：事件流处理

让我们来看一个重量级的例子。接下来，我们将会处理一个代表购买信息的事件流。每一个事件都包含了一个店铺号码、一个客户号码以及一个商品号码。我们的处理将会非常简单；将事件的流组织成一个以店铺号码为键的map，这样我们便可以根据店铺来对购买信息分组了。

除了处理事件流本身的代码之外，我们还需要一种简单的方式来产生测试数据。为此，我们将使用惰性序列（模式18）来产生一个无限长的购买信息测试数据的序列，从该序列中，我们将获取到任意数量的事件。让我们马上来看一看！

Scala实现

对Scala解决方案的讲解首先从一个名为Purchase的样本类开始，该样本类持有我们的购买信息。除了不可变和可变版本的测试函数之外，我们将会需要一组供测试用的购买信息序列。在这两个版本的代码中，我们都通过for推导来遍历购买信息测试数据，并从中抽取出店铺的号码，然后将它添加到持有该店铺其他购买信息的列表中，后续我们会将它放入一个以店铺号码为键的

map之中。

对于时间的统计，我们将重用与上一个例子相同的代码。让我们从Purchase类开始查看，它就是一个简单的样本类：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/fm/FocusedMutation.scala

```
case class Purchase(storeNumber: Int, customerNumber: Int, itemNumber: Int)
```

我们可以通过一个无限长的惰性序列来产生测试数据，从该序列中可以获取到所需要的任意数量的样本数据。如果你对该细节还不甚了解，那也没有关系，你可以在模式18中找到相关的内容。就当前的例子来说，我们可以通过使用take()从infiniteTestPurchases()产生测试数据了。代码如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/fm/FocusedMutation.scala

```
val r = new Random
def makeTestPurchase = Purchase(r.nextInt(100), r.nextInt(1000), r.nextInt(500))
def infiniteTestPurchases: Stream[Purchase] =
  makeTestPurchase #:: infiniteTestPurchases
```

如果我们想要从无穷序列获取5项内容，可以通过take()来实现，就像下面那样：

```
scala> val fiveTestPurchases = infiniteTestPurchases.take(5)
fiveTestPurchases: ...

scala> for(purchase <- fiveTestPurchases) println(purchase)
Purchase(71,704,442)
Purchase(23,718,87)
Purchase(39,736,3)
Purchase(33,3,233)
Purchase(86,985,152)
```

现在我们已经拥有了产生测试数据的方法，让我们先在不可变方案immutableSequence-EventProcessing()中充分利用一下。该函数接受购买信息测试数据的数量，然后从测试数据的无穷序列中获取购买信息测试数据，并将它们添加到一个如上文所述的根据店铺索引的map之中。

为了将新的购买信息放入map中，我们需要从购买信息中抽取出店铺号码，然后尝试从map中获取该店铺现有的任何购买信息。如果已经存在该店铺的信息，就将新的购买信息添加到现有的信息列表中，并创建一个新的map，该map包含了该键所对应项更新后的内容。代码如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/fm/FocusedMutation.scala

```
def immutableSequenceEventProcessing(count: Int) = {
  val testPurchases = infiniteTestPurchases.take(count)
  var mapOfPurchases = immutable.Map[Int, List[Purchase]]()

  for (purchase <- testPurchases)
    mapOfPurchases.get(purchase.storeNumber) match {
```

```
        case None => mapOfPurchases =
          mapOfPurchases + (purchase.storeNumber -> List(purchase))
        case Some(existing: List[Purchase]) => mapOfPurchases =
          mapOfPurchases + (purchase.storeNumber -> (purchase :: existing))
      }
    }
```

可变版本与不可变版本非常相似，只不过我们在完成对可变map的修改之后会将它转换成一个不可变的map，代码如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/fm/FocusedMutation.scala

```
def mutableSequenceEventProcessing(count: Int) = {
  val testPurchases = infiniteTestPurchases.take(count)
  val mapOfPurchases = mutable.Map[Int, List[Purchase]]()

  for (purchase <- testPurchases)
    mapOfPurchases.get(purchase.storeNumber) match {
      case None => mapOfPurchases.put(purchase.storeNumber, List(purchase))
      case Some(existing: List[Purchase]) =>
        mapOfPurchases.put(purchase.storeNumber, (purchase :: existing))
    }
  mapOfPurchases.toMap
}
```

那么我们将如何执行这两个解决方案呢？我们会通过它们来运行50万的样本数据，先从不可变的版本开始：

```
scala> timeRuns(immutableSequenceEventProcessing(fiveHundredThousand), 5)
Elapsed time: 647.948 msecs
Elapsed time: 523.477 msecs
Elapsed time: 551.897 msecs
Elapsed time: 505.083 msecs
Elapsed time: 538.568 msecs
```

接下来是可变版本：

```
scala> timeRuns(mutableSequenceEventProcessing(fiveHundredThousand), 5)
Elapsed time: 584.002 msecs
Elapsed time: 283.623 msecs
Elapsed time: 546.839 msecs
Elapsed time: 286.259 msecs
Elapsed time: 568.298 msecs
```

正如我们所看到的，可变版本仅有稍许性能上的提升。通过一些评测不难发现，之所以出现这一现象，主要原因在于该例子都将时间花费在了测试数据的生成上面，而非对map的处理上。

如果这些事件是从文件系统或网络上读取的，那么花费在上面的开销甚至会更大，而两个解决方案之间的差距则会更小。另一方面，即使每个事件处理所损耗的时间非常少，但是如果数据集足够的大话，最后也会成为要紧的问题。

Clojure实现

我们的Clojure解决方案与Scala的非常相似。与Scala一样，我们将会使用惰性序列来生成购买信息的无穷序列，并采用一个有穷的数字来获取所需的测试数据。我们将会考察测试函数的两组实现。其中第一个实现采用了标准的不可变map，而第二个则使用了一个可变的暂态形式的map。

先来看生成测试数据的代码。我们可以使用一个名为repeatedly的函数，顾名思义，它可以多次调用函数，然后便可以使用这些结果来创建惰性序列了。除此之外，我们只需要一个能自己创建购买信息测试数据的函数。下面是这部分内容的代码：

ClojureExamples/src/mbfpp/functional/fm/examples.clj

```
(defn make-test-purchase []
  {:store-number (rand-int 100)
   :customer-number (rand-int 100)
   :item-number (rand-int 500)})
(defn infinite-test-purchases []
  (repeatedly make-test-purchase))
```

现在我们需要祭出测试函数了。我们将会使用reduce来将一组购买信息的序列转换成一个按照店铺号码索引的map。与Scala的例子一样，我们将使用take来从购买信息的无穷序列中取出一组有限数量的测试数据。接着，在序列之上执行reduce操作，从而构建出以店铺号码索引的购买信息的map。

与之前一样，我们需要处理首次遇到一个店铺号码的场景，这一次我们可以传入一个默认的空列表来供我们自己获取。代码如下所示：

ClojureExamples/src/mbfpp/functional/fm/examples.clj

```
(defn immutable-sequence-event-processing [count]
  (let [test-purchases (take count (infinite-test-purchases))]
    (reduce
      (fn [map-of-purchases {:keys [store-number] :as current-purchase}]
        (let [purchases-for-store (get map-of-purchases store-number '())]
          (assoc map-of-purchases store-number
                 (conj purchases-for-store current-purchase))))
      {}
      test-purchases)))
```

由于Clojure拥有便携的暂态特性，所以可变的解决方案看上去会非常相似，除此之外，我们需要将map转型成暂态再转回来。对于暂态，我们需要使用assoc!来向其添加元素，如下面的代码所示：

ClojureExamples/src/mbfpp/functional/fm/examples.clj

```
(defn mutable-sequence-event-processing [count]
  (let [test-purchases (take count (infinite-test-purchases))]
  (persistent! (reduce
    (fn [map-of-purchases {:keys [store-number] :as current-purchase}]
      (let [purchases-for-store (get map-of-purchases store-number '())]
```

```
        (assoc! map-of-purchases store-number
                (conj purchases-for-store current-purchase))))
    (transient {})
    test-purchases))))
```

现在让我们来启动测试，从可变版本开始：

```
=> (time-runs (mutable-sequence-event-processing five-hundred-thousand) 5)
"Elapsed time: 445.841 msecs"
"Elapsed time: 457.66 msecs"
"Elapsed time: 452.743 msecs"
"Elapsed time: 374.041 msecs"
"Elapsed time: 403.498 msecs"
nil
```

接着是不可变的版本：

```
=> (time-runs (immutable-sequence-event-processing five-hundred-thousand) 5)
"Elapsed time: 481.547 msecs"
"Elapsed time: 413.121 msecs"
"Elapsed time: 460.379 msecs"
"Elapsed time: 441.686 msecs"
"Elapsed time: 445.772 msecs"
nil
```

正如我们所看到的，它们之间的差别非常之小，但是可变版本在速度上表现得略微出色一些。

讨论

集中的可变性是一个优化型的模式。前辈们通常会建议我们尽可能避免过早的优化。正如我们在本节中所看到的，Scala和Clojure的不可变数据结构的性能表现是非常不错的。相比它们所对应的可变数据结构并没有太大的差距！如果你一下子修改多个不可变的数据结构并且需要处理大量的数据，那么你将可能看到可变数据结构所带来的性能上的显著提升。然而，不可变数据结构应该是默认的选择——因为它们通常都足够快了。

在使用集中的可变性模式或任何小规模的性能优化之前，最好先对你的应用进行一次性能分析，以确保你的优化的方向是对的。否则，你可能会发现花了大量时间优化的一段代码其实很少被调用到，即它对整个程序的性能影响微乎其微。

模式 20

自定义控制流

目的

创建集中的自定义控制流抽象。

概述

通过使用正确的控制流抽象来完成任务，可以帮助我们编写出更加整洁的代码。举个例子，Ruby拥有一个unless操作符，它表示除非某个条件为真，否则就做某事。良好的Ruby代码会更多地使用unless操作符来替代if和not操作符，因为它阅读起来更加清晰。

然而，没有一种语言能拥有对所有有用控制流抽象的内建支持。函数式编程语言赋予了我们一种使用高阶函数来创建自己的控制流抽象的方式。举个例子，为了创建一个能对某段代码执行*n*次并打印出其平均运行时间的控制流结构，我们可以编写一个以另一个函数作为入参的函数，然后对传入的函数执行*n*次调用。

但是，仅仅利用高阶函数会让我们用以创建自定义控制流的语法显得非常啰嗦。我们可以做的更好。在Clojure中，我们可以使用宏系统，而在Scala中，我们有着囊括了各种技巧的百宝袋，包括构造块和按名称传递参数。

范例代码：三者选一

让我们先来看一个基本的自定义控制结构：choose，它可以在三个不同选项中选择一项。我们将会探索两种不同的实现：第一种实现采用了高阶函数，而第二种实现会通过添加一些语法糖对第一种实现进行改进。

Scala实现

我们的choose()函数接受一个1到3之间的整数以及三个函数作为入参。然后它会根据入参选择相对应的函数进行执行，如下面的代码所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/ccf/Choose.scala

```
def choose[E](num: Int, first: () => E, second: () => E, third: () => E) =
  if (num == 1) first()
  else if (num == 2) second()
  else if (num == 3) third()
```

这是对高阶函数的简单运用。让我们来看看如何使用该函数：

```
scala> simplerChoose(2,
     | () => println("hello, world"),
     | () => println("goodbye, cruel world"),
     | () => println("meh, indifferent world"))
goodbye, cruel world
```

该函数的运行结果符合我们的预期；然而，这种做法需要我们将行为包装在一个函数之中，这是相当麻烦的事情。我们希望能有一种更好的语法来实现这一功能，即可以直接向choose()函数传入普通的表达式，就像下面所想象的REPL会话一样：

```
scala> simplerChoose(2,
     | println("hello, world"),
```

```
     | println("goodbye, cruel world"),
     | println("meh, indifferent world"))
goodbye, cruel world
```

让我们来看看如何让这一语法梦想成真，先从一个简单的例子开始。在下面的REPL输出中，我们定义了一个只有一个入参expression的test()函数。该函数主体的职责就是负责尝试执行该表达式。我们可以以println("hello, world")作为入参来调用test()函数。

```
scala> def test[E](expression: E) = expression
test: (expression: Unit)Unit

scala> test(println("hello, world"))
hello, world
```

看上去似乎已经凑效了，鉴于"hello, world"已经输出到了控制台，表明我们的表达式已经完成了求值。但是，如果我们尝试对表达式执行两次会发生什么呢？让我们从下面的REPL片段中来寻找答案：

```
scala> def testTwice[E](expression: E) = {
     | expression
     | expression
     | }
testTwice: (expression: Unit)Unit

scala> testTwice(println("hello, world"))
hello, world
```

字符串"hello, world"仅在控制台打印了一次！这是因为Scala会默认将表达式传入函数时对表达式进行求值，然后将表达式求值后的结果传入函数。我们将这种形式称为按值传递（pass by value），而这种形式也通常是我们所希望及想要的。举个例子，在下面的实例中，这种形式可以防止表达式被求值两次：

```
scala> def printTwice[E](expression: E) = {
     | println(expression)
     | println(expression)
     | }
printTwice: [E](expression: E)Unit

scala> printTwice(5 * 5)
25
25
```

然而，这与我们需要用来编写控制流结构的需求却是背道而驰的。Scala赋予了我们一种可选的调用语义，称之为按名称传递（pass by name）。使用按名称传递意味着我们会将表达式的名称传入函数，而不会对表达式进行求值。然后根据需要，随时通过在函数体中引用该名称来完成表达式的求值。

为了让函数以按名称传递的方式进行工作，而不再按值传递，我们可以在参数名之后类型标注之前使用=>。下面的REPL片段便采用按名称传递的调用对测试函数进行了重写：

```
scala> def testByName[E](expression: => E) {
     | expression
     | expression
     | }
testByName: [E](expression: => E)Unit

scala> testByName(println("hello, world"))
hello, world
hello, world
```

现在我们了解了按值传递和按名称传递的区别，接下来编写一个以普通表达式作为入参的simplerChoose()函数。如下面的代码片段所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/ccf/Choose.scala

```
def simplerChoose[E](num: Int, first: => E, second: => E, third: => E) =
  if (num == 1) first
  else if (num == 2) second
  else if (num == 3) third
```

现在我们可以使用支持普通表达式的语法了，如下面的REPL输出所示：

```
scala> simplerChoose(2,
     | println("hello, world"),
     | println("goodbye, cruel world"),
     | println("meh, indifferent world"))
goodbye, cruel world
```

Clojure对于创建自定义控制流的方式有着很大的不同，同时还涉及了它强大的宏系统。让我们来一探究竟！

Clojure实现

现在开始讨论Clojure实例，先来看一个choose的简单版本，该版本依赖于高阶函数。我们输入三个函数以及一个表明接下来该执行哪一个函数的整数，如下面的代码所示：

ClojureExamples/src/mbfpp/functional/ccf/ccf_examples.clj

```
(defn choose [num first second third]
  (cond
    (= 1 num) (first)
    (= 2 num) (second)
    (= 3 num) (third)))
```

为了使用该函数，我们传入对应入参的整数及函数：

```
=> (choose 2
           (fn [] (println "hello, world"))
           (fn [] (println "goodbye, cruel world"))
           (fn [] (println "meh, indifferent world")))
goodbye, cruel world
nil
```

然而，我们希望能避免将行为包装在函数之中，而是像如下REPL会话这样来编写代码：

```
=> (choose 2
           (println "hello, world")
           (println "goodbye, cruel world")
           (println "meh, indifferent world"))
goodbye, cruel world
nil
```

为了理解如何来实现这一目标，我们会先进行一个短暂的有关Clojure最强大特性的补充介绍，即它的宏系统。在这个过程中，我们将会回答为何Lisp会拥有如此与众不同的语法这一古老的问题。

Clojure的宏

宏是元编程（metaprogramming）的一种形式：它们是一些可以转型成其他代码片段的代码片段。这一概念在Clojure和其他的Lisp语言中已经深入骨髓。

为了理解其中的原因，让我们来做一个思维实验。由不可变对象来替代生成器模式（模式4）中所提到的生成器编写起来相当的啰嗦。一种可以减少冗长代码的方式就是创建一个作为骨架的Java类，这些Java类只拥有一些属性，然后我们来编写代码从而生成基于这些属性的生成器。

该方式的流程图描述如下：

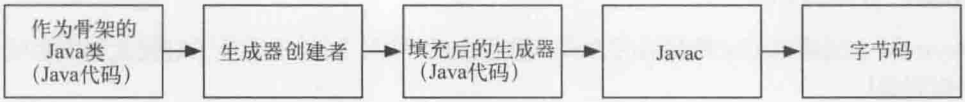


图4-6　Java中的元编程 。用代码来生成一个生成器类

图中提到的生成器创建者是一段代码，它负责基于作为骨架的Java类来生产出生成器，该Java类除了属性之外什么都没有。这跟IDE中对生成getter和setter方法的支持非常相似。

为了实现这一点，生成器创建者需要对输入的Java代码有一些了解。对于这么简单的一个任务，生成器创建者完全可以将文件作为文本处理，并逐行地读入该文件，然后识别出哪些行是与变量声明相关的。

然而，如果我们需要以一种更加复杂的方式来处理输入的Java代码该怎么办呢？比如，我们想要对某些方法进行修改，从而能通过日志记录下它们的调用时间。这样处理起来就比较困难了：如果我们逐行来扫描文件的话，如何才能知道某个方法的起始位置和结束位置呢？

困难之处在于我们的简易代码生成器是将Java文件作为普通文本来处理的。像编译器这样复杂的语言应用会对文件进行一系列的处理从而生成一棵抽象语法树，或简称AST。下面的流程图是对该过程的一个简化表示。

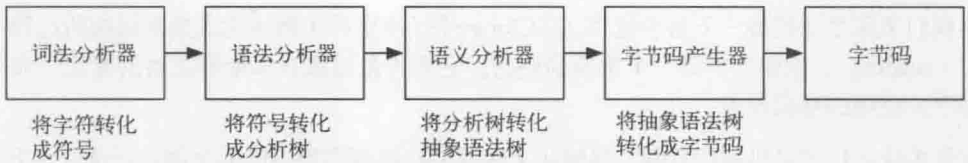


图4-7 简化后的编译器。简化后编译器的各个阶段

AST会按照方法和类这样的概念来进行处理，并以一种更加抽象的层次来表示代码，而不是只将代码视为简单的文本数据。举个例子，除了其他内容之外，一个以Java编写的Java编译器还会将方法和变量定义的类作为它AST的一部分。

这使AST成为了可供我们以编程方式处理代码的最方便的表示形式。然而，在大多数编程语言中，AST都被隐藏在编译器之后，并且需要具备一定的编译器知识才能对它进行处理。

Lisp，包括Clojure，绝对是与众不同的。Clojure的语法是按照Clojure的核心数据结构进行定义的，如列表和向量。作为实例，让我们来进一步看一个简易的函数定义：

```
(defn say-hello [name] (println (str "Hello, " name)))
```

这是一个拥有四个元素的列表。第一个元素是符号`defn`，第二个元素是符号`say-hello`，第三个元素是一个向量，第四个元素则是另一个列表。当Clojure对该列表进行求值的时候，它会认为第一个元素是某项像函数、宏或内建编译器这样可供调用的东西，而列表中的后续元素会被认为是参数。

除此之外，它只不过是一个跟其他任何列表一样的列表！我们可以看到，在后面的列表之前使用了一个单引号，它可以用来关闭Clojure原本用于列表的求值形式。在下面的片段中，我们分别从两个列表中取出了第一个元素。其中第一个列表是拥有四个整数的列表，而第二个则是我们刚才介绍的函数定义：

```
=> (first '(1 2 3 4))
1
=> (first '(defn say-hello [name] (println (str "Hello, " name))))
defn
```

因为Clojure的代码就是Clojure的数据，所以我们很容易就能编写代码来操作它。Clojure的宏系统为我们在编译时完成这样的操作提供了方便的钩子（hook），如图4-8所示。

图4-8 读入，求值，编译。从Clojure文本到字节码的转换过程

让我们来深度地挖掘一下这个流程。在Clojure中，从字符序列转化成数据结构的过程被称为“读入”（reading），正如图中第一个框所描述的。它不再是隐藏在编译器之后的魔法，而是一套可为程序员使用的基础设施。

有很多读入形式可供我们选择，例如从文件或字符串等多种不同的来源进行读入。下面，我们使用读入的字符串版本从字符串中读取一个向量，并取出它的第一个元素：

```
=> (first (read-string "[1 2 3]"))
1
```

读入拥有一个好搭档：eval，如图4-8中的第二步所示。它以数据结构作为输入，然后根据一套我们曾在2.4节中讨论过的简单的求值规则来对它进行求值。在下面的代码片段中，我们从字符串读入之后，采用了eval对其中的定义进行求值：

```
=> (eval (read-string "(def foo 1)"))
#'user/foo
=> foo
1
```

你或许曾在像Ruby或JavaScript这样的语言中见到过eval；然而，那些eval与Clojure中的eval有着一个重大的差别。在Clojure和其他Lisp语言中，eval可以应用于读入的数据结构之上，而不仅仅是字符串。

由于我们不一定使用原始的字符串操作来构建代码，这意味着我们将可以完成更多复杂的操作。而图4-8中所描述的宏展开这一步骤为我们这么做提供了一个方便的钩子。

宏除了几个关键的不同点之外，也算得上是一个函数。传递给宏的参数是不会被求值的，这就跟我们在Scala中所使用的按名称调用的参数差不多。宏是在编译之前运行的，它会返回需要编译的代码。这给了我们一种正式且内建的方式来进行在Java生成器产生器思维实验中所介绍的这一类操作。

我们采用内建的defmacro定义了一个宏。除此之外，一些其他的Clojure特性可以通过对求值的控制来帮助我们构建宏。这些特性包括反引号“`”和波浪符号“~”，我们分别称之为语法引述（syntax quote）和反引述（unquote）。

结合这些特性，我们便可以在宏之中构建出可供使用的代码模板了。语法引述关闭了它所应用的形式内的求值，并可以根据当前的命名空间将任何符号名称展开为完全限定的形式。反引述，顾名思义，让我们可以在语法引述之上打开求值。

在下面的片段中，我们结合使用了语法引述和反引述。符号foo和形式(+ 1 1)将不会被求值，但是当我们在number-one上应用反引述的时候，它便会被求值。

```
=> (def number-one 1)
#'mbfpp.functional.ccf.ccf-examples/number-one
=> `(foo (+ 1 1) ~number-one)
(mbfpp.functional.ccf.cff-examples/foo (clojure.core/+ 1 1) 1)
```

该输出看上去有一点杂乱，这是因为语法引述的foo和+是由命名空间限定的。

现在我们已经介绍了宏，让我们来看看如何使用它们来简化choose实例。我们编写了一个宏：simplerChoose。该simplerChoose宏接受一个数字和三个形式，然后返回一个cond表达式，该表达式会选择相应的形式进行求值。simplerChoose的代码如下面的片段所示：

ClojureExamples/src/mbfpp/functional/ccf/ccf_examples.clj

```
(defmacro simpler-choose [num first second third]
  `(cond
    (= 1 ~num) ~first
    (= 2 ~num) ~second
    (= 3 ~num) ~third))
```

在运行它之前，我们可以使用macroexpand-1来看看该宏所生成的代码，如下面的REPL会话所示：

```
=> (macroexpand-1
    '(simpler-choose 1 (println "foo") (println "bar") (println "baz")))
(clojure.core/cond
  (clojure.core/= 1 1) (println "foo")
  (clojure.core/= 2 1) (println "bar")
  (clojure.core/= 3 1) (println "baz"))
```

我们可以看到，正如我们所预期的，宏展开成为一个cond语句。现在如果运行它，它将会按我们预期的方式进行工作，无需再将行为包装进函数之中！

```
=> (simpler-choose 2
                   (println "hello, world")
                   (println "goodbye, cruel world")
                   (println "meh, indifferent world"))
goodbye, cruel world
nil
```

Clojure的宏系统是它最强大的特性之一，它同时也对Clojure为何具有这样的语法进行了说明。为了让魔法能够成真，Clojure的代码需要根据简单的Clojure数据结构来进行编写，这一属性我们称之为“同像”（homoiconicity）。

范例代码：平均时间

让我们来看一个关于自定义控制流的更加复杂的例子。接下来我们将创建一个自定义控制抽象，它会以给定数字作为次数来对某个表达式执行多次求值，然后返回执行的平均时间。这对快速随性的性能测试来说是非常有用的。

Scala实现

在Scala中，我们的解决方案有两个函数。第一个是timeRun()，它接受一个表达式并运行它，然后返回它所消耗的时间。第二个是avgTime()，它接受一个表达式和一个表示求值次数的数字，

然后返回消耗的平均时间。它将timeRun()作为帮助函数使用。

Scala解决方案的代码如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/ccf/Choose.scala

```
def timeRun[E](toTime: => E) = {
  val start = System.currentTimeMillis
  toTime
  System.currentTimeMillis - start
}
def avgTime[E](times: Int, toTime: => E) = {
  val allTimes = for (_ <- Range(0, times)) yield timeRun(toTime)
  allTimes.sum / times
}
```

正如该函数所描述的，它给了我们一种方式来获取语句的平均运行时间：

```
scala> avgTime(5, Thread.sleep(1000))
res0: Long = 1001
```

让我们使用REPL来对该方案进行拆解。函数avgTime()的主体代码便是下面的表达式：

```
val allTimes = for (_ <- Range(0, times)) yield timeRun(toTime)
```

如果手动来替换其中的一些表达式，我们将看到它产生了一个运行时间的序列。位于for绑定中的下划线表明，我们无需关心当前绑定了Range表达式中的什么值，因为我们只需将它作为运行次数来运行我们的语句：

```
scala> val allTimes = for (_ <- Range(0, 5)) yield timeRun(Thread.sleep(1000))
allTimes: scala.collection.immutable.IndexedSeq[Long] =
        Vector(1000, 1001, 1000, 1001, 1001)
```

到此为止，我们可以使用sum()来对所有的运行时间求和了，然后将它除以运行的次数，从而获得平均时间：

```
scala> allTimes.sum / 5
res2: Long = 1000
```

该解决方案中还有另一个有意思的地方，即我们是如何将一个参数以按名称传递的方式来穿越两个函数的。其中参数toTime被传递给avgTime()函数，然后又从那里传递到了timeRun()函数。直到在timeRun()中被使用之前，它一直都没有被求值。

使用按名称传递的参数来将调用串联起来是一个非常重要的特性，因为它可以让我们将代码拆解成更加复杂的自定义控制流实例。

Clojure实现

在Clojure中，我们的解决方案由一个宏avg-time和一个函数time-run组成。宏avg-time产生的代码使用了函数time-run来统计传入语句的运行时间，然后计算出它们的平均值。

Clojure解决方案的代码如下所示：

ClojureExamples/src/mbfpp/functional/ccf/ccf_examples.clj

```
(defn time-run [to-time]
  (let [start (System/currentTimeMillis)]
    (to-time)
    (- (System/currentTimeMillis) start)))

(defmacro avg-time [times to-time]
  `(let [total-time#
         (apply + (for [_# (range ~times)] (time-run (fn [] ~to-time))))]
     (float (/ total-time# ~times))))
```

下面，我们使用它来计算一个测试语句的平均运行时间：

```
=> (avg-time 5 (Thread/sleep 1000))
1000.8
```

我们来更深入地看看time-run的工作细节。先来介绍该例子中所使用的一个Clojure特性：自动生成符号，或简称为gensym。为了避免在宏中出现意外的变量名冲突，当我们需要在生成的代码中获取一个符号时，通常需要生成一个唯一的符号。

在Clojure中，当我们在语法引述中使用符号的时候，可以通过在符号名之后添加一个#标记来实现。如下面的片段所示，Clojure会将gensym展开为一个绝对唯一的符号：

```
=> `(foo# foo#)
(foo__2230__auto__ foo__2230__auto__)
```

在求平均时间的例子中，我们对total-time和_都使用了gensym。其中第二项可能看上去有一些另类，因为使用下划线可以表明我们无需关心范围中的值，这与我们在Scala中所做的是一样的。

如果我们不使用自动产生的符号，那么Clojure将会在当前命名空间中对符号进行限定，但是符号一旦经由gensym修饰便可以让Clojure为它生成一个唯一的符号。我们描述如下：

```
=> `(_)
(mbfpp.functional.ccf.ccf-examples/_)
=> `(-#)
(-__1308__auto__)
```

现在来看看avg-time的核心代码。下面由语法引述修饰的let语句将作为宏用以生成代码的模板。正如我们所看到的，该方案的主体是一个for语句，它将to-time中的表达式包装进一个函数之中，然后通过time-run来调用该函数，并使它运行请求所指定的次数：

```
`(let [total-time#
   (apply + (for [_# (range ~times)]
              (time-run (fn [] ~to-time))))] (float (/ total-time# ~times))))
```

为了验证我们的想法，可以使用macroexpand-1来看看该宏所生成的代码，如下面的REPL会话所示：

```
=> (macroexpand-1 '(avg-time 5 (Thread/sleep 100)))
```

```
(clojure.core/let
 [total-time__1489__auto__
  (clojure.core/apply
   clojure.core/+
   (clojure.core/for
    [___1490__auto__ (clojure.core/range 5)]
    (mbfpp.functional.ccf.cff-examples/time-run
     (clojure.core/fn [] (Thread/sleep 1000)))))]
 (clojure.core/float (clojure.core// total-time__1489__auto__ 5)))
nil
```

由于所有的符号要么是由gensym自动生成的，要么是由命名空间进行完全限定的，所以上面的代码有一点难以阅读。如果我对理解某个宏是如何工作的存在疑问，就会手动将`macroexpand-1`的输出转化成原本需要手动编写的代码。为了实现这一目的，你通常只需要将完全限定符号的命名空间和gensym生成的部分去掉。在下面的代码中我已经完成了这一工作：

```
(let
  [total-time
   (apply + (for [n (range 5)] (time-run (fn [] (Thread/sleep 100)))))]
  (float (/ total-time 5)))
```

正如你所看到的，上面较为清晰的输出更容易为我们所理解。我同样还发现，通过经历手动处理生成代码这一过程，可以帮助你发现宏中潜藏的任何bug。

讨论

Scala和Clojure都可以让我们创建自定义控制流抽象，但是它们所使用的方式却迥然不同。在Scala中，它们都是运行时的抽象。我们只需要编写函数，然后向它们传递语句。其中的技巧就是通过使用按名称传递参数来控制这些语句的求值时间。

在Clojure中，我们使用了宏系统，它利用了Clojure的同像特性。宏的关注点在于编译时而非运行时。正如我们所看到的，宏让我们可以非常简单地编写代码，这些代码可以通过将语法引述所修饰的代码作为模板来生成我们想要的代码。

Clojure的方式更加的通用，但这仅仅是因为Clojure拥有同像这一特性成为可能。为了在一门像Scala这样的非同像的语言中近似地模拟Clojure风格的宏，该语言需要为程序员提供可渗透进编译器的钩子，从而让程序员可以操纵AST以及其他的编译器功能。

这是一个困难的任务，但是Scala在2.10版中提供了对这一类编译时宏的实验性支持。使用这一风格的宏比使用Clojure风格的宏来的更加困难，因为它需要你掌握一些编译器内部的知识。

鉴于Scala的宏是实验性质的，并且Scala为实现自定义控制流提供了其他方式，所以本节将不再涉及这一部分的内容。

模式21

领域特定语言

目的

创建一门专门用于解决某个特定问题的小型编程语言。

概述

领域特定语言是一个非常常见的模式，它分为两个比较宽泛的大类：外部DSL和内部DSL。

外部DSL是一门成熟的编程语言，它拥有自己的语法和编译器。它的目标并不在于通用；相反，它通常只解决某些有针对性的问题。举个例子，SQL便是领域特定语言的一个实例，它专注于数据处理。另外还有ANTLR，它专注于创建解析器。

另一方面，我们也拥有一些内部的DSL，通常称之为“内嵌语言”（embedded language）。反观该模式的一些例子，往往构建在某些通用语言之上，并且其存在形式也受限于宿主语言的语法约束。

对于这两类DSL而言，它们的目的是一样的。我们正在尝试创建这样一门语言，以一种更贴近领域的方式来表达处理问题的解决方案。相比那些通用的语言，采用DSL可以以更少的代码来表达更清晰的解决方案。它同样还能让那些非软件开发者参与到某些领域问题的解决方案中来。

在本节，我们将来看看如何采用Scala和Clojure来构建内部的DSL。在使用这两门语言构建DSL的过程中，我们所采用的技术会截然不同，但它们的目标是一致的。

Scala实现

当前基于Scala的DSL大都依赖于Scala灵活的语法以及其他一些Scala的奇技淫巧。我们在本节中所探讨的DSL将会利用到多个Scala的高级特性。

首先，我们将会看到Scala的一个能力，即在后缀和中缀位置使用方法。这让我们可以像定义操作符那样来定义方法。

其次，使用模式10中所介绍的Scala的隐式转换。这项特性可以让我们为现有的类型添加新的行为。

最后，使用一个Scala的伴生对象来作为它对应的类的工厂。

Clojure实现

内部DSL是Clojure所传承的一项古老的Lisp技术。在Clojure和其他的Lisp语言中，领域特定

语言和框架或API之间的界限是非常模糊的。

良好的Clojure代码通常都会组织成多个DSL层，一层叠着一层，每一层都善于解决系统中特定层面的某个问题。

举个例子，在Clojure中，如果要设计一个分层系统来构建Web应用，那么首先可能会选择一个名为Ring的库。该库提供了一个基于HTTP的抽象，它会将HTTP请求转换成Clojure的map。在此之上，我们可以使用一个名为Compojure的DSL来将HTTP请求路由到相应的处理函数。最后，使用一个名为Enlive的DSL来为我们的页面创建模板。

Clojure的DSL通常围绕着一组核心的高阶函数构建而成，并通过宏为上层提供语法糖。这便是我们在本节中用来构建Clojure DSL的方式。

范例代码：为shell而准备的DSL

我发现自己经常会在使用Scala和Clojure编程的时候从shell中剪切命令，然后粘贴到REPL中。来看一个简单的DSL，它可以通过让我们直接在REPL中运行shell命令使这一过程变得更加自然。

除了运行命令之外，我们还希望能捕获到这些命令的退出状态、标准输出流以及标准错误流。最后，我们将会让这些命令以管道的方式串接起来，就像在普通的shell中所做的那样。

Scala实现

该例子的最终目标是让我们能在Scala的REPL中以一种自然的方式运行shell命令。对于每个独立的命令，可以像下面这样运行它们：

```
scala> "ls" run
```

并且我们希望像下面这样将命令以管道的方式串接起来：

```
scala> "ls" pipe "grep some-file" run
```

让我们迈出实现shell DSL旅程的第一步，首先来看命令在运行完之后应该返回给我们什么信息。我们需要能够检查一个shell命令的状态码，以及它的标准输出流和标准错误流。在下面的代码中，我们将这些信息打包进了一个名为CommandResult的样本类中：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/dsl/Example.scala

```
case class CommandResult(status: Int, output: String, error: String)
```

现在来看看如何真正地去运行一个命令。为此，让我们稍加研究一下Java的ProcessBuilder类。

类ProcessBuilder的构造器接受一组不定数量的字符串参数，这些参数表示需要运行的命令及其参数。在下面的REPL片段中，我们创建了一个用于运行ls -la的ProcessBuilder：

```
scala> val lsProcessBuilder = new ProcessBuilder("ls", "-la")
lsProcessBuilder: ProcessBuilder = java.lang.ProcessBuilder@5674c175
```

我们可以调用刚创建的ProcessBuilder上的start()方法来运行该进程。它会返回一个Process对象，作为我们引用该运行进程的句柄：

```
scala> val lsProcess = lsProcessBuilder.start
lsProcess: Process = java.lang.UNIXProcess@61a7c7e7
```

该Process对象让我们得以访问到所有需要的信息，但是来自标准输出流和标准错误流的输出内容却位于InputStream对象内，而不是在字符串中。我们可以在Scala的Source对象上使用fromInputStream()来将它们剥离出来，如下面的代码所示：

```
scala> Source.fromInputStream(lsProcess.getInputStream()).mkString("")
res0: String =
"total 96
drwxr-xr-x  12 mblinn  staff    408 Mar 17 10:23 .
drwxr-xr-x   8 mblinn  staff    272 Apr  6 15:12 ..
-rw-r--r--   1 mblinn  staff  35583 Jun  9 16:35 .cache
-rw-r--r--   1 mblinn  staff   1200 Mar 17 10:10 .classpath
-rw-r--r--   1 mblinn  staff    328 Mar 17 10:08 .project
drwxr-xr-x   3 mblinn  staff    102 Mar 16 13:29 .settings
drwxr-xr-x   9 mblinn  staff    306 Jun  9 15:58 .svn
drwxr-xr-x   2 mblinn  staff     68 Mar 13 20:34 bin
-rw-r--r--   1 mblinn  staff    262 Jun  9 13:12 build.sbt
drwxr-xr-x   6 mblinn  staff    204 Mar 13 20:33 project
drwxr-xr-x   5 mblinn  staff    170 Mar 13 19:52 src
drwxr-xr-x   6 mblinn  staff    204 Mar 16 13:33 target
"
```

请注意，这个让我们从标准输出中获得输出信息的方法叫作getInputStream()，这未免会让人感到困惑，难道不应该是getOutputStream()吗？这并不是一个拼写错误；该方法的名字是为了表达这样一个事实：标准输出被写入到一个Java的InputStream，这样调用它的代码才能消费这些输出内容。

现在我们可以将Command类结合起来使用了。Command接受一组字符串，这些字符串代表了命令本身及其参数，我们使用该命令来构造我们的ProcessBuilder。接下来，运行该进程，并等待它结束，然后剥离出结束进程的输出流和状态码。下面是该Command的实现代码：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/dsl/Example.scala

```
class Command(commandParts: List[String]) {
  def run() = {
    val processBuilder = new ProcessBuilder(commandParts)
    val process = processBuilder.start()
    val status = process.waitFor()
    val outputAsString =
      Source.fromInputStream(process.getInputStream()).mkString("")
    val errorAsString =
      Source.fromInputStream(process.getErrorStream()).mkString("")
    CommandResult(status, outputAsString, errorAsString)
  }
}
```

为了让Command类更易于构建，我们添加了一个工厂方法，该方法接受一个字符串，并对它进行分割，然后把它组装进一个Command的伴生对象中：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/dsl/Example.scala

```
object Command {
  def apply(commandString: String) = new Command(commandString.split("\\s").toList)
}
```

正如下面的REPL会话所展示的，这让我们离所期望的用以运行单个命令的语法又更近了一步：

```
scala> Command("ls -la").run
res1: com.mblinn.mbfpp.functional.dsl.ExtendedExample.CommandResult =
CommandResult(0,total 96
drwxr-xr-x  12 mblinn  staff    408 Mar 17 10:23 .
drwxr-xr-x   8 mblinn  staff    272 Apr  6 15:12 ..
-rw-r--r--   1 mblinn  staff  35592 Jun  9 16:57 .cache
-rw-r--r--   1 mblinn  staff   1200 Mar 17 10:10 .classpath
-rw-r--r--   1 mblinn  staff    328 Mar 17 10:08 .project
drwxr-xr-x   3 mblinn  staff    102 Mar 16 13:29 .settings
drwxr-xr-x   9 mblinn  staff    306 Jun  9 15:58 .svn
drwxr-xr-x   2 mblinn  staff     68 Mar 13 20:34 bin
-rw-r--r--   1 mblinn  staff    262 Jun  9 13:12 build.sbt
drwxr-xr-x   6 mblinn  staff    204 Mar 13 20:33 project
drwxr-xr-x   5 mblinn  staff    170 Mar 13 19:52 src
drwxr-xr-x   6 mblinn  staff    204 Mar 16 13:33 target
,)
```

为了走完剩下的路，我们将使用模式10中所介绍的隐式转换。我们将会创建一个隐式转换来将字符串转换成一个拥有run()方法的CommandString。该CommandString可以将它所转换的字符串传入Command，并调用它的run()方法。它的实现如下面的代码所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/dsl/Example.scala

```
implicit class CommandString(commandString: String) {
  def run() = Command(commandString).run
}
```

现在我们已经获得了希望用来运行单个命令的语法，如我们在下面的REPL输出中所展示的：

```
scala> "ls -la" run
res2: com.mblinn.mbfpp.functional.dsl.ExtendedExample.CommandResult =
CommandResult(0,total 96
drwxr-xr-x  12 mblinn  staff    408 Mar 17 10:23 .
drwxr-xr-x   8 mblinn  staff    272 Apr  6 15:12 ..
-rw-r--r--   1 mblinn  staff  35592 Jun  9 16:57 .cache
-rw-r--r--   1 mblinn  staff   1200 Mar 17 10:10 .classpath
-rw-r--r--   1 mblinn  staff    328 Mar 17 10:08 .project
drwxr-xr-x   3 mblinn  staff    102 Mar 16 13:29 .settings
drwxr-xr-x   9 mblinn  staff    306 Jun  9 15:58 .svn
drwxr-xr-x   2 mblinn  staff     68 Mar 13 20:34 bin
-rw-r--r--   1 mblinn  staff    262 Jun  9 13:12 build.sbt
```

```
drwxr-xr-x   6 mblinn  staff    204 Mar 13 20:33 project
drwxr-xr-x   5 mblinn  staff    170 Mar 13 19:52 src
drwxr-xr-x   6 mblinn  staff    204 Mar 16 13:33 target
,)
```

让我们来对DSL进行扩展，使之支持管道的功能。所使用的方法是将所有管道化的命令字符串收集到一个向量之中，并在构建完整个管道链之后运行它们。

先来审视一下需要对CommandString做出哪些扩展。记住，我们希望按下面的方式来运行命令管道："ls -la" pipe "grep build" run。这意味着我们需要添加一个pipe()方法，该方法接受一个字符串参数，并用于CommandString的隐式转换。当它被调用的时候，它会将那个被转换成CommandString的字符串以及传入pipe()方法的字符串参数填充到一个向量之中。扩展后的CommandString的代码如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/dsl/Example.scala

```
implicit class CommandString(firstCommandString: String) {
  def run = Command(firstCommandString).run
  def pipe(secondCommandString: String) =
    Vector(firstCommandString, secondCommandString)
}
```

现在，我们的隐式转换将会把"ls -la" pipe "grep build"转换成一个具有两个shell命令的向量。

```
scala> "ls -la" pipe "grep build"
res2: scala.collection.immutable.Vector[String] = Vector(ls -la, grep build)
```

下一步将再添加一个可以将Vector[String]转换成CommandVector的隐式转化，这与我们对单个字符串的处理差不多。该CommandVector类拥有一个run()方法和一个pipe()方法。

方法pipe()向存放命令的向量中添加了一个新的命令，然后返回该向量。而run()方法知道如何遍历及运行这些命令，它会以管道的方式将一个命令的输出传入下一个命令。CommandVector以及CommandVector所使用的一个位于Command伴生对象上的工厂方法的代码如下所示：

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/dsl/Example.scala

```
implicit class CommandVector(existingCommands: Vector[String]) {
  def run = {
    val pipedCommands = existingCommands.mkString(" | ")
    Command("/bin/sh", "-c", pipedCommands).run
  }
  def pipe(nextCommand: String): Vector[String] = {
    existingCommands :+ nextCommand
  }
}
object Command {
  def apply(commandString: String) = new Command(commandString.split("\\s").toList)
  def apply(commandParts: String*) = new Command(commandParts.toList)
}
```

现在我们已经拥有了完整的DSL，它具有管道和其他所有的功能！在下面的REPL会话中，我们使用该DSL来运行一些管道化的命令：

```
scala> "ls -la" pipe "grep build" run
res3: com.mblinn.mbfpp.functional.dsl.ExtendedExample.CommandResult =
CommandResult(0,-rw-r--r--   1 mblinn  staff    262 Jun  9 13:12 build.sbt
,)

scala> "ls -la" pipe "grep build" pipe "wc" run
res4: com.mblinn.mbfpp.functional.dsl.ExtendedExample.CommandResult =
CommandResult(0,       1       9      59
,)
```

对于该DSL，有两点值得注意。首先，它利用了Scala的特性从而得以将方法作为后缀操作符来使用。但是这种方式很容易被误用，因此Scala 2.10会在你这么做的时候产生一个警告，并且这一特性将会在Scala未来的版本中被默认禁用掉。为了在使用后缀操作符的时候避免收到警告，你可以在需要的时候将`scala.language.postfixOps`导入你的文件。

其次，这只是一个简单的DSL，它适合Scala REPL中的基本使用场景。Scala拥有一个已经构建好且与此非常相似的DSL版本，它位于`scala.sys.process`包之中，它较我们所开发的DSL更加完整。

Clojure实现

在Clojure中，我们的DSL由一个`command`函数组成，该函数可以创建执行shell命令的函数。接着我们将会创建一个`pipe`函数，该函数让我们可以通过使用函数组合来将多个命令以管道的方式串接起来。最后，我们将会创建两个宏：`def-command`和`def-pipe`，以便更容易地为管道和命令进行命名。

在进入DSL的主要代码之前，让我们先来看看如何与shell交互。我们将会使用一个内建于Clojure的库，该库位于命名空间`clojure.java.shell`中，它已经为我们完成了对Java的`Runtime.exec()`的封装。

在下面的REPL会话中，我们使用了`clojure.java.shell`中的`sh`函数来执行`ls`命令。正如我们所看到的，函数的输出是一个由进程状态码和任意进程向输出流写入的字符串所组成的map，这些输出流包括标准输出流和标准错误流，该会话如下所示：

```
=> (shell/sh "ls")
{:exit 0, :out "README.md\nclasses\nproject.clj\nsrc\ntarget\ntest\n", :err ""}
```

该输出并不易于阅读，因此让我们再来创建一个函数，该函数可以在输出的map返回之前以一种更易于阅读的方式将它打印出来。代码如下所示：

ClojureExamples/src/mbfpp/functional/dsl/examples.clj

```
(defn- print-output [output]
  (println (str "Exit Code: " (:exit output)))
```

```
  (if-not (str/blank? (:out output)) (println (:out output)))
  (if-not (str/blank? (:err output)) (println (:err output)))
  output)
```

现在可以使用sh来运行ls -a，然后获得以下可读的输出：

```
=> (print-output (shell/sh "ls" "-a"))
Exit Code: 0
.
..
.classpath
.project
.settings
.svn
README.md
classes
project.clj
src
target
test

{:exit 0,
:out ".\n..\n.classpath\n.project\n.settings\n.svn\n
          README.md\nclasses\nproject.clj\nsrc\ntarget\ntest\n",
:err ""}
```

先来看看该DSL的第一部分：command函数。该函数以字符串的形式接受一个我们想要执行的命令，并基于空格将它进行分割，从而获得一组由不同命令部分组成的序列，然后使用apply向序列应用sh函数。

最后，它通过print-output函数来运行返回的输出，并将所有内容打包进一个高阶函数，同时返回该函数。函数command的代码如下所示：

ClojureExamples/src/mbfpp/functional/dsl/examples.clj

```
(defn command [command-str]
  (let [command-parts (str/split command-str #"\s+")]
    (fn []
      (print-output (apply shell/sh command-parts)))))
```

现在，如果我们运行command函数所返回的函数，它将会运行封装在内部的那些shell命令：

```
=> ((command "pwd"))
Exit Code: 0
/Users/mblinn/Documents/mbfpp/Book/code/ClojureExamples
```

如果我们想要为命令命名，可以通过使用def来实现：

```
=> (def pwd (command "pwd"))
#'mbfpp.functional.dsl.examples/pwd
=> (pwd)
Exit Code: 0
/Users/mblinn/Documents/mbfpp/Book/code/ClojureExamples
```

现在我们已经可以运行单个独立的命令了，再来看看如何将它们以管道的方式串接起来。Unix shell中的管道可以将一个命令的输出作为另一个命令的输入。由于我们在这里是以字符串的方式来捕获命令输出的，因此我们需要一种方式来将字符串作为另一个命令的输入。

通过使用:in选项，函数sh可以让我们做到这一点：

```
=> (shell/sh "wc" :in "foo bar baz")
{:exit 0, :out "       0       3      11\n", :err ""}
```

让我们来修改command函数，接受一个来自另一个命令的输出map，并使用该map的标准输出字符串来作为输入。为了实现这一点，我们将会为command添加第二个参数，并将其作为传入的输出map。

函数command会对该map进行解构，从中分离出它的输出，然后将该输出作为输入传入sh。新command的代码如下所示：

ClojureExamples/src/mbfpp/functional/dsl/examples.clj

```
(defn command [command-str]
  (let [command-parts (str/split command-str #"\s+")]
    (fn
      ([] (print-output (apply shell/sh command-parts)))
      ([{old-out :out}]
        (print-output (apply shell/sh (concat command-parts [:in old-out])))))))
```

现在可以来定义另一个命令了，就像下面使用单词“README”进行的grep那样：

```
=> (def grep-readme (command "grep README"))
#'mbfpp.functional.dsl.examples/grep-readme
```

接着我们便可以将ls命令的输出传递给它，而ls的输出将会与grep以管道的方式串接起来。每个命令都将会把自己的输出打印到标准输出流，如下面的REPL会话所示：

```
=> (grep-readme (ls))
Exit Code: 0
README.md
classes
project.clj
src
target
test

Exit Code: 0
README.md

{:exit 0, :out "README.md\n", :err ""}
```

结合修改后的command函数，我们可以通过使用comp来将多个命令组合起来，从而创建一组命令的管道。如果我们想要按shell中那样的顺序来编写命令的话，只需要在组合这些命令之前将命令序列的顺序反转过来，就像我们在下面的pipe实现中所做的：

ClojureExamples/src/mbfpp/functional/dsl/examples.clj

```
(defn pipe [commands]
  (apply comp (reverse commands)))
```

现在可以创建一组命令管道了，如下面的REPL会话所示：

```
=> (def grep-readme-from-ls
     (pipe
       [(command "ls")
        (command "grep README")]))
#'mbfpp.functional.dsl.examples/grep-readme-from-ls
```

这与先运行ls命令，然后将其输出传入grep-readme的效果是一样的：

```
=> (grep-readme-from-ls)
Exit Code: 0
README.md
classes
project.clj
src
target
test

Exit Code: 0
README.md

{:exit 0, :out "README.md\n", :err ""}
```

现在已经可以定义命令和管道了，让我们通过使用宏来添加一些语法糖，从而让事情变得更加简单。有关Clojure的宏的介绍可以阅读模式20中“Clojure的宏”部分。首先，我们将会创建一个def-command宏。该宏接受一个名称和一个命令字符串，同时会定义一个用来执行该命令字符串的函数，def-command的代码如下所示：

ClojureExamples/src/mbfpp/functional/dsl/examples.clj

```
(defmacro def-command [name command-str]
  `(def ~name ~(command command-str)))
```

现在我们只需通过一次宏的调用便可以完成对一个命令的定义及命名了，如下面的REPL输出所示：

```
=> (def-command pwd "pwd")
#'mbfpp.functional.dsl.examples/pwd

=> (pwd)
Exit Code: 0
/Users/mblinn/Documents/mbfpp/Book/code/ClojureExamples

{:exit 0, :out "/Users/mblinn/Documents/mbfpp/Book/code/ClojureExamples\n", :err ""}
```

与为单个命令所做的工作差不多，现在让我们也为管道化命令创建一个def-pipe宏。该宏接受一个命令的名字以及一组不定数量的命令字符串，然后将每一个命名字符串转换成一个命

令，最后根据给定的名称来创建一个管道。宏def-pipe的代码如下所示：

ClojureExamples/src/mbfpp/functional/dsl/examples.clj

```
(defmacro def-pipe [name & command-strs]
  (let [commands (map command command-strs)
        pipe (pipe commands)]
    `(def ~name ~pipe)))
```

现在我们只需一招就可以创建出一个管道，如下面的代码所示：

```
=> (def-pipe grep-readme-from-ls "ls" "grep README")
#'mbfpp.functional.dsl.examples/grep-readme-from-ls
=> (grep-readme-from-ls)
Exit Code: 0
README.md
classes
project.clj
src
target
test

Exit Code: 0
README.md

{:exit 0, :out "README.md\n", :err ""}
```

到此为止，我们便结束了对Clojure DSL的讨论！

讨论

到目前为止，在实现领域特定语言的过程中，Scala和Clojure各自采用了非常不同的方式。其中Scala采用了一种灵活的语法以及一系列技巧，而Clojure则使用了高阶函数和宏。

Clojure的方式显得更加通用。事实上，Clojure语言自身的大部分内容是以一系列Clojure函数和宏来编写成的！高级的Scala DSL实现者或许可以突破Scala当前方式所带来的限制。

基于这一原因，Scala中加入了宏的特性。然而，正如我们在上一节的“讨论”部分所提到的，由于语法不够简单，以及缺少Clojure和其他Lisp家族语言所具备的“同像”特性，Scala中的宏不论是在实现方面还是使用方面都非常地费劲。

相关模式

模式20　自定义控制流

延展阅读

《领域专用语言实战》[①]

① 电子书网址http://www.ituring.com.cn/book/836。——编者注

第5章 结束语

我们对函数式编程中模式的讨论就到此结束了。通过这些讨论，希望你已经弄清楚了以下两件事情，即函数式编程工具是如何帮助我们编写出更加简短且更加清晰的代码的，以及不可变数据是如何帮助我们从程序中消除掉错误的主要来源的。

希望你已经动手尝试了Scala及Clojure。虽然它们都具有函数式的特性，但是它们之间却又相当不同。通过对由Scala和Clojure编写的这些实例的学习，你已经领略了各式各样的函数式技术。

最重要的是，我希望你能将在本书中所学到的知识运用于日常的编程工作，并在实践中不断提升你的编程技艺。

谢谢阅读！

参考文献

[AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, NY, 1977.

[Blo08] Joshua Bloch. *Effective Java*. Addison-Wesley, Reading, MA, 2008.

[FBBO99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, MA, 1999.

[FH11] Michael Fogus and Chris Houser. *The Joy of Clojure*. Manning Publications Co., Greenwich, CT, 2011.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[Gho10] Debasish Ghosh. *DSLs in Action*. Manning Publications Co., Greenwich, CT, 2010.

[Goe12] Brian Goetz. *JSR 335: Lambda Expressions for the Java Programming Language*. Java Community Process, http://jcp.org, 2012.

[Hal09] Stuart Halloway. *Programming Clojure*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2009.

[Lip11] Miran Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, 2011.

[MRB97] Robert C. Martin, Dirk Riehle, and Frank Buschmann. *Pattern Languages of Program Design 3*. Addison-Wesley, Reading, MA, 1997.

[Nor92] Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann Publishers, San Francisco, CA, 1992.

[Sub09] Venkat Subramaniam. *Programming Scala: Tackle Multi-Core Complexity on the Java Virtual Machine*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2009.

[Sue12] Joshua Suereth. *Scala In Depth*. Manning Publications Co., Greenwich, CT, 2012.

版 权 声 明

“本书是函数式编程领域的瑰宝，任何想要从面向对象过渡到函数式编程的开发者都应该读一读。你正在努力跨越这两个截然不同的世界？那么本书就是安全绳，请将它列入必读清单吧！”

——Colin Yates，QFI咨询公司技术小组负责人

“那些经验丰富的面向对象程序员，若想要尝试函数式风格，便是本书的理想读者，本书将会指引他们从熟悉的面向对象模式过渡到函数式模式。”

——Rod Hilton，时代华纳有线公司高级工程师

“本书为函数式模式与面向对象模式的函数式替代方案，分别设置了独立的章节，作者将函数式模式安排在书中偏后的部分，十分方便读者查阅。作为一名Scala程序员，我也能在书中学到一些新的技巧。”

——Justin James，康卡斯特软件工程师

Functional Programming Patterns in Scala and Clojure

Write Lean Programs for the JVM

StackOverflow刚刚发布了2015开发者调查，最受程序员喜爱的开发语言中，Clojure和Scala分别位列第五和第六。排名第一的是Swift，而Swift相比原先的Objective-C最重要的优点之一，就是对函数式编程提供了更好的支持。同时，新一代分布式计算系统Spark不仅在实现中选择了Scala，其提供的首选编程语言也是Scala。

所有迹象都显示，诞生50多年后，函数式编程却焕发了青春，越来越受到关注和青睐，从边缘地带步入了主流，除了Scala和Clojure这些新生函数式编程语言大行其道之外，Java等老牌面向对象的编程语言也开始支持匿名函数。函数式编程能简化开发过程，尤其是大型知识管理系统应用程序从中受益良多。

本书向读者展示了如何采用函数式方案来替代或简化面向对象编程中使用的诸多通用模式，同时还介绍了一些在函数式世界中广泛使用的模式。如果你是一名Java程序员，希望了解函数式编程能为你的工作效率带来[illegible]的提升，或者你是一名刚刚开始使用Scala和Clojure的新手，尚不能玩转函数式的问题解决方案，那么本书[illegible]你而准备的。

本书所有代码可在http://pragprog.com/titles/mbfpp/source_code免费下载。

The Pragmatic Programmers

图灵社区：iTuring.cn

热线：(010)51095186转600

分类建议 计算机/程序设计

人民邮电出版社网址：www.ptpress.com.cn

TURING 2761484 www.ituring.com.cn 来社区 领银子

ISBN 978-7-115-38894-0

ISBN 978-7-115-38894-0

定价：49.00元