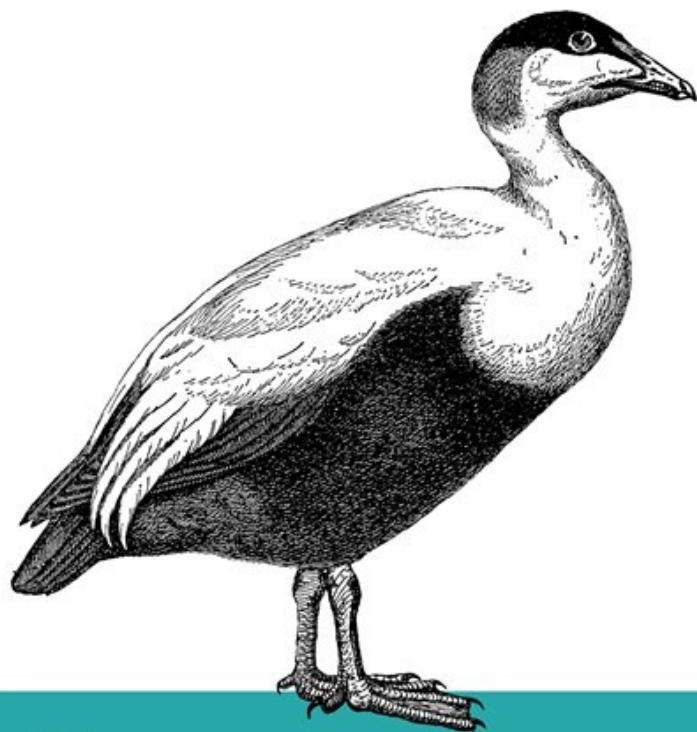


Functional JavaScript



JavaScript

函数式编程

[美] *Michael Fogus* 著
欧阳继超 王妮 译

O'REILLY®



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

目 录

[版权信息](#)

[版权声明](#)

[内容提要](#)

[O’ Reilly Media, Inc. 介绍](#)

[序一](#)

[序二](#)

[前言](#)

[第1章 JavaScript函数式编程简介](#)

[1.1 JavaScript案例](#)

[JavaScript的一些局限](#)

[1.2 开始函数式编程](#)

[1.2.1 为什么函数式编程很重要](#)

[1.2.2 以函数为抽象单元](#)

[1.2.3 封装和隐藏](#)

[1.2.4 以函数为行为单位](#)

[1.2.5 数据抽象](#)

[1.2.6 函数式JavaScript初试](#)

[1.2.7 加速](#)

[1.3 Underscore示例](#)

[1.4 总结](#)

[第2章 一等函数与Applicative编程](#)

[2.1 函数是一等公民](#)

[多种JavaScript编程方式](#)

[2.2 Applicative编程](#)

[2.2.1 集合中心编程](#)

[2.2.2 Applicative编程的其他实例](#)

[2.2.3 定义几个Applicative函数](#)

[2.3 数据思考](#)

[“表状（Tabel-Like）” 数据](#)

[2.4 总结](#)

[第3章 变量的作用域和闭包](#)

[3.1 全局作用域](#)

[3.2 词法作用域](#)

[3.3 动态作用域](#)

[JavaScript的动态作用域](#)

[3.4 函数作用域](#)

[3.5 闭包](#)

[3.5.1 模拟闭包](#)

[3.5.2 使用闭包](#)

[3.5.3 闭包的抽象](#)

[3.6 总结](#)

[第4章 高阶函数](#)

[4.1 以其他函数为参数的函数](#)

[4.1.1 关于传递函数的思考：max、finder和best](#)

[4.1.2 关于传递函数的更多思考：重复、反复和条件迭代 \(iterateUntil\)](#)

[4.2 返回其他函数的函数](#)

[4.2.1 高阶函数捕获参数](#)

[4.2.2 捕获变量的好处](#)

[4.2.3 防止不存在的函数：fnull](#)

[4.3 整合：对象校验器](#)

[4.4 总结](#)

[第5章 由函数构造函数](#)

[5.1 函数式组合的精华](#)

[突变 \(mutation\) 是底层的操作](#)

[5.2 柯里化 \(Currying\)](#)

[5.2.1 向右柯里化，还是向左](#)

[5.2.2 自动柯里化参数](#)

[5.2.3 柯里化流利的API](#)

[5.2.4 JavaScript柯里化的缺点](#)

[5.3 部分应用](#)

[5.3.1 部分应用一个和两个已知的参数](#)

[5.3.2 部分应用任意数量的参数](#)

[5.3.3 局部应用实战：前置条件](#)

[5.4 通过组合端至端的拼接函数](#)

[组合前置与后置条件](#)

[5.5 总结](#)

[第6章 递归](#)

[6.1 自吸收 \(self-absorbed\) 函数 \(调用自己的函数\)](#)

[6.1.1 用递归遍历图](#)

- [6.1.2 深度优先自递归搜索](#)
- [6.1.3 递归和组合函数: Conjoin和Disjoin](#)
- [6.2 相互关联函数 \(函数调用其他会再调用回它的函数\)](#)
- [6.2.1 使用递归深克隆](#)
- [6.2.2 遍历嵌套数组](#)
- [6.3 太多递归了](#)
- [6.3.1 生成器](#)
- [6.3.2 蹦床原理以及回调](#)
- [6.4 递归是一个底层操作](#)
- [6.5 总结](#)
- [第7章](#)
- [7.1 纯度](#)
- [7.1.1 纯度和测试之间的关系](#)
- [7.1.2 提取纯函数](#)
- [7.1.3 测试不纯函数的属性](#)
- [7.1.4 纯度与引用透明度的关系](#)
- [7.1.5 纯度和幂等性](#)
- [7.2 不变性](#)
- [7.2.1 如果一棵树倒在树林里, 有没有声音](#)
- [7.2.2 不变性与递归](#)
- [7.2.3 冻结和克隆](#)
- [7.2.4 在函数级别上观察不变性](#)
- [7.2.5 观察对象的不变性](#)
- [7.2.6 对象往往是一个低级别的操作](#)
- [7.3 控制变化的政策](#)
- [7.4 总结](#)
- [第8章 基于流的编程](#)
- [8.1 链接](#)
- [8.1.1 惰性链](#)
- [8.1.2 Promises](#)
- [8.2 管道](#)
- [8.3 数据流与控制流](#)
- [8.3.1 找个一般的形状](#)
- [8.3.2 函数可以简化创建action](#)
- [8.4 总结](#)
- [第9章 无类编程](#)
- [9.1 数据导向](#)

[通过函数来构建](#)

[9.2 Mixins](#)

[9.2.1 修改核心原型](#)

[9.2.2 类层次结构](#)

[9.2.3 改变层级结构](#)

[9.2.4 用Mixin扁平化层级结构](#)

[9.2.5 通过Mixin扩展新的语义](#)

[9.2.6 通过Mixin混合出新的类型](#)

[9.2.7 方法是低级别操作](#)

[9.3 总结](#)

[附录A 更多函数式JavaScript](#)

[A.1 JavaScript的函数式库](#)

[A.1.1 Functional JavaScript](#)

[A.1.2 Underscore-contrib](#)

[A.1.3 RxJS](#)

[A.1.4 Bilby](#)

[A.1.5 allong.es](#)

[A.1.6 其他函数式库](#)

[A.2 能编译成JavaScript的函数式语言](#)

[A.2.1 ClojureScript](#)

[A.2.2 CoffeeScript](#)

[A.2.3 Roy](#)

[A.2.4 Elm](#)

[作者简介](#)

[封面介绍](#)

[附录B 推荐书目](#)

[论文/书籍/博客文章/会谈](#)

[演讲](#)

[博客文章](#)

[杂志文章](#)

[欢迎来到异步社区！](#)

版权信息

书名：JavaScript函数式编程

ISBN：978-7-115-39060-8

本书由人民邮电出版社发行数字版。版权所有，侵权必究。

您购买的人民邮电出版社电子书仅供您个人使用，未经授权，不得以任何方式复制和传播本书内容。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

• 著 [美] Michael Fogus

译 欧阳继超 王 妮

责任编辑 陈冀康

• 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>

• 读者服务热线：(010)81055410

反盗版热线：(010)81055315

版权声明

Copyright © 2013 by O' Reilly Media, Inc.

Simplified Chinese Edition, jointly published by
O' Reilly Media, Inc. and Posts & Telecom Press, 2015.
Authorized translation of the English edition, 2013 O' Reilly
Media, Inc., the owner of all rights to publish and sell the
same.

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

本书中文简体版由O' Reilly Media, Inc. 授权人民邮电出版社出版。未经出版者书面许可，对本书的任何部分不得以任何方式复制或抄袭。

版权所有，侵权必究。

内容提要

JavaScript是近年来非常受瞩目的一门编程语言，它既支持面向对象编程，也支持函数式编程。本书专门介绍JavaScript函数式编程的特性。

全书共9章，分别介绍了JavaScript函数式编程、一等函数与Applicative编程、变量的作用域和闭包、高阶函数、由函数构造函数、递归、纯度和不变性以及更改政策、基于流的编程、无类编程。除此之外，附录中还介绍了其他和JavaScript函数式编程相关的知识。

本书内容全面，示例丰富，适合想要了解函数式编程的JavaScript程序员和学习JavaScript的函数式程序员阅读。

0' Reilly Media, Inc. 介绍

0' Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，0' Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，0' Reilly的发展充满了对创新的倡导、创造和发扬光大。

0' Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。0' Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，0' Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项0' Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“0' Reilly Radar博客有口皆碑。”

——Wired

“0' Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“0' Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O’ Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

序一

这是一本令人兴奋的书。

尽管开始是想作为嵌入HTML文档中的精简版Java (Java-lite) 的脚本语言，由此可以进行一点交互，但JavaScript却成为通用编程最灵活的语言之一。

你可以随意根据最适合你的特定方式来写JavaScript。在JavaScript中这样做比其他更刚性的语言更自然，究其原因，是支撑JavaScript的核心理念：更大程度上比面向对象语言如Ruby和Java扩展了一切都是对象（一切都是一个值）的理念。函数既是对象，也是值。任何对象都可以作为其他对象的原型（默认值）。函数只有一种，你可以随意使用它，可以作为一个纯函数、一个突变过程，或作为一个对象的方法。

JavaScript 可以但不会强制使用不同的编程风格。早期，我们倾向于把传统期望和“最佳”实践套用到JavaScript的学习中。当然，这会使得JavaScript很像没有类型的Java，或是将类型写到每个方法上面的注释里面。渐渐地，有人开始实验：在运行时生成函数，使用不可变的数据结构，创建不同于面向对象的设计模式，发现链式API的魔力，或是扩展内置原型来得到定制化的功能。

我最近特别热衷于使用函数式思路构建丰富的JavaScript应用。随着JavaScript从简单的表单验证和DOM动画到全功能的应用程序，JavaScript开始面临着各种特定问题，函数式也有了更有趣的舞台，如下所示。

- 通过可部分配置参数的函数构建大量API。
- 使用递归函数来平滑需要一段时间的事件。
- 使用无突变（mutation-free）的随意替换的管道构建复杂的业务逻辑。

本书正是探索这片领域的理想书籍。在书中的9章和附录里，友好的导游和疯狂的科学家Michael Fogus，将函数式编程层层剥开，让你

一探究竟。一般很少有编程书籍能带给读者惊喜，但是这一本绝对可以。

好好享受吧！

Jeremy Ashkenas

序二

我还记得当我第一次读到Douglas Crockford的《JavaScript精粹》时，我不仅从中学到了东西，而且Crockford只用了172页，就能带领读者避开JavaScript的各种问题，实在令人印象深刻。Crockford的书既简洁，又能让读者充分消化并从中受益。

接下来，你会发现，Michael Fogus给了我们一本类似Crockford的书。他吸收了Crockford以及其他前辈的中肯的意见，带我们深入函数式JavaScript编程的世界。我经常听说或看到（甚至我自己也会写到），JavaScript是一种函数式编程语言，但这种说法（包括我自己）都似乎难以领会。甚至Crockford也只用了一章阐述函数，像许多作家一样，都集中于JavaScript的对象支持。Fogus填补了这些重要的细节。

函数式编程从一开始就是计算机领域的重要的一部分，但它一直没有受到实践软件人员的广泛关注。但由于计算硬件速度和容量的不断提高，再加上我们的行业在创造并发、分布和大规模软件系统的需求不断扩大，函数式编程正迅速地普及。能获得这样的增长是因为对于开发者来说，函数式更容易推理、构建和维护。对函数式编程语言，如Scala、Clojure中，Erlang和Haskell的关注也达到了历史新高，并仍在增加，至今仍不见削减。

当你读完Michael具有深刻见解的JavaScript的函数式编程，你会对他所提供的信息的深度和广度留下深刻的印象。他首先会让事情保持简单，解释如何避免使用JavaScript功能强大的对象原型系统，而使用函数和“数据抽象”来建模类的方式。在之后的章节中，解释了函数式数据转换的简单模型可以产生复杂而高效的更高层次的抽象。我猜你会随着Fogus的每个章节对这种方式的层次深入感到惊讶。

大部分软件开发工作需要实用主义，好在Fogus也强调了这一项。如果不实用，就算有优雅、精致和简洁的代码，最终都是毫无意义的。这也是函数式编程隐藏在阴影中这么多年的很大一部分原因。Fogus通过帮助读者了解和评估与函数式编程相关的计算成本来解决这一问题。

当然，书就像软件，都少不了沟通。就像Crockford和Fogus的写作方式，既简短，又内容翔实，恰到好处。我没有夸大Michael的简洁和清晰的重要性，不然你会失去他所提供的令人难以置信的想法和见解的。你会发现，不仅Fogus所提供的方法和代码优雅，他表达的方式也一样优雅。

Steve Vinoski

前言

什么是Underscore

Underscore.js（以下简称Underscore）是支持函数式编程的JavaScript库。Underscore网站是这样描述的：

Underscore为JavaScript提供了大量的函数式编程的支持，类似Prototype.js（或Ruby）的utility-belt，但没有扩展JavaScript内置对象。

“utility belt”指的是一套能帮助你解决很多常见问题的工具。

获取Underscore

Underscore网站上有最新的版本。你可以从网站上下载并放入应用目录。

使用Underscore

你可以像使用所有其他库一样在你的项目中使用Underscore。然而，有几点需要注意的是，首先，默认情况下，Underscore定义了一个包含其所有函数的全局对象。要调用一个Underscore的函数，只需要调用“_”里的方法，如下面的代码：

```
_.times(4, function() { console.log("Major") });  
  
// (console) Major  
// (console) Major  
// (console) Major  
// (console) Major
```

很简单吧？

但如果你已经定义一个全局_变量，事情就没这么简单了。在这种情况下，Underscore提供了一个_.noConflict函数，将重新绑定旧的_，并返

回Underscore的引用。_.noConflict使用方式如下：

```
var underscore = _.noConflict();

underscore.times(4, function() { console.log("Major") });

// (console) Major
// (console) Major
// (console) Major
// (console) Major

_;
```

//=> Whatever you originally bound _ to

本书会介绍更多Underscore的细节，但记住，虽然我广泛使用（并认可）Underscore，但这并不是一本关于Underscore的书。

函数式JavaScript的源代码

许多年前，我想写基于函数式编程技术的JavaScript库。跟许多程序员一样，我曾通过实验、实践以及阅读Douglas Crockford的文章对JavaScript有所认识。虽然我继续完成了我的函数式库（Doris），但甚至连我自己都很少用它。

完成Doris后，我继续尝试广泛的函数式编程语言如Scala和Clojure。此外，我花了很多时间编写ClojureScript，特别是它的JavaScript编译器。基于这些经验，我非常了解函数式编程技术。因此，我决定尝试使用随后这几年学到的技术复活Doris。我将其命名为Lemonad，最后几乎是与本书同时完成的。

本书中大多数函数都是为了教学目的，但我扩展了一些并贡献到我的Lemonad库，以及Underscore-contrib库。

本书中的代码

本书的源代码可以在GitHub上获取。此外，你也可以进入本书的网址使用上面的REPL试试本书所有定义的函数。

符号约定

在编写本书（和一般编写JavaScript）的过程中，我得出以下比较好的约定。

- 避免多次赋值变量。
- 不要使用`eval`^[1]。
- 不要修改内核对象如`Array`和`Function`。
- 优先使用函数而不是方法。
- 如果项目一开始就定义函数，那么在接下来的阶段里也应该如此。

此外，我在本书中还用到了一些约定。

- 零参数的函数用于表示该参数并不重要。
- 在一些例子中，`……`用来表示其周围的代码段可忽略。
- `inst#method`表示实例方法引用。
- `Object.method`表示类型方法。
- 我倾向于用单行`if/else`语句，避免使用大括号。这样可以节省宝贵的垂直空间。
- 我喜欢用分号。

基本上除了函数式方面，这本书中的JavaScript代码就像现实中的大多数的JavaScript代码。

本书目标读者

我在几年前写一本Scheme编程语言的函数式编程的入门书籍的时候，就产生了写这本书的想法。尽管Scheme和JavaScript有一些共同的特点，但在许多重要方面截然不同。我想撇开语言来说说函数式编程。我写这本书介绍函数式编程是什么，什么是不可能JavaScript中找到的。

我期望读者对JavaScript有基本的理解。可以通过很多书籍以及网上的资源和讨论来学习这门语言，这里就不占用本书篇幅介绍了。我还期望读者能对面向对象编程有所了解，如Java和Ruby，Python和JavaScript。了解面向对象编程可以帮助你理解我偶尔使用的一些短语，但并不需要专家级的了解。

本书的合适读者是希望了解函数式编程的JavaScript程序员，或希望学习JavaScript的函数式程序员。对于后一种读者，还可以研究一些JavaScript的……古怪的部分，特别是可以参考Douglas Crockford的

《JavaScript精粹》（O’ Reilly出版）。最后，这本书还适合任何希望了解函数式编程，包括不打算使用JavaScript的读者。

本书组织结构

下面是JavaScript函数式编程的大纲。

第1章 JavaScript函数式编程简介

这本书通过引入一些主题来开始，包括函数式编程和Underscore.js。

第2章 一等函数与Applicative编程

第2章定义了一等函数，展示如何使用它们，并介绍了一些常见的应用。其中介绍了一个特别的技术，即利用一等函数实现Applicative编程。本章结尾还对软件开发中函数式编程的重要途径，即“数据思想”进行了探讨。

第3章 变量的作用域和闭包

第3章是一个过渡章，涵盖要了解函数式JavaScript编程需要注意的两个核心主题。通过覆盖变量作用域，包括在JavaScript中使用的方式：词法作用域，动态作用域和函数作用域。本章以闭包的介绍结尾，解释了工作原理，以及如何和为什么可能需要使用闭包。

第4章 高阶函数

本章建立在第2、3章基础上，介绍了一个重要的一等函数：高阶函数。虽然“高阶函数”听起来很复杂，本章会说明它其实是很直白的。

第5章 由函数构建函数

本章介绍了如何用其他函数“组合”新函数。组合函数是函数式编程的重要技术，本章将引导你了解这项技术。

第6章 递归

第6章是另一个过渡章节，将讨论递归，即一个直接或间接调用自身的函数。因为递归在JavaScript中是有局限的，因此不被经常使用；但是，本章会介绍几个绕过这些局限的方法。

第7章 纯度、不变性和更改政策

第7章介绍如何编写不会改变任何东西的函数。简单地说，函数式编程的便利性来源于不可变变量。本章将带你理解其中的含义。

第8章 基于流的编程

第8章涉及如何将任务甚至是整个系统，看作变换数据的“装配线”。

第9章 无类编程

最后一章的重点是介绍函数式编程是完全不同于基于类的面向对象编程的结构化应用程序的方式。

在这些章节之后补充了附录A。

本书使用的约定

本书使用以下字体排版约定。

斜体

表示新的术语、网址、电子邮件地址、文件名和文件扩展名。

等宽字体

用于程序代码清单，出现在段落之内则表示程序中的元素，如变量、函数名、数据库、数据类型、环境变量、程序语句和关键字。

等宽加粗体

显示命令或其他应当由用户键入的文本。

等宽斜体

表示该文本应当更换为用户提供的值或者由上下文所决定的值。

示例代码的使用

本书的目的是为了帮助你完成工作任务。在一般情况下，这本书中包括的代码示例，你可以将其应用到你的程序和文档中。除非需要复制这些示例代码的相当大部分，否则无需联系我们以获得许可。比如说，当你编写的程序用到了本书中的若干示例代码，这并不需要特别许可。但是，销售或分发含有O’Reilly书籍附带的示例程序的光盘则需要获得许可。当你

在回答他人问题时援引本书内容，或者引用书中的范例代码，也不用申请许可；而如果要把本书中的代码大量地引用到你的产品文档中，则需要许可。

对于引用时署名本书，我们表示感谢，但并不要求。一个署名通常包括标题、作者、出版商和ISBN，例如“Functional JavaScript Michael Fogus (O’ Reilly出版)。版权所有2013 Michael • Fogus, 978-1-449-36072-6”。

如果你觉得你使用示例代码的情况超出了以上描述的不需要许可的范围，请随时联系我们：
permissions@oreilly.com。

Safari®在线图书

记录

Safari在线图书 (www.safaribooksonline.com) 是一个虚拟图书馆，让你可以轻松搜寻成千上万的顶尖技术书籍。

科技人才，软件开发人员，网页设计师，以及商业和创意专业人士都将Safari在线图书作为研究、解决问题、学习和认证培训的主要资源。

Safari的联机丛书提供了一系列的产品并为组织、政府机构和个人提供不同定价。用户可以访问和搜索出版社O’ Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology等的数字内容。有关Safari在线图书的更多信息，请访问我们的在线网站。

如何联系我们

请将对本书的有关意见和问题告知出版商：

美国：

O’ Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国:

北京市西城区西直门南大街2号成铭大厦C座807室 (100035)

奥莱利技术咨询 (北京) 有限公司

我们为这本书制作了网页, 其中包含了勘误表、范例, 以及其他补充资料。你可以通过这个地址访问: http://oreilly/functional_js。

请发送电子邮件至bookquestions@oreilly.com发表评论或询问关于本书的技术问题。

更多关于这本书、课程、会议和新闻的信息, 查看下面的网站:
<http://www.oreilly.com>。

我们的Facebook网站: <http://facebook.com/oreilly>。

在Twitter上follow我们: <http://twitter.com/oreillymedia>。

YouTube上的频道: <http://www.youtube.com/oreillymedia>。

致谢

写一本书需要各方面的支持和努力, 本书也不例外。首先, 我要感谢我的好朋友Rob Friesel抽出时间提供反馈意见。此外, 我要感谢Jeremy Ashkenas把我介绍给O'Reilly, 能让本书得以出版。而且是他写了非同小可的Underscore.js库。

我还要感谢这些人给我的反馈和启发: Chris Houser、David Nolen、Stuart Halloway、Tim Ewald、Russ Olsen、Alan Kay、Peter Seibel、Sam Aaron、Brenton Ashworth、Craig Andera、Lynn Grogan、Matthew Flatt、Brian McKenna、Bodil Stokke、Oleg Kiselyov、Dave Herman、Mashaaricda Barmajada ee Mahmud、Patrick Logan、Alan Dipert、Alex Redington、Justin Gehtland、Carin Meier、Phil Bagwell、Steve Vinoski、Reginald Braithwaite、Daniel Friedman、Jamie Kite、William Byrd、Larry Albright、Michael Nygard、Sacha Chua、Daniel Spiewak、Christophe Grand、Sam Aaron、Meikel Brandmeyer、Dean

Wampler、Clinton Dreisbach、Matthew Podwysocki、Steve Yegge、David Liebke、Rich Hickey。

我编写本书时的配乐由Pantha du Prince、Black Ace、Brian Eno、Béla Bartók、Dieter Moebius、Sun Ra、Broadcast、Scientist、John Coltrane提供。

最后，所有这一切都要感谢我生命中的三位挚爱：Keita、Shota、Yuki。

[\[1\]](#) 跟所有强大的工具一样，eval和Function常量都是双刃剑，我并不反对使用，但是建议尽可能少用。

第1章 JavaScript函数式编程简介

本章是本书后续内容的基础。本章将介绍什么是Underscore以及如何开始使用它；除此之外，也将对后续用到的术语和本书的目标进行解释。

1.1 JavaScript案例

“为什么选择JavaScript”，这个问题的答案很简单：灵活。换句话说，或许除了Java，目前没有比JavaScript更加流行的语言了。所有浏览器以及现有新兴技术的大范围支持，使得JavaScript成了满足可移植性的不错的选择，甚至是唯一的选择。

随着客户端服务和单页布局应用架构的再度出现，JavaScript越来越广泛地应用于附加在大量网络服务中的分离式应用当中（如单页布局）。比如所有的谷歌应用程序都由JavaScript编写，这也是单页布局应用的范例。

如果在学习JavaScript之前，你已对函数式编程有所研究，那么好消息是JavaScript“天然”支持函数式技术（例如，函数是JavaScript的一个核心概念）。举个例子，如果你对JavaScript有所了解，那么应该见过下面的代码：

```
[1, 2, 3].forEach(alert);  
// alert box with "1" pops up  
// alert box with "2" pops up  
// alert box with "3" pops up
```

Array#forEach方法于ECMA-262语言标准第5版加入，它接收一个函数（本例中的alert）并将数组中的每一个元素依次交给该函数执行。除此之外，JavaScript提供大量的能够以其他函数为参数的方法和函数。在本书的后续内容中，我将进一步以此类编程风格进行讨论。

JavaScript有坚实的语言原语基础，这是非常好的事情，但同时也是一把双刃剑。从函数、闭包、原型，到相当不错的动态核心，JavaScript提供了一系列非常好的工具集^[1]。此外，JavaScript也提供了一种非常开放和灵活的执行模型。举一个小例子：所有的JavaScript函数都有一个apply方法，它使得我们可以用一个数组来调用函数，其中，数组的元素作为函数的参数。使用apply，我们可以创建一个名为splat的函数。它接受

一个函数fun，并返回另一个函数，该函数接受一个数组并用apply来执行函数fun。这样一来，传入函数splat的数组的元素是函数fun的参数：

```
function splat(fun) {  
  return function(array) {  
    return fun.apply(null, array);  
  };  
}  
  
var addArrayElements = splat(function(x, y) { return x + y });  
  
addArrayElements([1, 2]);  
//=> 3
```

这是我们的函数式编程初试——一个返回函数的函数——我们待会再来仔细研究。需要注意的是，JavaScript是一种非常灵活的语言，apply仅仅只是它实现函数式编程的一种方法而已。

另外一个展现JavaScript灵活性的地方是，我们可以随时以任意多个任意类型的参数来调用任意一个函数。我们可以创建一个与splat功能相反的函数unsplat，它接受一个函数fun并返回另一个接受任意多个参数的函数，将参数转为数组传入函数fun并调用它：

```
function unsplat(fun) {  
  return function() {  
    return fun.call(null, _.toArray(arguments));  
  };  
}  
  
var joinElements = unsplat(function(array) { return array.join(' ') });  
  
joinElements(1, 2);  
//=> "1 2"  
  
joinElements('-', '$', '/', '!', ':');  
//=> "- $ / ! :"
```

每个JavaScript函数都可以访问一个名为arguments的局部对象，它以类似于数组的形式存储了调用本函数时的实际参数。arguments非常强大，并能够产生惊人的效果。另外，call方法与apply方法类似，只不过apply方法将参数放到数组中来调用函数，而call方法则是直接将参数逐一传递

给函数。Apply、call和arguments的同时存在只是JavaScript强大灵活性的一个小例子。

随着用JavaScript来创建各种规模应用的趋势，你可能会担心该语言本身发展及其运行时支持会停滞不前。但是随意阅读一下ECMAScript.next就可以发现，JavaScript是一种不断发展（尽管速度缓慢）的语言[2]。同样，会不断有像V8引擎那样经得起时间考验的新兴技术来改进和提升JavaScript的速度和效率。

JavaScript的一些局限

从JavaScript的出现、演变、发展到普遍存在的角度来讲，JavaScript的局限性很小。很多人会诟病JavaScript的种种奇怪用法和鲁棒性缺陷，但事实上，JavaScript确实存活了下来，并且将会一直存在下去。但无论如何，我们需要承认JavaScript是一门存在缺陷的语言[3]。事实上，目前最流行的介绍JavaScript的书：Douglas Crockford的*JavaScript: The Good Parts*（O’Reilly出版），花了大量的篇幅来讨论JavaScript的不好的部分。这门语言确实有古怪之处，而且总体来说在表达方面也不是很简洁。然而，修复JavaScript中存在的问题恐怕会“破坏网络世界”，这恐怕也是不能被大众所接受的。也正是因为这些问题的存在，针对于JavaScript的编译平台不断增多；这确实是一片非常多产的领域[4]。

从语言支持角度来讲，经过时间的选择，我们发现命令式语言技术和对全局作用域的依赖使得JavaScript存在不安全性。这是因为，若在创建程序时将关键点放在处理易变性上，会给程序扩展带来潜在的混乱。同样，这门语言也提供了很多可用来实现其他语言中默认存在的高级功能的方法。如在主要版本的ECMAScript 6之前，JavaScript没有提供模块系统，但其实可用原生对象来简便地创建模块。这个版本的JavaScript提供了一系列松散的、互不兼容的基本部分集合，可以用来保证一系列自定义模块的实现。

古怪的语言、不安全的功能以及一系列互相竞争的库，这三个理由使得我们很难考虑选择JavaScript。然而，希望还是有的。利用一系列的规范和规约，JavaScript代码可以做到不仅安全，而且容易理解和测试，除此之外，也能够成比例缩减代码库大小。本书将带你掌握这样的方法：函数式编程。

1.2 开始函数式编程

或许你已经从最喜欢的新闻聚合网站上听说过函数式编程，也可能你已使用过支持函数式编程的技术。如果你写过JavaScript代码（在本书中，我们默认读者写过），那么你确实已使用过支持函数式编程的语言。然而，有种情况是，你可能没有从函数式编程的角度使用过JavaScript。本书突出了函数式编程风格，它可以帮助我们简化自己的库和应用程序，并帮助我们驯服那只使得JavaScript变得复杂的“野兽”。

我们可以用下面一句话来直白地描述函数式编程：

函数式编程通过使用函数来将值转换成抽象单元，接着用于构建软件系统。

这是一种简单粗糙的解释，但对于本书的前面部分来说已够用。本书使用Underscore作为库来实现函数式表达式，并且大部分内容也都遵循上面的定义。然而，这个定义并没有解释清楚“为什么”使用函数式编程。

1.2.1 为什么函数式编程很重要

对我来说，重大演变还是向更加函数式的风格的发展，它使得我们放弃很多旧的习惯，并从一些面向对象思想中逐渐退出。

——John Carmack

如果你熟悉面向对象编程，那么你可能会同意它的主要目标是问题分解，如图1-1所示（Gamma，1995）。

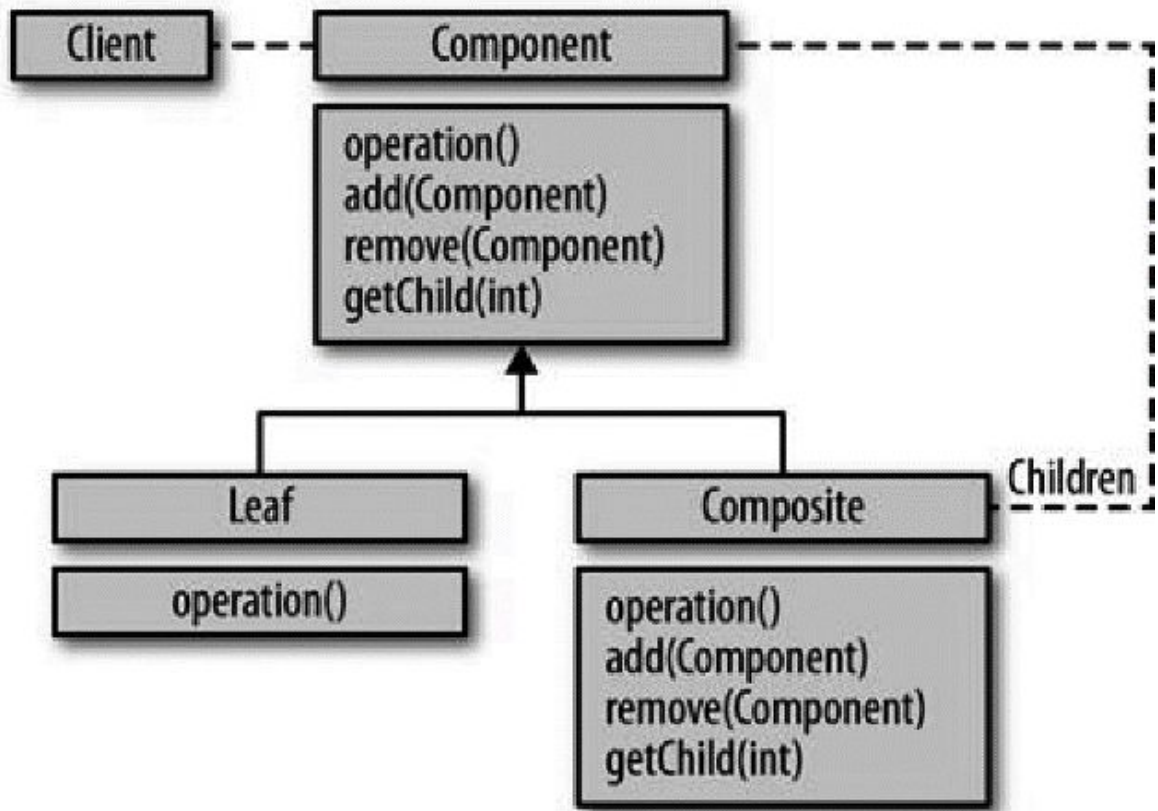


图1-1 将一个问题分解为面向对象的几个部件

同样，如图1-2 所示，这些部件/对象可以被聚集在一起，并组合成更大的部件。

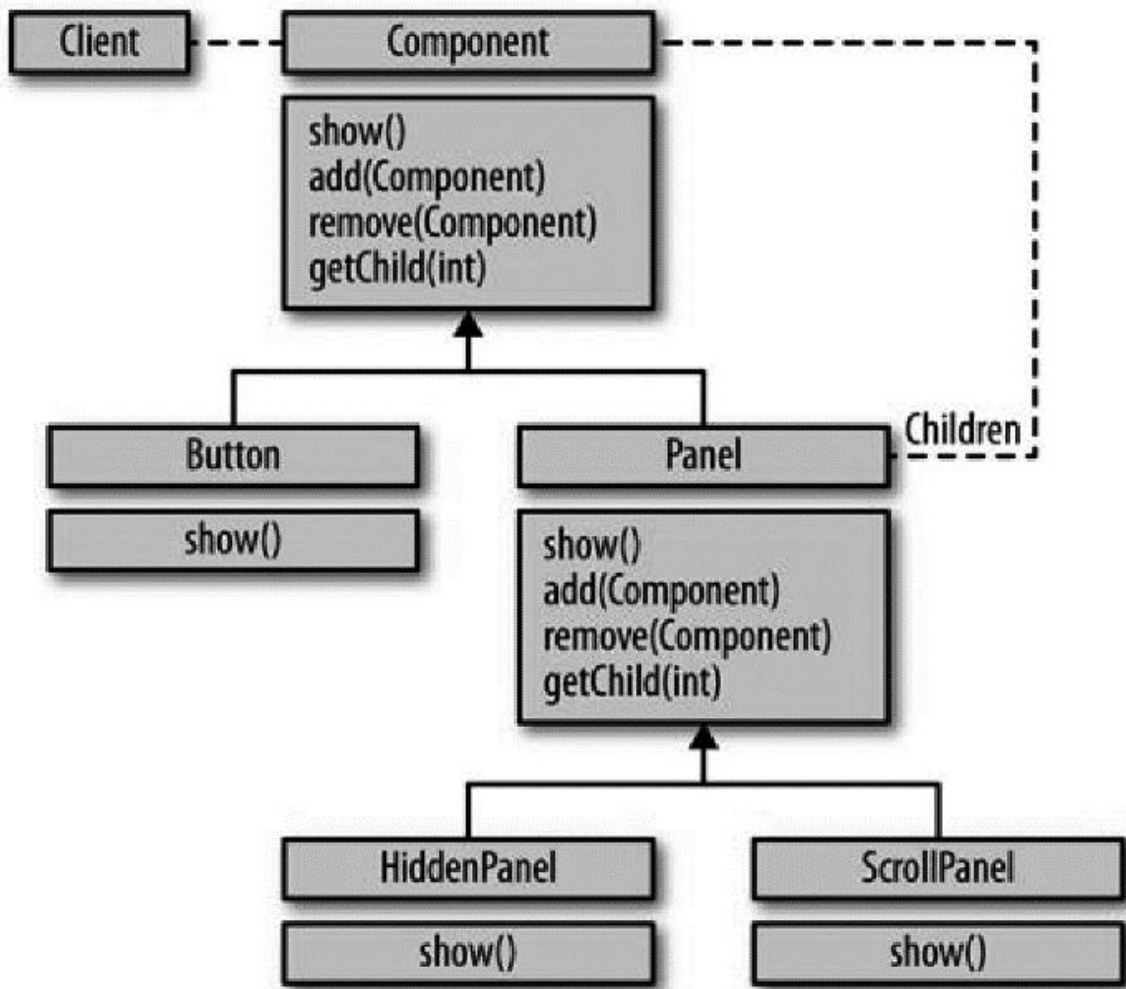


图1-2 对象“组合”在一起形成更大的对象

基于这些部件和它们之间的组合关系，我们就可以从部件之间的交互和值来描述一个系统，如图1-3所示。

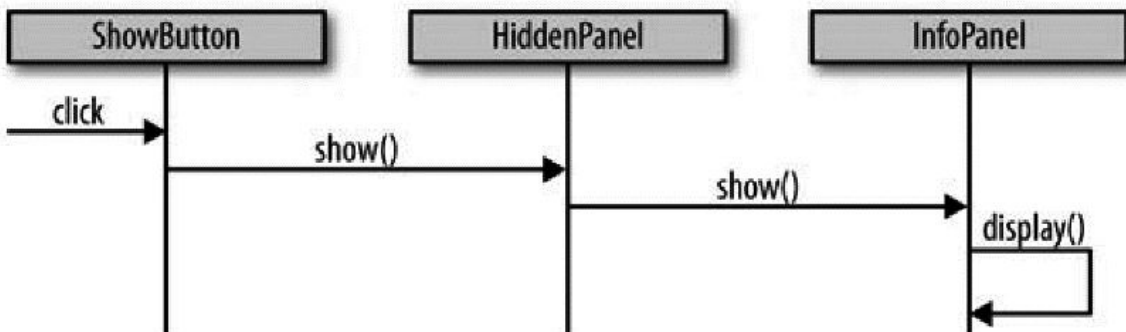


图1-3 一个描述面向对象系统及其交互的序列图

这里只对如何构建面向对象系统进行了简单粗糙的解释，但这种抽象的解释已经能够说明问题。

相比较而言，用严格的函数式编程的方法来解决问題，也会将一个问題分成几部分（函数）来解决，如图1-4所示。

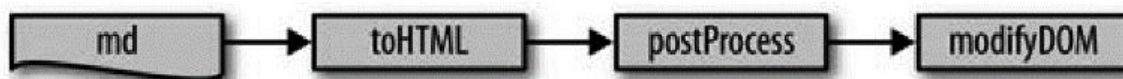


图1-4 将一个问题分解成几个函数式的部分

与面向对象方法将问題分解成多组“名词”或对象不同，函数式方法将相同的问题分解成多组“动词”或函数^[5]。与面向对象编程类似的是，函数式编程也通过“黏结”或“组合”其他函数的方式来构建更大的函数，以实现更加抽象的行为，如图1-5 所示。

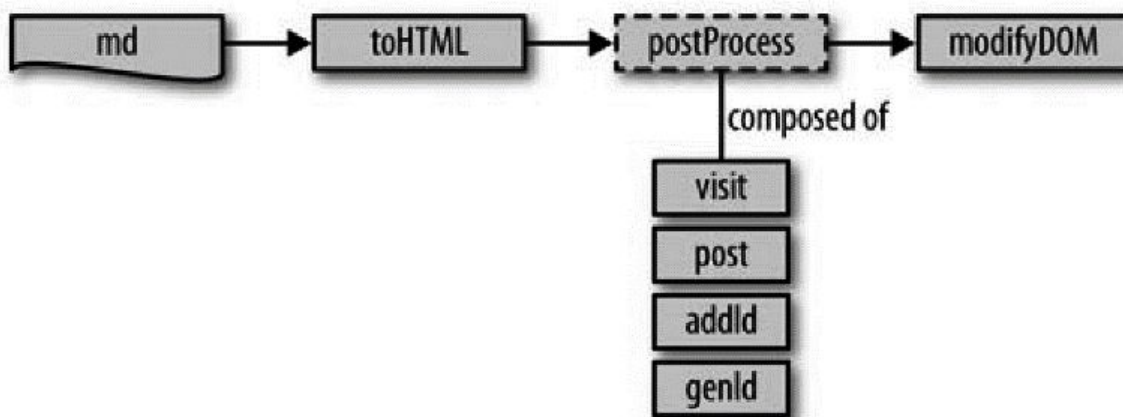


图1-5 通过函数组合来实现更多的行为

最终，一种将函数式的部件组成一个完整的系统的方法（见图1-6）是取一个值，逐渐地将它“改变”——通过一个原始的或组合的函数——成另一个值。

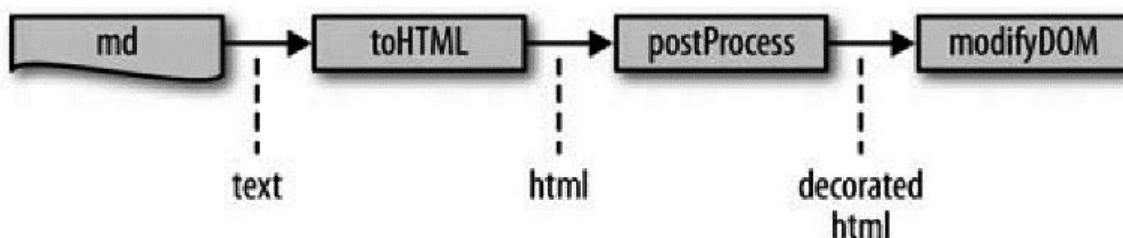


图1-6 一个通过数据转换进行交互的函数式系统

在一个面向对象系统的内部，我们发现对象间的交互会引起各个对象内部状态的变化，而整个系统的状态转变则是由许许多多小的、细微的状态变化混合来形成的。这些相互关联的状态变化形成了一个概念上的“变化网”，我们时不时会因为它而感到困惑。当需要了解其带来的微妙且广泛的状态变化时，这种困惑就会成为一个问题。

相比之下，函数式系统则努力减少可见的状态修改。因此，向一个遵循函数式原则的系统添加新功能就成了理解如何在存在局限的上下文环境中——无破坏性的数据转换（例如原始数据永不发生变化）——来实现新的函数。然而，我并不愿意在函数式风格和面向对象风格之间画一条明显的界线，说它们应该是对立关系。因为既然JavaScript同时支持这两种模式，那就说明一个系统可以也应该由这两种模式共同组成。如何平衡函数式风格和面向对象风格是一件需要技巧的事情，我们将会在第9章讨论mixin时解答这个问题。然而，既然本书是在介绍函数式编程在JavaScript中实现，那么我们将会将大量篇幅放在函数式风格而非面向对象风格上。

这样说来，一个美好的基于函数式原则而构建的系统将是一个能够从输入终端接收未加工原料并逐渐从输出终端生产出产品的装配线设备（见图1-7）。

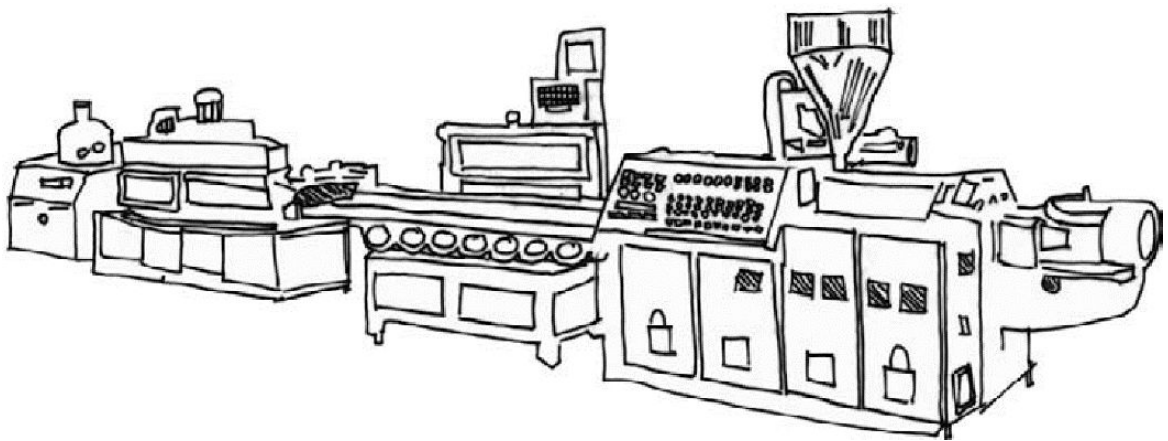


图1-7 函数式程序类似于一个用来转换数据的机器

当然，这种装配线的类比并不完全准确，因为我们知道每个机器生产产品都需要消耗加工原料。相比之下，函数式编程以命令式的方式构建系统，并通过将显性的状态改变缩减到最小来变得更加模块化（Hughes，1984）。实践中的函数式编程并不以消除状态改变为主要目的，而是将任何已知系统中突变的出现尽量压缩到最小区域中去。

1.2.2 以函数为抽象单元

抽象方法是指隐藏了实现细节的函数。事实上，函数是一种非常好的工作单元，它使得我们能够坚持一句由巴特勒兰普森提出的、在UNIX社区长期奉行的格言：

使之运行，使之正确，使之快速。

同样，抽象函数使得我们能够完全理解Kent Beck的关于测试驱动开发（TDD）的类似的说法：

使之运行，再使之正确，再使之快速。

例如，对于错误和警告的报告，我们可以写出如下代码：

```
function parseAge(age) {  
  if (!_isString(age)) throw new Error("Expecting a string");  
  var a;  
  
  console.log("Attempting to parse an age");  
  
  a = parseInt(age, 10);  
  
  if (_isNaN(a)) {  
    console.log(["Could not parse age:", age].join(' '));  
    a = 0;  
  }  
  
  return a;  
}
```

虽然这个函数并不能全面地解析年龄字符串，但却是一个好例子。我们可以用如下方法来调用parseAge：

```
parseAge("42");  
// (console) Attempting to parse an age  
//=> 42
```

```
parseAge(42);  
// Error: Expecting a string
```

```
parseAge("frob");  
// (console) Attempting to parse an age  
// (console) Could not parse age: frob  
//=> 0
```

parseAge函数工作正常，但如果我们想修改输出错误、信息和警告呈现的方式，那么就需要修改相应的代码行，以及其他地方的类似输出模式。一个较好的方法是将错误、信息和警告的概念抽象成不同的函数：

```
function fail(thing) {  
    throw new Error(thing);  
}
```

```
function warn(thing) {  
    console.log(["WARNING:", thing].join(' '));  
}
```

```
function note(thing) {  
    console.log(["NOTE:", thing].join(' '));  
}
```

有了这些函数，我们就可将parseAge函数改写成：

```

function parseAge(age) {
  if (!_.isString(age)) fail("Expecting a string");
  var a;

  note("Attempting to parse an age");
  a = parseInt(age, 10);

  if (_.isNaN(a)) {
    warn(["Could not parse age:", age].join(' '));
    a = 0;
  }

  return a;
}

```

下面是新函数的行为：

```

parseAge("frob");
// (console) NOTE: Attempting to parse an age
// (console) WARNING: Could not parse age: frob
//=> 0

```

新的行为与旧的行为差别不大，不同的是现在报告错误、信息和警告的想法已经被抽象化了。错误、信息和警告的报告结果也因此完全被修改：

```

function note() {}
function warn(str) {
    alert("That doesn't look like a valid age");
}

parseAge("frob");
// (alert box) That doesn't look like a valid age
//=> 0

```

因此，由于行为包含在单一的函数中，所以函数可以被能够提供类似行为的新函数取代，或直接被完全不同的行为所取代（Abelson and Sussman, 1996）。

1.2.3 封装和隐藏

多年来，我们一直被教导说封装是面向对象的基石。在面向对象术语中，封装是指一种将若干个数据与用来操纵它们的特定操作包装起来的方式，如图1-8所示。

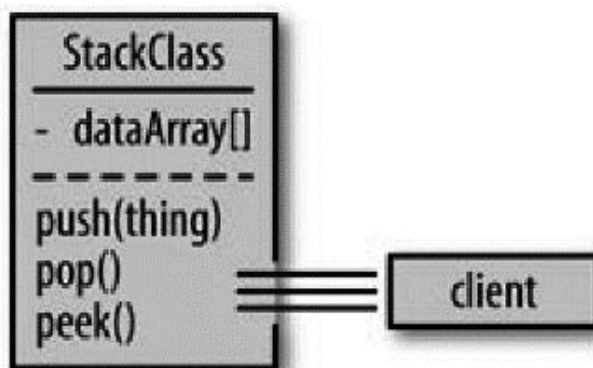


图1-8 大多数面向对象语言使用对象边界来包装数据元素和它们的操作；因此，一个Stack类将一个元素的数组和用来操作这个数组的push、pop和peek方法包装在一起

JavaScript提供了一个对象系统，它也确实能够封装数据与操作。然而，有时封装被用来限制某些元素的可见性，称为数据隐藏。在JavaScript的对象系统中，并没有提供直接隐藏数据的方式，因此使用一种叫做闭包的方式来隐藏数据，如图 1-9所示。

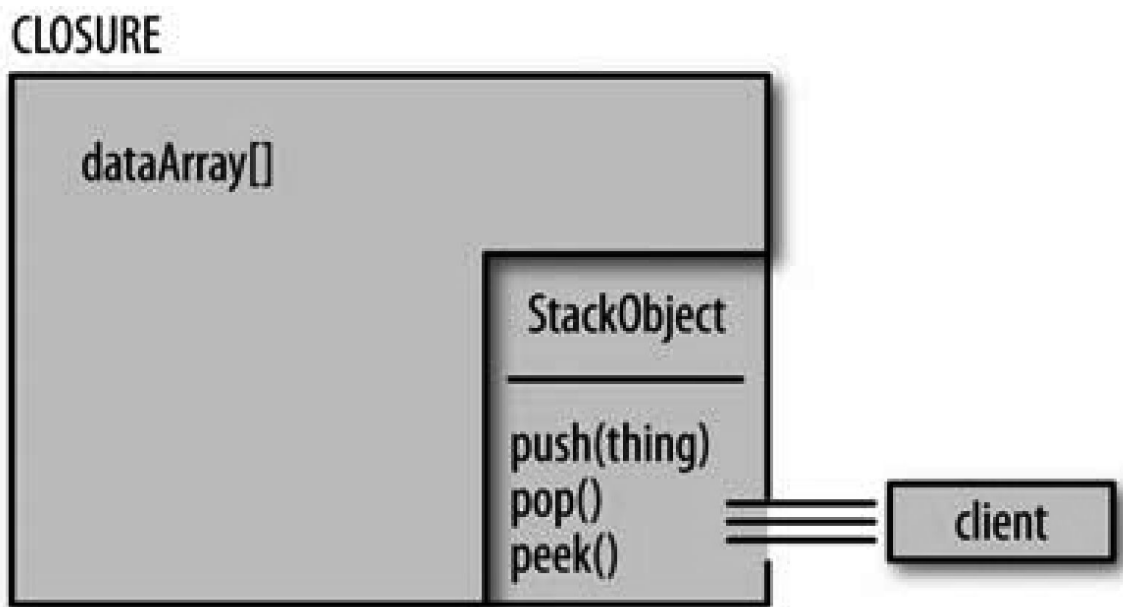


图1-9 使用闭包来封装数据是一种函数式的向客户端隐藏细节的方式

在第3章之前，我们不会深入介绍闭包，但现在需要你记住的是，闭包也是一种函数。通过使用包含了闭包的函数式技术，我们能够与大多数面向对象语言一样，实现有效的数据隐藏，尽管我不愿说函数式封装和面向对象式封装究竟谁更好。虽然在实践中它们是不同的，但它们实际上都提供了建立某种抽象的类似的方法。事实上，本书并不鼓励大家为了学习函数式编程而扔掉曾经学到的一切，从而喜欢学习函数式编程；相反，我们旨在就其本身来讨论函数式编程，这样你就可以确定它是否合适你的需求。

1.2.4 以函数为行为单位

隐藏数据和行为（通常不方便于快速修改）只是一种将函数作为抽象单元的方式。另外一种方式是提供一种简单地存储和传递基本行为的离散单元。举个例子，用JavaScript语法来索引数组中的一个值：

```
var letters = ['a', 'b', 'c'];  
  
letters[1];  
//=> 'b'
```

虽然数组索引是JavaScript的一个核心行为，但并没有办法可以在不把它放到函数里的前提下，获取这个行为并根据需要来使用它。因此，举

一个函数的简单例子就是抽象数组索引行为，我们称它为nth。nth的简单实现如下所示：

```
function naiveNth(a, index) {  
  return a[index];  
}
```

或许正如你所猜测的，nth的主逻辑工作正常：

```
naiveNth(letters, 1);  
//=> "b"
```

然而，当传入意想不到的值时，nth就会出错：

```
naiveNth({}, 1);  
//=> undefined
```

因此，如果想围绕nth来实现函数抽象，我们或许会设计出下面的声明：nth返回一个存储在允许索引访问的数据类型中的有效元素。这段声明的关键在于索引数据类型的概念。为了判断什么是索引的数据类型，我们可以创建一个isIndexed函数，实现如下所示：

```
function isIndexed(data) {  
  return _.isArray(data) || _.isString(data);  
}
```

函数isIndexed也是一个提供了判断某个数据是否是字符串或数组的函数抽象。在抽象之上实现新的抽象，nth的实现也就如下所示：

```
function nth(a, index) {  
  if (!_.isNumber(index)) fail("Expected a number as the index");  
  if (!isIndexed(a)) fail("Not supported on non-indexed type");  
  if ((index < 0) || (index > a.length - 1))  
    fail("Index value is out of bounds");  
  
  return a[index];  
}
```

完整的nth函数的使用方法如下所示：

```
nth(letters, 1);  
//=> 'b'
```

```
nth("abc", 0);  
//=> "a"
```

```
nth({}, 2);  
// Error: Not supported on non-indexed type
```

```
nth(letters, 4000);  
// Error: Index value is out of bounds
```

```
nth(letters, 'aaaaa');  
// Error: Expected a number as the index
```

与amp我们从index抽象中构建nth函数抽象的方式一样，我们也可以以同样的方式来构建一个second抽象：

```
function second(a) {  
  return nth(a, 1);  
}
```

函数second允许我们在一个不同但相关的情况下，正确使用nth函数：

```
second(['a', 'b']);  
//=> "b"
```

```
second("fogus");  
//=> "o"
```

```
second({});  
// Error: Not supported on non-indexed type
```

另外一个JavaScript的基本行为单元是比较器。比较器是一个函数，它接受两个参数，如果第一个参数值小于第二个参数值，则返回<1；如果第一个参数值大于第二个参数值，则返回>1；如果两个参数值相等，则返回0。事实上，JavaScript本身似乎可以利用数字本身的性质，提供一个默认的sort方法：

```
[2, 3, -6, 0, -108, 42].sort();  
//=> [-108, -6, 0, 2, 3, 42]
```

但是当有不同数据混合出现时，就会出错：

```
[0, -1, -2].sort();  
//=> [-1, -2, 0]
```

```
[2, 3, -1, -6, 0, -108, 42, 10].sort();  
//=> [-1, -108, -6, 0, 10, 2, 3, 42]
```

问题在于，在没有给定参数的情况下，Array#sort方法执行字符串的比较。然而，每一个JavaScript程序员都知道，Array#sort需要一个比较器，因此应该写成：

```
[2, 3, -1, -6, 0, -108, 42, 10].sort(function(x,y) {  
    if (x < y) return -1;  
    if (y < x) return 1;  
    return 0;  
});  
  
//=> [-108, -6, -1, 0, 2, 3, 10, 42]
```

现在看起来似乎好多了，但是有更为通用的方法。毕竟我们可能还要在其他的代码中用到这样的排序，所以把这个匿名函数抽出来，给它起一个名字，或许会更好一些：

```
function compareLessThanOrEqual(x, y) {
  if (x < y) return -1;
  if (y < x) return 1;
  return 0;
}
```

```
[2, 3, -1, -6, 0, -108, 42, 10].sort(compareLessThanOrEqual);
//=> [-108, -6, -1, 0, 2, 3, 10, 42]
```

但函数compareLessThanOrEqual的问题在于，它被耦合到了“比较器”的概念当中，并不容易被单独当作一个通用的比较器来用：

```
if (compareLessThanOrEqual(1,1))
  console.log("less or equal");
```

```
// nothing prints
```

为了达到预期的效果，我们需要了解函数compareLessThanOrEqual作为一个比较器的性质：

```
if (_.contains([0, -1], compareLessThanOrEqual(1,1)))
  console.log("less or equal");
```

```
// less or equal
```

但这并不令人满意，特别是将来函数compareLessThanOrEqual的返回值可能会被其他开发人员修改为用-42来代表比较结果。一个较好的实现compareLessThanOrEqual函数的方式是：

```
function lessOrEqual(x, y) {
  return x <= y;
}
```

总是返回一个布尔值（只会返回true或false）的函数被称为谓词。因此，函数lessOrEqual事实上并不是一个精心设计的比较器，它只是对内建操作符<=的一个简单包装。

```
[2, 3, -1, -6, 0, -108, 42, 10].sort(lessOrEqual);  
//=> [100, 10, 1, 0, -1, -1, -2]
```

看到这里，你恐怕会有转行的意向。但是，更深一步考虑，这事实上是合理的。如果sort函数需要一个比较器，并且lessThan函数只会返回true或false，那么我们需要通过某种方式在不重复一堆的if/then/else模板的情况下，从后者的世界转换到前者当中来。解决方案是创建一个comparator函数，它接受一个谓词，并将其结果转化成comparator函数所期待的-1/0/1：

```
function comparator(pred) {  
  return function(x, y) {  
    if (truthy(pred(x, y)))  
      return -1;  
    else if (truthy(pred(y, x)))  
      return 1;  
    else  
      return 0;  
  };  
};
```

现在，我们可以用comparator函数来返回一个能够将谓词lessOrEqual的结果（true或false）“映射”成比较器所期待的结果（-1，0或1）的新函数了，如图1-10 所示。

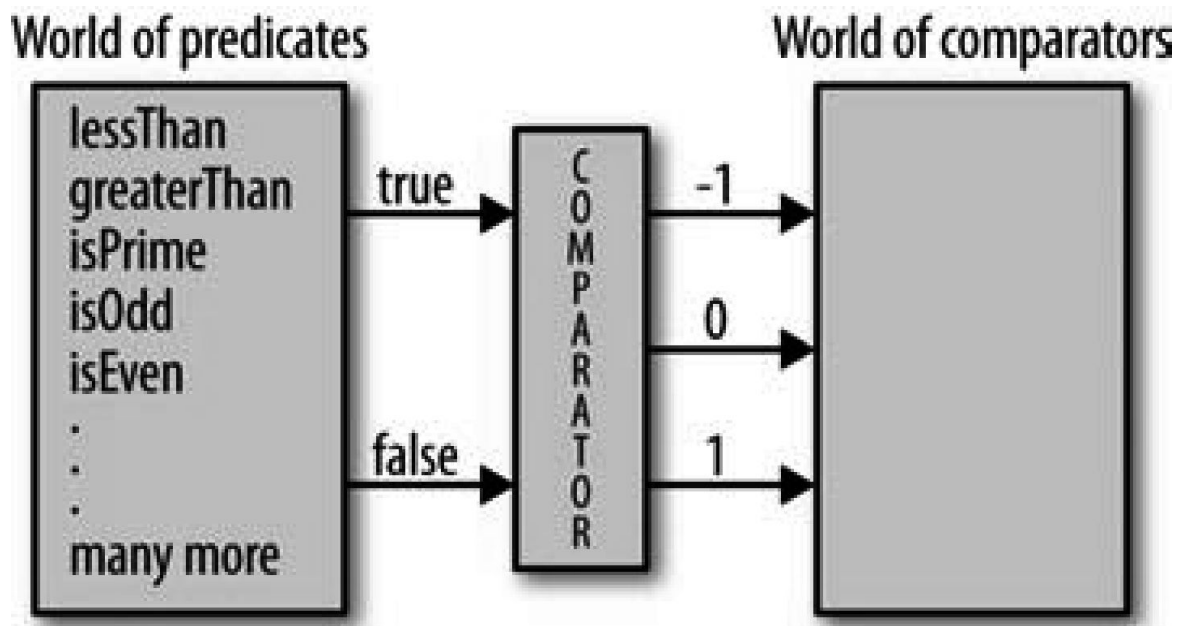


图1-10 用比较器功能来桥接两个“世界”之间的差异

在函数式编程中，我们将经常看到这类用于允许将一种类型数据转换为另一种类型数据的函数。我们来看看comparator的用法：

```
[100, 1, 0, 10, -1, -2, -1].sort(comparator(lessOrEqual));  
//=> [-2, -1, -1, 0, 1, 10, 100]
```

comparator函数可以将任何返回“真”或“假”的函数映射到“比较器”的概念上来。这个话题将在第4章进行更深入的讨论，但是值得注意的是，comparator是一个高阶函数（因为它接受一个函数，并返回一个新的函数）。请记住，并不是每一个谓词都应该与comparator函数一起使用。例如，将_.isEqual函数作为一个comparator的基础意味着什么？尝试一下，看看会发生什么。

在本书中，我们将会讨论多种用函数式技术提供并促进创建抽象的方式。正如我们接下来要讨论的，抽象的函数与数据之间有一个漂亮的协同作用。

1.2.5 数据抽象

JavaScript的对象原型模型是一个丰富且基础的数据方案。就其本身而言，原型模型提供了在许多其他主流编程语言中没有发现的一定级别的灵活性。然而，出于习惯，许多JavaScript程序员会立即尝试利用原型或

闭包（或两者都用）来建立一个基于类的对象系统^[6]。尽管类系统有其长处，但很多时候一个JavaScript应用程序的数据需求比类中的简单得多^[7]。

相反，使用JavaScript的原始数据、对象和数组，以及目前由类创建的大部分数据模型任务都属于一个范畴。从历史上看，函数式编程已经致力于构建能够实现更高层次行为以及能够工作在非常简单的数据结构上的函数^[8]。事实上，在这本书（以及Underscore）中，重点是如何处理数组和对象。这两个简单数据类型的灵活性是惊人的。不幸的是，它们常常被因为另一种基于类的系统而被忽视。

假设我们有一个任务，需要用编写JavaScript程序来处理逗号分隔值（CSV）文件（一种用来代表数据表的标准方法）。例如，假设我们有一个如下所示的 CSV 文件：

```
name,   age, hair
Merble, 35,  red
Bob,    64,  blonde
```

很明显，这个数据代表一个有三列（姓名、年龄和头发）和三行（第一个是标题行，并且其余的为数据行）的表。一个用来解析这个非常有限的CSV格式表示的字符串的小函数的实现如下：

```
function lameCSV(str) {
  return _.reduce(str.split("\n"), function(table, row) {
    table.push(_.map(row.split(","), function(c) { return c.trim();}));
    return table;
  }, []);
};
```

我们发现，函数lameCSV一行接一行地处理，用\n分离出行，再将每一个表格中的空白去除^[9]。整个数据表是一个包含了数组的数组，每个数组都包含了字符串。从表1-1 所示的概念图可以看出，嵌套的数组可以被看作一个表。

表1-1 简单的嵌套数组是一种抽象的数据表的方式

姓 名	年 龄	头 发 颜 色
Merble	35	red
Bob	64	blonde

姓 名	年 龄	头 发 颜 色
Merble	35	红色
Bob	64	金黄

如下所示，使用lameCSV来解析存储在一个字符串中的数据：

```
var peopleTable = lameCSV("name,age,hair\nMerble,35,red\nBob,64,blonde");

peopleTable;
//=> [{"name", "age", "hair"},
//     ["Merble", "35", "red"],
//     ["Bob", "64", "blonde"]]
```

使用选择性间距突出了返回数组的表性质。在函数式编程中，像lameCSV这样的函数以及先前定义的comparator是将一个数据类型转换为另一个的关键。图 1-11 描述了一般的数据转换是如何被看作从一个“世界”进入另一个“世界”的。

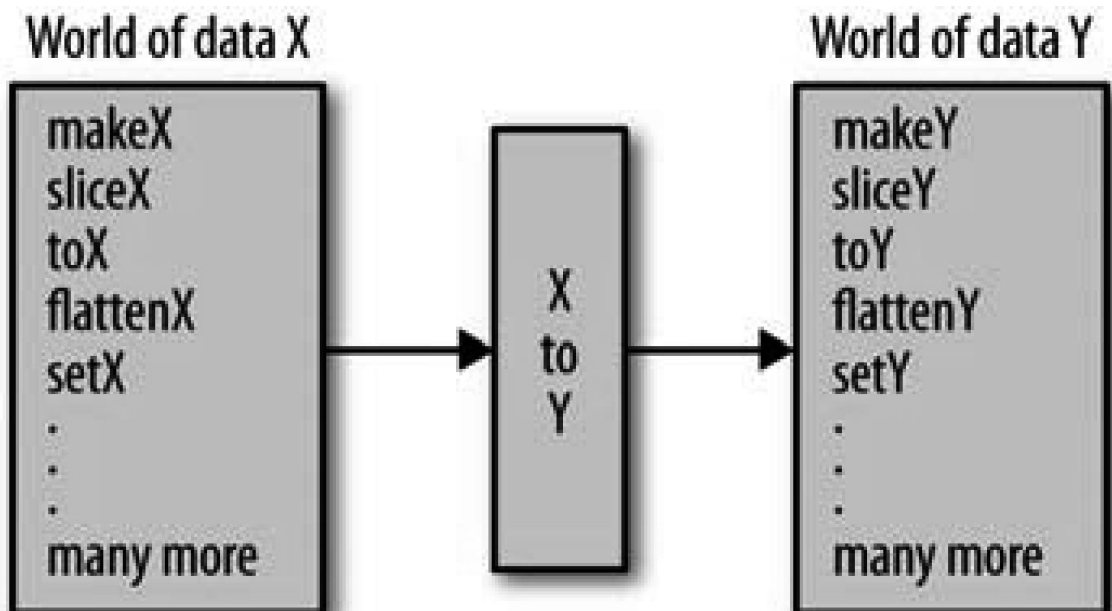


图1-11 函数是跨越两个“世界”之间的桥梁

我们有更好的方法来表示表数据，但是这个嵌套数组目前来说已经足够了。事实上，很少有动机建立一个复杂的类层次结构来代表无论是表本身、行、人或任何其他数据。相反，保持数据表示最小，使得我们可以方便地使用已有的数组字段和数组处理函数和方法：

```
_.rest(peopleTable).sort();

//=> [ ["Bob",    "64",  "blonde"],
//     ["Merble", "35",  "red"] ]
```

同样，由于我们知道原始数据的形式，可以创建更有描述性的名字选择函数来访问数据：

```
function selectNames(table) {
  return _.rest(_.map(table, _.first));
}

function selectAges(table) {
  return _.rest(_.map(table, second));
}

function selectHairColor(table) {
  return _.rest(_.map(table, function(row) {
    return nth(row, 2);
  }));
}

var mergeResults = _.zip;
```

这里定义的select函数使用了现有的数组处理函数，帮助我们流畅地访问简单数据类型：

```
selectNames(peopleTable);  
//=> ["Merble", "Bob"]  
  
selectAges(peopleTable);  
//=> ["35", "64"]  
  
selectHairColor(peopleTable);  
//=> ["red", "blonde"]  
  
mergeResults(selectNames(peopleTable), selectAges(peopleTable));  
//=> [["Merble", "35"], ["Bob", "64"]]
```

一个能令人信服的说法是，实施和使用的简易性是使用JavaScript的核心数据结构进行数据建模的目的。这并不是说面向对象或基于类的方法就完全没有用。根据我的经验，我发现以处理集合为中心的函数式方式更适合处理与人有关的数据，而面向对象的方法最适合模拟人^[10]。

如果愿意的话，我们可以将数据表改为自定义的基于类的模型。只要你使用选择器抽象，那么用户将永远不知道，也不关心。然而，在这本书中，我努力保持数据需求尽可能简单，并构建操作这些数据的抽象函数。有趣的是，通过将自己约束在对简单数据的操作上，增加了灵活性。你可能会对这些基本类型将会带我们走多远感到惊讶。

1.2.6 函数式JavaScript初试

这不是一本围绕JavaScript众多怪癖的书。已经有很多其他的书以这种方式来帮助你学习JavaScript。然而，在开始任何JavaScript项目之前，这里定义了两个我们常常会需要的有用的函数：existy和truthy。

函数existy旨在定义事物的存在。JavaScript中有两个值表示不存在——null和undefined。因此，existy函数主要检查其参数是否是这类值，它的实现如下：

```
function existy(x) { return x != null };
```

使用松散不等式运算符（!=），就可以区分null，undefined和其他所有对象。existy函数的使用方法如下所示：

```
existy(null);  
//=> false
```

```
existy(undefined);  
//=> false
```

```
existy({}.noHere);  
//=> false
```

```
existy((function(){})(()));  
//=> false
```

```
existy(0);  
//=> true
```

```
existy(false);  
//=> true
```

使用existy函数简化了JavaScript中对象是否存在的判断。至少，它将存在性检查并置成了一个简单易用的函数。上面说到的第二个函数truthy的定义如下所示^[11]。

```
function truthy(x) { return (x !== false) && existy(x) };
```

函数truthy用来判断一个对象是否应该被认为是true的同义词，它的使用方法如下所示^[12]。

```
truthy(false);  
//=> false
```

```
truthy(undefined);  
//=> false
```

```
truthy(0);  
//=> true
```

```
truthy('');  
//=> true
```

在JavaScript中，有时只有在某个条件为真的情况下执行某些操作，否则返回类似undefined或null的值。一般模式如下所示：

```
{
  if(condition)
    return _.isFunction(doSomething) ? doSomething() : doSomething;
  else
    return undefined;
}
```

使用truthy函数，我们可以将该逻辑通过以下方式封装起来：

```
function doWhen(cond, action) {
  if(truthy(cond))
    return action();
  else
    return undefined;
}
```

现在，每当出现这种丑陋的模式，我们可以用以下操作来代替^[13]：

```
function executeIfHasField(target, name) {
  return doWhen(existy(target[name]), function() {
    var result = _.result(target, name);
    console.log(['The result is', result].join(' '));
    return result;
  });
}
```

函数executeIfHasField的成功执行和出错的情况如下所示：

```
executeIfHasField([1,2,3], 'reverse');  
// (console) The result is 3, 2, 1  
//=> [3, 2, 1]
```

```
executeIfHasField({foo: 42}, 'foo');  
// (console) The result is 42  
//=> 42
```

```
executeIfHasField([1,2,3], 'notHere');  
//=> undefined
```

没什么大不了的，对不对？所以，我们定义了两个函数，这很难称得上是函数式编程。函数式理念来自于它们的使用。你可能已经熟悉了在许多JavaScript实现中的Array#map的方法。它旨在接收一个函数，用一个数组中的每一个元素来调用它，并返回一个存储了新值的新数组。它的使用方法如下所示：

```
[null, undefined, 1, 2, false].map(existy);  
//=> [false, false, true, true, true]
```

```
[null, undefined, 1, 2, false].map(truthy);  
//=> [false, false, true, true, false]
```

以下就是函数式编程。

- 一个对“存在”的抽象函数的定义。
- 一个建立在存在函数之上的，对“真”的抽象函数的定义。
- 通过其他函数来使用上面的两个函数，以实现更多的行为。

类似于这样的代码会遍布在本书之中。

1.2.7 加速

我知道此刻你在想什么。这函数式编程执行起来必定慢得让人受不了，是吗？

没有办法否认的是，使用数组索引形式的`array[0]`会执行得比任何`nth(array, 0)`或`_.first(array)`都快。同样，以下形式的命令式循环也是非常快的^[14]：

```
for (var i=0, len=array.length; i < len; i++) {  
    doSomething(array[i]);  
}
```

类似的功能，在所有因素都相同的情况下，使用Underscore的`_.each`函数确实会慢一些：

```
_.each(array, function(elem) {  
    doSomething(array[i]);  
});
```

然而，很有可能的是所有因素不会相等。当然，如果一个函数对执行速度有需求，那么一个将内部使用的`_.each`转换成类似功能的`for`或`while`的工作是合理的。令人高兴的是，笨重缓慢的JavaScript的日子即将结束，在某些情况下已经是过去的事情了。例如，谷歌的V8引擎的发布引来了一直在向所有JavaScript引擎供应商推动的（BAK 2012）运行时优化的时代^[15]。即使其他厂商没有跟随谷歌的带领，V8引擎的使用率仍然在增长，而事实上，它驱动着非常流行的Chrome浏览器和Node.js本身。然而，其他厂商都是跟着V8引擎的引领，并引入了运行时速度增强功能，如本机代码的执行，即时编译，更快的垃圾收集，客户端缓存，并嵌入到他们自己的JavaScript引擎中^[16]。

然而，对于一些JavaScript程序员来说，对老浏览器——如Internet Explorer 6——的支持是一个非常现实的要求。当面对传统平台时，有两个因素要考虑：（1）IE6及其同类型浏览器的使用正在逐渐消失，（2）在代码到达浏览器之前，有其他的方法来提高速度^[17]。例如，在代码内联是一个有趣的话题，因为许多采用代码内联的优化工作可以静态地展开，或者甚至是在代码运行之前展开。内联代码是将一段代码放在一个函数中，并“粘贴”到调用函数的地方中去。让我们来看一个例子，以便更清楚地理解。在Underscore的`_.each`方法的实现内部，是一个类似于如前面所示`for`循环的循环代码（为清楚起见，对原代码做了一些修改）：


```

_.each = function(obj, iterator, context) {
  // bounds checking
  // check for native method
  // check for length property
  for (var i = 0, l = obj.length; i < l; i++) {
    // call the given function
  }
}

```

假设我们有一段代码，类似于：

```

function performTask(array) {
  _.each(array, function(elem) {
    doSomething(array[i]);
  });
}

// ... some time later

```

```
performTask([1,2,3,4,5]);
```

静态优化器可以将performTask的函数体转换为如下样子：

```

function performTask(array) {
  for (var i = 0, l = array.length; i < l; i++) {
    doSomething(array[i]);
  }
}

```

成熟的优化工具可以通过完全消除函数调用来对其进行进一步的优化：

```
// ... some time later

var array123 = [1,2,3,4,5];

for (var i = 0, l = array123.length; i < l; i++) {
    doSomething(array[i]);
}
```

最后，非常棒的静态分析器甚至可以将它进一步优化为五个独立的调用：

```
// ... some time later

doSomething(array[1]);
doSomething(array[2]);
doSomething(array[3]);
doSomething(array[4]);
doSomething(array[5]);
```

最理想的优化转换器是这样的：假设上面的调用不会带来什么影响或根本不会被调用，那么最优变换为：

```
// ... some time later
```

也就是说，如果一段代码可以被确定为“死代码”（不被调用），那么它可以安全地通过代码省略（code elision）来消除。目前已经有针对

于JavaScript的此类优化器——以谷歌的闭包编译器（Google's Closure compiler）为主。闭包编译器是一个非常好的能够将JavaScript高度优化的工程^[18]。

有很多种不同的基于最佳实践和优化工具组合的方式，能够实现加速甚至是高度函数化的代码。可是，很多时候我们太急于考虑运算速度，甚至是在写出一段正确的代码之前就开始考虑运算速度。同样，有时候我发现我在不断依据速度来考虑问题，即使我所创建的系统根本不要考虑速度。Underscore是一个非常流行的函数式编程JavaScript库，而大量的应用程序只是用用它而已。对于成就了许多函数式习语的JavaScript库重量级冠军——jQuery——也是同样。

当然，也有以速度考虑为首的场景（例如游戏编程和低延迟系统）。然而，即使在这样的系统的执行要求面前，函数式技术也并不一定会降低速度。我们应该不会希望将类似于nth的函数放在渲染界面的核心代码中，但总体而言，函数式结构仍然可以带来好处。

对我个人而言，编程风格的第一条规则是：写漂亮的代码。在我的职业生涯中，我已在不同程度上实现了这个目标，但它仍然是我追求的东西。写漂亮的代码使得我从另一个方面优化了时间：坐在一张桌子边打字的时间。我发现如果做得好，函数式代码风格可以很漂亮。我希望在快要看完本书的时候，你会同意这个观点。

1.3 Underscore示例

在开始有趣的内容之前，我想解释一下为什么选择用Underscore来作为本书的表述方式。首先，Underscore是一个非常好的库，它提供了一套漂亮实用的函数式风格API。如果我从头开始重新实现一遍所有对理解函数式编程有用的函数是没有意义的。为什么需要在“映射化”的理念更重要时候来实现map方法？当然，并不是说我不会在本书中实现核心的函数式工具，但还是以Underscore为基础^[19]。

其次，读者在练习时可能会发现对Array#map调用不起作用。这种问题的原因可能是，运行环境中没有实现数组的 map 方法。另外，我也想尽可能地避免陷入跨浏览器兼容问题泥潭。在学习过程中，这种类型的影响是非常重要的，它会使得我在向大家介绍函数式编程的过程中分心去处理这类问题。使用Underscore几乎可以完全消除这类影响^[20]。

最后，JavaScript的本质使得程序员能够经常重新发明轮子。JavaScript本身可将强大的低级别构建与中高层语言完美地组合。正是这种奇怪的情况，使得人们几乎不敢用较低级别部件来创造新的语言特性。语言本身的进化可以消除重新造轮子的需求（例如模块系统），但我们不太可能看到这种需求的完全消失^[21]。不过，我们相信在可以的条件，应该重用已有的高品质代码库^[22]。重新实现Underscore的功能将会非常有趣，但这并不能给你我（或者我的员工们）带来很大的益处。

1.4 总结

以学习和使用JavaScript为动机，本章涵盖了一些介绍性主题。在现有稳定的主流编程语言中，很少有可以跟JavaScript的增长趋势相匹敌的。同样，这种增长潜力似乎是无限的。然而，JavaScript是一门有缺陷的语言，它需要依赖于强大的技术、规范或两者的混合才能有效地被运用起来。一种用于构建JavaScript应用程序的技术称为“函数式编程”。概括地说，它包括以下技术。

- 确定抽象，并为其构建函数。
- 利用已有的函数来构建更为复杂的抽象。
- 通过将现有的函数传给其他的函数来构建更加复杂的抽象。

但是，仅仅只是构建函数是不够的。事实上，与强大的数据抽象相结合来实现函数式编程往往效果最好。在函数式编程和数据之间，存在一个美丽的对称性。下一章会对该对称性进行深入的讨论。

[1] 与所有的工具一样，如果你不小心，还是可能切到或粉碎你的拇指。

[2] 可以在http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts查看ES.next的草案。

[3] 值得讨论的是缺陷有多严重。

[4] 一些能编译成JavaScript的语言有ClojureScript, CoffeeScript, Roy, Elm, TypeScript, Dart, Flapjax, Java。

[5] 这种思路比较容易从面向对象转换到函数式编程上，而且我将在接下来的书中混合这两种思路。

[6] ECMAScript.next 正讨论支持类的可能性。然而，这个特性颇受争议。不管怎么样，类不知道什么时候才能加入到JavaScript。

[7] 基于类的对象系统的一个有力的论据是实现用户界面的历史使用。

[8] 不久你就能看到函数式是注重列表数据结构。而对于JavaScript来说，array就可以替代该数据结构。

[9] 函数lameCSV在这里只用作说明用途，并不是完整功能的CSV 解释器。

[10] 这种面向对象的范式从模拟社区如雨后春笋般以simula的编程语言形式涌现出来并不是巧合。通过写模拟系统，我强烈感觉到面向对象或基于角色的模型很适合用simula。

[11] 我这里定义的truthy是指JavaScript的原始真类型。虽然知道JavaScript认为什么为真很重要，我还是在我的应用中简化这些规则。

[12] 这里数字0是故意设计成“truthy”的。原本代表false是继承自C语言。如果你还想使用该属性，就在期待0的时候不要用truthy函数。

[13] 我使用existy(target[name])而不是Underscore的has(target, name)，是因为后者只会检查自己的字段。

[14] 我用来评测JavaScript性能的网站是<http://www.jsperf.com>。

[15] 在任何故事中都是有前传的。在V8之前，WebKit项目用SquirrelFish引擎编译JavaScript代码。在SquirrelFish之前是Tamarin VM，它由Mozilla基于Adobe的ActionScript VM 2开发。有趣的是，大部分JavaScript的优化技术都来自Self和Smalltalk这些老的语言。

[16] 不懂这些优化技术没关系，它们不是本书的重点。但我还是强烈推荐研究这些话题。

[17] 有一个有意思的网站监测全球IE6的使用率：<http://www.ie6countdown.com>。

[18] 使用Google Closure compiler时生成的特殊风格的代码颇受争议。但是正如我在ClojureScript编译器工作时学到的，如果有用的话，就会特别有用。

[19] 还有一些其他的函数式编程库如Functional JavaScript, Bilby 甚至jQuery。然而我选择的是Underscore。

[20] 当接触到底层方法调用时，Underscore会考虑到浏览器兼容性的问题。

[21] 我觉得这是编程的本质。

[22] 我特别迷恋于用microjs网站来发现有趣的JavaScript库（译者注：也可以到该网站的github上添加你觉得有趣的JavaScript库——<https://github.com/madrobby/microjs.com>）。

第2章 一等函数与Applicative编程

本章将介绍函数式编程中的一等公民——函数。我会从map、reduce和filter这三个常用函数开始介绍。由于程序员对这些函数比较熟悉，以此为出发点将会为本书后面的内容打下基础。

2.1 函数是一等公民

很多熟悉JavaScript的程序员，包括我自己在内，都认为这是一种函数式语言。当然，也有人对此表述有分歧。产生这种分歧的原因在于，对函数式编程的定义往往是相对的，就像从次要和主要的区别来区分行动派和思想家一样^[1]。

这种分歧事实上并不是件好事。但值得庆幸的是，各种关于函数式编程的观点似乎都同意一点：函数式编程语言应该是促进创造和使用函数的。

函数式编程通常还伴随着一些其他定义，包括但不限于静态类型、模式匹配、不变性、纯度等。然而，使用这些特性来验证函数式编程语言的某些实现通常是不可行的。如果要由其本质来定义的话，只需要符合“促进”和“一等函数”这两点，那么无论Haskell还是JavaScript都可以涵盖入内，后者则是本书的重点。谢天谢地终于用一段话解释完了函数式编程，看来还需要再用一段话来解释一等公民函数^[2]。

“一等”这个术语通常用来描述值。当函数被看作“一等公民”时，那它就可以去任何值可以去的地方，很少有限制。比如数字在JavaScript里就是一等公民，同样作为一等公民的函数就会拥有类似数字的性质。

- 函数与数字一样可以存储为变量。

```
var fortytwo = function() { return 42 };
```

- 函数与数字一样可以存储为数组的一个元素。

```
var fortytwos = [42, function() { return 42 }];
```

- 函数与数字一样可以作为对象的成员变量。

```
var fortytwos = {number: 42, fun: function() { return 42 }};
```

- 函数与数字一样可以在使用时直接创建出来。

```
42 + (function() { return 42 })();  
//=> 84
```

- 函数与数字一样可以被传递给另一个函数。

```
function weirdAdd(n, f) { return n + f() }
```

```
weirdAdd(42, function() { return 42 });  
//=> 84
```

- 函数与数字一样可以被另一个函数返回。

```
return 42;
```

```
return function() { return 42 };
```

最后两点其实就是“高阶”函数的定义：一个高阶函数应该可以执行下列至少一项操作。

- 以一个函数作为参数。
- 返回一个函数作为结果。

第1章列举了一个comparator的高阶函数例子，在这里我们再给出一个例子：

```
_.each(['whiskey', 'tango', 'foxtrot'], function(word) {  
  console.log(word.charAt(0).toUpperCase() + word.substr(1));  
});  
  
// (console) Whiskey  
// (console) Tango  
// (console) Foxtrot
```

Underscore的`_.each`函数接受一个集合（对象或数组），并遍历其元素，然后调用作为`each`的第二个参数被传进来的函数。

我会在第4章 更深入地讲解高阶函数。现在，我会再用几个篇幅来谈谈JavaScript本身，因为正如你可能已经知道的，JavaScript不仅支持函数式编程风格，也同样支持一些其他的编程范式。

多种JavaScript编程方式

当然，JavaScript并不仅限于函数式编程语言，使用其他编程方式也是很方便的。

(1) 命令式编程

通过详细描述行为的编程方式。

(2) 基于原型的面向对象编程

基于原型对象及其实例的编程方式。

(3) 元编程

对JavaScript执行模型数据进行编写和操作的编程方式。

仅用命令式、面向对象和元编程会限制我们使用这些语言结构本身直接支持的编程范式。其实，你可以进一步支持其他范式，比如面向类型和事件编程，或是用语言本身作为实现媒介，但是这本书不会深入这些话题。在我开始进行定义和详细介绍JavaScript对一等函数的支持之前，让我再花一点时间阐明上述三种范式与函数式编程的区别。我将每一种范式的深入讨论贯穿整本书中，所以从现在开始可能再用一两段话就足够过渡到函数式编程的讨论中。

1. 命令式编程

命令式编程风格是由其细腻的（而且往往令人生气的）注意算法的实现细节来进行分类的。此外，命令式编程往往是建立在直接操作和检查程序状态之上。例如，假设你想编写一个程序来为歌曲“99瓶啤酒”作词，用最直接的方式来描述这个程序的要求。

- 开始数字 X 为99。
- 重复唱下面内容直到数字降到1。
 - 墙上有 X 瓶啤酒。
 - X 瓶啤酒。
 - 拿一个下来，分给大家。
 - 墙上还有 $X-1$ 瓶啤酒。
- 把最后得到的数字减1，得到的结果作为 X 重新开始。

- 当最终 x 变成1，最后一句改为：

——墙上已经没有啤酒了。

事实证明，这个描述可以直接翻译成JavaScript的命令式实现，如下所示：

```
var lyrics = [];

for (var bottles = 99; bottles > 0; bottles--) {
  lyrics.push(bottles + " bottles of beer on the wall");
  lyrics.push(bottles + " bottles of beer");
  lyrics.push("Take one down, pass it around");

  if (bottles > 1) {
    lyrics.push((bottles - 1) + " bottles of beer on the wall.");
  }
  else {
    lyrics.push("No more bottles of beer on the wall!");
  }
}
```

这个命令式的版本虽然有点做作，但却是命令式编程的标志性风格。也就是说，对“99瓶啤酒”的命令式描述的实现就是“99瓶啤酒”的完整程序。因为命令性代码运行在如此细节的层面，它们往往需要一次性实现或尝试最好的，否则难以再利用。此外，命令式语言通常局限于细节，这虽有利于编译器，却不利于程序员（Sokolowski 1991）。

相比之下，这个问题的函数式解决方法如下所示：

```
function lyricSegment(n) {
  return _.chain([])
    .push(n + " bottles of beer on the wall")
    .push(n + " bottles of beer")
    .push("Take one down, pass it around")
    .tap(function(lyrics) {
```

```

        if (n > 1)
            lyrics.push((n - 1) + " bottles of beer on the wall.");
        else
            lyrics.push("No more bottles of beer on the wall!");
    })
    .value();
}

```

该lyricSegment函数其实做很少的事情。事实上，它只是产生给定数量的单一的歌词：

```

lyricSegment(9);

//=> ["9 bottles of beer on the wall",
//    "9 bottles of beer",
//    "Take one down, pass it around",
//    "8 bottles of beer on the wall."]

```

函数式编程的思路是将程序拆分并抽象成多个函数再组装回去。按照这种思维方式，你可以想象，函数lyricSegment是作为“99瓶啤酒”程序的歌词生成这部分，被抽了出来。这样，抽象的歌词片段组装成整首歌的程序将如下所示：

```

function song(start, end, lyricGen) {
    return _.reduce(_.range(start, end, -1),
        function(acc, n) {
            return acc.concat(lyricGen(n));
        }, []);
}

```

使用起来也很方便，只要：

```
song(99, 0, lyricSegment);
```

```
//=> ["99 bottles of beer on the wall",  
//    ...  
//    "No more bottles of beer on the wall!"]
```

以这种方式可以让你从一般的歌词装配过程中分离出的逻辑区域（例如生成歌词片段）。如果你愿意，你可以通过将不同的函数，如 `germanLyricSegment` 或 `agreementLyricSegment`，传入函数 `song`，以产生完全不同的歌词。在这本书中，我将逐渐深入解释并使用这种技术。

2. 基于原型的面向对象编程

JavaScript的构造函数是类（至少在实现层面上是如此），这一点非常类似于Java或C#，但使用的方法更为底层。在Java程序中的每个实例是以类作为它的模板生成的，而JavaScript却是利用现有的对象作为原型来生成特定的实例^[3]。这种对象特定，以及将调用导向至被称为“原型链”的内置调度逻辑的编程方式，远比面向类型编程更为底层，但极为优雅且功能强大。我会在后面第9章 谈谈如何利用JavaScript的原型链。

现在，可以用Underscore本身作为一个完美的例证，解释函数也可以像对象中字段的值一样存在：

```
_.each;
```

```
//=> function (array, n, guard) {  
//    ...  
// }
```

这样很优美，不是吗？嗯……不完全是。由于JavaScript是面向对象语言，它必须有一个语义的自引用。但事实上，它的自引用语义与函数式编程的概念是相冲突的。注意观察以下代码：

```
var a = {name: "a", fun: function () { return this; }};  
  
a.fun();  
//=> {name: "a", fun: ...};
```

你会发现，嵌入在函数fun的自引用this返回的是对象a本身。这可能正是你所期望的。但是，注意观察下列代码：

```
var bFunc = function () { return this };
var b = {name: "b", fun: bFunc};

b.fun();
//=> some global object, probably Window
```

这会得到出乎意料的结果。当函数是在对象实例的上下文之外被创建，它的this指向的是全局对象。因此，当我将bFunc绑定到b的字段b.fun后^[4]，其引用并没有更新为b。大多数编程语言都提供了函数式和面向对象的风格，需要权衡的是其处理子引用的方式。JavaScript有它自己的方法，而Python和Scala也有不同的方法。在本书中，你会发现一个面向对象与函数式的风格之间存在先天性的冲突，但Underscore提供了一些工具来消除或是缓解这种冲突。我将在后面进行更深入的讨论。但现在请记住，当我使用“函数”这个词的时候，指独立存在的函数，而“方法”则指的是在对象的上下文中创建出来的函数。

3. 元编程

JavaScript对元编程的支持为其基于原型的面向对象编程提供了便利。许多编程语言都支持元编程，但很少有JavaScript强大。对元编程比较好的定义应该是这样的：编写代码来做一些事情叫作编程，而元编程是当你写的代码改变了某些代码被解释的方式。让我们来看看元编程的一个例子，这样可以更好地理解。

在JavaScript的情况下，this引用的动态性质可以用来元编程。例如，观察下面的构造函数：

```
function Point2D(x, y) {
  this._x = x;
  this._y = y;
}
```

当使用new来生成Point2D函数的对象实例，会得到你所期望的字段值：

```
new Point2D(0, 1);
```

```
//=> {_x: 0, _y: 1}
```

然而，可以使用方法`Function.call`来进行元编程，将`Point2D`派生为新`Point3D`的构造器：

```
function Point3D(x, y, z) {  
    Point2D.call(this, x, y);  
    this._z = z;  
}
```

这样会创建一个正如我们期待的新实例：

```
new Point3D(10, -1, 100);
```

```
//=> {_x: 10, _y: -1, _z: 100}
```

`Point3D`并没有显式地设置`this._x`和`this._y`的值，而是通过调用`Point2D`的`call`方法动态绑定`this`，这样可以改变构造属性的目标。

因为它与函数式编程正交，我不会在这本书太深入介绍JavaScript元编程，但会偶尔用到它^[5]。

2.2 Applicative编程

本书到目前为止，只展示了函数式编程处理函数功能很窄的一个方面，即Applicative编程。Applicative编程定义为函数A作为参数提供给函数B。在这本书中，我不会过多地使用术语“Applicative”，因为这个词出现在不同的上下文中有不同的含义，但是接下来你还会见到这个术语。Applicative编程的三个典型的例子是map、reduce和filter。观察它们是如何运作的：

```
var nums = [1,2,3,4,5];

function doubleAll(array) {
  return _.map(array, function(n) { return n*2 });
}

doubleAll(nums);
//=> [2, 4, 6, 8, 10]

function average(array) {
  var sum = _.reduce(array, function(a, b) { return a+b });
  return sum / _.size(array);
}

average(nums);
//=> 3

/* grab only even numbers in nums */
function onlyEven(array) {
  return _.filter(array, function(n) {
    return (n%2) === 0;
  });
}

onlyEven(nums);
//=> [2, 4]
```

你可能猜到函数`map`、`reduce`和`filter`会在某个地方最终调用作为参数传入的这些匿名函数，事实确实是这样的。实际上，这些函数的语义可以由这个调用关系来定义。

- `_.map`遍历集合并对其每一个值调用一个函数，返回结果的集合。
- `_.reduce`利用函数将值的集合合并成一个值，该函数接收一个积累值和本次处理的值。
- `_.filter`对集合每一个值调用一个谓词函数（返回`true`或`false`的函数），抽取谓词函数返回`true`的值的集合。

函数`map`、`reduce`和`filter`是最简单和最具有象征性的Applicative函数式编程，但Underscore提供了许多其他函数供你使用。在介绍这些函数之前，让我花点时间来解释集合中心编程的概念，它常常与函数式编程一起出现。

2.2.1 集合中心编程

函数式编程对于需要操作集合中元素的任务非常有用。当然，数组`[1, 2, 3, 4, 5]`是一个数字集合，但我们也可以设想，一个对象`{a: 1, b: 2}`是键值对的集合。拿`_.map`使用`_.identity`函数（返回其参数的函数）作为简单例子：

```
_.map({a: 1, b: 2}, _.identity);  
//=> [1, 2]
```

似乎`_.map`只涉及键值对的值部分，但这种限制只是使用上的问题。如果我们要处理键/值对，我们只需要提供一个接受键值对的函数：

```
_.map({a: 1, b: 2}, function(v,k) {  
  return [k,v];  
});  
//=> [['a', 1], ['b', 2]]
```

`_.map`还接收集合本身作为第三个参数：

```
_.map({a: 1, b: 2}, function(v,k,coll) {
  return [k, v, _.keys(coll)];
});
//=> [['a', 1, ['a', 'b']], ['b', 2, ['a', 'b']]]
```

从以集合为中心的角度看，Underscore和一般函数式编程所提倡的是要建立一个统一的处理形式，使我们可以重用一套综合的函数。正如伟大的首位图灵奖获得者Alan Perlis曾经说的：

用100个函数操作一个数据结构，比用10个函数操作10个数据结构要好。

在这本书中，我强调通过用集合中心的泛型函数处理数据的概念。

2.2.2 Applicative编程的其他实例

我提供了一些例子和讨论来作为Applicative编程的收尾。

1. reduceRight

前面已经介绍过`_.reduce`函数，但我没有提到它的兄弟`_.reduceRight`。这两种函数以大致相同的方式操作，不同之处在于`_.reduce`从左至右操作，而`_.reduceRight`从右到左。注意观察差异之处：

```
var nums = [100,2,25];

function div(x,y) { return x/y };

_.reduce(nums, div);
//=> 2

_.reduceRight(nums, div);
//=> 0.125
```

`_.reduce`的操作是 $(100/2) / 25$ ，而`_.reduceRight`是 $(25/2) / 100$ 。如果提供给reduce兄弟的函数符合结合律，那么它们返回的值相同；否

则，这种顺序的差异会造成不同的结果。有很多函数都用到`_.reduceRight`。下面是几个例子：

```
function allOf(/* funcs */) {  
  return _.reduceRight(arguments, function(truth, f) {  
    return truth && f();  
  }, true);  
}
```

```
function anyOf(/* funcs */) {  
  return _.reduceRight(arguments, function(truth, f) {  
    return truth || f();  
  }, false);  
}
```

下面是使用`allOf`和`anyOf`的示例：

```
function T() { return true }  
function F() { return false }
```

```
allOf();  
//=> true  
allOf(T, T);  
//=> true  
allOf(T, T, T , T , F);  
//=> false  
anyOf(T, T, F);  
//=> true  
anyOf(F, F, F, F);  
//=> false  
anyOf();  
//=> false
```

`_.reduceRight`函数在提供惰性求值的语言中更有优势，但由于JavaScript不是这样的语言，计算顺序成为了关键因素（Bird, 1988）[\[6\]](#)。

2. find

`find`函数理解起来非常容易，它接收一个集合和一个谓词函数，并返回该谓词为`true` 时的第一个元素。如下是`find`的一个例子：

```
_.find(['a', 'b', 3, 'd'], _.isNumber);  
//=> 3
```

注意这里使用了内置函数`_.isNumber`作为谓词函数。Underscore有许多谓词函数可以使用，包括`_.isEqual`、`_.isEmpty`、`_.isElement`、`_.isArray`、`_.isObject`、`_.isArguments`、`_.isFunction`、`_.isString`、`_.isNumber`、`_.isFinite`、`_.isBoolean`、`_.isDate`、`_.isRegExp`、`_.isNaN`、`_.isNull`和`_.isUndefined`。我将会在这本书中用到它们中的一些或全部。

3. reject

Underscore的`_.reject`本质上是`_.filter`的逆，它接收一个谓词并返回除了谓词为`true`的值的集合。例如

```
_.reject(['a', 'b', 3, 'd'], _.isNumber);  
//=> ['a', 'b', 'd']
```

这与颠倒`_.filter`的谓词其实是一样的。事实上，一个简单函数补集（complement）就可以完成这样的任务[\[7\]](#)。

```
function complement(pred) {  
  return function() {  
    return !pred.apply(null, _.toArray(arguments));  
  };  
}
```

该`complement`函数接受一个谓词，并返回一个反转谓词结果的函数。然后可以用它将`_.filter`实现`_.reject`相同的效果：

```
_.filter(['a', 'b', 3, 'd'], complement(_.isNumber));  
//=> ['a', 'b', 'd']
```

`complement`函数就是一个高阶函数的例子。虽然我在之前简单地触及过高阶函数的定义，我会推迟更深层次的讨论，直至第3章。

4. `all`

`_.all`函数接收一个集合和一个谓词，当对于所有的元素谓词函数都返回`true`时，返回`true`，例如。

```
_.all([1, 2, 3, 4], _.isNumber);  
//=> true
```

当然，如果任何元素对谓词测试失败，则`_.all`返回`false`。

5. `any`

`_.any`函数接收一个集合和一个谓词，如果任何元素的谓词检查返回了`true`，则返回`true`。例如

```
_.any([1, 2, 'c', 4], _.isString);  
//=> true
```

当然，如果所有的元素谓词都失败，`_.any`返回`false`。

6. `sortBy`，`groupBy`和`countBy`

我将讨论的这最后三个Applicative函数是相关的，因为它们都基于给定的条件函数（criteria function）的结果来做出相应的行动。先解释第一个函数 `_.sortBy`，它接收一个集合和一个函数，并返回由传入的函数确定的条件来对集合排序的结果。例如

```
var people = [{name: "Rick", age: 30}, {name: "Jaka", age: 24}];  
  
_.sortBy(people, function(p) { return p.age });  
  
//=> [{name: "Jaka", age: 24}, {name: "Rick", age: 30}]
```

`_.groupBy`函数接收一个集合和一个条件函数，并返回一个对象，其中键是由传入函数所返回的条件，值是与其相对应的元素。例如

```

var albums = [{title: "Sabbath Bloody Sabbath", genre: "Metal"},
               {title: "Scientist", genre: "Dub"},
               {title: "Undertow", genre: "Metal"}];

_.groupBy(albums, function(a) { return a.genre });

//=> {Metal:[{title:"Sabbath Bloody Sabbath", genre:"Metal"},
//           {title:"Undertow", genre:"Metal"}],
//     Dub: [{title:"Scientist", genre:"Dub"}]}

```

_.groupBy函数是非常方便的，并且会在本书中出现相当多次。

我将要讨论的最后一个applicative函数是_.countBy。此函数类似于_.groupBy，只不过它返回一个对象，包含符合匹配条件的键及其个数，如下：

```

_.countBy(albums, function(a) { return a.genre });

//=> {Metal: 2, Dub: 1}

```

至此，终于圆满完成了applicative函数式编程的讨论。我从JavaScript代码中可能最常遇到的情况入手，这样能在深入到旷野冒险前让你掌握一定的背景知识。接下来，我会介绍JavaScript代码中常见的话题：闭包。

2.2.3 定义几个Applicative函数

我已经展示了许多Underscore提供的applicative函数，但是如何自己创建一些呢？这个过程是相当简单的：定义一个函数，让它接收一个函数，然后调用它。

一个简单的、接收一定数量的参数并连接它们的函数并不是applicative：

```

function cat() {
  var head = _.first(arguments);
  if (existy(head))
    return head.concat.apply(head, _.rest(arguments));
  else
    return [];
}

```

```

cat([1,2,3], [4,5], [6,7,8]);
//=> [1, 2, 3, 4, 5, 6, 7, 8]

```

虽然相当有用，但cat不指望得到任何函数作为参数^[8]。同样，函数construct接受一个元素和一个数组，并用cat将元素放置在数组前方：

```

function construct(head, tail) {
  return cat([head], _.toArray(tail));
}

construct(42, [1,2,3]);
//=> [42, 1, 2, 3]

```

虽然construct在函数中用到cat，但它并没有将cat作为参数传入，所以它不符合applicative的要求。

然而，定义如下的一个mapcat函数是applicative：

```

function mapcat(fun, coll) {
  return cat.apply(null, _.map(coll, fun));
}

```

mapcat函数接收一个函数fun，与_.map用了相同的方式，对给定集合中的每个元素进行调用。这种fun的使用是mapcat的applicative本质。此外，mapcat串连_.map结果中的所有元素：


```
mapcat(function(e) {
  return construct(e, [""]);
}, [1,2,3]);
//=> [1, "", 2, "", 3, ""]
```

当映射函数返回一个数组，mapcat可将其展平。然后，我们可以使用mapcat和另一个函数butLast，来定义第三个函数 interpose:

```
function butLast(coll) {
  return _.toArray(coll).slice(0, -1);
}

function interpose (inter, coll) {
  return butLast(mapcat(function(e) {
    return construct(e, [inter]);
  },
  coll));
}
```

使用interpose很简单:

```
interpose("", [1,2,3]);

//=> [1, "", 2, "", 3]
```

这是函数式编程的一个关键方面：用较低级别的函数来逐步定义和使用离散功能。很多时候，你会看到（在这本书中，我会大声宣传）一串函数链一个接一个地调用，每一个函数将逐渐转变的结果传递到后一个函数，最后得到解决方案。

2.3 数据思考

在这本书中，我将使用最少的数据类型来表示抽象，如集合、树和表。但是在JavaScript中，尽管它的对象类型是非常强大的，但与其一起工作的工具并不完全是函数式的。相反，用JavaScript对象更常用的模式是，以多态调度为目的来附加方法。值得庆幸的是，你还可以把一个未命名的JavaScript对象（不是通过构造函数生成的）看作一个简单的关联性数据存储^[9]。

如果我们能够对Book对象或Employee类型的实例执行的唯一操作是setTitle或getSSN的话，那么我们已经将数据转换成了单件信息的微语言（Hickey, 2011）。关联数据技术是一种更灵活的建模数据方式。JavaScript对象，即使不考虑原型机理，依然是理想的关联数据建模载体，其中可以结构化具名值，以形成更高层次的数据模型，提供统一的访问方式^[10]。

虽然把JavaScript对象当成数据映射来操作和访问的工具本身很稀少，但是幸好Underscore提供了有用的一系列操作。其中最易于掌握的函数有_.keys, _.values和_.pluck。_.keys和_.values 都是根据它们的功能命名，都是接收一个对象并返回它的键或值的数组：

```
var zombie = {name: "Bub", film: "Day of the Dead"};

_.keys(zombie);
//=> ["name", "film"]

_.values(zombie);
//=> ["Bub", "Day of the Dead"]
```

_.pluck函数接收一个对象数组和一个字符串，并返回在给定的键数组中对应对象的值：

```
_.pluck([ {title: "Chthon", author: "Anthony"},  
          {title: "Grendel", author: "Gardner"},  
          {title: "After Dark"} ],  
        'author');
```

```
//=> ["Anthony", "Gardner", undefined]
```

这三个函数都是分解给定的对象并放入数组中，并且允许你执行顺序操作。另一种查看JavaScript对象的方式是把它看作数组的数组，每个数组包含一个键和一个值。Underscore提供了一个名为_.pairs的函数，它接收一个对象，并把它变成这个嵌套数组：

```
_.pairs(zombie);
```

```
//=> [ ["name", "Bub"], ["film", "Day of the Dead"] ]
```

可以对此嵌套数组视图进行顺序操作，还可以使用Underscore的_.object函数将其重新组合成一个新的对象：

```
_.object(_.map(_.pairs(zombie), function(pair) {  
    return [pair[0].toUpperCase(), pair[1]];  
}));
```

```
//=> { FILM: "Day of the Dead", NAME: "Bub" };
```

除了以某种微妙的方式来改变键，另一种常见的函数是通过_.invert函数翻转键和值：

```
_.invert(zombie);
```

```
//=> { "Bub": "name", "Day of the Dead": "film" }
```

值得一提的是，不像许多其他语言，JavaScript对象的键永远只能是字符串。因此偶尔会在使用_.invert时引起混乱：

```
_.keys(_.invert({a: 138, b: 9}));  
//=> ['9', '138']
```

Underscore还提供了增加，或根据参数的值从对象中移除值的函数：

```
_.pluck(_.map([{title: "Chthon", author: "Anthony"},  
               {title: "Grendel", author: "Gardner"},  
               {title: "After Dark"}]),  
         function(obj) {  
           return _.defaults(obj, {author: "Unknown"})  
         }),  
         'author');
```



```
//=> ["Anthony", "Gardner", "Unknown"]
```

在这个例子中，每一个对象都被`_.defaults`函数预处理，以确保该`author`字段包含一个有用的值（而不是`undefined`）。`_.defaults`用于扩充传入的对象，而`_.pick`和`_.omit`这两个函数会根据它们的参数（潜在的）筛选对象：

```
var person = {name: "Romy", token: "j3983ij", password: "tigress"};  
  
var info = _.omit(person, 'token', 'password');  
info;  
//=> {name: "Romy"}  
  
var creds = _.pick(person, 'token', 'password');  
creds;  
  
//=> {password: "tigress", token: "j3983ij"};
```

使用相同的“危险”键：`token`和`password`，`_.omit`函数接收一个黑名单，从对象中删除键，而`_.pick`根据白名单保留相应键（都不会破坏原对象）。

最后，Underscore提供了`_.findWhere`和`_.where`两个选择器函数，可以方便地根据关键的条件寻找特定对象。`_.findWhere`函数接收一个对象的数组，并返回第一个与参数给出的条件匹配的对象：

```
var library = [{title: "SICP", isbn: "0262010771", ed: 1},
               {title: "SICP", isbn: "0262510871", ed: 2},
               {title: "Joy of Clojure", isbn: "1935182641", ed: 1}];
```

```
_.findWhere(library, {title: "SICP", ed: 2});
```

```
//=> {title: "SICP", isbn: "0262510871", ed: 2}
```

.where函数使用起来与.findWhere非常相似，只是它会返回所有符合条件的对象：

```
_.where(library, {title: "SICP"});
```

```
//=> [{title: "SICP", isbn: "0262010771", ed: 1},
//    {title: "SICP", isbn: "0262510871", ed: 2}]
```

这种类型的使用模式引出了一个很重要的数据抽象：表。事实上，使用Underscore的函数来操作对象，你会觉得非常类似于SQL，都是根据一个强大的声明规约进行过滤和处理逻辑数据表。不过，我还会实现更加流畅的表处理的API，我需要加强并超越Underscore所提供的函数，并使用函数式的技术优势。在本节中所创建的函数只实现所有的SQL引擎都基于的关系代数（2003年）的一个子集。我不会深入关系代数，而只停留在SQL伪代码层面。假定读者已有类SQL语言的基本熟练水平。

“表状（Table-Like）”数据

表2-1 给出了一种查看library数组中数据的方式。

表2-1 JavaScript对象数组的数据表视图

title	Isbn	ed
SICP	0262010771	1
SICP	0262510871	2
Joy of Clojure	1935182641	1

在这里，每一行就相当于一个JavaScript对象，每一格都是对象中的一个键/值对。表2-2中的信息相当于SQL查询SELECT title FROM library的结果。

表2-2 标题表

Title
SICP
SICP
Joy of Clojure

用我目前发现的工具可以达到同样的效果：

```
_.pluck(library, 'title');
```

```
//=> ["SICP", "SICP", "Joy of Clojure"]
```

问题是，`_.pluck`函数的结果并不同于表的抽象。虽然技术上说字符串的数组也是一个对象数组，但`_.pluck`打破了这种抽象。相反，你需要一个允许类似于SQL的SELECT语句的函数，同时还保留表抽象。函数`project`将作为SELECT的替身^[11]。

```
function project(table, keys) {  
  return _.map(table, function(obj) {  
    return _.pick.apply(null, construct(obj, keys));  
  });  
};
```

`project`函数用`_.pick`函数作用在数组中的每个对象，并选出匹配白名单的对象，从而保持了抽象的表：

```

var editionResults = project(library, ['title', 'isbn']);

editionResults;
//=> [{isbn: "0262010771", title: "SICP"},
//     {isbn: "0262510871", title: "SICP"},
//     {isbn: "1935182641", title: "Joy of Clojure"}];

```

如上所示，project函数返回表状的数据结构，用project还可以进一步处理：

```

var isbnResults = project(editionResults, ['isbn']);

isbnResults;
//=> [{isbn: "0262010771"}, {isbn: "0262510871"}, {isbn: "1935182641"}]

```

最后，可以有意地选出所需的数据来打破抽象：

```

_.pluck(isbnResults, 'isbn');
//=> ["0262010771", "0262510871", "1935182641"]

```

这种提取是一种有意地从一个模块或函数“传递”数据到下一个行为。project函数作用于抽象表，而另一个虚拟的函数populateISBNSelectBox会作用于字符串数组，然后构造表单的DOM元素<option value="1935182641">1935182641</option>。函数式程序员会深入思考他们应用的数据，以及每一层交接的格式。在视觉上，你能想象高层次的以数据为中心的思维，如图2-1所示（Gruber，2004）。

在深入到数据的转换和传递前，让我们再来探讨一下这个表的抽象。例如，大多数SQL引擎提供AS用于给列声明一个别名。在SQL中，AS用法如下：

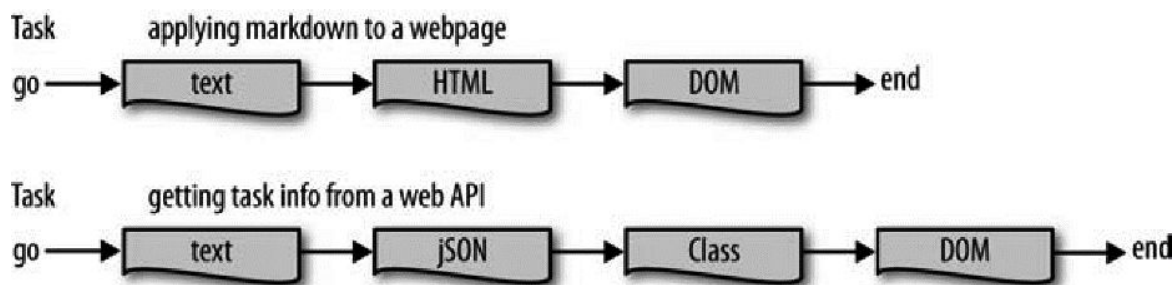


图2-1 使用数据转换抽象任务

```
SELECT ed AS edition FROM library;
```

前面的查询将输出表2-3 中显示的结果。

表2-3 别名edition的表

edition
1
2
1

然而，在我实现as之前，有必要创建一个工具函数rename，可根据给定的重命名条件的映射表来重命名：

```
function rename(obj, newNames) {  
  return _.reduce(newNames, function(o, nu, old) {  
    if (_.has(obj, old)) {  
      o[nu] = obj[old];  
      return o;  
    }  
    else  
      return o;  
  },  
  _.omit.apply(null, construct(obj, _.keys(newNames))));  
};
```

实现rename的一个重点是，它使用了_.reduce函数来重新构造一个对象，通过使用Underscore的另一种模式遍历保留累加器的“mappiness”的键/值对。我根据renaming的映射直接操作数组来进行重命名键。举个例子会更清楚地解释这是如何工作的：


```
rename({a: 1, b: 2}, {'a': 'AAA'});
```

```
//=> {AAA: 1, b: 2}
```

我可以使用函数rename实现as，来操作表单抽象：

```
function as(table, newNames) {  
  return _.map(table, function(obj) {  
    return rename(obj, newNames);  
  });  
};
```

正如你可能发现的，as通过简单的映射rename到表中的每个对象。注意：

```
as(library, {ed: 'edition'});
```

```
//=> [{title: "SICP", isbn: "0262010771", edition: 1},  
//    {title: "SICP", isbn: "0262510871", edition: 2},  
//    {title: "Joy of Clojure", isbn: "1935182641", edition: 1}]
```

因为无论as还是project都操作同一抽象，我可以将调用连在一起，这样就与执行SQL语句一样得到一个新表：

```
project(as(library, {ed: 'edition'}), ['edition']);
```

```
//=> [{edition: 1}, {edition: 2}, {edition: 1}];
```

最后，我可以实现类似于SQL的WHERE子句，这样就可以提供基本的操作表抽象的SQL功能，命名为restrict（2011年）：

```

function restrict(table, pred) {
  return _.reduce(table, function(newTable, obj) {
    if (truthy(pred(obj)))
      return newTable;
    else
      return _.without(newTable, obj);
  }, table);
};

```

restrict函数接收一个函数，作为对表中的每个对象的谓词。每当谓词返回false值时，该对象不会出现在最终表中。下面是restrict如何删除所有书的第一版：

```

restrict(library, function(book) {
  return book.ed > 1;
});

```

```
//=> [{title: "SICP", isbn: "0262510871", ed: 2}]
```

像其他表抽象的函数一样，restrict可以被链接：

```

restrict(
  project(
    as(library, {ed: 'edition'}),
    ['title', 'isbn', 'edition']),
  function(book) {
    return book.edition > 1;
  });

```

```
//=> [{title: "SICP", isbn: "0262510871", edition: 2},]
```

等效的SQL语句可以写成如下：

```
SELECT title, isbn, edition FROM (  
    SELECT ed AS edition FROM library  
) EDS  
WHERE edition > 1;
```

尽管它们没有SQL那么有魅力，但函数project、as和restrict共同操作同一张表的抽象——简单的对象数组。这就是数据思考。

2.4 总结

本章的重点是一等函数。一等函数被看成与其他数据一样的函数。

- 它们可以存储在变量中。
- 它们可以被存储在数组中的插槽中。
- 它们可以存储在对象的字段中。
- 它们可以根据需要来创建。
- 它们可以被传递到其他函数中。
- 它们可以被其他函数返回。

JavaScript支持一等函数是实践函数式编程的一大福音。一个大多数读者都熟悉的特殊函数式编程形式，被称为applicative编程。本章列举了几个applicative编程的例子：`_.map`、`_.reduce`和`_.filter`，后来也创建了一些新的applicative函数。

applicative编程实用性比较强的原因是，大多数的JavaScript应用程序都专注于处理数据集合，无论是数组、对象、对象数组，或是包含数组的对象。把重点放在基本集合类型让我们建立了一套类似SQL的工作对一个简单的“表”，从对象的数组构建抽象关系运算符。

下一章将是一个过渡章节，涵盖了变量的作用域和闭包的基本主题。

[1] 实际上，程序员甚至很难对最基本的术语达成一致。

[2] Haskell处理I/O时通常采用命令式编程，但很少有人会因此说Haskell不是函数式语言。

[3] 这仿佛是一个先有鸡还是先有蛋的问题。

[4] 译者注：作者好像犯了一个错误，这段代码其实返回的也是对象**b**，`this`其实指向调用时的上下文。

[5] 如果你对JavaScript元编程感兴趣，可以向O’Reilly出版社申请让我写一本。

[6] `allOf`和`anyOf`函数也可以用Underscore的`reduce`函数轻松实现，但我选择前者以便能更好地解释`reduceRight`。

[7] 值得一提的是，将null作为第一个参数传入apply。之前说过，apply的第一个参数是设置this引用的“目标”对象。由于我并不知道应该设置什么作为目标对象，甚至是不是需要设置目标对象，这里使用null表示this应指向全局对象。

[8] cat函数当然可以接受函数数组作为参数，但这样就有些离题了。

[9] ECMAScript对于提供不使用原型系统的简单map（以及set）类型有过一些讨论。具体可以参考http://wiki.ecmascript.org/doku.php?id=harmony:simple_maps_and_sets。

[10] JavaScript提供统一访问其关联数据类型的能力，使得你能编写强大的通用数据操作函数套件。如JavaScript的for...in循环以及访问索引操作符成为了Underscore实现的基础。

[11] 我们习惯的SQL中的SELECT语句其实是关系代数中的PROJECT语句。我将使用project，因为Underscore已经提供了select作为filter的别名。

第3章 变量的作用域和闭包

本章介绍变量的作用域，不论是对函数式编程，还是对一般的JavaScript编程，这都是一个重要的基础话题。“绑定（binding）”一词指的是通过var关键字、函数参数、this传递，或属性分配给JavaScript中的值分配名字的行为。本章首先会谈谈到动态作用域，如JavaScript的this引用，接着介绍函数级别的作用域以及它是如何工作的。所有这一切将会引出闭包的讨论，即函数在创建时，捕捉附近变量绑定的讨论。本章会覆盖闭包的机制，及其常用用例。

“作用域（scope）”这个术语在JavaScript程序员日常使用中有所不同含义。

- this绑定的值。
- this绑定的值定义的执行上下文。
- 一个变量的“生命周期”。
- 变量的值解析方案，或词法绑定。

出于本书的目的，我将使用作用域来引出变量值解析方案的一般想法。我将从最简单的全局作用域开始，深入挖掘各类解析方案从而涵盖JavaScript提供的所有作用域。

3.1 全局作用域

作用域的外延（extent）是指变量的生命周期（一个变量多长时间内保持一定的值）。我将从拥有最长生命周期的变量—全局变量开始，全局变量的生命周期将跨越整个程序。

在JavaScript中，下列变量将具有全局作用域：

```
aGlobalVariable = 'livin la vida global';
```

在JavaScript中任何没有用var关键字声明的变量都是全局变量，全局变量能被程序中任何函数和方法访问到。注意观察以下代码：

```
_.map(_.range(2), function() { return aGlobalVariable });  
//=> ["livin la vida global", "livin la vida global"]
```

变量aGlobalVariable在_.map函数的参数中的匿名函数（创建时未命名的函数）中被访问。全局作用域简单易懂，常用于JavaScript程序中（有时影响会很大）。事实上，Underscore创建了一个包含其所有函数的全局变量_。尽管这样没有命名空间，但JavaScript就是这么用的，并且Underscore至少给自己留了一个逃生舱口_.noConflict函数。

对于JavaScript变量，有趣的是，它们默认情况下是可变的（mutable）（例如，你可以随时改变它们的值）：

```
aGlobalVariable = 'i drink your milkshake';
```

```
aGlobalVariable;  
//=> "i drink your milkshake"
```

用全局变量的问题，以及广受非议的原因，就是任何一段代码都可以随意改变它们。这种反常的状态可能会引起之后的一些烦恼。不管怎样，你都应该清楚全局作用域的危险。然而，将一个变量声明为全局变量的方式不仅仅是将其定义在文件顶部，或是不用var声明。任何对象对变化都是开放的（除非它被冻结，我将在第7章 中讨论）：

```
function makeEmptyObject() {  
  return new Object();  
}
```

makeEmptyObject函数正如它的名字那样：创建一个空的对象。我可以随意往该函数返回的对象添加属性，这样的话，任何可以获得该对象的代码都可以对其进行操作。任何可变对象的属性都可以在全局范围内被修改。这样，如果我愿意，可以改变Underscore对象中的每个函数，让它们返回字符串'nerf herder'，而且没有人能阻止我这样做。这会给JavaScript的函数式编程带来了一定程度的困难。不过，我将在本书中展示如何使用一些方式减轻隐含的全局作用域 问题。

全局变量存在于程序整个生命周期，但并不意味着通过其引用就一定能取到值。当我们引入词法作用域（Lexical scope）时，作用域这个话题就更有意思了。

3.2 词法作用域

词法作用域是指一个变量的可见性，及其文本表述的模拟值。例如，看下面的代码：

```
aVariable = "Outer";

function afun() {
  var aVariable = "Middle";

  return _.map([1,2,3], function(e) {
    var aVariable = "In";

    return [aVariable, e].join(' ');
  });
}
```

调用afun会输出什么？

```
afun();
//=> ["In 1", "In 2", "In 3"]
```

最内层的变量值In，优先传递给_.map函数中的。词法作用域决定了这个结果，是因为赋值In到aVariable是文本上发生的最近操作，则认为In是使用时的值。图3-1 显示了这种情况的图形化表述。

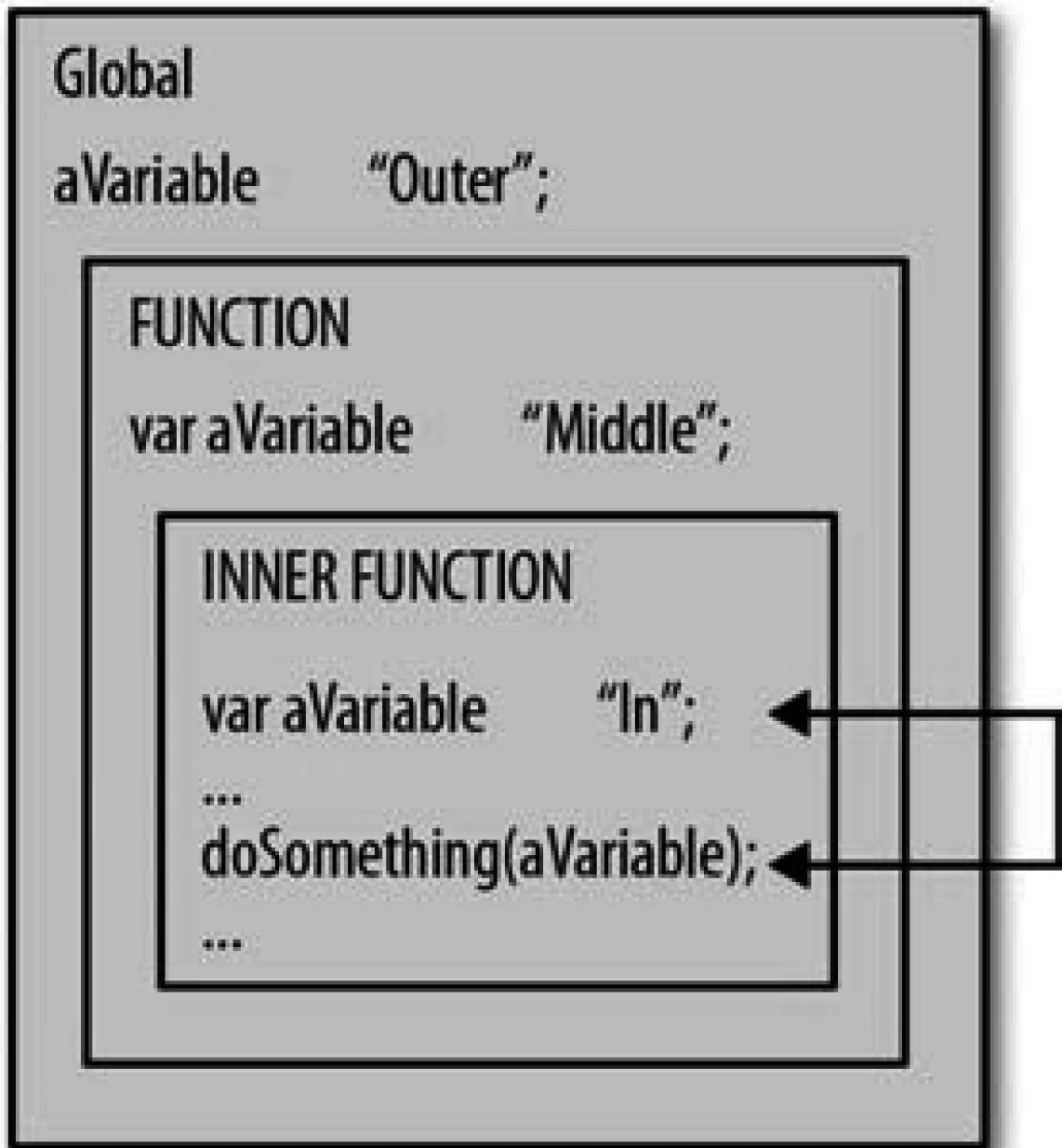


图3-1 变量查找从最内层范围向外扩展

简单情况下，变量的查找开始于最接近的绑定（binding）上下文而向外扩展，直到找到第一个绑定^[1]。图3-1 描述了词法作用域，就像根据周围的源代码将名称与值分组。本节将会涵盖JavaScript所支持的不同查找方案的机制，那么让我们先从动态作用域开始。

3.3 动态作用域

在编程中最容易被低估和过度滥用的概念就是动态作用域。原因是，很少有语言使用的动态作用域作为绑定解析方案。目前只有极少数的现代编程语言使用动态作用域这种简单方案来作为主要的作用域机制，除了Lisp语言的最早的版本之外，再没有更广泛的应用了^[2]。模拟一个原生的动态作用域机制只需要非常少的代码：

```
var globals = {};
```

首先，动态作用域的基础是值的一个全局表^[3]。在任何JavaScript引擎的核心中，你都会看到（就算不是这么实现的，也大体都是这个思想）一个查找表：

```
function makeBindFun(resolver) {  
  return function(k, v) {  
    var stack = globals[k] || [];  
    globals[k] = resolver(stack, v);  
    return globals;  
  };  
}
```

有了globals和makeBindFun，我们开始考虑如何增加绑定到globals的变量：

```
var stackBinder = makeBindFun(function(stack, v) {  
  stack.push(v);  
  return stack;  
});  
  
var stackUnbinder = makeBindFun(function(stack) {  
  stack.pop();
```

```

    return stack;
});

```

函数stackBinder执行一个非常简单的任务（例如，它需要一个键和值对，并将值推至对应建的全局绑定映射）。动态作用域的核心是维护一个命名绑定栈的全局映射，如图3-2所示。

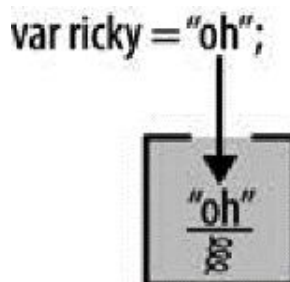


图3-2 可以想象，任何时候声明的变量都有一个对应的栈来存储其值，栈顶则为动态的值

stackUnbinder函数是stackBinder的逆，它弹出与名字相关联的堆栈顶部的值。最后，我们需要一个函数来查询绑定的值：

```

var dynamicLookup = function(k) {
    var slot = globals[k] || [];
    return _.last(slot);
};

```

dynamicLookup函数提供一种便捷的方式查询指定值的绑定栈。图3-3描述了this引用的解析。

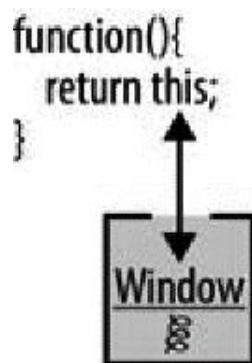


图3-3 函数引用“this”将指向全局对象（例如，在浏览器的Window）

既然现在我们有了绑定和查找函数，可以试一下模拟动态作用域：

```
stackBinder('a', 1);  
stackBinder('b', 100);  
  
dynamicLookup('a');  
//=> 1  
  
globals;  
//=> {'a': [1], 'b': [100]}
```

到目前为止，前面的代码可能都正如你所期望的。例如键控阵列globals的堆栈，a和b仅绑定一次，堆栈将只有单个值。虽然dynamicLookup不能轻易模仿this在对象中的方法解析，但可以把它看作入栈操作，如图3-4 所示。

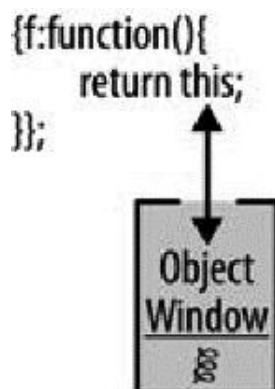


图3-4 一个对象的方法引用“this”将处理对象本身

在动态作用域的方案中，在栈顶的值是绑定的当前值。如果我们再绑定一次，会发生什么：

```
stackBinder('a', '*');
```

```
dynamicLookup('a');
```

```
//=> '*'
```

```
globals;
```

```
//=> {'a': [1, '*'], 'b': [100]}
```

绑定在a的新堆栈包含[1, ''], 所以该条件发生的任何查询将会返回。
需要检索以前的绑定很简单，只要通过解除绑定将其弹出堆栈：

```
stackUnbinder('a');
```

```
dynamicLookup('a');
```

```
//=> 1
```

你可能已经猜到（或已经知道）像这样一个方案（全局命名栈的操作）可能会导致一些麻烦：

```
function f() { return dynamicLookup('a'); };
```

```
function g() { stackBinder('a', 'g'); return f(); };
```

```
f();
```

```
//=> 1
```

```
g();
```

```
//=> 'g'
```

```
globals;
```

```
// {a: [1, "g"], b: [100]}
```

在这里，虽然f从来没有操作a的绑定，但它看起来的值取决于调用的g函数！这是动态作用域的缺点：任何给定的绑定的值，在确定调用其函数之前，都是不可知的。

上述代码中另一点值得注意的是，不得不显式地“取消”动态绑定，而在支持动态绑定的编程语言中，这个任务是在关闭动态绑定的上下文中自动完成的。

JavaScript的动态作用域

本节并不是理论的实践，而是对JavaScript中的动态作用域this引用的讨论。第2章 中提到了this引用会根据第一次创建时的上下文指向不同的值，但实际情况会更糟。this引用的值，就像a的绑定，也由调用者确定，如以下代码所示：

```
function globalThis() { return this; }

globalThis();
//=> some global object, probably window

globalThis.call('barnabas');
//=> 'barnabas'

globalThis.apply('orsulak', [])
//=> 'orsulak'
```

是的，this引用的值是通过apply或call 直接操作的，如图3-5 所示。也就是说，传入到它们的第一个参数就是被引用的对象。例如jQuery库使用这种方式来传递上下文对象和事件目标到一等函数，如果使用得当的话，这是一种很强大的技术。然而，使用动态作用域时很容易混淆this。

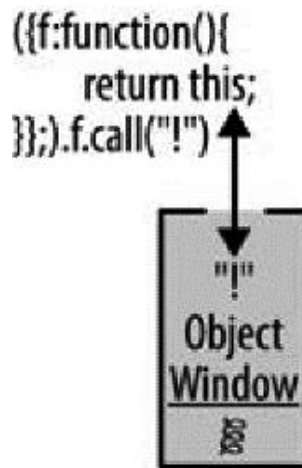


图3-5 使用Function#call调用允许设置“this”引用到一个已知的值

值得庆幸的是，如果this引用永远不传递给call或apply，或者如果它被绑定到null，是不会出现这个问题的。此外，Underscore提供了_.bind函数，可以锁定this使其不被更改：

```
var nopeThis = _.bind(globalThis, 'nope');
```

```
nopeThis.call('wat');  
//=> 'nope';
```

因为this引用是动态作用域，你会发现，尤其是在事件处理函数中，如单击按钮时得到的this通常是没用的，并可能会破坏你的应用程序。为了解决这样的问题，可以使用_.bindAll函数来锁定this引用到对应的命名函数，如下所示：


```
var target = {name: 'the right value',
              aux: function() { return this.name; },
              act: function() { return this.aux(); }};

target.act.call('wat');
// TypeError: Object [object String] has no method 'aux'

_.bindAll(target, 'aux', 'act');

target.act.call('wat');
//=> 'the right value'
```

这样，Underscore解除了动态作用域的危险。现在，我已经详细介绍动态作用域，接下来介绍函数作用域。

3.4 函数作用域

为了说明动态作用域与函数作用域的区别，我需要修改绑定和查询的逻辑。不是在全局哈希映射表中访问绑定，而是构造新的模型，使其将所有绑定局限在最小的作业范围内（函数）。这是根据JavaScript作用域模型^[4]模拟一个函数作用域方案，需要一点想象力。每个JavaScript函数可以引用this。在上一节中，我谈到this动态性质的危险性。而是为了说明，我将用它来证明另一个观点。首先，观察JavaScript的默认行为：

```
function strangeIdentity(n) {  
    // intentionally strange  
    for(var i=0; i<n; i++);  
    return i;  
}
```

```
strangeIdentity(138);  
//=> 138
```

在Java语言中，如果试图访问for循环中的局部变量i，将引发一个访问错误。然而，在JavaScript中，所有在函数体内的var声明都会隐式地移到函数的顶部。JavaScript重新排列变量声明的动作称为吊装（hoisting）。换句话说，前面定义的函数相当于^[5]

```
function strangeIdentity(n) {  
    var i;  
    for(i=0; i<n; i++);  
    return i;  
}
```

这样做的意义是，函数内定义的变量对函数内任何一段代码都是可见的。不用说，这可能会不时地带来问题，特别是如果不注意变量是如何通过闭包（在下一节讨论）获得的。

在此期间，我可以展示如何使用this引用轻松模拟函数作用域：

```
function strangerIdentity(n) {  
    // intentionally stranger still  
    for(this['i'] = 0; this['i']<n; this['i']++);  
    return this['i'];  
}
```

```
strangerIdentity(108);  
//=> 108
```

当然，这并不是一个真正的模拟，因为在这种情况下实际上修改了全局对象：

```
i;  
//=> 108
```

如果能提供对要操作的函数的暂存空间就更好了。感谢神奇的this引用，可以用call函数来实现this的绑定：

```
strangerIdentity.call({}, 10000);  
//=> 10000
```

```
i;  
//=> 108
```

虽然原来全局i仍然存在，至少已经停止修改全局环境，我还是没有给出一个真正的模拟器，因为现在我只能访问函数的局部变量。但是，没有任何理由使用一个空的对象传递上下文。事实上，对于这个伪造出来的JavaScript，使用全局（但不直接）传递上下文似乎更合适。clone正好可以解决：

```
function f () {  
    this['a'] = 200;  
    return this['a'] + this['b'];  
}
```

```
var globals = {'b': 2};
```

```
f.call(_.clone(globals));  
//=> 202
```

检查全局上下文是否被污染:

```
globals;  
//=> {'b': 2}
```

用这个模型来运作的函数作用域似乎很合理。确实，JavaScript就是这么做的，除了变量访问是在函数体内隐式地进行，而不需要显式地在this中查找。不管你现在对函数作用域是怎么想的，至少JavaScript的底层机制已经帮我们打理好了一切。

3.5 闭包

闭包是JavaScript的一大谜团。最近的一项调查显示，有关JavaScript的闭包的博客文章占23%左右^[6]。闭包，无论出于何种原因，对于相当数量的程序员仍是一个谜。在本节中，我需要一些时间来仔细讲解JavaScript的闭包。幸运的是，它其实是很简单的。事实上，在本节中，我将建立一个小型的库来模拟作用域规则和闭包。我将使用这个库来探索这一章的细节，包括全局作用域、函数作用域、自由变量以及闭包。

开始之前，值得一提的是，闭包与一等函数是齐头并进的。虽然没有一等函数的语言也能支持闭包，但往往有很大阻碍。值得庆幸的是，JavaScript是支持一等函数的，所以它的闭包可以以强大的方式来绕过临时封装状态。

备注

在本章和下一章中，我将所有由闭包捕获的变量大写。这并不是JavaScript的标准实践，也不鼓励这样做，这里只是为了方便教学。这两章过后，我将不再使用这个约定。

闭包是一个函数，该函数在生成时会“捕获”附近的值。图3-6 是闭包的图形表示。

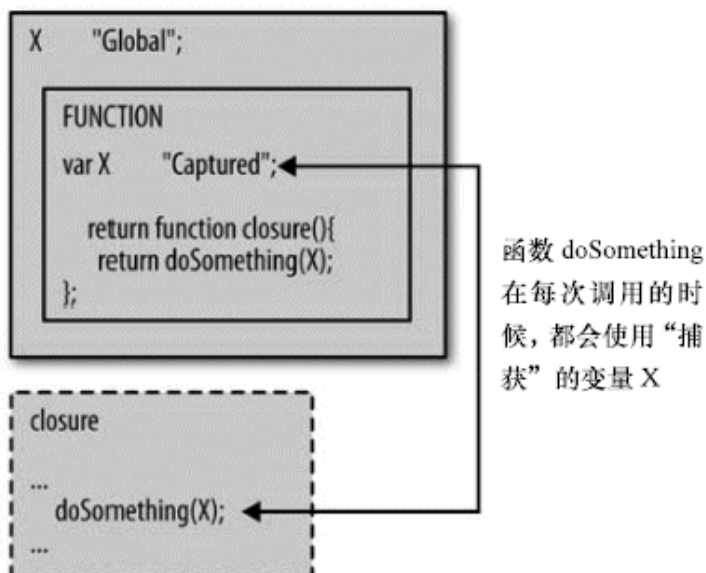


图3-6 闭包是生成时会“捕获”附近的值的函数

在接下来的几节中，我会从一个闭包模拟器开始，深入闭包。

3.5.1 模拟闭包

经历了30年，闭包终于成为主流的编程语言的主要特点。什么是闭包？一句话概括：闭包就是一个函数，捕获作用域内的外部绑定（例如，不是自己的参数）。这些绑定是为之后使用（即使在该作用域已结束）而被定义的。

在我们进一步模拟闭包之前，先来看看它们的默认行为。闭包的最简单的例子是一等函数，捕获局部变量供以后使用：

```
function whatWasTheLocal() {  
    var CAPTURED = "Oh hai";  
  
    return function() {  
        return "The local was: " + CAPTURED;  
    };  
}
```

使用whatWasTheLocal函数：

```
var reportLocal = whatWasTheLocal();
```

我已经谈到了函数的局部变量的生命周期只限于在函数体内，但是当一个闭包捕获这个变量，它在一定程度上还能继续存在^[7]。

```
reportLocal();  
//=> "The local was: Oh hai"
```

如此看来，局部变量CAPTURED好像能够通过whatWasTheLocal返回的闭包被得到，实际上就是这么回事。但局部变量并不是唯一可以捕获的东西。函数参数也是可以被捕获的：

```

function createScaleFunction(FACTOR) {
  return function(v) {
    return _.map(v, function(n) {
      return (n * FACTOR);
    });
  };
}

```

```

var scale10 = createScaleFunction(10);

```

```

scale10([1,2,3]);
//=> [10, 20, 30]

```

函数createScaleFunction需要一个比例因子，并返回一个函数，给定数字集合，返回其元素与缩放因子乘积的列表。你可能已经注意到了，一旦createScaleFunction函数退出，返回的函数引用的变量FACTOR似乎超出范围。这种观察只是部分正确，因为事实上，该变量FACTOR被保持在返回的缩放函数的主体内，并能在该函数被调用时访问。这个变量保存恰恰就是闭包的定义。

那么，如何用我们上一节的函数作用范围的this暂存器来模拟闭包呢？首先，我需要设计一个捕获已封闭变量的方法，并使其保持跟正常变量一样不被关闭。最直接的方式是拿到外函数的变量并绑定到返回函数的this，如下所示：

```

function createWeirdScaleFunction(FACTOR) {
  return function(v) {
    this['FACTOR'] = FACTOR;
    var captures = this;

    return _.map(v, _.bind(function(n) {

```

```

        return (n * this['FACTOR']);
    }, captures));
};
}

var scale10 = createWeirdScaleFunction(10);

scale10.call({}, [5,6,7]);
//=> [50, 60, 70];

```

决定需要跟踪内部函数中的哪些变量似乎非常棘手。如果像这个例子一样需要手动跟踪，那么JavaScript的编写将极其困难^[8]。值得庆幸的是，JavaScript的变量捕获是自动的，且可以直接使用。

1. 自由变量

自由变量与闭包的关系是，自由变量闭合于闭包的创建。闭包背后的基本原理是，如果一个函数包含内部函数，那么它们都可以看到其中声明的变量；这些变量被称为“自由”变量^[9]。然而，这些变量可以被内部函数捕获，从高阶函数中return实现“越狱”，以供以后使用^[10]。唯一需要注意的是，捕获函数必须在外部函数内定义。函数内在没有任何局部声明之前（既不是被传入，也不是局部声明）使用的变量就是被捕获的变量。注意观察：

```

function makeAdder(CAPTURED) {
    return function(free) {
        return free + CAPTURED;
    };
}

var add10 = makeAdder(10);

add10(32);
//=> 42

```


外部函数中的变量CAPTURED被执行加法的返回函数捕获，因为内部函数从未声明过CAPTURED，而只是引用了它。之后，从makeAdder中创建并返回的函数保留了变量CAPTURED，并用它进行加法运算。创建另一个加法器将捕捉到同名变量CAPTURED，但有不同的值，因为它是在调用makeAdder之后被创建：

```
var add1024 = makeAdder(1024);
add1024(11);
//=> 1035
```

```
add10(98);
//=> 108
```

最后，如上述代码所示，每一个新的加法器函数都保留了自己创建时捕获的CAPTURED实例。捕捉到的值可以是任何类型，包括函数。下面这个函数averageDamp，捕获一个函数，并返回一个计算该函数结果与另一个值的平均值函数^[11]。

```
function averageDamp(FUN) {
  return function(n) {
    return average([n, FUN(n)]);
  }
}
```

```
var averageSq = averageDamp(function(n) { return n * n });
averageSq(10);
//=> 55
```

捕获其他函数的高阶函数是构建抽象的强大技术。我将在本书中继续使用这种技术。

如果你用与更高层的作用域的变量同名的变量创建函数，会发生什么？我将简要地谈谈这个话题。

2. 遮蔽 (Shadowing)

在JavaScript中，当变量x在一定作用域内声明，然后另一个同名变量在一个较低的作用域声明，会发生变量的遮蔽。下面来看遮蔽的一个简单

的例子：

```
var name = "Fogus";  
var name = "Renamed";  
  
name;  
//=> "Renamed"
```

两个连续的同名声明，分配第二个值的变量应该是毫不奇怪。然而，函数参数的遮蔽就相对复杂：

```
var shadowed = 0;  
  
function argShadow(shadowed) {  
    return ["Value is", shadowed].join(' ');  
}
```

调用argShadow(108)函数的返回值是什么？注意：

```
argShadow(108)  
//=> "Value is 108"  
  
argShadow();  
//=> "Value is "
```

函数argShadow的参数shadowed覆盖了全局作用域内的同名变量。即使没有传递任何参数，仍然还是绑定shadowed。在任何情况下，离得“最近”的变量绑定优先最高。例如：

```
var shadowed = 0;  
  
function varShadow(shadowed) {  
    var shadowed = 4320000;  
    return ["Value is", shadowed].join(' ');  
}
```

如果你猜`varShadow(108)`的返回值是“ Value is 4320000 ”，那就对了。遮蔽变量同样发生在闭包内，如下所示：

```
function captureShadow(SHADOWED) {  
    return function(SHADOWED) {  
        return SHADOWED + 1;  
    };  
}  
  
var closureShadow = captureShadow(108);  
  
closureShadow(2);  
//=> 3 (it would stink if I were expecting 109 here)
```

我倾向于编写JavaScript代码时避免遮掩变量，所以需要注意变量命名。如果命名不小心，那么可能会造成混淆。在结束本章前，我将快速介绍一些闭包用法的示例。

3.5.2 使用闭包

在本节中，我将简要介绍闭包的用法。由于本书的后续部分将广泛使用闭包，没有必要过度讨论，所以这里只展示几个有用的例子。

如果你回想一下Applicative编程的例子，`complement` 函数接受一个谓词，并返回一个反转谓词结果的函数。当时没有点明，其实`complement`已经将闭包用到了极致：

```
function complement(PRED) {  
    return function() {  
        return !PRED.apply(null, _.toArray(arguments));  
    };  
}
```

谓词函数PRED被返回的函数捕获。例如一个判断偶数的谓词函数：

```
function isEven(n) { return (n%2) === 0 }
```

我们可以用complement来定义判断奇数的函数isOdd:

```
var isOdd = complement(isEven);
```

```
isOdd(2);  
//=> false
```

```
isOdd(413);  
//=> true
```

但如果isEven之后发生了变化呢?

```
function isEven(n) { return false }
```

```
isEven(10);  
//=> false
```

是否会改变isOdd的行为呢? 注意:

```
isOdd(13);  
//=> true;
```

```
isOdd(12);  
//=> false
```

正如你所看到的, 变量的捕获发生在创建闭包的时候 (这个例子中的 PRED)。因为我通过一个新的变量创建了新的isEven引用, 所以这种变化是会被isOdd察觉的。让我们运行下列代码:

```
function showObject(OBJ) {  
    return function() {  
        return OBJ;  
    };  
}
```

```
var o = {a: 42};  
var showO = showObject(o);
```

```
showO();  
//=> {a: 42};
```

好像都对，不是吗？不完全是：

```
o.newField = 108;  
showO();  
//=> {a: 42, newField: 108};
```

由于o的引用同时存在于闭包内部和外部，它的变化可以跨越看似私有的界限。这很容易导致混乱，所以通常的使用情况是最大限度地减少暴露捕获变量的风险。JavaScript经常使用下面这种模式，把捕获的变量作为私有数据：

```
var pingpong = (function() {  
    var PRIVATE = 0;  
  
    return {  
        inc: function(n) {  
            return PRIVATE += n;  
        },  
        dec: function(n) {  
            return PRIVATE -= n;  
        }  
    };  
})();
```

对象pingpong是由作为块作用域的匿名函数构建，并包含两个闭包inc和dec。最有趣的部分是，捕获的变量PRIVATE是这两个闭包的私有变量，除了通过调用这两个函数之一，无法通过任何手段进行访问：

```
pingpong.inc(10);  
//=> 10
```

```
pingpong.dec(7);  
//=> 3
```

甚至添加其他函数也是安全的：

```
pingpong.div = function(n) { return PRIVATE / n };

pingpong.div(3);
// ReferenceError: PRIVATE is not defined
```

通过这种闭包模式提供访问保护是一个强大的技术，能使JavaScript程序员在应对软件复杂性时保持头脑清楚。

3.5.3 闭包的抽象

闭包为JavaScript提供了私有访问，这是一种提供抽象的好方法（例如闭包允许你在创建函数时做一些“配置”）。makeAdder和complement的实现就是关于这个技术很好的例子。另一个例子是一个名为plucker的函数，接收一个键并将其传给一个关联结构，如数组或对象，并返回给定的结构，返回键值的函数。具体实现如下：

```
function plucker(FIELD) {
  return function(obj) {
    return (obj && obj[FIELD]);
  };
}
```

通过测试这个实现可以看出其行为：

```
var best = {title: "Infinite Jest", author: "DFW"};

var getTitle = plucker('title');

getTitle(best);
//=> "Infinite Jest"
```

正如我所提到的，plucker也可以操作数组：

```
var books = [{title: "Chthon"}, {stars: 5}, {title: "Botchan"}];

var third = plucker(2);

third(books);
//=> {title: "Botchan"}
```

plucker与`_.filter`一起使用很方便。下例是从数组抓取对象的给定字段：

```
_.filter(books, getTitle);
//=> [{title: "Chthon"}, {title: "Botchan"}]
```

本书的后续过程中，我将会探讨闭包的其他用途与优点。现在我觉得已经奠定了足够的基础。

3.6 总结

本章主要集中在一般函数式编程两个基本的主题：变量作用域和闭包。

从变量的全局作用域开始，逐步引入词法作用域和函数作用域的工作方式。另外，还介绍了动态作用域，特别是使用的`call`和`apply`方法，体现了`this`引用的使用与行为。这样可能会造成混乱，但可以通过Underscore的`_.bind`和`_.bindAll`函数固定`this`绑定到一个固定值。

本章还模拟了JavaScript的闭包，包括动态`this`引用。之后，除了展示如何在自己的函数中使用闭包，还展示了如何使用闭包“调整”现有的函数，从而实现新的功能的抽象。

在下一章中，我将再扩展一等函数并深入高阶函数，可以通过这些点来定义高阶函数。

- 可以传递给其他函数的函数。
- 可以从函数返回的函数。

如果你还不清楚这章的内容，那么最好在进入下一章之前回去重读一下。许多高阶函数的强大之处都与变量作用域，尤其是闭包密切相关。

[1] JavaScript也提供其他复杂查找的作用域模式，包括`this`解析、函数作用域，以及块。除了最后一个，我将在本书覆盖其他。

[2] 想知道早期的Lisp如何做的，可以阅读John McCarthy写的“Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I”以及McCarthy, Abrahams, Edwards, Hart, and Levin的“LISP 1.5 Programmer’s Manual”。

[3] 这是实现动态作用域的一种方式，而且是比较简单的方式。

[4] ECMAScript.next定义了很多定义“词法”变量作用域的方式。词法作用域与函数作用域非常相似，但绑定更“紧”一些。也就是说，它在JavaScript块中绑定时不会把声明提升到函数顶部。我不会深入词法作用域的细节，但此话题值得读者自己研究。

[5] ECMAScript.next 正在研究提供比函数作用域更细粒度的块作用域规范。目前还不清楚这个特性何时能出现自JavaScript内核中。一旦出现，将（但愿能）有助于本书下一版本的解释。

[6] 如果算上Hacker News（译者注：Y Combinator创建的关于计算机黑客和创业公司的社交新闻网站）中的讨论，这个数值会接近36%。我并没有这个数值的证据，只是为了好玩编造出来的。

[7] 闭包犹如编程语言中的吸血鬼，他们捕获部下并给其永久的生命，知道自己被摧毁。唯一的区别在于，闭包不会在阳光下化为灰烬。

[8] 我可以直接引用captures而不是通过Underscore的bind方法动态地传递到内部函数map，但这样做的话，我就相当于作弊，是用闭包来模拟闭包了。

[9] 这里的free不是免费啤酒中免费的意思，也不是自由的意思，而是相当于小偷偷来的“免费”（译者注：此处应当就是自由的意思，因为free和CAPTURED分别对应于数学中的自由变量和约束变量。读者可以注意例子中的add 10其实是把CAPTURED固定了，而add 10的参数可以是任意的。）

[10] 返回另一个函数的函数叫做高阶函数，在第4章中我会深入介绍。

[11] 我在第2章中定义的average。

第4章 高阶函数

本章在第3章的基础上，对函数是一等元素的说法进行扩展。也就是说，本章将解释函数不仅可以保存在数据结构中，并以数据的形式来传递，也可以从函数中返回。对这些“高阶”函数的讨论将是本章的主要内容。

高阶函数遵循一个非常明确的定义。

- 它是一等公民（如果需要复习这个话题，请回到第2章）。
- 以一个函数作为参数。
- 以一个函数作为返回结果。

前面我们已经演示了很多以其他函数作为参数的函数，但这是一个值得深入探索的领域，特别是因为函数式编程风格是它的明显优势。

4.1 以其他函数为参数的函数

我们已经看到很多以其他函数为参数的函数，其中最突出的当属`_.map`，`_.reduce`和`_.filter`。所有这些函数遵循高阶函数的定义。然而，简单地展示几个用法不足以说明以函数为参数的函数在函数式编程中的重要性。所以，我会花一些时间更多地讨论以函数为参数的函数，并与闭包的讨论一起进行练习。再次，每当所展示的代码用到闭包时，我会将被捕获的变量名大写。值得重申的是，将被捕获的变量名大写不是推荐的做法，只是在本书的写作中加以区分。

4.1.1 关于传递函数的思考：`max`、`finder`和`best`

在讨论以函数为参数的函数之初，我们有必要来看几个例子。很多编程语言甚至是一些核心库，都包括一个称为`max`的函数，用来找到一个列表或数组中的最大值（通常是一个数字）。事实上，`Underscore`本身也有这样的函数，执行如下代码：

```
_.max([1, 2, 3, 4, 5]);  
//=> 5
```

```
_.max([1, 2, 3, 4.75, 4.5])  
//=> 4.75
```

执行结果并没有什么奇怪，但是这个特定的用例中存在一个限制。就是说，如果我们想从一个不是数字数组的对象中找到最大值，该怎么办？值得庆幸的是，`_.max`是一个高阶函数，它接受一个可选的第二个参数。你可能已经猜到了，这第二个参数是用来从被比较对象中获得一个数值的函数^[1]。例如

```
var people = [{name: "Fred", age: 65}, {name: "Lucy", age: 36}];  
  
_.max(people, function(p) { return p.age });  
  
//=> {name: "Fred", age: 65}
```

这是一个非常有用的函数构建方法，因为相对于比较数值，`_.max`提供了比较任意对象的方法。但是，在某些方面，这个函数仍然是受限的，并

不是真正的函数式。具体说来，对`_.max`而言，比较总是需要通过大于运算符（`>`）来完成。

不过，我们可以创建一个新的函数`finder`。它接收两个函数：一个用来生成可比较的值，而另一个用来比较两个值并返回当中的“最佳”值。`finder`的实现如下所示：

```
function finder(valueFun, bestFun, coll) {  
  return _.reduce(coll, function(best, current) {  
    var bestValue = valueFun(best);  
    var currentValue = valueFun(current);  
  
    return (bestValue === bestFun(bestValue, currentValue)) ? best : current;  
  });  
}
```

现在，使用`finder`函数，可以模拟Underscore的`_.max`操作：

```
finder(_.identity, Math.max, [1,2,3,4,5]);
```

```
//=> 5
```

你应该注意到了例子中使用了便利的`_.identity`函数，它只是单纯地接收一个值，并返回该值。似乎有点没用，对不对？也许吧。但是在函数式编程领域，我们需要从函数的角度来思考问题，即便在最佳值就是该值本身的情况下。

在任何情况下，我们现在都可以用`finder`来找到不同类型的“最佳”值：

```
finder(plucker('age'), Math.max, people);
```

```
//=> {name: "Fred", age: 65}
```

```
finder(plucker('name'),  
  function(x,y) { return (x.charAt(0) === "L") ? x : y },  
  people);
```

```
//=> {name: "Lucy", age: 36}
```

看起来这个功能更喜欢以字母L开头的名字。

缩减一点

函数finder的实现短小精干，并且能按照我们的预期来工作。但为了满足最大程度的灵活性，它重复了一些逻辑。请注意在finder实现中与比较最佳值的高阶函数中的比较逻辑：

```
// in finder
return (bestValue === bestFun(bestValue, currentValue)) ? best : current);

// in the best-value function
return (x.charAt(0) === "L") ? x : y;
```

你会发现，这两个逻辑是完全相同的。也就是说，这两种算法都返回最佳值或当前值。finder的实现可以根据以下两个假设来缩减。

- 如果第一个参数比第二个参数“更好”，比较最佳值的函数返回true。
- 比较最佳值的函数知道如何“分解”它的参数。

在这些假设基础之上，我们可以用以下方式实现一个更简洁的best函数（Graham, 1993）：

```
function best(fun, coll) {
  return _.reduce(coll, function(x, y) {
    return fun(x, y) ? x : y
  });
}

best(function(x,y) { return x > y }, [1,2,3,4,5]);
//=> 5
```

删除重复逻辑之后，我们有了一个更严格、更优雅的解决方案。事实上，前面的例子再次表明，best(function(x,y) { return x > y }, ...)提供了与Underscore的_.max甚至是Math.max.apply(null, [1,2,3,4,5])相同的模式。第5章 将讨论函数式程序员如何通过抓取这类模式来创建一套合适有用的函数，所以现在我会推迟这个话题的讨论，而是将重点放在高阶函数上。

4.1.2 关于传递函数的更多思考：重复、反复和条件迭代（iterateUntil）

在上一节中，我创建了一个接收两个函数的函数finder。正如前面所看到的，以两个函数为参数（一个用来解析数据，另一个用来进行比较），对简化函数best来说是大材小用。事实上，你会发现，在JavaScript中，创建以多个函数为参数的函数常常会显得大材小用。然而，正如我下面将要讨论的一样，在某些情况下，这样做是完全合理的。

去除finder多余的参数是因为，根据对best函数所给出的假设，这里对两个函数参数的需求被取消了。然而，在某些情况下给一个算法设立假设是不恰当的。

我将逐个介绍三个相关的函数，并在这个过程中讨论如何将它们变得更加通用（以及最佳折中方案）。

首先，让我们从一个很简单的函数repeat开始。它以一个数字和一个值为参数，将该值进行多次复制，并放入一个数组当中：

```
function repeat(times, VALUE) {  
  return _.map(_.range(times), function() { return VALUE; });  
}
```

```
repeat(4, "Major");  
//=> ["Major", "Major", "Major", "Major"]
```

repeat的实现使用_.map函数来遍历从0到times-1的数组，并将VALUE丢到数组中。你会发现，里面的匿名函数使用了VALUE变量，但是目前这不是很重要（也不是很有趣）。目前有很多可以替代_.map来实现上面的功能的函数，但这里我主要用它来强调一个很重要的观点，可概括为“使用函数，而不是值”。

1. 使用函数，而不是值

将repeat的实现隔离出来不是一件坏事。然而，作为“repeatness”的常规实现方法，它的实现仍然有提高的空间。也就是说，当一个函数将一个值重复多次是可以的，但将运算重复多次则更好。我略微地修改了一下repeat，按如下方式来执行：

```
function repeatedly(times, fun) {  
  return _.map(_.range(times), fun);  
}  
  
repeatedly(3, function() {  
  return Math.floor((Math.random()*10)+1);  
});  
//=> [1, 3, 8]
```

函数repeatedly是展示函数式思维方式力量的一个很好的例证。通过将参数从值替换为函数，我给“repeatness”打开了一个充满可能性的世界。与repeat类似，在调用端，我们可以用一个可以填充任何东西的数组来替换一个固定的值。如果我们真的想在repeatedly中用常量，那么我们只需要执行以下操作：

```
repeatedly(3, function() { return "Odelay!"; });  
  
//=> ["Odelay!", "Odelay!", "Odelay!"]
```

事实上，该模式是一个无论其参数是什么都返回常量的函数的例子，此类函数将多次出现在本书以及其他外界的函数库当中。我将在下一节以及第5章 中进行更多的讨论。

你会发现，我未能一一列出能够提供给repeatedly函数的所有参数。但这是权宜之计，因为我选择了不使用传入的参数。事实上，由于repeatedly的实现是对_.range结果进行_.map应用到函数上，因此函数可以接收到当前重复的次数。我发现这种技术在用来生成一些已知数量的DOM节点时非常有用，其中每个节点都用带计数值的id，如下所示：


```

repeatedly(3, function(n) {
    var id = 'id' + n;
    $('body').append($("<p>Odelay!</p>").attr('id', id));
    return id;
});

// Page now has three Odelays
//=> ["id0", "id1", "id2"]

```

如上所示，我使用了jQuery库来添加一些节点。这是完全合法的使用方式，但它对函数之外的“世界”进行了修改。在第7章 我会讨论为什么这是一个潜在问题，但现在我想让repeatedly更加通用一些。

2. 再次强调，“使用函数，而不是值”

我已经将一个在repeat中使用静态值的函数变成了接收一个函数的repeatedly 函数。虽然这确实使repeatedly更加开放，但它的通用性仍然没有达到期望。我们仍然需要一个确定需要调用给定的函数多少次的数值。我们常常会准确地知道函数应该被调用多少次，但有时候也知道什么时候退出并不取决于“次数”，而是条件。换句话说，你可能需要不断调用一个函数，直到它的返回值超过了某个阈值，或改变了符号，或变为大写等，这样一来，一个简单的值是远远不够的。相反，我可以定义另一个名为iterateUntil的函数，它称得上repeat和repeatedly合乎逻辑的进化；实现如下所示：

```

function iterateUntil(fun, check, init) {
    var ret = [];
    var result = fun(init);

    while (check(result)) {
        ret.push(result);
        result = fun(result);
    }

    return ret;
};

```

函数`iterateUntil`接收两个函数：一个用来执行一些动作，另外一个用来进行结果检查，当结果满足“结束”值时返回`false`。这算得上是真正将`repeatedly`带到了一个新的水平，现在甚至连重复次数都是开放的，受到一个函数执行结果的影响。那么，该如何使用`iterateUntil`呢？一种简单的用法是收集一些重复计算的结果，直到值超过某个阈值。例如，假设你想找到所有大于2小于1024的2的倍数：

```
iterateUntil(function(n) { return n+n },  
              function(n) { return n <= 1024 },  
              1);
```

```
//=> [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
```

正如你所知，在使用`repeatedly`来实现相同的功能之前，为了得到正确的结果数组，需要提前确定调用我们的函数的次数：

```
repeatedly(10, function(exp) { return Math.pow(2,exp+1) });
```

```
//=> [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
```

有时候，你可能知道需要计算多少次，但有时你只知道计算何时停止。`iterateUntil`提供的另外一个优点是，循环体是一个前馈函数。换句话说，一些函数的执行结果被当作下一个函数的参数。我将会在“管道（`Pipelining`）”一节展示这种条件结束的强大，不过我们先继续下一节的内容，来谈谈返回其他函数的函数。

4.2 返回其他函数的函数

你已经看到了几个以函数为返回结果的函数—— `makeAdder`、`complement`和`plucker`。你可能已经猜到了，所有这些函数都是高阶函数。在本节中，我将更深入地讨论并了解返回（或接收）函数和闭包的高阶函数。我们以回忆`repeatedly`作为开始：它使用了一个忽略其参数并仅仅返回一个常量的函数：

```
repeatedly(3, function() { return "Odelay!"; });  
  
//=> ["Odelay!", "Odelay!", "Odelay!"]
```

这种返回一个常量的函数非常有用，所以几乎是函数式编程的一个设计模式，经常被简称为`k`。然而，为了清晰起见，我称它为`always`；其实现方式如下所示：

```
function always(VALUE) {  
  return function() {  
    return VALUE;  
  };  
};
```

`always`的行为可以用来解释闭包的一些关键点。首先，闭包会捕获一个值（或引用），并多次返回相同的值：

```
var f = always(function(){});  
  
f() === f();  
//=> true
```

因为该函数总是产生一个确定的值，你可以看到，从每次调用`always`而得到的与`VALUE`绑定的函数始终都是相同的（Braithwaite, 2013）。

任何一个用`function`创建的函数都会返回一个特定的实例，无论它的函数体是什么。使用`(function() {})`是一种确保生成特定值的简便方法。请记住，闭包的第二个重点是，每一个新的闭包都会捕获不一样的值：

```
var g = always(function(){});
```

```
f() === g();  
//=> false
```

使用闭包时牢记这两个规则，可以帮助我们避免混淆。

接下来，用always来替换之前的匿名函数，会更加简洁一些：

```
repeatedly(3, always("Odelay!"));
```

```
//=> ["Odelay!", "Odelay!", "Odelay!"]
```

像always这样的函数被称为组合子（combinator）。这本书不会过分专注于组合子，从某种程度上讲，它是值得涵盖的话题，因为你会在函数式风格的代码库中见到它们。不过，我会将这个问题的讨论推迟到第5章；现在，我将会展示更多的以函数为返回值的函数，特别专注于闭包是如何推动这种方法的。

然而，在继续下面的内容之前，我会展示另一种以函数为返回值的函数的实现，invoker：接收一个方法，并在任何给定的对象上调用它。如下所示：

```
function invoker (NAME, METHOD) {  
  return function(target /* args ... */) {  
    if (!existy(target)) fail("Must provide a target");  
  
    var targetMethod = target[NAME];  
    var args = _.rest(arguments);  
  
    return doWhen((existy(targetMethod) && METHOD === targetMethod), function() {  
      return targetMethod.apply(target, args);  
    });  
  };  
};
```

invoker的形式与always非常像；也就是说，它在后续调用中使用了一个原始参数：METHOD。在这种情况下，返回的函数是一个闭包。然而，与返回常量不同，invoker在原始调用基础上执行了一系列特殊操作。invoker的使用如下所示：

```
var rev = invoker('reverse', Array.prototype.reverse);

_.map([[1,2,3]], rev);
//=> [[3,2,1]]
```

虽然直接对一个实例执行特定方法是完全合法的，但函数式的风格更倾向于将调用对象作为参数。invoker会在一个对象不包含给定方法时返回undefined，这样可以使用JavaScript天然的多态性来构建多态函数。不过，我将在第5章 再来讨论。

4.2.1 高阶函数捕获参数

另一个为什么你可能需要创建一个以函数为返回值的函数的理由是，高阶函数的参数是用来“配置”返回函数的行为的。对于高阶函数makeAdder而言，它的参数配置了其返回函数每次添加数值的大小。注意观察以下代码：

```
var add100 = makeAdder(100);
add100(38);
//=> 138
```

具体来说，通过将函数makeAdder的返回函数命名为add100，这里特别强调了返回函数是如何被“配置”的。也就是说，它的配置是始终向传入的任何值加100。这是一个非常有用的技术，但其能力有限。不同的是，你会经常看到一个函数返回了一个捕获变量的函数，而这正是我现在要讨论的内容。

4.2.2 捕获变量的好处

假设你需要一个能够生成唯一字符串的函数。实现可能如下所示^[2]。

```
function uniqueString(len) {
  return Math.random().toString(36).substr(2, len);
};

uniqueString(10);
//=> "3rm6ww5w0x"
```

然而，如果需要生成具有特定前缀的唯一字符串，该怎么办？可将 `uniqueString` 修改为：

```
function uniqueString(prefix) {  
    return [prefix, new Date().getTime()].join('');  
};
```

```
uniqueString("argento");  
//=> "argento1356107740868"
```

新 `uniqueString` 似乎可以工作。但是，如果需要再次变更，需要返回一个添加了前缀，并且后缀从某一个值开始增长的字符串，该怎么办？在这种情况下，函数应该像下面这样工作：

```
uniqueString("ghosts");  
//=> "ghosts0"
```

```
uniqueString("turkey");  
//=> "turkey1"
```

新的实现可以用闭包来捕获增加值，并用作后缀：

```
function makeUniqueStringFunction(start) {  
  var COUNTER = start;  
  
  return function(prefix) {  
    return [prefix, COUNTER++].join('');  
  }  
};  
  
var uniqueString = makeUniqueStringFunction(0);  
  
uniqueString("dari");  
//=> "dari0"  
  
uniqueString("dari");  
//=> "dari1"
```

对于makeUniqueStringFunction函数，变量COUNTER被函数返回并捕获。这似乎工作得很好，但我们能不能在一个对象上实现相同的功能？例如：

```
var generator = {  
  count: 0,  
  uniqueString: function(prefix) {  
    return [prefix, this.count++].join('');  
  }  
};
```

```
generator.uniqueString("bohr");  
//=> bohr0
```

```
generator.uniqueString("bohr");  
//=> bohr1
```

但它有一个缺点（除了它不是函数式），即不够安全：

```
// reassign the count  
generator.count = "gotcha";  
generator.uniqueString("bohr");  
//=> "bohrNaN"  
  
// dynamically bind this  
generator.uniqueString.call({count: 1337}, "bohr");  
//=> "bohr1337"
```

此时，事实上你的系统处于一个危险的状态。

makeUniqueStringFunction所使用的方法隐藏了COUNTER。也就是说，COUNTER变量是返回闭包“私有”的。现在，我并不是在讲究私有变量和对象属性，但有时候隐藏关键实现细节是很重要的。事实上，我们可以使用JavaScript的秘密武器将COUNTER隐藏在generator中：


```

var omgenerator = (function(init) {
    var COUNTER = init;

    return {
        uniqueString: function(prefix) {
            return [prefix, COUNTER++].join('-');
        }
    };
})(0);

omgenerator.uniqueString("lichking-");
//=> "lichking-0"

```

但这有什么意义呢？创建这样奇怪的实现有时是必要的，特别是在建立模块/命名空间时，但它不是我想使用的方法^[3]。闭包的方式很干净、简单，并且很优雅，但它也充满了陷阱。

改变值时要小心

我打算在第7章 更细致地讨论改变变量的危险性，但现在我可以花点时间简单介绍一下。makeUniqueStringFunction使用了名为COUNTER的变量来追踪当前值。虽然对于外界操作来说，该变量是安全的，但它的存在会增加复杂度。当一个函数的返回值只依赖于它的参数时，被称为具有引用透明（referential transparency）。

这似乎是一个很花哨的名词，但它只是意味着你应该能够在不破坏自己代码的情况下，用预期值来替换一个函数的任意调用。当你使用一个会改变内部代码的闭包时，你不一定能做到这一点，因为它返回的值是完全依赖于它的调用次数的。也就是说，调用uniqueString10次所返回的值与调用10 000次所返回的值是不同的。用值来替换uniqueString的唯一方法是，你确切地知道它会在任意时间点被调用多少次，但这是不可能的。

再次说明，我会在第7章 对这个问题进行更多的讨论，但值得指出的是，我会避免类似于makeUniqueStringFunctions这样的函数，除非它们是绝对必要的。相反，我觉得你会惊讶地发现在函数式编程中对状态改变的要求甚少。在第一次面对设计函数式程序时，你需要时间来改变你的思维

定式，但我希望在读完这本书后，你会对为什么一个可变的狀態是有潜在危险的有更好的理解，并且你会努力去避免它。

4.2.3 防止不存在的函数：fnull

在进入第5章 之前，我想创建几个高阶函数作为例子。我们要讨论的第一个高阶函数叫作fnull。为了描述fnull的目的，我想展示一些它想要解决的错误。假设我们有一组需要执行乘法的数字数组：

```
var nums = [1,2,3,null,5];

_.reduce(nums, function(total, n) { return total * n });
//=> 0
```

很明显，乘以null是不会给我们任何有用的答案的。另一个有问题的场景是：一个函数接收一个配置对象作为输入来执行一些动作：

```
doSomething({whoCares: 42, critical: null});
// explodes
```

在这两种情况下，有一个fnull函数将是很有用的。fnull接收一个函数及一些额外的参数，并返回一个只是调用给定的原始函数的函数。fnull神奇的地方在于，对于任何是null或undefined的参数，都用原来“默认”的参数来代替。这里将要展示的fnull是实现最为复杂的高阶函数，但它仍然是相当合理的。注意观察以下代码：

```

function fnull(fun /*, defaults */) {
  var defaults = _.rest(arguments);

  return function(/* args */) {
    var args = _.map(arguments, function(e, i) {
      return existy(e) ? e : defaults[i];
    });

    return fun.apply(null, args);
  };
};

```

fnull的工作原理是，检查进来的参数是否是null或undefined，如果是则用默认值来替换，然后再调用函数。fnull的一个特别有趣的地方是，只有在守卫函数被调用时，用来遍历默认值的成本才会产生。也就是说，分配默认值是以懒惰方式完成的，只在需要的时候发生。

你可以用以下方式使用fnull：

```

var safeMult = fnull(function(total, n) { return total * n }, 1, 1);

_.reduce(nums, safeMult);
//=> 30

```

使用fnull来创建safeMult函数可以防止其受null或undefined影响。这也带来了另一个好处：能够在没有给出参数时得到一个确定值的乘法函数。

为了解决配置对象的问题，可以通过以下方式使用fnull：

```

function defaults(d) {
  return function(o, k) {
    var val = fnull(_.identity, d[k]);
    return o && val(o[k]);
  };
}

function doSomething(config) {
  var lookup = defaults({critical: 108});

  return lookup(config, 'critical');
}

doSomething({critical: 9});
//=> 9

doSomething({});
//=> 108

```

对于任何给定的配置对象，利用fnull可以保证临界值被设置为合理的默认值。这有助于避免函数开头的长长的警告，以及`o[k] || someDefault`模式的必要性。在defaults内部使用fnull函数，说明在函数式编程中从较低级别的函数中提取高阶函数的倾向。同样，defaults返回了一个函数，这对提供一个访问数组的额外层很有帮助^[4]。因此，使用函数式编程风格可以封装默认值，并检查被隔离的函数逻辑，从doSomething函数中分离出来。继续这个话题，我将用一个建立对象字段验证的函数来结束本章。

4.3 整合：对象校验器

为了结束本章，我将解决一个JavaScript中普遍的需求：判断对象是否有效。例如，假设你要创建一个通过JSON对象接收外部命令的应用程序。这些命令的基本格式如下：

```
{message: "Hi!",  
  type: "display"  
  from: "http://localhost:8080/node/frob"}
```

在简单地直接使用这条消息并遍历它之前，如果有个简单的方法来验证这条消息，那就好了。我想看到更加流畅且容易组合的方法，它能够报告给定的命令对象中的所有错误。在函数式编程中，接收和返回其他函数的函数所提供的灵活性不能低估。事实上，解决命令验证问题是一个挺普遍的方法，只是为了提供友好的错误报告而显得有一点复杂。

这里我给出一个名为checker的函数，它接收一组谓词函数（返回true或false的函数），并返回一个验证函数。返回的验证函数在给定对象上执行每个谓词，并对每一个返回false的谓词增加一个特殊的错误字符串到一个数组中。如果所有的谓词返回true，那么最终返回的结果是一个空数组；否则，结果为错误消息的数组。checker的实现如下所示：

```
function checker(/* validators */) {  
  var validators = _.toArray(arguments);  
  
  return function(obj) {  
    return _.reduce(validators, function(errs, check) {  
      if (check(obj))  
        return errs  
      else  
        return _.chain(errs).push(check.message).value();  
    }, []);  
  };  
}
```

在这种情况下使用`_.reduce`是合适的，因为在每个谓词被验证时，`errs`数组将会被增加或不发生变化。顺便说一句，我喜欢用Underscore的`_.chain`函数来避免下面这种可怕的模式：

```
{
  errs.push(check.message);
  return errs;
}
```

使用`_.chain`肯定需要更多的字符，但它很好地隐藏了数组变化（我将在第 7 章更多地讨论隐藏变化）。请注意，`checker`函数在谓词对象中去查找一个`message`属性。为了这样，我打算用那些将错误信息作为伪元数据（pseudo-metadata）包含在特殊验证函数内。这不是一个通用解决方案，但在我所实现的代码中，它是一个有效的用例。

对验证命令对象的一个基本测试如下所示：

```
var alwaysPasses = checker(always(true), always(true));
alwaysPasses({});
//=> []

var fails = always(false);
fails.message = "a failure in life";
var alwaysFails = checker(fails);

alwaysFails({});
//=> ["a failure in life"]
```

需要在每次创建验证器时都记得去设置一个`message`会让人感觉有点痛苦。同样，如果能够避免在不属于自己的验证器上添加属性将会更好。可以想到的是，`message`是一个非常普通的属性名，如果给它设置值将可能会抹掉正常的属性值。我可以用类似于`_message`这样的名字来混淆属性名，但这并不利于记忆。相反，我希望能有一个特定的API来创建验证器——能够一目了然的那种。我的解决方案是一个叫做`validator`的高阶函数，其定义如下：

```

function validator(message, fun) {
  var f = function(/* args */) {
    return fun.apply(fun, arguments);
  };

  f['message'] = message;
  return f;
}

```

validator函数的简单使用证明了这一策略：

```

var gonnaFail = checker(validator("ZOMG!", always(false)));

gonnaFail(100);
//=> ["ZOMG!"]

```

相较于在使用的地方直接定义，我更愿意单独定义每一个“checkers”。这使得我可以给它们起更具描述性的名字，例如：

```

function aMap(obj) {
  return _.isObject(obj);
}

```

aMap函数可以被用作checker的一个参数，来实现一个虚拟的句子：

```

var checkCommand = checker(validator("must be a map", aMap));

```

当然，正如你所料，它的使用方法如下所示：

```

checkCommand({});
//=> true

checkCommand(42);
//=> ["must be a map"]

```

添加简单的checker很简单。然而，保持高水平的流畅可能需要一些有趣的技巧。如果回忆一下本章前面的内容，我提到过对于返回函数的函

数，其参数可以作为所返回的闭包的行为配置。牢记这一点，可以让你随时随地在需要函数的地方返回配置过的闭包。

以验证命令对象是否包含某些特定的键为例。形容这个checker的最好方法是什么？我要说的是拿到一个所需键的简单列表将会很流畅——例如 `hasKeys('msg', 'type')`。为了让`hasKeys`顺应这个调用约定，我们让它返回一个闭包和一个错误数组，其实现如下所示：

```
function hasKeys() {
  var KEYS = _.toArray(arguments);

  var fun = function(obj) {
    return _.every(KEYS, function(k) {
      return _.has(obj, k);
    });
  };

  fun.message = cat(["Must have values for keys:"], KEYS).join(" ");
  return fun;
}
```

你会发现闭包（捕获KEYS）用来检查给定的对象是否有效^[5]。`hasKeys`函数的目的是向`fun`提供执行配置。此外，通过直接返回一个函数，我提供了一个很好的用于描述所需键的流畅的界面。从一个函数返回另一个函数的技术——在这个过程中捕获参数——被称为“柯里化（currying）”

（我将会在第5章 更多地讨论柯里化）。最终，在返回绑定到`fun`的闭包之前，我附上一个有用的`message`属性，来存储所有需要键的列表。还可以通过其他一些额外的工作让它的信息更丰富一些，但作为例子，它已经够用了。

`hasKeys`函数的使用方法如下所示：

```
var checkCommand = checker(validator("must be a map", aMap),
  hasKeys('msg', 'type'));
```

`checkCommand`函数的组成很有意思。你可以将它的操作当作编译链中的一个验证模块，其中参数在各种检查关卡中传递，并在验证过程中被执行。事实上，正如你在本书中所经历的，你会发现，函数式编程的确可以被看作一种构建虚拟装配生产线的方式，其中数据从一个函数式“机器”输入，经过一系列的转变和验证（可选），最终以另外一种东西被返回。

在任何情况下，使用新的checkCommand检查器来构建“句子的一致性”，都会如你所想象的那样工作：

```
checkCommand({msg: "blah", type: "display"});  
//=> []
```

```
checkCommand(32);  
//=> ["must be a map", "Must have values for keys: msg type"]
```

```
checkCommand({});  
//=> ["Must have values for keys: msg type"]
```

这很好地突出了本章要涵盖的重点内容。我将进一步深入探讨这些话题，checker也将在本书中再次露面。

4.4 总结

在本章中，我讨论了属于头等函数的高阶函数，它能够实现下面的一项或两项。

- 以一个函数作为参数。
- 以一个函数作为返回结果。

为了说明如何传递函数给别的函数，本章给出了很多例子，其中包括max、finder、best、repeatedly和iterateUntil。很多时候，为了实现一些行为而将值传递给函数是有价值的，但有时可以通过传递函数使得这样的任务变得更加通用。

对返回函数的函数的讲解开始于有用的always函数。always的一个有趣的特点是，它返回一个闭包，一种你会经常在JavaScript中见到的技术。此外，返回函数的函数能够构建强大的函数，如fnull对null的处理，以及对默认参数的支持。同样，我们使用了很少的代码，用高阶函数建立了一个强大的一致性检查系统：checker。

在下一章中，我将带着我们目前所学到的一切，完全用其他函数来“组合”新函数。

[1] 与Underscore的min函数类似。

[2] 实际上能保证产生唯一的字符串，但我希望的意图是明确的。

[3] ECMAScript.next正准备努力通过模块系统，用简单的声明来处理可见性问题。更多信息请参考<http://wiki.ecmascript.org/doku.php?id=harmony:modules>。

[4] ECMAScript.next 努力制定函数可选参数的规范。目前还不清楚是否要往 JavaScript 内核加入该特性，但我觉得是个不错的特性。更多信息请参考http://wiki.ecmascript.org/doku.php?id=harmony:parameter_default_values。

[5] hasKeys函数中的_.has函数可检查对象是否存在，而当我使用existy(obj[k]) 时，对象是null或undefined时会没用。

第5章 由函数构建函数

本章建立在一等函数的思想基础上，解释如何以及为什么需要构建函数，并探讨了如何像搭乐高积木一样，通过细小部件能构建出丰富的函数。

5.1 函数式组合的精华

回想一下，第4章 中用函数invoker建立了一个接收对象作为其第一个参数的函数，并试图调用该方法。你应该还记得，如果调到目标对象不具有的方法，invoker返回undefined。这样可以将多个invoker组合在一起，形成多态函数，或根据不同参数产生不同行为的函数。要做到这一点，需要一种接收一个或多个函数，然后不断尝试依次调用这些函数的方法，直到返回一个非undefined的值。这样的函数dispatch，可以被命令式地定义如下：

```
function dispatch(/* funs */) {  
  var funs = _.toArray(arguments);  
  var size = funs.length;  
  
  return function(target /*, args */) {  
    var ret = undefined;  
    var args = _.rest(arguments);  
  
    for (var funIndex = 0; funIndex < size; funIndex++) {  
      var fun = funs[funIndex];  
      ret = fun.apply(fun, construct(target, args));  
  
      if (existy(ret)) return ret;  
    }  
  
    return ret;  
  };  
}
```

对于一个简单的任务来说，这段代码会显得太啰嗦^[1]。

在这里我们想做的只是返回一个遍历函数数组，并apply给一个目标对象的函数，返回第一个存在的值。然而，尽管看似复杂，dispatch却满足了多态JavaScript函数的定义。这样做简化了委托具体方法的任务。例如，在Underscore的实现中，你会经常看到许多不同的函数重复这样的模式。

1. 确保目标的存在。
 2. 检查是否有原生版本，如果是则使用它。
 3. 如果没有，那么做一些实现这些行为的具体任务。
- 做特定类型的任务（如适用）。
 - 做特定参数的任务（如适用）。
 - 做特定个参数的任务（如适用）。

同样的模式也体现在Underscore的函数`_.map`的实现中：

```
_.map = _.collect = function(obj, iterator, context) {  
  var results = [];  
  if (obj == null) return results;  
  if (nativeMap && obj.map === nativeMap) return obj.map(iterator, context);  
  each(obj, function(value, index, list) {  
    results[results.length] = iterator.call(context, value, index, list);  
  });  
  return results;  
};
```

使用`dispatch`可以简化一些这方面的代码，并且更容易扩展。想象一下，你正在写一个可以为数组和字符串类型生成字符描述的函数。使用`dispatch`则可以优雅地实现：

```
var str = dispatch(invoker('toString', Array.prototype.toString),  
                   invoker('toString', String.prototype.toString));  
  
str("a");  
//=> "a"  
  
str(_.range(10));  
//=> "0,1,2,3,4,5,6,7,8,9"
```

也就是说，通过耦合`invoker`与`dispatch`，可以向下代理给具体实现，比如用`Array.prototype.toString`，而不是通过`if-then-else`进行群体类型和存在检查的单一函数^[2]。

当然，`dispatch`操作是不依赖于`invoker`的，而是附着在一个特定的约定上。约定持续尝试执行函数，直到返回`existy`值为止。可以通过提供一个坚持约定的函数来获取这种约定，例如`stringReverse`：

```
function stringReverse(s) {
  if (!_isString(s)) return undefined;
  return s.split('').reverse().join("");
}
```

```
stringReverse("abc");
//=> "cba"
```

```
stringReverse(1);
//=> undefined
```

现在stringReverse可以与Array#reverse方法组合来定义一个新的多态函数：rev，如下所示：

```
var rev = dispatch(invoker('reverse', Array.prototype.reverse), stringReverse);

rev([1,2,3]);
//=> [3, 2, 1]

rev("abc");
//=> "cba"
```

此外，我们可以利用dispatch的约定组成一个带有默认行为的终止函数，如总是返回已有值或抛出异常。作为一个不错的奖励，通过dispatch创建的函数也可以作为参数传递给dispatch：

```
var sillyReverse = dispatch(rev, always(42));

sillyReverse([1,2,3]);
//=> [3, 2, 1]

sillyReverse("abc");
//=> "cba"

sillyReverse(1000000);
//=> 42
```

dispatch的一个更有趣的模式是可以消除下面这样的switch语句调度:

```
function performCommandHardcoded(command) {  
    var result;  
  
    switch (command.type)  
    {  
    case 'notify':  
        result = notify(command.message);  
        break;  
    case 'join':  
        result = changeView(command.target);  
        break;  
    default:  
        alert(command.type);  
    }  
  
    return result;  
}
```

performCommandHardcoded函数中的switch语句通过command对象的一个字段, 分派给相关的代码:

```
performCommandHardcoded({type: 'notify', message: 'hi!'});  
// does the notify action  
  
performCommandHardcoded({type: 'join', target: 'waiting-room'});  
// does the changeView action  
  
performCommandHardcoded({type: 'wat'});  
// pops up an alert box
```

可以使用下面的方式消除这种模式:


```

function isa(type, action) {
  return function(obj) {
    if (type === obj.type)
      return action(obj);
  }
}

var performCommand = dispatch(
  isa('notify', function(obj) { return notify(obj.message) }),
  isa('join', function(obj) { return changeView(obj.target) }),
  function(obj) { alert(obj.type) });

```

上面的代码中的isa函数，接受一个type字符串和一个action函数，并返回一个新的函数。返回的函数会在type字符串和obj.type相等时调用action函数；否则返回undefined。返回的undefined会促使dispatch去尝试下一函数^[3]。

为了扩展performCommandHardcoded函数，需要去改变实际的switch语句。你还可以通过简单地封装performCommand到另一个dispatch函数来扩展它：

```

var performAdminCommand = dispatch(
  isa('kill', function(obj) { return shutdown(obj.hostname) }),
  performCommand);

```

新创建的performAdminCommand函数，首先尝试分派kill命令，如果失败则尝试通过调用performCommand处理：

```

performAdminCommand({type: 'kill', hostname: 'localhost'});
// does the shutdown action

performAdminCommand({type: 'flail'});
// alert box pops up

performAdminCommand({type: 'join', target: 'foo'});
// does the changeView action

```

你还可以通过重载先前在分发链中的命令来限制一些行为：

```
var performTrialUserCommand = dispatch(  
  isa('join', function(obj) { alert("Cannot join until approved") }),  
  performCommand);
```

通过几个例子可以看出其新行为：

```
performTrialUserCommand({type: 'join', target: 'foo'});  
// alert box denial pops up  
  
performTrialUserCommand({type: 'notify', message: 'Hi new user'});  
// does the notify action
```

这是函数组合的精华：使用现有的零部件来建立新的行为，这些新行为同样也成为了已有的零部件。在本章的其余部分，我将从介绍柯里化的概念开始，讨论以其他方式组合函数来创建新的行为。

突变（mutation）是底层的操作

本书已经举过一个以命令式的方式实现函数的例子，本书的后续部分会有更多这样的例子。虽然理想情况是可以以函数式的方式编写代码，但是为追求速度或方便，一些库的原语也会使用命令式编程技术来完成。函数是抽象的量子，函数中最重要的是，它需要遵循一定的约定。没有人关心一个永远无法逃离的变量是否在函数的作用域内突变。突变有时是必要的，但我认为这是一个底层的操作，眼不见则心不烦。

本书并不是关于函数式编程的优点教条。很多函数式的技术可以控制软件开发的复杂性，但其实很多时候，有更好的方法可以实现单个部件（见图5-1）。

无论何时，在构建一个应用程序时，通过探索需要的参数来确定实现方法是否适当总是明智的。本书针对函数式编程，主张对问题和解决方案空间的充分了解，从而得出最佳解决方案。我将在第7章中贯穿讨论这个主题，不过现在我准备好了一份美味柯里（curry）食谱^[4]。



图5-1 一个“进化了”的程序员知道在合适的时候使用合适的工具

5.2 柯里化（Currying）

你其实已经见过了柯里化函数（invoker）。柯里化函数为每一个逻辑参数返回一个新函数。例如invoker，可以想象它的另一种略微不同的实现方式：

```
function rightAwayInvoker() {  
    var args = _.toArray(arguments);  
    var method = args.shift();  
    var target = args.shift();  
  
    return method.apply(target, args);  
}
```

```
rightAwayInvoker(Array.prototype.reverse, [1,2,3])  
//=> [3, 2, 1]
```

也就是说，函数rightAwayInvoker不返回一个等待目标对象的函数，而是用第二个参数也就是目标对象调用了方法。而invoker函数是柯里化，它意味着给定目标的方法调用被推迟直到参数的逻辑个数（例中为两个）耗尽。你可以通过以下代码看得出来：

```
invoker('reverse', Array.prototype.reverse)([1,2,3]);  
//=> [3, 2, 1]
```

这两个括号说明了这里发生了什么（从函数invoker返回的绑定到执行reverse的函数立即由数组[1, 2, 3]调用）？

回顾一下这个思想，返回一个已配置做特定操作的函数（闭包）是非常有用的。同样的想法可以扩展到柯里化函数。也就是说，对于每一个逻辑参数，柯里化函数会逐渐返回已配置的函数，直到所有的参数用完（见图5-2）。

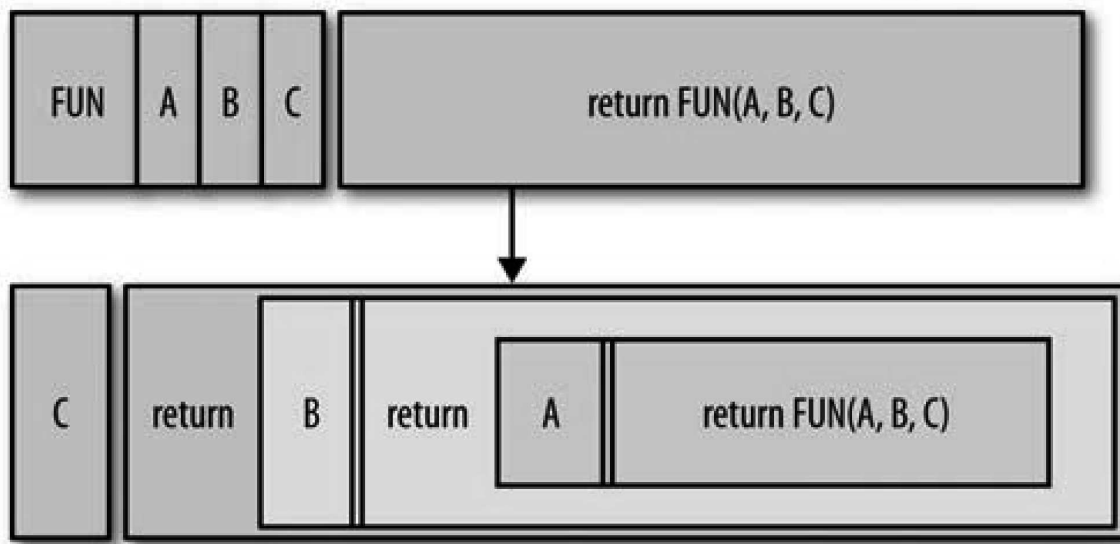


图5-2 柯里化的图形描述

柯里化的想法其实很简单，但有一点需要注意：如果一个柯里化函数为每个参数返回一个函数，那么“反柯里化”（uncurrying）应该从哪个参数开始，到哪个参数停止呢？

5.2.1 向右柯里化，还是向左

其实柯里化的方向并不重要，重要的是这个选择将会对你的API有一些影响。本书（以及我个人）偏好从最右边的参数向左柯里化。像JavaScript这样的语言，可以通过任意数量的参数，从右到左让你能修复可选参数的值。

为了说明参数方向选择的差异，请看以下两个例子：

```
function leftCurryDiv(n) {
  return function(d) {
    return n/d;
  };
}
```

```
function rightCurryDiv(d) {
  return function(n) {
    return n/d;
  };
}
```

使用除法运算很好地说明了柯里化方向的区别，因为结果会因为参数顺序而变化。观察leftCurryDiv函数如何柯里化参数产生结果的：

```
var divide10By = leftCurryDiv(10);
```

函数以参数10初始化并命名为divide10By，该函数配置为做10/?操作，其中?是下次调用时最右边的参数：

```
divide10By(2);
//=> 5
```

第二次调用柯里函数（命名为divide10By）开始执行10/2操作，得出值为5。然而，如果使用rightCurryDiv函数，情况就完全不一样了：

```
var divideBy10 = rightCurryDiv(10);
```

现在命名为divideBy10的柯里函数体变为?/10，在执行前等待最左边的参数：

```
divideBy10(2);
//=> 0.2
```

正如我所说，我将开始从最右边的参数开始柯里化，计算顺序将如图5-2 所示。

从右柯里化的另一个原因是，部分应用从左向右操作（部分应用将在下一节深入讨论）。部分应用和柯里化覆盖两个方向，允许全面的参数特化。总而言之，我将实现手动柯里（如leftCurryDiv和rightCurryDiv）和自动柯里化的函数。

5.2.2 自动柯里化参数

over10与divideBy10都是手动柯里化函数。也就是说，我显式地返回对应参数数量的函数。同样，为了说明的目的，我的函数rightCurryDiv返回一个函数，相当于接收两个参数的除法函数。然而，这是一个简单的高阶函数，它接受一个函数，并返回一个只接收一个参数的函数；我将命名该函数为curry，其实现如下：

```
function curry(fun) {  
  return function(arg) {  
    return fun(arg);  
  };  
}
```

curry的操作可以概括为：

- 接受一个函数。
- 返回一个只接收一个参数的函数。

这似乎是一个相当没用的函数。为什么不直接使用该函数呢？在许多函数式编程语言中，很少能有令人信服的理由来提供一个像curry这样无修饰的代理，但JavaScript却略有不同。很多时候在JavaScript中，函数会接收期望参数以及额外的“特化”参数。例如，JavaScript函数parseInt接受一个字符串并返回其相应的整数：

```
parseInt('11');  
//=> 11
```

此外，parseInt还可接受一个基数：

```
parseInt('11', 2);  
//=> 3
```

上述调用时给定的值的基数为2，意味着按二进制解析该数字。按照一等函数的方式使用parseInt会带来一些副作用：

```
['11', '11', '11', '11'].map(parseInt)
//=> [11, NaN, 3, 4]
```

这里的问题是，在一些版本的JavaScript中，给Array#map中的函数提供的参数包括数组元素、元素索引，以及数组本身^[5]。因此，调用parseInt时传入的基数参数将会依次为0，1，2，然后是3。看！如果使用curry，你可以强制parseInt在每次调用时只接收一个参数：

```
['11', '11', '11', '11'].map(curry(parseInt));
//=> [11, 11, 11, 11]
```

我可以很容易地编写一个接收任意数量参数的函数，然后考虑如何柯里余下参数，但我更喜欢使用柯里化将参数明确下来。其原因是，使用像curry这样的函数，让我们能显式地控制接收固定以及可选的特化参数的函数行为。

举个例子，一个柯里化两个参数的curry2函数：

```
function curry2(fun) {
  return function(secondArg) {
    return function(firstArg) {
      return fun(firstArg, secondArg);
    };
  };
}
```

curry2函数接受一个函数并将其柯里化成两个深层参数的函数。可以用它来实现先前定义的divideBy10函数：


```
function div(n, d) { return n / d }
```

```
var div10 = curry2(div)(10);
```

```
div10(50);
```

```
//=> 5
```

就像rightCurryDiv，div10函数的逻辑体对应为`n/10`。curry2还可用于固化parseInt的行为，使其解析时只处理二进制数：

```
var parseBinaryString = curry2(parseInt)(2);
```

```
parseBinaryString("111");
```

```
//=> 7
```

```
parseBinaryString("10");
```

```
//=> 2
```

柯里化有利于指定JavaScript函数行为，并将现有函数“组合”为新函数。

1. 使用柯里化构建新函数

我展示了如何使用curry2来构建一个简单div10函数，其中的除法运算符期待一个分子，但是这并没有充分展现其实用性。事实上，正如使用闭包可以根据捕获变量来定制化函数行为，柯里化以同样的方式做同样的事情。例如，Underscore提供的`_.countBy`函数给定一个数组，返回以函数返回值为键的对象：

```

var plays = [{artist: "Burial", track: "Archangel"},
              {artist: "Ben Frost", track: "Stomp"},
              {artist: "Ben Frost", track: "Stomp"},
              {artist: "Burial", track: "Archangel"},
              {artist: "Emeralds", track: "Snores"},
              {artist: "Burial", track: "Archangel"}];

_.countBy(plays, function(song) {
  return [song.artist, song.track].join(" - ");
});

//=> {"Ben Frost - Stomp": 2,
//     "Burial - Archangel": 3,
//     "Emeralds - Snores": 1}

```

`_.countBy`接收任意的函数作为第二个参数这一事实，应该能给你如何使用`curry2`构建定制函数一点提示。也就是说，你可以用`_.countBy`柯里化有用的函数来实现定制的计数功能。在这个计算艺术家的例子中，我们还可以建立一个名为`songCount`的函数：

```

function songToString(song) {
  return [song.artist, song.track].join(" - ");
}

var songCount = curry2(_.countBy)(songToString);

songCount(plays);
//=> {"Ben Frost - Stomp": 2,
//     "Burial - Archangel": 3,
//     "Emeralds - Snores": 1}

```

使用柯里化形成了一个虚拟的句子，有效地陈述“实现`songCount`，`countBy songToString`。”你会经常看到这样使用柯里化来使得函数式接口更可读的用法。

2. 柯里化三个参数来实现HTML十六进制颜色构建器

使用实现curry2相同的模式，可以定义柯里化3个参数的函数：

```
function curry3(fun) {  
  return function(last) {  
    return function(middle) {  
      return function(first) {  
        return fun(first, middle, last);  
      };  
    };  
  };  
};
```

我可以以各种有趣的方式使用curry3，包括使用Underscore的_.uniq函数来构建所有唯一演奏过的歌曲数组：

```
var songsPlayed = curry3(_.uniq)(false)(songToString);  
  
songsPlayed(plays);  
  
//=> [{artist: "Burial", track: "Archangel"},  
//    {artist: "Ben Frost", track: "Stomp"},  
//    {artist: "Emeralds", track: "Snores"}]
```

通过比较curry3与_.uniq的直接调用，你可能会看到二者清晰的关系：

```
_.uniq(plays, false, songToString);  
  
curry3(_.uniq)      (false) (songToString);
```

再回到如何用 curry3 来实现为特定的色彩生成 HTML 十六进制值。我们从函数rgbToHexString开始：

```

function toHex(n) {
  var hex = n.toString(16);
  return (hex.length < 2) ? [0, hex].join(''): hex;
}

function rgbToHexString(r, g, b) {
  return ["#", toHex(r), toHex(g), toHex(b)].join('');
}

rgbToHexString(255, 255, 255);
//=> "#ffffff"

```

可以继续柯里化该函数来生成特定的颜色或色调：

```

var blueGreenish = curry3(rgbToHexString)(255)(200);

blueGreenish(0);
//=> "#00c8ff"

```

5.2.3 柯里化流利的API

使用柯里化比较容易产生流利的函数式API。在Haskell编程语言中，函数是默认柯里化的，所以它的库自然可以利用这一事实。但在JavaScript中，函数式API的设计必须利用柯里化，而且必须文档化。然而，确定是否应该使用柯里化的通用规则是：API是否要利用高阶函数？如果答案是肯定的，那么至少一个参数的柯里化函数是合适的。例如第4章构建的checker，它接收一个函数来检查值的正确性。使用柯里化函数构建一个流畅的checker非常简单：

```

var greaterThan = curry2(function (lhs, rhs) { return lhs > rhs });
var lessThan    = curry2(function (lhs, rhs) { return lhs < rhs });

```

直接将柯里化函数greaterThan与lessThan作为谓词传入validator：

```

var withinRange = checker(
  validator("arg must be greater than 10", greaterThan(10)),
  validator("arg must be less than 20", lessThan(20)));

```

这种柯里化的用法比直接使用匿名函数要容易阅读。当然，withinRange检查器工作正常：

```
withinRange(15);  
//=> []
```

```
withinRange(1);  
//=> ["arg must be greater than 10"]
```

```
withinRange(100);  
//=> ["arg must be less than 20"]
```

现在你应该同意，使用柯里化有利于创建更流畅的接口。代码阅读起来越像它行为的描述则越好。本书会努力满足这个条件。

5.2.4 JavaScript柯里化的缺点

curry2和curry3看起来都很不错，要是能有进行任意深度柯里化的函数curryAll就更好了。事实上，建立这样的函数是可能的，但并不是很实用。像Haskell或Shen这样的编程语言，柯里化是自动完成的，API都建立在柯里化函数的基础上。而JavaScript允许可变数目的参数这点很容易让人费解，并看上去像是反柯里化的。事实上，Underscore库中提供了大量的根据不同参数类型和数目有不同行为的函数，因此，只要注意应用，也是可以实现柯里化的。

curry2和curry3对于API的设计是有用的，而且它们也可以优雅地实现函数式组合。不过，在任意层次上部分应用函数要比柯里化更实用。

5.3 部分应用

前面已经解释过，柯里化函数逐渐返回消耗参数的函数，直到所有参数耗尽。然而，部分应用函数是一个“部分”执行，等待接收剩余的参数立即执行的函数，如图5-3 所示^[6]。

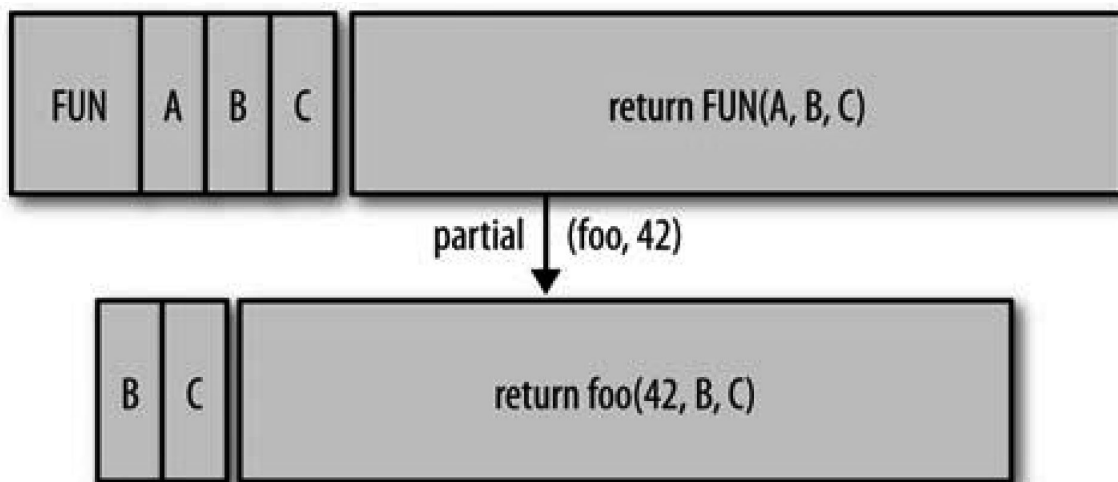


图5-3 部分应用

文字与图片说明都不错，但要了解部分应用的最好方法就是实践。想象一下over10的另一种实现：

```
fucntion divPart(n) {  
  return fucntion(d) {  
    return n / d;  
  };  
}
```

```
var over10Part = divPart(10);  
over10Part(2);  
//=> 5
```

over10Part的实现看起来与leftCurryDiv一样，这其实就是柯里化和部分应用程序之间的关系。在这里柯里化函数与部分应用在执行前都只期

待一个参数。但是，部分应用程序并不一定每次只处理一个参数，而是应用并存储部分参数，并接收剩下的参数等待运行。

柯里化和局部应用之间的关系可参考图5-4 。

虽然柯里化与部分应用程序是相关的，但它们的用法完全不同。请不要留意curry2和curry3函数是由右至左处理参数列表，虽然单凭这一事实就足以形成不同的API与使用模式。但它们之间的主要区别是，对于可变参（varargs）函数，前者不会产生太多困惑。JavaScript函数的可变参通常直接绑定前几个参数并保留末尾的参数作为可选或特化参数。换句话说，部分应用函数就是利用指定的一组已知参数必然会导致特定行为这一点，后续将介绍该部分。

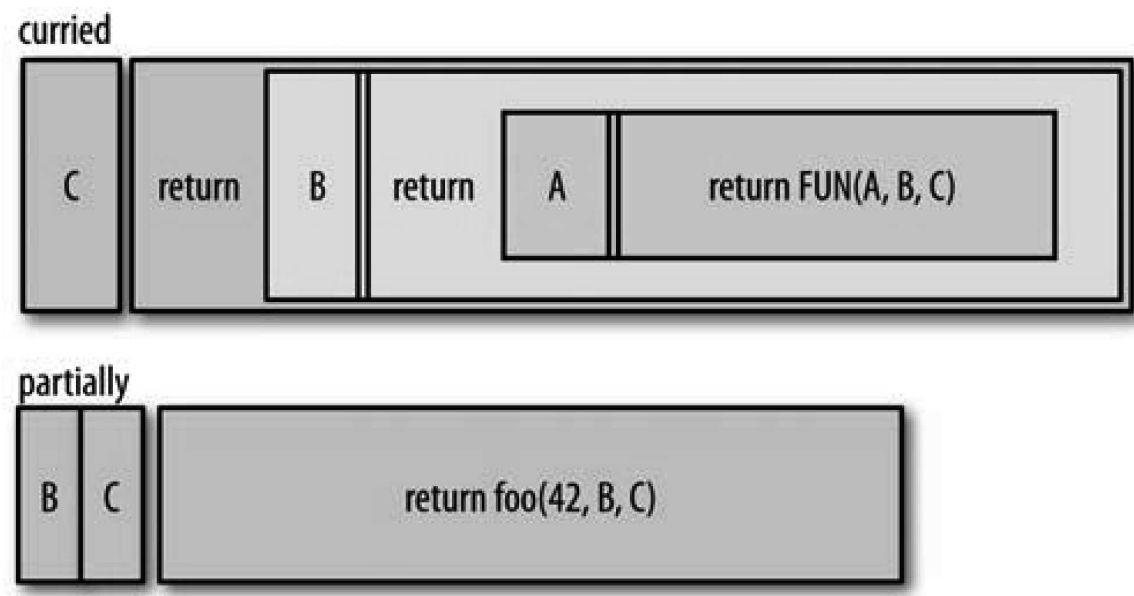


图 5-4 柯里化与部分应用的关系：柯里化函数在FUN运行之前需要三次级联调用（例如curried（3）（2）（1）），而部分应用函数已准备好被调用，只需要一次带两个参数的调用（例如partially（2，3））

5.3.1 部分应用一个和两个已知的参数

跟柯里化一样，最好是从简单例子开始。部分应用一个参数的函数 [7]：

```
function partial1(fun, arg1) {
  return function(/* args */) {
    var args = construct(arg1, arguments);
    return fun.apply(fun, args);
  };
}
```

注意，从函数partial1返回的函数捕获参数arg1，并把它放在执行调用的参数列表arglist前面。你可以通过运行该函数看出这些操作：

```
var over10Part1 = partial1(div, 10);

over10Part1(5);
//=> 2
```

我通过组合另一个函数与一个“配置”参数重新创建了函数over10^[8]。部分应用两个参数的函数也比较类似：

```
function partial2(fun, arg1, arg2) {
  return function(/* args */) {
    var args = cat([arg1, arg2], arguments);
    return fun.apply(fun, args);
  };
}
```

```
var div10By2 = partial2(div, 10, 2)

div10By2()
//=> 5
```

你会在实践中经常看到部分应用一个或两个参数，而捕捉任意数量的参数供之后执行会更有用。

5.3.2 部分应用任意数量的参数

不像要对付JavaScript复杂可变参的柯里化，部分应用任意数量的参数是正统的组合策略。值得庆幸的是，函数partial的实现并不明显比partial1或partial2更复杂。事实上，之前的实现在这里仍适用：

```
function partial(fun /*, pargs */) {  
    var pargs = _.rest(arguments);  
  
    return function(/* arguments */) {  
        var args = cat(pargs, _.toArray(arguments));  
        return fun.apply(fun, args);  
    };  
}
```

你可能已经注意到了，原理是一样的：partial捕获了一定数量的参数，并返回用它们作为参数的调用函数^[9]。运行partial也完全一样：

```
var over10Partial = partial(div, 10);  
over10Partial(2);  
//=> 5
```

而JavaScript中可变参数的存在并没有完全打败部分应用程序的实用性，它仍然可以使问题复杂化，如下所示^[10]：

```
var div10By2By4By5000Partial = partial(div, 10, 2, 4, 5000);  
div10By2By4By5000Partial();  
//=> 5
```

你可能知道部分应用只期待一个参数，而事实上却可以接收任何数量的参数，对此你可能会产生疑惑。实际上，部分应用div函数只是调用一次，参数是10和2，其余的参数都被忽略了。继续增加部分应用会使得问题更加混乱。但好消息是，实践中很少会碰到这种情况。

5.3.3 局部应用实战：前置条件

回忆第4章 的validator函数：

```
validator("arg must be a map", aMap)(42);  
//=> false
```

高阶函数`validator`接受一个验证的谓词函数并返回一个验证错误的数组。如果错误数组为空，则所有验证是通过的。`validator`可用于更广泛的用途，如手动验证函数参数：

```
var zero = validator("cannot be zero", function(n) { return 0 === n });  
var number = validator("arg must be a number", _.isNumber);  
  
function sqr(n) {  
  if (!number(n)) throw new Error(number.message);  
  if (zero(n))    throw new Error(zero.message);  
  
  return n * n;  
}
```

调用`sqr`函数检查如下：

```
sqr(10);  
//=> 100
```

```
sqr(0);  
// Error: cannot be zero
```

```
sqr('');  
// Error: arg must be a number
```

读起来相当不错，但如果使用部分应用将更好。有一类错误会被基本数据检查出来，但有一类错误会被忽略。也就是说，有一类错误涉及计算的担保。对于这类错误，有两种类型的担保。

(1) 前置条件

函数的调用者的担保。

(2) 后置条件

假设的前置条件成立，则保证函数调用的结果。

前置与后置条件之间的关系可以这么描述：给出函数能处理的数据，那么就能确保符合特定条件的结果。

之前提到的函数—sqr—有关于“numberness”和“zeroness”两个前提条件。我们可以随时验证这些条件，结果都是对的。但实际上，它们需要保证sqr关联在正在运行的应用程序的上下文中。因此，我可以使用一个新的函数condition1，以及部分应用将前置条件与计算要素分离：

```
function condition1(/* validators */) {  
  var validators = _.toArray(arguments);  
  
  return function(fun, arg) {  
    var errors = mapcat(function(isValid) {  
      return isValid(arg) ? [] : [isValid.message];  
    }, validators);  
  
    if (!_.isEmpty(errors))  
      throw new Error(errors.join(", "));  
  
    return fun(arg);  
  };  
}
```

你会注意到，从condition1返回的函数注定只需要一个参数。这里主要是为了说明的目的，因为vararg确实比较复杂并且令人困惑^[11]。问题的关键是，condition1返回的函数接受一个函数和一组函数，这组函数由validator创建，要么抛出异常Error，要么返回fun的运行结果。这是一个非常简单但功能强大的模式：

```
var sqrPre = condition1(  
  validator("arg must not be zero", complement(zero)),  
  validator("arg must be a number", _.isNumber));
```

这个验证API可读性非常好。你会很快发现，通过函数组合，你的代码变得更像声明一样（描述要做什么，而不是怎么做）。下面是运行condition1返回的sqrPre过程：

```
sqrPre(_.identity, 10);  
//=> 10
```

```
sqrPre(_.identity, '');  
// Error: arg must be a number
```

```
sqrPre(_.identity, 0);  
// Error: arg must not be zero
```

回顾一下sqr的定义，你可能已经猜到了我们如何使用sqrPre来检查它的参数。如果没有，那么试想一下“不安全”版本的sqr定义：

```
function uncheckedSqr(n) { return n * n };  
  
uncheckedSqr('');  
//=> 0
```

显然，即使它可以用JavaScript的弱点当借口，空字符串的平方也不应该是0。值得庆幸的是，我已经建立了一套工具，用validator，partial1，condition1和sqrPre可以解决这个问题^[12]：

```
var checkedSqr = partial1(sqrPre, uncheckedSqr);
```

通过新建立的函数checkedSqr，组合了sqrPre与uncheckedSqr的功能：

```
checkedSqr(10);  
//=> 100
```

```
checkedSqr('');  
// Error: arg must be a number
```

```
checkedSqr(0);  
// Error: arg must not be zero
```

checkedSqr与之前的sqr用起来完全一样，但是分离了计算与有效性验证，这样的灵活性已经达到了预期。也就是说，我现在只需要不传入验证函数即可关闭验证，同样也可以增加额外的验证：

```
var sillySquare = partial1(
  condition1(validator("should be even", isEven)),
  checkedSqr);
```

因为condition1返回的是一个期待另一个函数的函数，使用partial1结合了这两个函数：

```
sillySquare(10);
//=> 100

sillySquare(11);
// Error: should be even

sillySquare('');
// Error: arg must be a number

sillySquare(0);
// Error: arg must not be zero
```

明显你不希望数字的平方被限制到这种无聊的程度，但我想已经阐明了我的观点。组成其他函数的函数也是由别的函数组合而成的。在讲下一节之前，我们再回头看看如何重新实现一遍命令对象，看如何重新实现有验证命令对象（第4章）：

```
var validateCommand = condition1(
  validator("arg must be a map", _.isObject),
  validator("arg must have the correct keys", hasKeys('msg', 'type')));

var createCommand = partial(validateCommand, _.identity);
```

为什么要使用`_.identity`函数作为createCommand函数的逻辑部分？在JavaScript中，我们需要通过规矩和缜密的思考来达到安全性。createCommand的情况是，其目的在于提供同一种函数用于创建和验证命令对象：

```
createCommand({});  
// Error: arg must have right keys
```

```
createCommand(21);  
// Error: arg must be a map, arg must have right keys
```

```
createCommand({msg: "", type: ""});  
//=> {msg: "", type: ""}
```

但是，使用函数组合可以让你后来为了定义实际的建筑逻辑或验证本身建立在现有的创作抽象之上。如果想建立需要的另一个派生命令的类型，那么你可以像下面这样组合：

```
var createLaunchCommand = partial1(  
  condition1(  
    validator("arg must have the count down", hasKeys('countDown'))),  
  createCommand);
```

createLaunchCommand如预期的一样：

```
createCommand({msg: "", type: ""});  
// Error: arg must have the count down
```

```
createCommand({msg: "", type: "", countDown: 10});  
//=> {msg: "", type: "", countDown: 10}
```

无论用柯里化还是部分应用来构建函数都有局限性：都只按照它们的参数来组合。然而，你可能要根据参数与它们的返回值之间的关系来组合函数。在下一节中，我将谈到函数compose，允许函数的端到端的拼接。

5.4 通过组合端至端的拼接函数

一个理想化的函数式程序是向函数流水线的一端输送的一块数据，从另一端输出一个全新的数据块。事实上，JavaScript程序员一直在这样做：

```
!_.isString(name)
```

这个流水线由`_.isString`与`!`组成，其中包括如下几种。

- `_.isString`接收一个对象，并返回一个布尔值。
- `!`接收一个布尔值，并返回一个布尔值。

通过组合多个函数及其数据转换建立新的函数：

```
function isntString(str) {  
  return !_.isString(str);  
}
```

```
isntString(1);  
//=> true
```

还可以使用Underscore的`_.compose`函数实现同样的功能：

```
var isntString = _.compose(function(x) { return !x }, _.isString);  
  
isntString([]);  
//=> true
```

`_.compose`函数从右往左执行。也就是说，最右边函数的结果会被送入其左侧的函数，一个接一个。通过格式化代码，你可以看到该函数与原函数的对应：

```
!      _.isString("a");  
  
_.compose(function(str) { return !str }, _.isString)("a");
```

事实上，`!`操作符还可以封装到它自己的函数：

```
function not(x) { return !x }
```

用not函数组合成你期待的函数：

```
var isntString = _.compose(not, _.isString);
```

使用组合的方式轻松将字符串转换成布尔变量。这种组合模型就像搭乐高积木一样，通过组合函数来创建新的函数。

之前定义的函数mapcat，还可以用_.compose重新定义^[13]：

```
var composedMapcat = _.compose(splat(cat), _.map);
```

```
composedMapcat([[1,2],[3,4],[5]], _.identity);  
//=> [1, 2, 3, 4, 5]
```

还有无数种方式组合函数，以形成更多的功能。现在我就先介绍一种。

组合前置与后置条件

在上一节中，我提到前置条件下定义的函数的操作会产生一个遵循一组不同的约束条件的值。这些生产限制被称为后置条件。使用condition1和partial，能够建立一个函数（checkedSqr）来检查输入参数uncheckedSqr是否符合其前提条件。不过，如果我想定义平方的后置条件，那么需要这样定义condition1：

```
var sqrPost = condition1(  
  validator("result should be a number", _.isNumber),  
  validator("result should not be zero", complement(zero)),  
  validator("result should be positive", greaterThan(0)));
```

我可以一一检查出现错误的情况：


```
sqrPost(_.identity, 0);  
// Error: result should not be zero, result should be positive  
  
sqrPost(_.identity, -1);  
// Error: result should be positive  
  
sqrPost(_.identity, '');  
// Error: result should be a number, result should be positive  
  
sqrPost(_.identity, 100);  
//=> 100
```

但问题出现了：我怎么能将后置条件函数check粘到现有的uncheckedSqr和sqrPre上呢？当然，答案是使用`_.compose`粘合[\[14\]](#)。

```
var megaCheckedSqr = _.compose(partial(sqrPost, _.identity), checkedSqr);
```

用法与checkedSqr是一样的：

```
megaCheckedSqr(10);  
//=> 100
```

```
megaCheckedSqr(0);  
// Error: arg must not be zero
```

以下情况除外：

```
megaCheckedSqr(NaN);  
// Error: result should be positive
```

当然，如果函数抛出过一个后置条件的错误，那么这意味着前提条件可能不足，或者后置条件过足，也有可能使内部逻辑错乱。一切后置条件的失败都永远是函数的提供者的错。

5.5 总结

本章中介绍了无论是通用还是专用函数，新的函数可以从现有的函数来构建的理念。第一阶段的组合是调用了一个又一个函数，然后将这些调用包装在另一个函数中。但是，使用专门的组合函数往往更容易阅读与理解。

第一个介绍的组合函数是`_.curry`，它接收一个函数和一定数量的参数并返回一个接收剩余参数的函数。因为JavaScript允许可变数目的参数，我们用静态的函数`curry`和`curry2`来创建已知的参数大小的函数。为了引入柯里化，我还使用该技术实现了一些有趣的函数。

第二个引入的组合函数是`partial`，它接收一个函数和一定数量的参数并返回一个固定了第一个参数的函数。通过局部应用`partial`，`partial1`和`partial2`，证明了一个比柯里化更广泛适用的技术。

最后一个组合函数是`_.compose`，接收一定数量的函数并将它们从右至左拼接。高阶函数`_.compose`是用来建立在第4章中`checker`实现的教训之上的，使用了特别少量的代码，用前置和后置函数做“装饰”。

下一章仍然是过渡性章节，将再次覆盖JavaScript中不是很流行但在函数式中很普通的话题：递归。

[1] `construct`函数在第2章中定义。

[2] 直接使用`Array.prototype.toString`。

[3] 有些语言提供自动调度的方式，如通过谓词列表或任意函数的结果来确定函数的行为。

[4] 这里的“柯里”与美味的食物没有任何关系。相反，它是以数学家Haskell Curry的名字命名的，他重新发掘了由另一个名数学家Moses Schönfinkel设计的技术。尽管Haskell Curry应该是对计算机科学贡献很大的人，但我们似乎已经错过了一个有趣的叫作schönfinkeling编程方法的机会。

[5] Underscore的`map`函数也会被这一问题困扰。

[6] 最近的JavaScript的版本提供了Function.bind方法执行部分应用（Herman 2012）。

[7] 你也可以使用原生的bind方法（如果有的话）来实现partial1，替换内容为return fun.bind(undefined, arg1);。

[8] 同样，over10可以通过原生的bind，如var over10 = div.bind(undefined, 10);。你也可以使用原生bind的方法实现partial2，只需要替换成return fun.bind(undefined, arg1, arg2);。

[9] JavaScript的原生bind，让你部分应用任意数量参数的函数。为了达到与partial相同的效果，你可以执行以下操作：
fun.bind.apply(fun, construct(undefined, args))。

[10] Underscore也有本章提及的partial函数。然而，Underscore的本质是，默认参数顺序与其使用是没有关系的。partial真正的意义是在于使用现有函数创建新的函数。正如Underscore最受欢迎的地方，集合优先，消除了通过部分应用修饰（modifier）函数作为第一个参数的特化高阶函数的动力。

[11] 我将此部分作为练习留给读者。

[12] 在这个例子中，我本可以用partial而不是partial1，但我喜欢代码更明确。

[13] 我早在第1章就定义了splat。

[14] 另一种方法是重写condition1，操作名为Either的一个中间对象类型，Either持有所得到的值或一个错误字符串。

第6章 递归

本章是一个过渡性的章节，旨在平滑地从思考函数转向更深程度的函数式风格的思考，包括何时打破它，以及为什么有时候这样做很好。具体来说，本章包含了递归，直接或通过其他函数调用自己的函数。

6.1 自吸收 (self-absorbed) 函数 (调用自己的函数)

从历史上看，递归和函数式编程是相关的，或者至少是它们经常被一起介绍。在本章中，我将解释它们是如何相关的。但现在我可以说，理解递归对理解函数式编程来说非常重要，原因有三。

- 递归的解决方案包括使用对一个普通问题子集的单一抽象的使用。
- 递归可以隐藏可变状态。
- 递归是一种实现懒惰和无限大结构的方法。

如果你考虑一个函数本质，如myLength接受一个数组，并返回它的长度（元素的数量），那么你可能会得到下面的描述^[1]。

1. 从零开始计算数组大小。
2. 遍历数组，每遍历一个元素，数组大小加1。
3. 遍历到数组结束，那么数组大小就是它的长度。

这是一个对myLength的正确描述，但它不涉及递归的思想。相反，一个递归式的描述将如下所示。

1. 如果数组是空的，那么它的长度是零。
2. 对数组的其余部分，添加一个结果到myLength。

我可以直接在myLength的实现中执行这两个规则，如下所示：

```
function myLength(ary) {  
  if (_.isEmpty(ary))  
    return 0;  
  else  
    return 1 + myLength(_.rest(ary));  
}
```

递归函数非常有利于构建值。实现递归的关键在于认识到某个值是由一个更大问题的子问题构建出来的。对于myLength而言，整个解决方案可以被看作向一个空数组的长度增加一定数量的单元素数组。由于myLength用`_.rest`的结果来调用自己，每个递归调用得到少了一个元素的数组，直到最后调用得到一个空数组（见图6-1）。

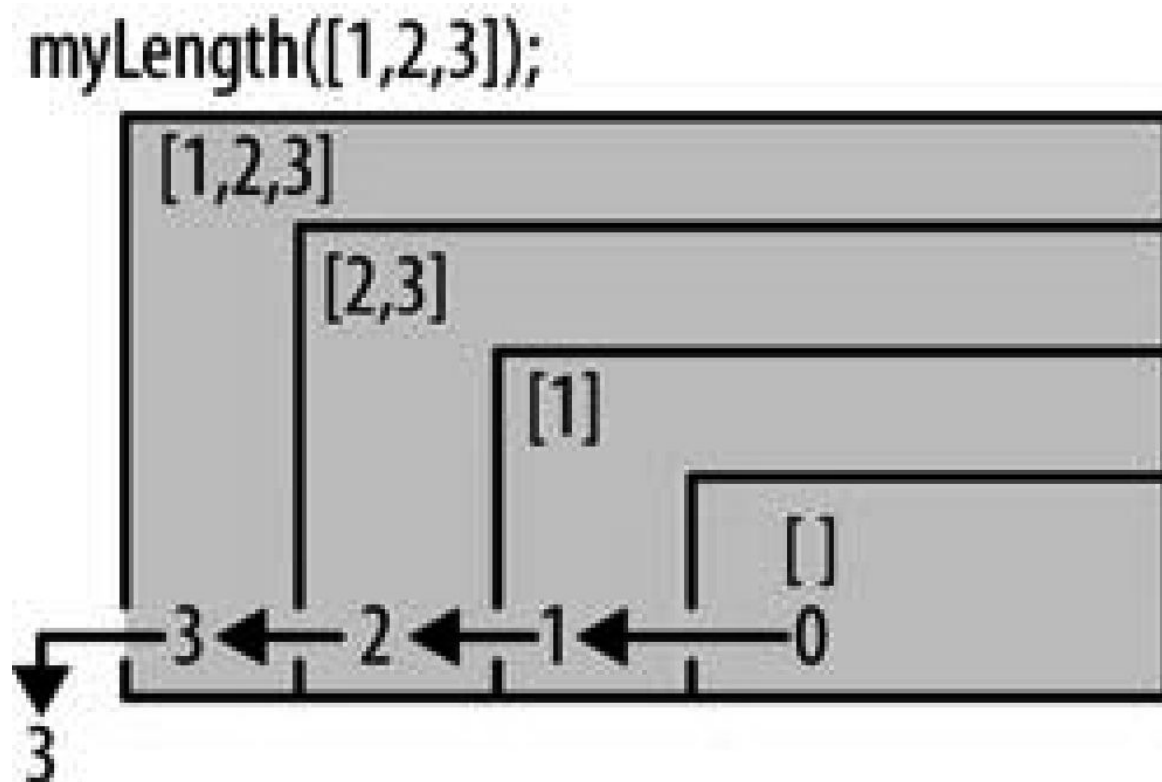


图6-1 递归式的myLength会“消耗”数组

观察下面myLength的行为：

```
myLength(_.range(10));  
//=> 10
```

```
myLength([]);  
//=> 0
```

```
myLength(_.range(1000));  
//=> 1000
```

需要知道的是，不应该改变传给递归函数的参数：

```
var a = _.range(10);
```

```
myLength(a);  
//=> 10
```

```
a;  
//=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

虽然一个递归函数可能在逻辑上消耗它的参数，但它不应该真的这样去做。虽然在myLength 中，它利用输入构建了一个整数返回值，但事实上递归函数可以创建任何类型的合法值，包括数组和对象。例如，对于函数cycle，它接受一个数字和一个数组，可以构建一个新数组，新数组中一定次数的重复包含了输入数组：

```
function cycle(times, ary) {  
  if (times <= 0)  
    return [];  
  else  
    return cat(ary, cycle(times - 1, ary));  
}
```

cycle函数的形式看起来与myLength函数类似。也就是说，myLength “消耗” 了输入数组，而cycle “消耗” 了重复次数。同样，每个步骤中构建的是一个新的数组。这种消费/构建的过程如图6-2所示。

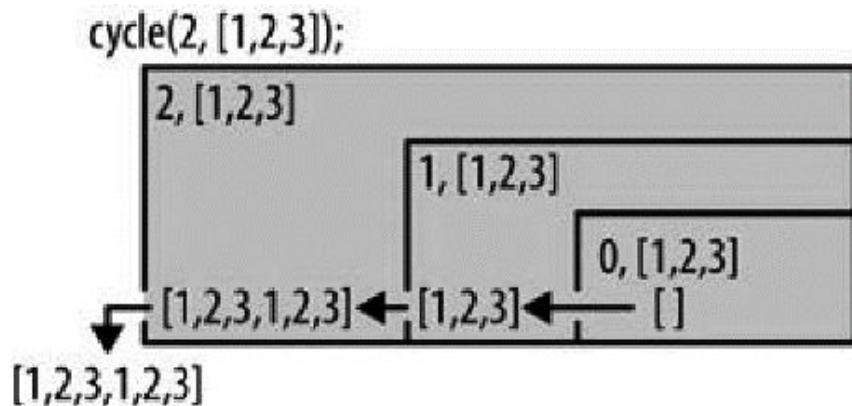


图6-2 构建一个数组的递归循环

以下是cycle的使用方法：

```
cycle(2, [1,2,3]);  
//=> [1, 2, 3, 1, 2, 3]  
  
_.take(cycle(20, [1,2,3]), 11);  
//=> [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2]
```

我将创建的另一个递归函数叫作unzip，它是对Underscore的_.zip函数的逆。_.zip函数正如拉链一般，用于将两个数组中的值一一配对，如下所示：

```
_.zip(['a', 'b', 'c'], [1, 2, 3]);  
  
//=> [['a', 1], ['b', 2], ['c', 3]]
```

Underscore的_.zip接受两个数组，将第一个数组中的每个元素与对应的第二个数组中元素组成对。要“解压”那些由函数_.zip生成的数组，我认为需要考虑解压数组的“对（pairness）”。换句话说，如果我考虑这样的基本情况：一个数组需要解压缩，然后我可以开始考虑如何通过解构这个问题来解决它：

```
var zipped1 = [['a', 1]];
```

比zipped1更基础的是空数组[]，但是解压缩空数组的结果是数组[[], []]（这似乎是一个很好的终止条件，所以现在先把它记住）。zipped1是一个简单且有趣的案例，它是解压数组[['a'], [1]]的结果。所以在给出终止条件[[], []]和基本用例zipped的情况下，如何得到[['a'], [1]]？

其实就是从zipped1拿第一个元素并将其放入终止条件的第一个数组中，对于第二个元素也一样。我可以把这种操作抽象成函数constructPair：


```
function constructPair(pair, rests) {
    return [construct(_.first(pair), _.first(rests)),
           construct(second(pair), second(rests))];
}
```

为了提供一般的“unzippability”，单单constructPair是不够的，比如我可以得到一个解压版本的zipped1：

```
constructPair(['a', 1], [[],[ ]]);
//=> [['a'], [1]]
```

```
_.zip(['a'], [1]);
//=> [['a', 1]]
```

```
_.zip.apply(null, constructPair(['a', 1], [[],[ ]]));
//=> [['a', 1]]
```

同样，我可以使用constructPair逐步建立已解压版本的更大的压缩数组，如下所示：

```
constructPair(['a', 1],
  constructPair(['b', 2],
    constructPair(['c', 3], [[],[ ]])));

//=> [['a', 'b', 'c'], [1, 2, 3]]
```

该操作如图6-3所示。

因此，使用constructPair的知识，现在可以建立一个自递归函数unzip：

```
function unzip(pairs) {
    if (_.isEmpty(pairs)) return [[],[ ]];

    return constructPair(_.first(pairs), unzip(_.rest(pairs)));
}
```

在递归调用unzip会遍历所有的压缩组合，直到它到达一个空数组。然后，它遍历子数组，沿路利用constructPair构造一个解压缩的数组。实现unzip之后，我能够“撤销”调用_.zip的结果：

```
unzip(_.zip([1,2,3],[4,5,6]));  
//=> [[1,2,3],[4,5,6]]
```

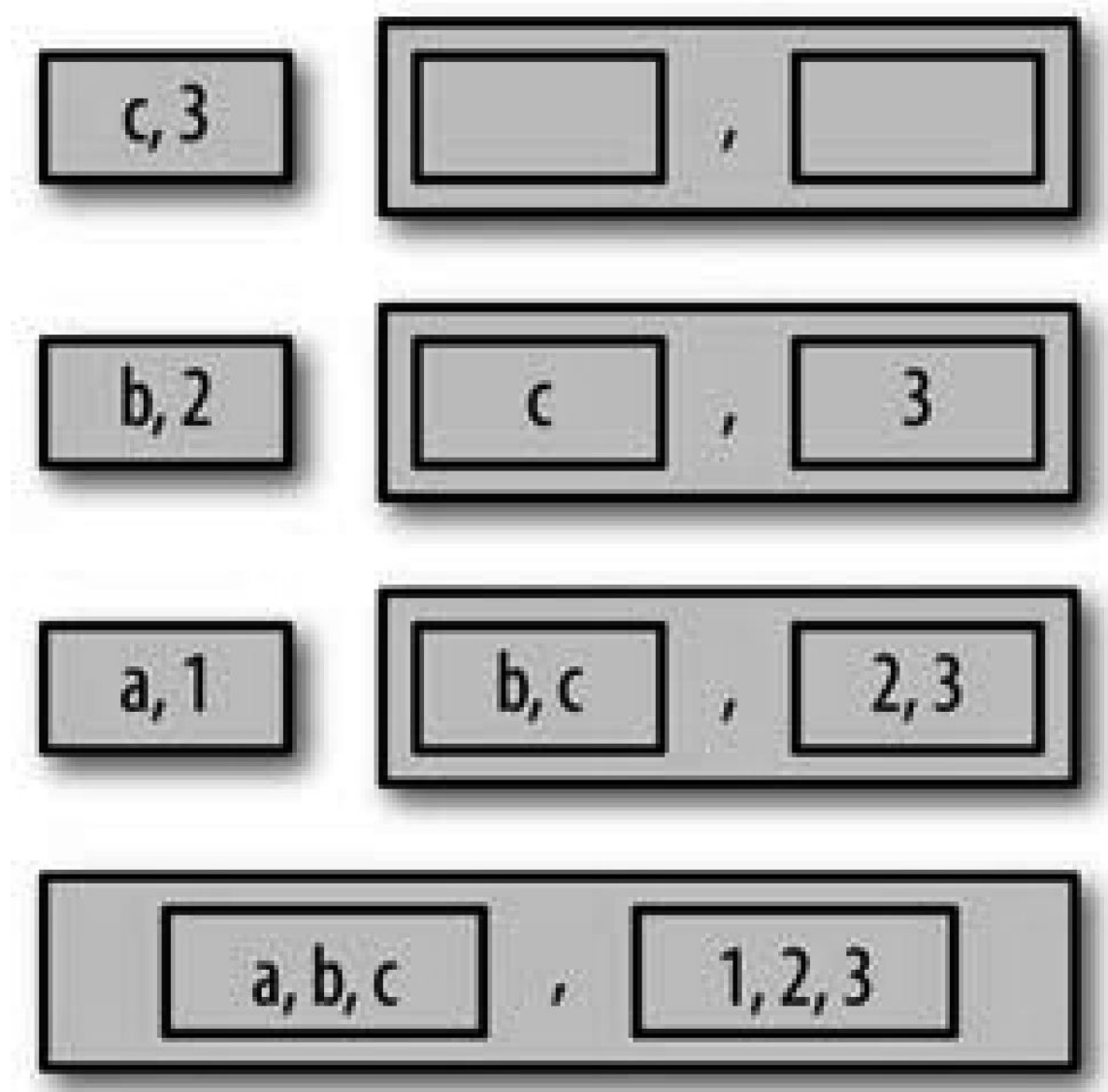


图6-3 `constructPair`操作的图形示意

`myLength` 的所有实例，如`cycle`，以及`unzip`是自递归（或者换句话说，调用自身的函数）的例子。编写自递归函数时，规则如下（Touretzky, 1990）。

- 知道什么时候停止。
- 决定怎样算一个步骤。
- 把问题分解成一个步骤和一个较小的问题。

表6-1列出了遵守这些规则如何运作的表格。

表6-1 自递归的规则

函 数	何时停止	进行一个步骤	小一些的问题
myLength	<code>_.isEmpty</code>	<code>1 + ...</code>	<code>_.rest</code>
cycle	<code>times <= 0</code>	<code>cat(array ...</code>	<code>times - 1</code>
unzip	<code>_.isEmpty</code>	<code>constructPair(_.first ...</code>	<code>_.rest</code>

这三个规则相当于编写自递归函数的模板。为了说明可以将此模板套用到任何复杂的自递归函数，我将运行一个更复杂的例子，并解释相似之处。

6.1.1 用递归遍历图

能优雅地用递归解决的问题之一是遍历数据结构图的每一个节点。很难再找到比用递归更优雅的方式解决在图内搜索路径的方法。图6-4的特别之处在于，它显示了间接或直接影响了JavaScript的编程语言。

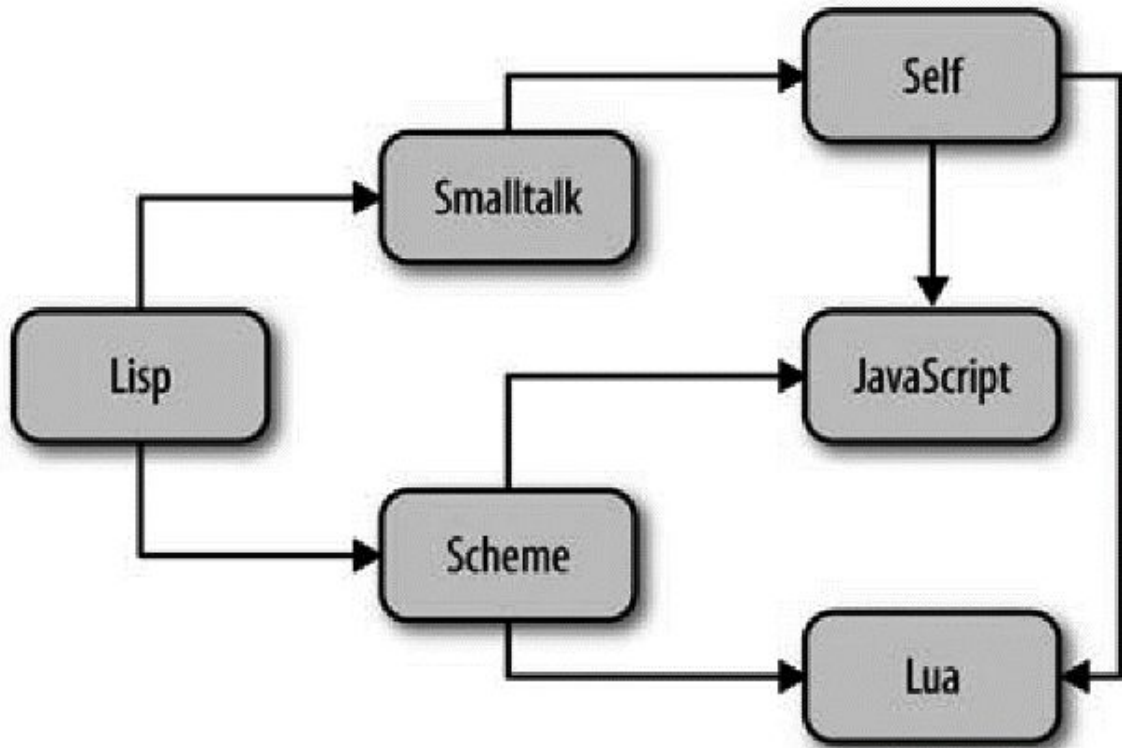


图6-4 编程语言影响

其实我也可以使用基于类或基于对象的角度来呈现，其中每一种语言和连接可以通过Node和Arc类型的对象来表示，但我更愿意先从简单的东西入手，比如字符串数组的数组：

```
var influences = [
  ['Lisp', 'Smalltalk'],
  ['Lisp', 'Scheme'],
  ['Smalltalk', 'Self'],
  ['Scheme', 'JavaScript'],
  ['Scheme', 'Lua'],
  ['Self', 'Lua'],
  ['Self', 'JavaScript']];
```

influences 中的每个嵌套数组表示“影响者”的连接，例如，第一个数组表示 Lisp语言 Smalltalk 的影响，最终形成如图 6-4 所示的图。递归函数 nexts，递归的定义如下（Paulson 1996）：

```

function nexts(graph, node) {
  if (_.isEmpty(graph)) return [];

  var pair = _.first(graph);
  var from = _.first(pair);
  var to    = second(pair);
  var more  = _.rest(graph);

  if (_.isEqual(node, from))
    return construct(to, nexts(more, node));
  else
    return nexts(more, node);
}

```

用函数nexts递归遍历，并建立由给定节点影响的编程语言的数组，如下所示：

```

nexts(influences, 'Lisp');
//=> ["Smalltalk", "Scheme"]

```

nexts里面的递归调用跟你到目前为止看到的完全不同；在if语句的两个分支都进行了递归调用。nexts中的“then”分支直接处理有关的目标节点，而else分支负责忽略不重要的节点。

表6-2 nexts自递归的规则

函 数	停止条件	一个步骤	缩小的问题
nexts	_.isEmpty	construct(...)	_.rest

让nexts能够遍历多个节点其实只需要再一点点努力，但我决定将此作为一个练习留给读者。我现在将介绍的图遍历的一个递归算法，称为深度优先搜索。

6.1.2 深度优先自递归搜索

在本节中，我会简单介绍一下图的搜索，并提供深度优先搜索的函数实现。在函数式编程，你会经常需要搜索数据结构中的数据块。在有层级关系的图（如influences）中，搜索的解决方案自然是递归。然而，要找到图中一个给定的节点，你（可能）需要访问图中的每一个节点，确定是否就是你要找的。一个节点遍历策略称为深度优先遍历，优先访问最左侧分支，然后再访问右侧分支上的节点（见图6-5）。

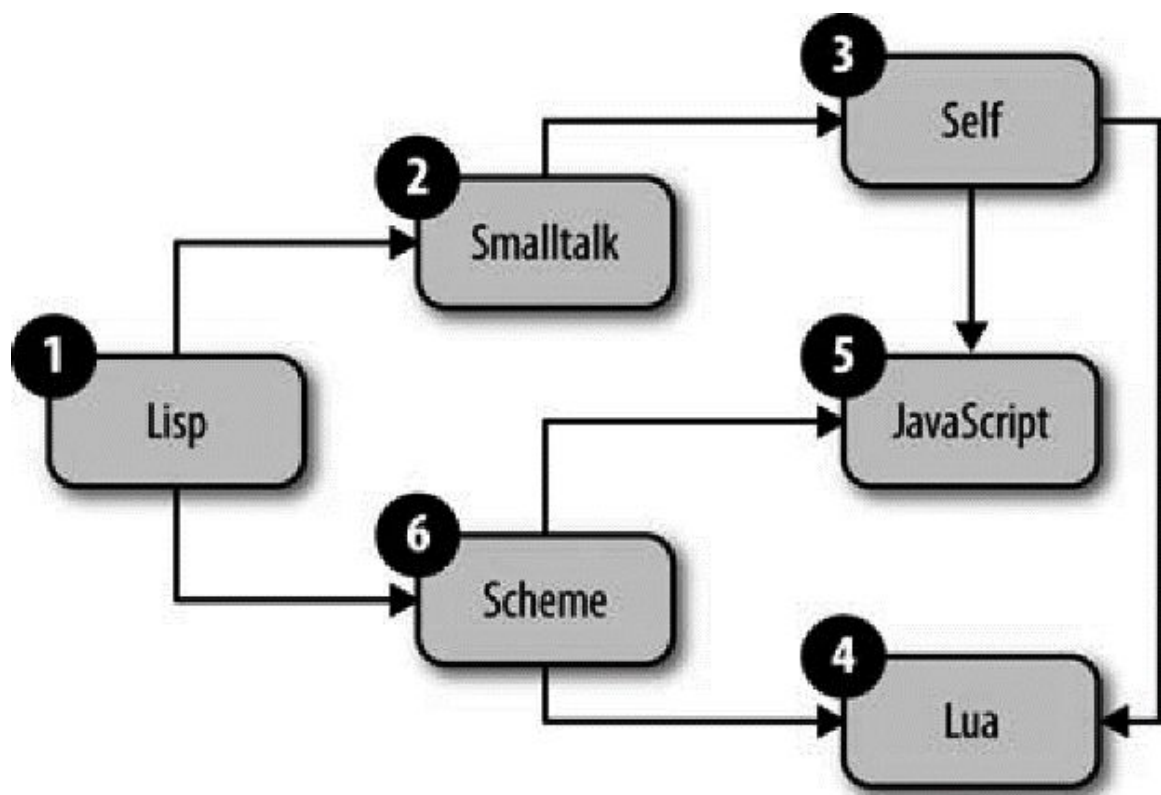


图6-5 深度优先遍历影响图

不同于以往的递归实现，函数depthSearch应保存遍历过的节点。原因是图中又可能会出现环，所以如果没有记录，会一直无线循环在环中搜索。然而，由于自递归调用可以通过参数与下个递归进行交互，遍历的记录可以从一个调用传给下一个调用，这个参数称为“累加器”。累加器参数是递归的常见技术，经常用于从一个递归调用到下一个的信息通信。使用累加器实现的depthSearch如下：

```

function depthSearch(graph, nodes, seen) {
  if (_.isEmpty(nodes)) return rev(seen);

  var node = _.first(nodes);
  var more = _.rest(nodes);

  if (_.contains(seen, node))
    return depthSearch(graph, more, seen);
  else
    return depthSearch(graph,
                        cat(nexts(graph, node), more),
                        construct(node, seen));
}

```

你可能已经发现，第三个参数seen就是用来存放已遍历节点的累加器。depthSearch的实际应用如下：

```

depthSearch(influences, ['Lisp'], []);
//=> ["Lisp", "Smalltalk", "Self", "Lua", "JavaScript", "Scheme"]

depthSearch(influences, ['Smalltalk', 'Self'], []);
//=> ["Smalltalk", "Self", "Lua", "JavaScript"]

depthSearch(construct(['Lua', 'Io'], influences), ['Lisp'], []);
//=> ["Lisp", "Smalltalk", "Self", "Lua", "Io", "JavaScript", "Scheme"]

```

你可能已经注意到，depthSearch函数实际上什么事情也没做。反而，它只是建立了按照深度顺序排列的节点的数组。稍后会用函数组合以及相互递归再次实现深度优先的策略。首先，请允许我花点时间谈“尾递归”。

尾（自）递归

虽然一般形式的depthSearch看起来与之前的非常相似，但是有一个区别很关键。为了突出我的意思，看看表6-3。

表6-3 depthSearch自递归的规则

函 数	停止条件	一个步骤	缩小的问题
nexts	<code>_.isEmpty</code>	<code>construct(...</code>	<code>_.rest</code>
<code>depthSearch</code>	<code>_.isEmpty</code>	<code>depthSearch(more...</code>	<code>depthSearch(cat...</code>

其中明显的区别是，“一个步骤”和“缩小的问题”中的元素都要进行`depthSearch`递归调用。当对任何这些元素进行递归调用时，该函数被称为尾递归。换句话说，就是函数（除了停止条件返回值）的最后一个动作是递归调用。由于`depthSearch`的最后一次调用是一个递归调用，所以函数体将永远不会再次使用。例如 Scheme 语言就是用这一方式来释放尾递归函数体中使用的资源。

`myLength`的尾递归实现如下所示：

```
function tcLength(ary, n) {
  var l = n ? n : 0;

  if (_.isEmpty(ary))
    return l;
  else
    return tcLength(_.rest(ary), l + 1);
}

tcLength(_.range(10));
//=> 10
```

而之前的递归调用`myLength (1+...)`会重复访问函数体来执行最终的加法。也许有一天，JavaScript引擎将优化尾递归函数来保持内存。但现在，事实是我们被诅咒在调用堆栈上深深的递归调用中^[2]。然而，正如你将在本章后面看到的，函数的尾部位置仍然是有趣的。

6.1.3 递归和组合函数：Conjoin和Disjoin

在这本书中，我已经实现了一些常用的函数组合功能：curry1，partial和compose，但我没有描述如何自己创建一个。幸运的是，几乎不太有这种需求，因为Underscore 已经实现了这些函数。然而，我将在本节中创建两个新的组合函数：orify和andify，并使用递归实现。

回想一下早在第4章实现的checker函数，它接收一些谓词函数并返回一个能判断是否每个参数对那些谓词都是真的函数。我可以实现一个递归版本的checker，叫作andify^[3]：

```
function andify(/* preds */) {  
  var preds = _.toArray(arguments);  
  
  return function(/* args */) {  
    var args = _.toArray(arguments);  
  
    var everything = function(ps, truth) {  
      if (_.isEmpty(ps))  
        return truth;  
      else  
        return _.every(args, _.first(ps))  
          && everything(_.rest(ps), truth);  
    };  
  
    return everything(preds, true);  
  };  
}
```

注意，andify返回的递归调用非常有意思。由于逻辑与运算符&&是“惰性的”，递归调用不会发生，因此_**every**测试会失败。这种类型的懒惰被称为“短路”，它在避免不必要的计算时很有用。值得注意的是，我用的是本地函数everything，来消耗在andify中的谓词函数。使用嵌套函数是隐藏递归调用累加器的常用方法。

观察andify的操作：

```
var evenNums = andify(_.isNumber, isEven);
```

```
evenNums(1,2);
```

```
//=> false
```

```
evenNums(2,4,6,8);
```

```
//=> true
```

```
evenNums(2,4,6,8,9);
```

```
//=> false
```

orify的实现几乎与andify一样，除了一些逻辑会反转^[4]。

```
function orify(/* preds */) {  
  var preds = _.toArray(arguments);  
  
  return function(/* args */) {  
    var args = _.toArray(arguments);  
  
    var something = function(ps, truth) {  
      if (_.isEmpty(ps))  
        return truth;  
      else  
        return _.some(args, _.first(ps))  
          || something(_.rest(ps), truth);  
    };  
  
    return something(preds, false);  
  };  
}
```

跟andify一样，`_.some`函数也不会成功，`orify`返回的函数也发生了短路。注意：

```
var zeroOrOdd = orify(isOdd, zero);
```

```
zeroOrOdd();
```

```
//=> false
```

```
zeroOrOdd(0,2,4,6);
```

```
//=> true
```

```
zeroOrOdd(2,4,6);
```

```
//=> false
```

自递归函数的讨论就到此为止，但我还没有用递归做任何事情。别着急，还有另一种方式来实现递归，它还有一个朗朗上口的名字：相互递归。

6.2 相互关联函数（函数调用其他会再调用回它的函数）

两个或多个函数相互调用被称为相互递归。下面来看两个非常简单的相互递归函数的例子，用谓词函数来检查偶数和奇数：

```
function evenSteven(n) {  
  if (n === 0)  
    return true;  
  else  
    return oddJohn(Math.abs(n) - 1);  
}  
  
function oddJohn(n) {  
  if (n === 0)  
    return false;  
  else  
    return evenSteven(Math.abs(n) - 1);  
}
```

相互递归调用来回反弹彼此之间递减某个绝对的值，直到一方或另一方达到零。这是一个相当优雅解决方式：

```
evenSteven(4);  
//=> true  
  
oddJohn(11);  
//=> true
```

如果你坚持严格使用高阶函数，那么你很可能会频繁地遇到互斥函数。举个例子，比如说`_.map`函数。把自身传入`_.map`其实就是一种间接相互递归的方式。下面来看看一个将数字展平的函数^[5]：

```
function flat(array) {
  if (_.isArray(array))
    return cat.apply(cat, _.map(array, flat));
  else
    return [array];
}
```

flat有点微妙，但问题是，为了展平一个嵌套数组，它为每一个嵌套元素都建立一个数组，并在递归返回时将它们都串联起来。

```
flat([[1,2],[3,4]]);
//=> [1, 2, 3, 4]
```

```
flat([[1,2],[3,4,[5,6,[[[7]]],8]]]);
//=> [1, 2, 3, 4, 5, 6, 7, 8]
```

这种使用相互递归的方式相当模糊，但是非常适合使用高阶函数。

6.2.1 使用递归深克隆

另一个例子，用递归来实现“深”克隆对象的函数是个不错的选择。Underscore有一个`_.clone`函数，但它只是“浅”克隆（只复制对象的第一级）：

```

var x = [{a: [1, 2, 3], b: 42}, {c: {d: []}}];

var y = _.clone(x);

y;
//=> [{a: [1, 2, 3], b: 42}, {c: {d: []}}];

x[1]['c']['d'] = 1000000;

y;
//=> [{a: [1, 2, 3], b: 42}, {c: {d: 1000000}}];

```

虽然在很多情况下，`_.clone`可能是有用的，但有些时候你真的想克隆一个对象及其所有子对象^[6]。递归是完成这个任务的完美方式，因为它让我们可以遍历一个对象上的所有子对象，在遍历的过程可以进行复制。`deepClone`的递归实现如下所示，可能鲁棒性还不足以使用在产品环境中：

```

function deepClone(obj) {
  if (!existy(obj) || !_.isObject(obj))
    return obj;

  var temp = new obj.constructor();
  for (var key in obj)
    if (obj.hasOwnProperty(key))
      temp[key] = deepClone(obj[key]);

  return temp;
}

```

当`deepClone`遇到一个原始类型，如数字，将会直接返回。然而，它遇到一个对象时，它会认为是一个关联结构，会递归地拷贝所有的键/值映射关系。我选择使用`obj.hasOwnProperty(key)`，以确保不复制原型上的字段。我倾向于使用对象作为关联的数据结构（如`map`），并避免将数据放置到原型上，除非必须这样做。`deepClone`的使用方式如下所示：

```
var x = [{a: [1, 2, 3], b: 42}, {c: {d: []}}];
```

```
var y = deepClone(x);
```

```
_.isEqual(x, y);  
//=> true
```

```
y[1]['c']['d'] = 42;
```

```
_.isEqual(x, y);  
//=> false
```

deepClone的实现是不是非常有趣，特别是JavaScript一切都是对象这一事实让该递归解决方案既紧凑又优雅。在下一节中，我将使用相互递归重新实现depthSearch，这次会真正做一些事情。

6.2.2 遍历嵌套数组

像deepClone一样遍历嵌套对象看起来不错，但是却不是必要的。然而，一个更为普遍的现象是需要遍历一个嵌套数组的数组。很多时候，你会看到下面这种模式：

```
doSomethingWithResult(_.map(someArray, someFun));
```

调用_.map的结果被传递到另一个函数，以便进一步处理。确保自身的抽象是很常见的，我把它叫做visit，在这里实现：

```
function visit(mapFun, resultFun, array) {  
  if (_.isArray(array))  
    return resultFun(_.map(array, mapFun));  
  else  
    return resultFun(array);  
}
```

`visit` 函数需要两个函数以及一个数组来处理。`mapFun` 对数组中的每个元素进行调用，将得到的数组传递到`resultFun`进行最后的处理。如果传入`array`的不是数组，那么我只需运行`resultFun`就可以了。用部分应用实现这样的函数是非常有用的，因为其中个别函数可以用部分应用来缩减`visit`中的多余操作。现在来观察如何使用`visit`：

```
visit(_.identity, _.isNumber, 42);  
//=> true
```

```
visit(_.isNumber, _.identity, [1, 2, null, 3]);  
//=> [true, true, false, true]
```

```
visit(function(n) { return n*2 }, rev, _.range(10));  
//=> [18, 16, 14, 12, 10, 8, 6, 4, 2, 0]
```

使用与`flat`同样的原理，可以使用`visit`来实现的相互递归版本的`depthSearch`，叫做`postDepth`^[7]。

```
function postDepth(fun, ary) {  
  return visit(partial1(postDepth, fun), fun, ary);  
}
```

叫做`postDepth`的原因是，深度优先遍历在扩展每个节点的子节点后进行`mapFun`。一个相关的函数`preDepth`，在扩展子节点前调用执行`mapFun`，其实现如下：

```
function preDepth(fun, ary) {  
  return visit(partial1(preDepth, fun), fun, fun(ary));  
}
```

前置深度优先搜索中有大量的`fun`，但原理非常简单，在移动到其他元素之前执行函数调用。让我们来看看`postDepth`如何工作：

```
postDepth(_.identity, influences);  
//=> [['Lisp', 'Smalltalk'], ['Lisp', 'Scheme'], ...]
```

将`_.identity`函数传到`*Depth`函数并返回`influences`数组的副本。函数`evenSteven`，`oddJohn`，`postDepth`和`visit`的执行策略本身就是一个类

图的模型，如图6-6所示。

如果我想大写Lisp的所有实例，有一个函数可以做到这一点：

```
postDepth(function(x) {  
  if (x === "Lisp")  
    return "LISP";  
  else  
    return x;  
}, influences);
```

```
//=> [['LISP', 'Smalltalk'], ['LISP', 'Scheme'], ...]
```

所以规则是，如果我想改变一个节点，然后我用它做点什么，并返回新的值；否则，我只是返回的节点。当然，原始数组不会被修改：

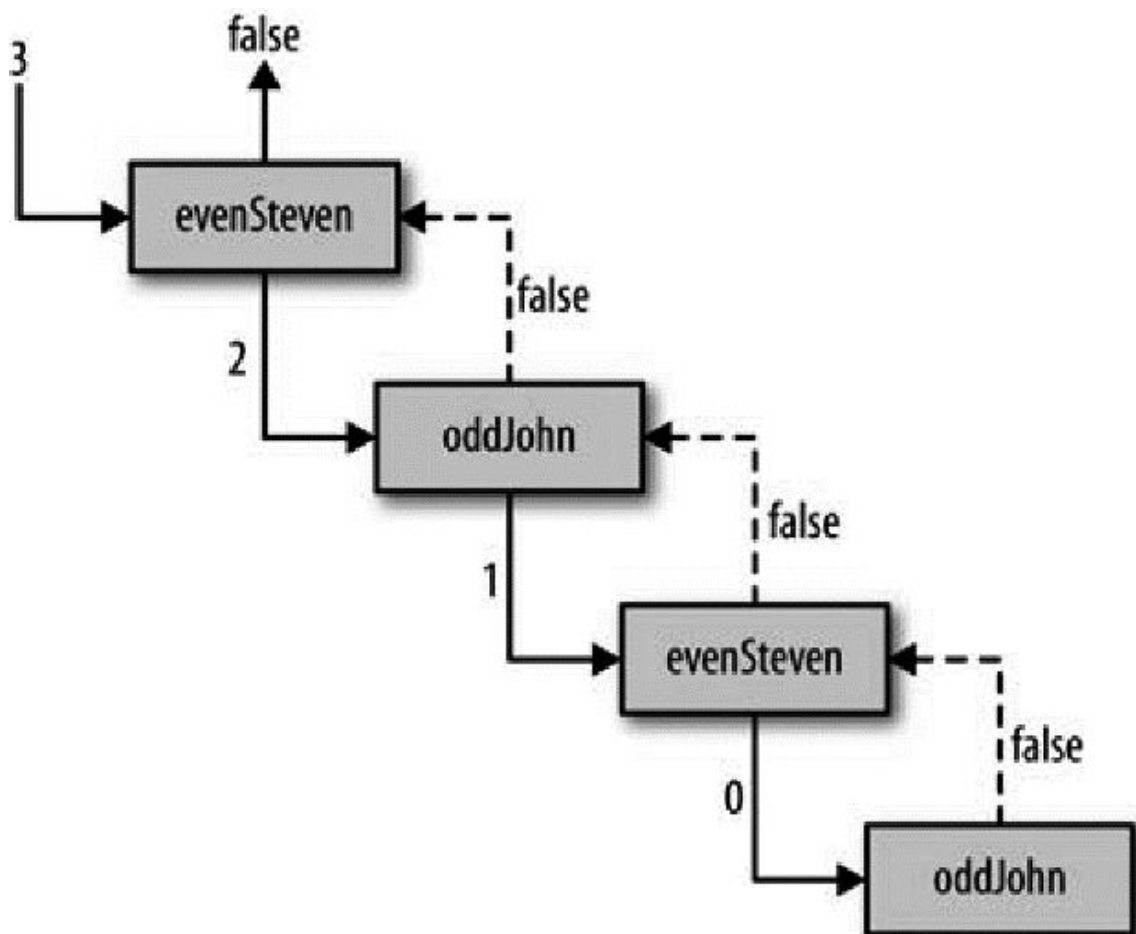


图6-6 相互递归函数以类似图的方式执行

```
influences;  
//=> [['Lisp', 'Smalltalk'], ['Lisp', 'Scheme'], ...
```

如果我想建立所有被影响的语言数组，我可以按照如下步骤来实现：

```
function influencedWithStrategy(strategy, lang, graph) {  
  var results = [];  
  
  strategy(function(x) {  
    if (_.isArray(x) && _.first(x) === lang)  
      results.push(second(x));  
  
    return x;  
  }, graph);  
  
  return results;  
}
```

influencedWithStrategy是采用深度优先搜索的函数之一。它遍历整个图，并沿途建立影响语言的数组：

```
influencedWithStrategy(postDepth, "Lisp", influences);  
//=> ["Smalltalk", "Scheme"]
```

虽然我通过改变一个数组来生成构建的结果，但是该操作被限制在influenced WithStrategy内部。

6.3 太多递归了

正如前面尾递归那节提到的，目前即使技术上是可行的，但JavaScript引擎没有优化递归调用。因此，在使用或写递归函数时，你可能会偶尔碰到下面的错误：

```
evenSteven(100000);  
// Too much recursion (or some variant)
```

这种错误（称为“blowing the stack”^[8]）是因为evenSteven和oddJohn互相调用，使每个函数都被调用数千次才使得其中一方达到零。因为大多数JavaScript实现对递归调用的次数有限制，这样的函数很容易“blow stack”（Zakas, 2010）。

在本节中，我将简单介绍一下所谓的蹦床（trampoline）原理的控制结构，这将有助于消除这些类型的错误。它的基本原理是，使用蹦床展平调用，而不是深度嵌套的递归调用。在继续之前，我们来看看如何手动修复evenSteven和oddJohn使得递归调用不会溢出。一个办法是返回一个函数，它包装调用，而不是直接调用。我可以使用partial1来实现这一点：

```
function evenOline(n) {  
  if (n === 0)  
    return true;  
  else  
    return partial1(oddOline, Math.abs(n) - 1);  
}  
  
function oddOline(n) {  
  if (n === 0)  
    return false;  
  else  
    return partial1(evenOline, Math.abs(n) - 1);  
}
```

我们返回一个包装函数而不是直接进行evenOline和oddOline的互相调用。调用两个函数的终止情况都正常工作：

```
evenOline(0);  
//=> true
```

```
oddOline(0);  
//=> false
```

现在，我可以手动试试用相互递归来展平数组：

```
oddOline(3);  
//=> function () { return evenOline(Math.abs(n) - 1) }
```

```
oddOline(3)();  
//=> function () { return oddOline(Math.abs(n) - 1) }
```

```
oddOline(3)();()  
//=> function () { return evenOline(Math.abs(n) - 1) }
```

```
oddOline(3)();()();  
//=> true
```

```
oddOline(2000000001)();()(); //... a bunch more ()s  
//=> true
```

我想可以向用户暴露这个API了。但你可能想提供另一个函数trampoline，从程序执行来进行扁平化处理：

```

function trampoline(fun /*, args */) {
  var result = fun.apply(fun, _.rest(arguments));

  while (_.isFunction(result)) {
    result = result();
  }

  return result;
}

```

trampoline所做的是不断调用函数的返回值，直到它不再是一个函数。你可以看看它具体是如何工作的：

```

trampoline(oddOline, 3);
//=> true

trampoline(evenOline, 200000);
//=> true

trampoline(oddOline, 300000);
//=> false

trampoline(evenOline, 2000000000);
// wait a few seconds
//=> true

```

由于调用链的间接性，使用蹦床增加了相互递归函数的一些开销。然而，慢总比溢出好。同样，你可能不希望强迫用户使用trampoline，只是为了避免堆栈溢出。其实也可以隐藏其外观：

```

function isEvenSafe(n) {
  if (n === 0)
    return true;
  else
    return trampoline(partial1(oddOline, Math.abs(n) - 1));
}

function isOddSafe(n) {
  if (n === 0)
    return false;
  else
    return trampoline(partial1(evenOline, Math.abs(n) - 1));
}

```

试试这些函数是否正常工作：

```

isOddSafe(2000001);
//=>true

```

```

isEvenSafe(2000001);
//=> false

```

6.3.1 生成器

我将用几个无限的例子结束本节。使用递归，我可以演示如何构建和操作无限的“惰性”数据流，并同样相互调用函数，直到太阳燃尽。我说的惰性的意思是直到需要的时候才会计算。下面来看看前面定义的cycle函数：

```

_.take(cycle(20, [1,2,3]), 11);
//=> [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2]

```

在这个调用中，cycle创建的数组不是惰性的，因为它在传递给_.take之前就已经构造好了。尽管_.take只需要11个元素，但是却传入了全部60个元素。这是非常低效的，但Underscore 以及 JavaScript 本身却默认这样干。

然而，对数组应该把它看成是由第一个元素以及其余部分组成的。事实上，Underscore也提供了`_.first`和`_.rest`方法来支持这种看法。一个无限数组同样可以被看作第一个元素“first”以及其余部分“tail”。然而，与有限的阵列不同的是，无限阵列的尾部可能不存在。把头部和尾部放到一个对象里可能有助于理解这个概念（Houser，2013）：

```
{head: aValue, tail: ???}
```

问题出现了：`tail`存放的是什么？答案很简单，跟`odd0line`一样，只需要一个可以计算尾部的尾部函数。而且这个函数应该是一个递归函数。

这个头尾部对象需有两个函数：一个用来计算当前元素值的函数和另一个用来计算下一元素“种子”值的函数。实际上，这种结构正是所谓的生成器的概念，或者说是一个按照需求产生后续值的函数。理解了这一点，那么来看看generator的实现^[9]。

```
function generator(seed, current, step) {  
  return {  
    head: current(seed),  
    tail: function() {  
      console.log("forced");  
      return generator(step(seed), current, step);  
    }  
  };  
}
```

如上所示，`current`参数是计算头部位位置的值的函数，`step`用于将值传给递归调用。`tail`的关键在于，它是包裹在一个`function`中，只有调用时才会是值。还可以实现一些操作生成器的函数：

```
function genHead(gen) { return gen.head }  
function genTail(gen) { return gen.tail() }
```

`genHead` 和 `genTail` 函数会返回头部和尾部。然而，尾部会被“强制”执行。这里先让我创建一个生成器：

```
var ints = generator(0, _.identity, function(n) { return n+1 }));
```

使用generator函数，可以定义整数范围。现在，使用访问头尾部的函数，我就可以开始采摘走头部：

```
genHead(ints);  
//=> 0
```

```
genTail(ints);  
// (console) forced  
//=> {head: 1, tail: function}
```

调用genHead不会取ints的尾部，但调用genTail时会去取。嵌套的调用genTail会迫使生成器深度生成响应深度的数字：

```
genTail(genTail(ints));  
// (console) forced  
// (console) forced  
//=> {head: 2, tail: function}
```

只需用这两个函数，我就建立一个更强大的存取函数genTake，可以随着调用生成相应的前n项：

```
function genTake(n, gen) {  
  var doTake = function(x, g, ret) {  
    if (x === 0)  
      return ret;  
    else  
      return partial(doTake, x-1, genTail(g), cat(ret, genHead(g)));  
  };  
  
  return trampoline(doTake, n, gen, []);  
}
```

如上所示，genTake是利用蹦床实现的。终于，这种有可能引起溢出的调用似乎有用了。下面来看看如何使用genTake：


```
genTake(10, ints);  
// (console) forced x 10  
//=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
  
genTake(100, ints);  
// (console) forced x 100  
//=> [0, 1, 2, 3, 4, 5, 6, ..., 98, 99]  
  
genTake(1000, ints);  
// (console) forced x 1000  
//=> Array[1000]  
  
genTake(10000, ints);  
// (console) forced x 10000  
// wait a second  
//=> Array[10000]  
  
genTake(100000, ints);  
// (console) forced x 100000  
// wait a minute  
//=> Array[100000]  
  
genTake(1000000, ints);  
// (console) forced x 1000000  
// wait an hour  
//=> Array[1000000]
```

利用“蹦床原理”可以定义一个无限大的结构，按照需求计算而不会堆栈溢出。但是创建生成器generator有一个巨大的缺陷：尾部元素在被访问之前都不会被计算，也意味着每次访问都需要再重新计算一次：

```
genTake(10, ints);  
// (console) forced x 10  
//=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

比如genTake计算前10项，如果不再跑一次，其实看起来也不错，然而创建一个全面的生成器已经超出本书的范围^[10]。

当然，天下没有免费的午餐，使用蹦床时也是如此。我会尽可能使用堆，而不是栈。因为堆比调用栈大得多，所以你使用蹦床时不太可能碰到内存耗尽的问题。

如果不直接使用蹦床，一般的“蹦床化（trampolineness）”的思想在JavaScript中其实一文不值。

6.3.2 蹦床原理以及回调

对于异步JavaScript API如setTimeout或XMLHttpRequest库（jQuery的\$.ajax），相关的递归的讨论是十分有趣的。异步库工作在事件循环上是非阻塞的。也就是说，如果你使用异步API来安排一个可能需要很长的时间的任务，那么浏览器或运行时不会阻塞等待它完成。相反，每个异步API需要一个任务完成时可以调用的“回调”（就是函数或闭包）。这可以让你（有效）执行并发任务，其中一些需要长时间运行的操作，而不会阻塞你的应用程序的运行^[11]。

非阻塞API的一个有趣的特点是，调用在回调执行之前立即返回。而这些回调会在不太遥远的未来被执行^[12]。

```
setTimeout(function() { console.log("hi") }, 2000);  
//=> returns some value right away  
// ... about 2 seconds later  
// hi
```

立即返回真正有趣的是，JavaScript每一次时钟会清理事件循环上的调用堆栈。这样异步回调总是会在一次新的时钟到来时执行！注意观察以下代码：

```

function asyncGetAny(interval, urls, onsuccess, onfailure) {
    var n = urls.length;

    var looper = function(i) {
        setTimeout(function() {
            if (i >= n) {
                onfailure("failed");
                return;
            }

            $.get(urls[i], onsuccess)
                .always(function() { console.log("try: " + urls[i]) })
                .fail(function() {
                    looper(i + 1);
                });
        }, interval);
    }

    looper(0);
    return "go";
}

```

你会发现，在调用jQuery的异步调用\$.get函数失败时会递归调用looper。这个调用（原则上）跟其他相互递归调用没有什么不同，唯一不一样的是在于每次调用发生堆栈上的调用栈都被清空。下面来看看完整的例子asyncGetAny^[13]：

```

var urls = ['http://dsfgfgs.com', 'http://sghjgsj.biz', '_html', 'foo.txt'];

asyncGetAny(2000,

```

```
    urls,
    function(data) { alert("Got some data") },
    function(data) { console.log("all failed") });
//=> "go"

// (console after 2 seconds) try: http://dsfgfgs.com
// (console after 2 seconds) try: http://sghjgsj.biz
// (console after 2 seconds) try: _.html

// an alert box pops up with 'Got some data' (on my computer)
```

有一些比本书更好地描述JavaScript异步编程的资源，但我觉得提到事件循环和递归的独特性能就足够了。在实践中，使用事件循环可以使JavaScript应用程序获得最大的效率。

6.4 递归是一个底层操作

本章主要介绍如何创建、使用和理解递归。尽管递归很有用，但我还是要提醒一句：递归应该被看作一个底层操作，应该尽可能地避免。更好的途径是利用现有的高阶函数来创造新的函数。例如我实现的 `influencedWithStrategy`，虽然方式很巧妙，但却是完全不必要的。我应该知道用已有的函数可以混合出同样的效果。首先，我可以创建两个辅助函数：

```
var groupFrom = curry2(_.groupBy)(_.first);  
var groupTo   = curry2(_.groupBy)(second);
```

因为我使用的是一个简单的嵌套数组来表示我的图关系，创造新的函数来进行操作跟重用已有的数值函数似乎没什么区别。看看 `groupFrom` 和 `groupTo` 的效果：

```
groupFrom(influences);  
//=> {Lisp:[["Lisp", "Smalltalk"], ["Lisp", "Scheme"]],  
//     Smalltalk:[["Smalltalk", "Self"]],  
//     Scheme:[["Scheme", "JavaScript"], ["Scheme", "Lua"]],  
//     Self:[["Self", "Lua"], ["Self", "JavaScript"]]}  
  
groupTo(influences);  
//=> {Smalltalk:[["Lisp", "Smalltalk"]],  
//     Scheme:[["Lisp", "Scheme"]],  
//     Self:[["Smalltalk", "Self"]],  
//     JavaScript:[["Scheme", "JavaScript"], ["Self", "JavaScript"]],  
//     Lua:[["Scheme", "Lua"], ["Self", "Lua"]]}  

```

这些函数比较好玩，但它们还不够。再用函数 `influenced` 来找到图中的环：

```
function influenced(graph, node) {  
  return _.map(groupFrom(graph)[node], second);  
}
```

这就是与递归 `influencedWithStrategy` 同样功能的函数：

```
influencedWithStrategy(preDepth, 'Lisp', influences);  
//=> ["Smalltalk", "Scheme"]
```

```
influenced(influences, 'Lisp');  
//=>["Smalltalk", "Scheme"]
```

这样influences的实现不仅减少了代码量，而且概念也是非常容易理解的。我已经知道`_.groupBy`、`_.first`、`second`和`_.map`的用法，所以要了解influenced是了解数据在函数间是如何转换的。这就是函数式编程的一大优势所在，如同乐高积木，数据流在函数管道中不断转换成最终需要的数据形式。

这才是美丽的编程。

6.5 总结

本章涉及递归，也就是函数直接或间接地调用自己。自调用函数是搜索以及处理嵌套数据结构的强大工具。对于搜索，我通过使用 `visit` 函数遍历树的例子，介绍了深度优先搜索函数。

虽然树的遍历是一个强大的技术，但是却遇到JavaScript对递归调用的基本限制。然而，使用一种称为蹦床的技巧，展示了如何通过闭包数组间接地互相调用。

最后，我觉得有必要退后一步，谨慎使用递归。很多时候，递归函数没有组合高阶函数那么直接。普遍的共识是，首先使用函数组合，仅当需要时才使用递归和蹦床。

在下一章中，我将介绍函数式编程的一个常见话题——突变（`mutation`），也就是改变变量的值，以及如何限制及避免修改变量。

[1] 你可能会想：“为什么不在字段中直接使用`length`呢？”而这种务实的想法对构建好的系统很重要，但不会对递归的学习有所帮助。

[2] ECMAScript6提议尾递归调用优化，我们只能是默默期待了。
http://wiki.ecmascript.org/doku.php?id=harmony:proper_tail_calls。

[3] 用Underscore的`every`方法也能使`andify`短路，知道怎么做到的吗？

[4] 你知道吗，`orify`函数也可以用Underscore的某函数实现。

[5] 更好的方式是用Underscore的`flatten`方法。

[6] 我使用了JavaScript圈里常用的术语“克隆”。在其他原型语言（例如，`Self`或`IO`）中，克隆操作会委托给原来的对象，直到做了变更出现，再由此克隆出副本。

[7] 该`JSON.parse`方法接收一个可选的“`reviver`”函数，而且行为与`postDepth`类似。也就是说，解析完成会传入`reviver`函数，`reviver`的返

回值会变成JSON.parse的返回值。Reviver的用法很多，但也许最常见的是将Date转成字符串。

[8] 译者注：这里是双关，blowing stack本意是大发雷霆，这里指栈溢出。

[9] 调用console.log仅用于演示目的。

[10] ECMAScript.next正试着设计生成器。更多信息可以参考ECMAScript的网站。

[11] 需要注意的是，你的应用程序可能还是会因为各种原因阻塞。但如果正确使用，JavaScript的事件架构将帮助你避免这些阻塞。

[12] 也可能会是下周末。

[13] 我使用了jQuery的promise接口always与fail，使得GET请求的接口更流利。因为并发执行的原因，不能保证console打印会在得到GET结果之前还是之后打印出来。我会在第8章再解释jQuery的promises。

第7章

纯度、不变性和更改政策

本章要探索完全函数式以及实用风格。函数式编程不仅仅只关心函数，也是思考如何尽量降低软件复杂性的一种方式。而降低复杂性的一种方法是减少甚至消除程序中的状态变化。

7.1 纯度

如果你需要一个大于0且小于某个数的（伪）随机数，完全可以使用Underscore的`_.random`函数，不过其默认为包括零。我们可以简单地创建一个随机数函数`rand`：

```
var rand = partial1(_.random, 1);
```

使用`rand`很简单，只要执行以下操作：

```
rand(10);  
//=> 7
```

```
repeatedly(10, partial1(rand, 10));  
//=> [2, 6, 6, 7, 7, 4, 4, 10, 8, 5]
```

```
_.take(repeatedly(100, partial1(rand, 10)), 5);  
//=> [9, 6, 6, 4, 6]
```

还可以利用`rand`生成一定长度的随机小写ASCII字符串与数字，如下所示：

```
function randString(len) {  
  var ascii = repeatedly(len, partial1(rand, 26));  
  return _.map(ascii, function(n) {  
    return n.toString(36);  
  }).join('');  
}
```

使用`randString`的方法如下：

```
randString(0);  
//=> ""  
  
randString(1);  
//=> "f"  
  
randString(10);  
//=> "k52k7bae8p"
```

构建randString函数的方式已经贯穿于整本书的过程中。用函数来构造更高层次函数是我们一贯的作风。虽然randString技术上符合这个定义，但是randString与之前的函数有一点大不相同。究竟是什么？来先回答另外一个问题：如何测试 randString？

7.1.1 纯度和测试之间的关系

你会如何测试函数 randString ？也就是说，如果你正在使用类似Jasmine^[1]为randString写一个测试，你将如何完成下面的代码片段？

```
describe("randString", function() {  
  it("builds a string of lowercase ASCII letters/digits", function() {  
    expect(randString()).toBe???(?);  
  });  
});
```

标记???中到底应该放什么才能使得测试通过？你可以尝试给定一个字符串，但是这将是浪费时间，因为结果是随机的。现在randString的问题已经很明显是没有办法预测调用的结果。这就跟函数_.map每次调用的结果都是由参数确定的情况完全不一样：

```
describe("_map", function() {  
  it("should return an array made from...", function(){  
    expect(_.map([1,2,3], sqr)).toEqual([1, 4, 9]);  
  });  
});
```

```
});  
{  
    expect(_.map([1,2,3], sqr)).toEqual([1, 4, 9]);  
});  
});
```

`_.map`函数的操作可以用“纯（pure）”来描述。纯函数坚持以下属性。

- 其结果只能从它的参数的值来计算。
- 不能依赖于能被外部操作改变的数据。
- 不能改变外部状态。

以`randString`为例，它就违反了纯度的第一条规则，因为它的计算没有使用任何参数。同时，也违反了第二条规则，因为它的结果是完全依赖于JavaScript的随机数生成器，这相当于从一个黑箱子中拿出完全跟输入参数无关的值。这是语言级别上而不是随机生成器的问题。也就是说，你可以通过依赖调用者提供的“种子”值，创建一个纯粹的随机数生成器。

打破第一条规则的另一个例子：

```
PI = 3.14;
```

```
function areaOfACircle(radius) {  
    return PI * sqr(radius);  
}
```

```
areaOfACircle(3);  
//=> 28.26
```

你可能已经看到了问题所在，为了完整起见，我们假定在网页中另一个库中载入了下面的代码片段：

```
// ... some code
```

```
PI = "Magnum";
```

```
// ... more code
```

调用areaOfACircle的结果是什么？

```
areaOfACircle(3);
```

```
//=> NaN
```

这种问题是因为在运行时可以加载任意代码，这样对象或者变量很容易被更改。因此，编写依赖于它的控制范围之外的数据的函数容易造成混乱。通常情况下，当你尝试测试依赖于外部条件的函数，测试用例也必须运行在相同的条件下。坚持纯度的标准不仅将有助于使程序更容易测试，也更容易推理。

7.1.2 提取纯函数

因为JavaScript的Math.rand方法从设计上讲都是不纯的，任何使用它的函数都将变成非纯函数，因此难以测试。纯函数可以通过建立一组输入值和输出的数据进行测试。JavaScript中还有一些会导致不纯的函数或方法如Date.now, console.log, this和全局变量。事实上，因为JavaScript可以传递对象的引用，所以每一个接受对象或者数组的函数都有可能不纯。我将在本节介绍如何缓解这些问题，但重点是，虽然JavaScript是不可能完全纯净的（可能我们也不希望这样），但我们可以将变化的影响降到最低限度。

randString函数无疑是不纯的，但是有办法分离出纯的部分。如何划分randString还是比较清楚的：一个字符生成的部分，以及拼装的部分。要分离纯的部分很简单，只要创建两个函数：

```

function generateRandomCharacter() {
  return rand(26).toString(36);
}

function generateString(charGen, len) {
  return repeatedly(len, charGen).join('');
}

```

变更generateString的实现（让它显式地接收一个字符生成函数），就可以按如下模式使用：

```

generateString(generateRandomCharacter, 20);
//=> "2lfhj045n2nfnpb7m7e"

```

此外，由于generateString是一个高阶函数，我可以使用partial来组合原本不纯的randomString：

```

var composedRandomString = partial1(generateString, generateRandomCharacter);

composedRandomString(10);
//=> "j18obj1jc"

```

现在所有纯的部分已经封装在函数内，可以单独对它进行测试：

```

describe("generateString", function() {
  var result = generateString(always("a"), 10);

  it("should return a string of a specific length", function() {
    expect(result.constructor).toBe(String);
    expect(result.length).toBe(10);
  });

  it("should return a string congruent with its char generator", function() {
    expect(result).toEqual("aaaaaaaaaa");
  });
});

```

要测试不纯的generateRandomCharacter函数还是有问题，但是generateString已经变得通用，易测试。

7.1.3 测试不纯函数的属性

如果一个函数是不纯的，它的返回值是受其外部条件的影响的，那么如何进行测试呢？假设你能成功地缩减不纯的部分到最低限度，像 `generateRandomCharacter` 那样，那么测试会变得比较容易。虽然你不能测试特定用例的返回值，但你可以测试它的某些特性。在 `generateRandomCharacter` 这个例子里，我可以测试以下特性。

- ASCII。
- 数字。
- 字符串。
- 字符。
- 小写。

然而，测试所有这些特性需要大量的数据：

```
describe("generateRandomCharacter", function() {
  var result = repeatedly(10000, generateRandomCharacter);

  it("should return only strings of length 1", function() {
    expect(_.every(result, _.isString)).toBeTruthy();
    expect(_.every(result, function(s) { return s.length === 1 })).toBeTruthy();
  });

  it("should return a string of only lowercase ASCII letters or digits", function() {
    {
      expect(_.every(result, function(s) {
        return /[a-z0-9]/.test(s) }))).toBeTruthy();

      expect(_.any(result, function(s) { return /[A-Z]/.test(s) })).toBeFalsy();
    }
  });
});
```

只测试 `generateRandomCharacter` 的 10000 个结果离完整的测试覆盖率是远远不够的。你可以增加迭代次数，但永远不会完全满意。同样，还好知道生成的字符会落在一定范围内。事实上，我的实现限制了生成所有合法小写ASCII字符，那么我们到底在测什么？我一直在测试错误的解决方案。解决创建错误东西的问题是一个哲学的问题，而且远远超出了本书的深度。对于随机密码生成而言，这可能是一个问题，但对于展现不纯的代码片段的分离和检测的目的，我的实现应该足够了。

7.1.4 纯度与引用透明度的关系

使用纯函数编程可能看起来限制特别多。JavaScript作为一个高度动态的语言，允许用户定义和使用不需要严格遵守参数或返回值类型的函数定义。有时，这种松散标准导致了不确定性（例如`true + 1 === 2`），但你也可以利用这一点，获得更灵活的优势。很多时候，JavaScript程序员将允许随意形态的变量，对象和数组插槽看作动态性的精髓部分。然而，当你实际上随意使用状态突变，其实会限制组合的能力，并且使得推理变得更复杂；当你随意改变状态，实际上局限了组合的可能性，并使得代码难以梳理，导致测试难以进行。

而使用纯函数，则使得函数组合更容易，甚至可以随意替换具有同等功能的函数，甚至期望值。举个例子，我们使用`nth`函数来定义第1章的`second`函数：

```
function second(a) {  
    return nth(a, 1);  
}
```

这里`nth`函数是一个纯函数。也就是说，它的唯一依赖就是数组参数。它对给定的数组和索引的返回总是相同的：

```
nth(['a', 'b', 'c'], 1);  
//=> 'b'
```

```
nth(['a', 'b', 'c'], 1);  
// 'b'
```

你可以运行这个调用10亿次，只要`nth`接收数组`['a', 'b', 'c']`和数字1，它总是返回字符串`'b'`。同样，`nth`函数永远不会修改传给它的数组：


```
var a = ['a', 'b', 'c'];
```

```
nth(a, 1);  
//=> 'b'
```

```
a === a;  
//=> true
```

```
nth(a, 1);  
//=> 'b'
```

```
_.isEqual(a, ['a', 'b', 'c']);  
//=> true
```

但是 JavaScript 有一个我们不得不接受的限制因素，就是nth函数可能会返回对象、数组，甚至是不纯的函数：

```
nth([ {a: 1}, {b: 2}], 0);  
//=> {a: 1}
```

```
nth([function() { console.log('blah') }], 0);  
//=> function ...
```

解决这个问题的唯一方法是严格遵守不修改这一点，同时也不依赖于外部值，将影响减至最低。当认识到必须保持纯洁的函数的重要性时，你会得到更多的编程选项。例如，我可以替换nth函数，而保持功能完全一样[\[2\]](#)。

```
function second(a) {  
  return a[1];  
}
```

或者

```
function second(a) {  
  return _.first(_.first(a));  
}
```

在这两种情况下，second的行为都没有任何改变。因为nth是一个纯函数，这样替换实在太轻松了。实际上，由于nth是纯函数，可以很轻松地更换它的返回值而仍然能保持程序的一致性：

```
function second() {  
  return 'b';  
}  
  
second(['a', 'b', 'c'], 1);  
//=> 'b'
```

这种能随意更换新函数而不需要关心随之带来的混淆显示出程序组合的灵活性。接下来我将介绍有关纯粹性与引用透明的话题。

7.1.5 纯度和幂等性

随着RESTful风格API的日益盛行，幂等的想法最近得到了广泛认同。幂等性是指执行无数次后还具有相同的效果。幂等性在函数式编程中与纯度相关，但不尽相同。形式上，一个幂等函数可以做到：

```
someFun(arg) == _.compose(someFun, someFun)(arg);
```

换句话说，对同一的参数运行一次函数应该与连续两次运行是一个结果，相当于someFun(someFun(arg))。例如前面提到的second函数，就不是幂等的：

```
var a = [1, [10, 20, 30], 3];  
  
var secondTwice = _.compose(second, second);  
  
second(a) === secondTwice(a);  
//=> false
```

当然调用一次second时返回数组[10, 20, 30]，因此secondTwice返回嵌套值20。最简单的幂函数可能是Underscore的`_.identity`函数：

```
var dissociativeIdentity = _.compose(_.identity, _.identity);  
  
_.identity(42) === dissociativeIdentity(42);  
//=> true
```

JavaScript的`Math.abs`方法也是幂等的：

```
Math.abs(Math.abs(-42));  
//=> 42
```

其实也不必为了坚持追求纯函数而牺牲了动态性。但是，请记住，任何时候显式地改变一个变量时，不管它是在一个容器对象（在本章后面）或封装在一个闭包，甚至直接修改，你都引入了一个时间敏感的状态。也就是说，在程序执行的任何给定的时刻，程序的状态都取决于微妙变化之间的相互作用。即使你可能无法消除程序中的所有状态的变换，也应该尽量减少。我会在本章后面部分深入如何分离变化，但首先，与函数纯度相关是不变性的想法，或者说缺乏明确的状态变化。

7.2 不变性

JavaScript中很少有默认不可变的数据类型。而字符串是为数不多的不可变数据类型的例子：

```
var s = "Lemongrab";
```

```
s.toUpperCase();
```

```
//=> "LEMONGRAB"
```

```
s;
```

```
//=> "Lemongrab"
```

字符串不可变其实是一件好事，因为很可能会出现类似下面这种情况[3]。

```
var key = "lemongrab";
```

```
var obj = {lemongrab: "Earl"};
```

```
obj[key] === "Earl";
```

```
//=> true
```

```
doSomethingThatMutatesStrings(key);
```

```
obj[key];
```

```
//=> undefined
```

```
obj["lemonjon"];
```

```
//=> "Earl"
```

这是一个不可预测的事件。平常可能难以遇到这样的问题，但如果字符串突变成为一种普遍现象，那么这类问题将会频繁出现。值得庆幸的

是，在JavaScript中，字符串是不可变的，因此不会出现这样讨厌的问题。但是，下面这种突变在JavaScript 中是允许的^[4]。

```
var obj = {lemongrab: "Earl"};

(function(o) {
  _.extend(o, {lemongrab: "King"});
})(obj);

obj;
//=> {lemongrab: "King"}
```

虽然庆幸字符串不可变，但是JavaScript对象是可变的。事实上，大部分的JavaScript都借此可变性的优势。然而，随着JavaScript在行业中逐渐流行，使用它编写的程序也越来越大。想象一下，如图7-1所示的小程序中的突变依赖关系。

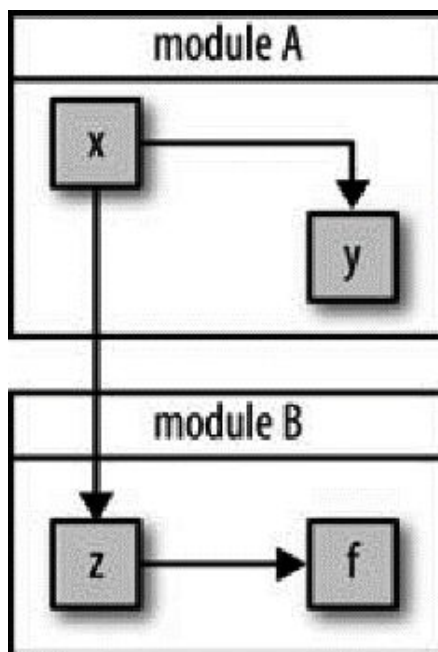


图7-1 即便是可管理的小程序中，“突变的网络”也很复杂

然而，由于该程序的增长，“突变网络”随之增加，每个节点都会依赖于更多节点，如图7-2所示。

这些状态不容易维护。如果每次变化都会通过突变网络广泛扩散，那么任何更改都会影响整体^[5]。在函数式编程中，理想的情况是没有突变。如果你一开始就遵守不变性原则，你会惊奇地发现你能走得很远。在本节中，我将讨论不变性的优点，以及如何遵守它的规范。

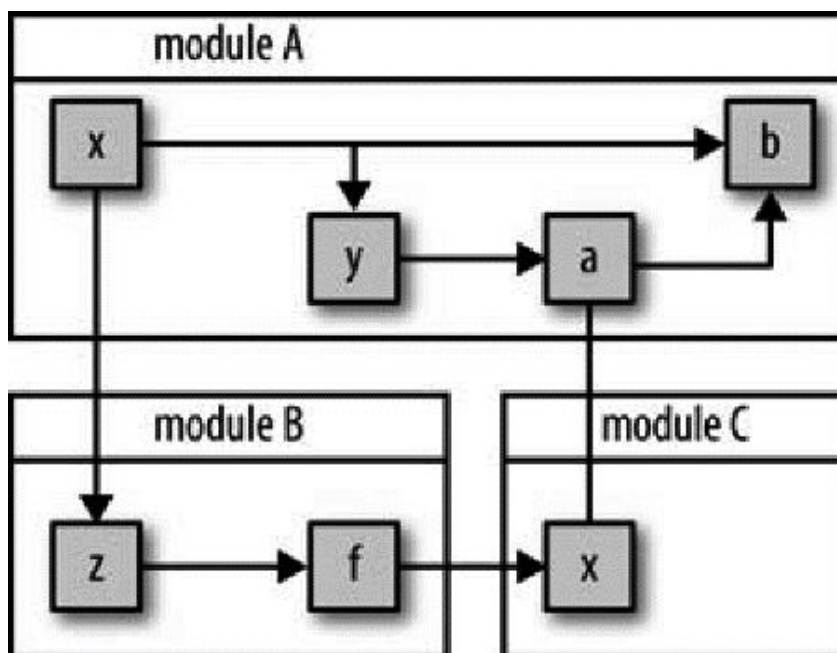


图7-2 随着程序的扩大，“突变网络”也在蔓延

7.2.1 如果一棵树倒在树林里，有没有声音

通过本书你会发现，我经常在函数的实现中使用可变数组和对象。为了说明我的意思，请注意看函数skipTake的实现，当给定一个数n和一个数组，返回一个包含每第n个元素的数组：

```

function skipTake(n, coll) {
  var ret = [];
  var sz = _.size(coll);

  for(var index = 0; index < sz; index += n) {
    ret.push(coll[index]);
  }

  return ret;
}

```

使用skipTake的方法如下：

```

skipTake(2, [1,2,3,4]);
//=> [1, 3]

skipTake(3, _.range(20));
//=> [0, 3, 6, 9, 12, 15, 18]

```

在skipTake的实现中，我很刻意地用数组再加上一个命令式的循环进行Array#push。有很多函数式的技术可以用来实现skipTake，可以避免显式的突变。然而，用for循环简单，快捷。更重要的是，这种方式被完全封装在skipTake函数中而不会对用户暴露。把函数看成抽象的基本单元的优点是，任何给定的函数范围内，实现细节都是无关紧要的，只要它们不“泄漏（leak out）”。“泄露”的意思是，你可以使用一个函数作为状态突变的边界，完全隔绝于外部代码的变化。

不管我用_.foldRight还是while来实现skipTake，这对用户都是无关紧要的。他们所要知道或关心的只是获得一个新的数组，而且他们传入的数组不会被改变。

如果一棵树倒在树林里，有没有声音？

如果一个纯函数改变了一些局部数据来产生一个不可变的返回值，这样做可以吗？

——Rich Hickey <http://clojure.org/transients>

事实证明，答案是肯定的^[6]。

7.2.2 不变性与递归

如果你读过跟我一样多的关于函数式编程的书籍，那么一个有趣的现象就出现了。几乎所有的书都会覆盖递归和递归技术的话题。出现这种情况的原因有很多，但其中一个重要的原因是涉及纯度。在许多函数式编程语言中，你很难只使用局部变量实现函数`summ`：

```
function summ(array) {  
    var result = 0;  
    var sz = array.length;  
  
    for (var i = 0; i < sz; i++)  
        result += array[i];  
  
    return result;  
}  
  
summ(_.range(1,11));  
//=> 55
```

该函数`summ`要改变两个局部变量：`i`和`result`。然而，在传统的函数语言中，局部变量实际上不是变量，是不可变的。唯一修改局部变量值的方法是通过调用堆栈去改变它，而这正是递归。下面是一个递归实现。


```
function summRec(array, seed) {  
  if (_.isEmpty(array))  
    return seed;  
  else  
    return summRec(_.rest(array), _.first(array) + seed);  
}  
  
summRec([], 0);  
//=> 0  
  
summRec(_.range(1,11), 0);  
//=> 55
```

当使用递归，状态是通过函数参数管理，而变化是通过从一个递归调用的参数传给下一个[\[7\]](#)。JavaScript允许这种递归的状态管理，但是有第6章中提到的递归深度限制。那么，为什么不使用更省时间的方式？我将在下一节讨论，改变局部状态的注意事项。

7.2.3 冻结和克隆

由于JavaScript按引用传递数组和对象，因此没有什么是不可变的。同样，JavaScript对象的字段总是可见的，没有什么简单的方法使它们不可变（2005戈茨）。不过确实有办法封装、隐藏数据，以避免意外的变化。但在最顶层，所有的JavaScript对象是可变的，除非冻结它们。

JavaScript的最新版本提供了一种方法，当对一个对象或数组调用Object#freeze，将导致所有后续的突变失败。在严格模式下使用的情况下，失败将会抛出一个TypeError；否则，所有突变都会悄悄地失败。freeze方法的工作原理如下：

```
var a = [1, 2, 3];
```

```
a[1] = 42;
```

```
a;  
//=> [1, 42, 3]
```

```
Object.freeze(a);
```

一个正常的数组是默认可变的，但调用`Object#freeze`之后，发生以下情况：

```
a[1] = 108;
```

```
a;  
//=> [1, 42, 3]
```

也就是说，突变将不再生效。你还可以使用`Object#isFrozen`方法来检查`a`是否确实冻结：

```
Object.isFrozen(a);  
//=> true
```

使用`Object#freeze`来确保不变性容易有两个问题。

- 除非你能完全控制代码库，否则可能会导致微妙的（和不那么微妙的）错误。
- `Object#freeze`是浅操作。

关于对象的冻结，虽然这可能是实践不可变性的一个好主意，但不是所有的库都支持这样做。冻结对象，并通过随意传给其他API可能会引起麻烦。然而，更深层次的问题是，`Object#freeze`是一个浅操作。也就是说，冻结只会发生在最顶层，不会遍历嵌套对象。注意：

```
var x = [{a: [1, 2, 3], b: 42}, {c: {d: []}}];
```

```
Object.freeze(x);
```

```
x[0] = "";
```

```
x;
```

```
//=> [{a: [1, 2, 3], b: 42}, {c: {d: []}}];
```

试图改变数组第一个元素会失败。然而，改变a内的嵌套结构却成功了：

```
x[1]['c']['d'] = 100000;
```

```
x;
```

```
//=> [{a: [1, 2, 3], b: 42}, {c: {d: 100000}}];
```

如果想要深度冻结一个对象，我需要使用递归遍历数据结构，很像第6章的deepClone：

```
function deepFreeze(obj) {  
  if (!Object.isFrozen(obj))  
    Object.freeze(obj);  
  
  for (var key in obj) {  
    if (!obj.hasOwnProperty(key) || !_.isObject(obj[key]))  
      continue;  
  
    deepFreeze(obj[key]);  
  }  
}
```

deepFreeze函数与你所期望的一样：

```

var x = [{a: [1, 2, 3], b: 42}, {c: {d: []}}];

deepFreeze(x);

x[0] = null;

x;
//=> [{a: [1, 2, 3], b: 42}, {c: {d: []}}];

x[1]['c']['d'] = 42;

x;
//=> [{a: [1, 2, 3], b: 42}, {c: {d: []}}];

```

不过，正如我之前提到的，随意冻结对象可能会在与第三方的API进行交互时引入微妙的错误。因此，你只有以下三个选项。

- 使用`_.clone`，如果你知道一个浅拷贝就足够了。
- 使用`deepClone`拷贝整套副本。
- 将你的代码建立在纯函数之上。

在这本书中，我选择了第三个选项，但你将会在第8章看到，我需要使用`deepClone`，以确保函数纯度。现在，让我们来探讨函数式和以对象为中心的API保持不变性的想法。

7.2.4 在函数级别上观察不变性

根据一些规则以及下列技术，你可以创建不可变的对象和纯函数。

这本书中许多实现的函数，包括`Unscore`，都有一个共同的特点：它们接收集合并从它建立另一个集合。举个例子，一个函数`freq`，接收数字或字符串数组，并返回以其元素作为键，出现次数作为值的对象：

```

var freq = curry2(_.countBy)(_.identity);

```

因为我知道，函数`_.countBy`是一种非破坏性的操作（不改变输入数组），那么它与`_.identity`的组合应该形成一个纯函数。注意：

```
var a = repeatedly(1000, partial1(rand, 3));
var copy = _.clone(a);

freq(a);
//=> {1: 498, 2: 502}
```

正如计数掷硬币正反面结果几乎是50/50一样。同样有趣的是，操作`freq`不会改变原数组`a`：

```
_.isEqual(a, copy);
//=> true
```

注意观察纯函数实现有助于消除函数组合出来新的行为正确与否的担忧。如果你组合纯函数，结果也是纯函数。

因为我实现的`skipTake`也是纯函数，即使它在内部使用了可变的结构，它也可以安全地用于组合：

```
freq(skipTake(2, a));
//=> {1: 236, 2: 264}

_.isEqual(a, copy);
//=> true
```

然而，有时有一些函数不是那么配合，它们会改变对象的内容，从而带来不纯的因素。例如，`_.extend`函数会合并左边的对象到右边，从而产生一个新的对象，如：

```
var person = {fname: "Simon"};

_.extend(person, {lname: "Petrikov"}, {age: 28}, {age: 108});
//=> {age: 108, fname: "Simon", lname: "Petrikov"}
```

当然，问题是，`_.extend`改变的是参数列表中的第一个对象：

```
person;  
//=> {age: 108, fname: "Simon", lname: "Petrikov"}
```

所以`_.extend`的出现关闭了函数组合的大门，对不对？嗯，还没有。函数式编程的优点在于，只需稍稍修改就可以创建新的抽象。也就是说，比起使用“`extend`”，也许“`merge`”会更合适：

```
function merge(/*args*/) {  
  return _.extend.apply(null, construct({}, arguments));  
}
```

比起使用第一个参数作为目标对象，我们把一个局部的空对象传给`_.extend`并改变它的值。结果大不相同，但可能是你想要的：

```
var person = {fname: "Simon"};  
  
merge(person, {lname: "Petrikov"}, {age: 28}, {age: 108})  
//=> {age: 108, fname: "Simon", lname: "Petrikov"}  
  
person;  
//=> {fname: "Simon"};
```

现在`merge`函数可以与其他纯函数完全安全地组合了，它隐藏了可变性。从调用者的角度看，什么也没有被改变。

7.2.5 观察对象的不变性

对于JavaScript的内置类型和对象，你很难做到普遍的不变性，除非普遍使用冻结或严格使用规范。事实上，在自己使用JavaScript对象时，规范变得更加引人注目。为了说明问题，我将定义一个`Point`对象的片段，其定义如下：

```
function Point(x, y) {  
  this._x = x;  
  this._y = y;  
}
```

我大概可以借助各种封闭封装技巧^[8]以隐瞒Point实例没有公开访问的字段的事实。不过，我喜欢用一个更简单的方法来定义与标记“私有”字段（Bolin, 2010）。

我很快就会展示，一个API是如何只提供操作接口而不暴露实现细节的。然而，就目前而言，我将实现两个“change”的方法：withX 和 withY，而且我会坚持不变性的政策^[9]。

```
Point.prototype = {
  withX: function(val) {
    return new Point(val, this._y);
  },
  withY: function(val) {
    return new Point(this._x, val);
  }
};
```

对于Point的原型，我添加了两个方法作为“modifier”，这两种操作都不会造成任何的改变。无论是withX还是withY都返回Point新实例。下面是withX的实例：

```
var p = new Point(0, 1);

p.withX(1000);
//=> {_x: 1000, _y: 1}
```

调用withX返回Point对象带有_x字段设置为1000的实例，但是什么被改变了呢？什么都没有：

```
p;
//=> {_x: 0, _y: 1}
```

原本的p实例是与最初创建时一样的[0, 1]。事实上，设计不可变对象应该把自己的值在构造的时候就固定而之后再也不能改变。此外，不可变对象的所有操作应该返回新的实例。作为解决突变的问题的奖励，这样做还可以得到一个不错的链式API：

```
(new Point(0, 1))  
  .withX(100)  
  .withY(-100);  
  
//=> {_x: 100, _y: -100}
```

所以需要记住的点有。

- 不可变对象应该在构造时固定它们的值而之后再不能修改。
- 不可变对象操作并返回新对象^[10]。

即使只看这两个规则，你可能也会发现问题。例如，当一个Queue类型实现时指定一个数值，并提供（局部的）队列的访问逻辑^[11]：

```
function Queue(elems) {  
  this._q = elems;  
}  
  
Queue.prototype = {  
  enqueue: function(thing) {  
    return new Queue(this._q + thing);  
  }  
};
```

如同Point，Queue对象构造时得到它的种子值。此外，Queue提供了一个enqueue方法，用于添加作为种子的新元素到新的实例。Queue的使用过程如下：

```
var seed = [1, 2, 3];  
  
var q = new Queue(seed);  
  
q;  
//=> {_q: [1, 2, 3]}
```


在构造时，q实例接收到三个元素的数组作为它的种子数据。调用enqueue方法会返回一个新实例：

```
var q2 = q.enqueue(108);  
//=> {_q: [1, 2, 3, 108]}
```

事实上，q的值似乎是正确的：

```
q;  
//=> {_q: [1, 2, 3]}
```

但实际却不然：

```
seed.push(10000);
```

```
q;  
//=> {_q: [1, 2, 3, 1000]}
```

好吧，修改原始的seed改变了Queue的实例。问题是，我在构造时直接使用引用而不是建立克隆。这一次，我将实现一个新的对象SaferQueue，这将避免这一缺陷：

```
var SaferQueue = function(elems) {  
  this._q = _.clone(elems);  
}
```

deepClone可能不是必要的，因为Queue实例的目的是为了元素的添加和删除，而不是一个数据结构。其实，最好还是能在元素级别保持不变性，如新的enqueue的做法：

```
SaferQueue.prototype = {  
  enqueue: function(thing) {  
    return new SaferQueue(cat(this._q, [thing]));  
  }  
};
```

使用安全不变的cat功能将消除SaferQueue实例之间共享引用的问题：

```
var seed = [1,2,3];  
var q = new SaferQueue(seed);
```

```
var q2 = q.enqueue(36);  
//=> {_q: [1, 2, 3, 36]}
```

```
seed.push(1000);
```

```
q;  
//=> {_q: [1, 2, 3]}
```

我不敢说一切都是安全的。正如前面所提到的，每个q实例都有一个公共字段_q，我可以很容易地直接修改它们：

```
q._q.push(-1111);
```

```
q;  
//=> {_q: [1, 2, 3, -1111]}
```

同样，我可以很容易地替换SaferQueue.prototype中的方法来做我想做的事情：

```
SaferQueue.prototype.enqueue = sqr;
```

```
q2.enqueue(42);  
//=> 1764
```

JavaScript只能提供这么多的安全了，因此，责任在于我们要坚持编程规范，以确保我们的编程实践是尽可能安全的^[12]。

7.2.6 对象往往是一个低级别的操作

在继续讲如何控制突变之前，我还有最后一点需要强调：用不用new来创建对象都是允许的，但是会出现这种情况：

```
var q = SaferQueue([1,2,3]);
```

```
q.enqueue(32);
```

```
// TypeError: Cannot call method 'enqueue' of undefined
```

呀！我忘了new了。当然还有其它方法来避免这种问题，要么允许或不允许使用new构造的对象，我发现这些解决方案不是很有用。而我更喜欢使用构造函数，如下所示：

```
function queue() {  
    return new SaferQueue(_.toArray(arguments));  
}
```

因此，每当我需要一个队列，都可以使用构造函数构造：

```
var q = queue(1,2,3);
```

此外，我可以用invoker函数来创建一个函数代理给enqueue：

```
var enqueue = invoker('enqueue', SaferQueue.prototype.enqueue);
```

```
enqueue(q, 42);
```

```
//=> {_q: [1, 2, 3, 42]}
```

使用函数而不是裸的方法调用给了我很大的灵活性，包括但不限于以下内容。

- 我不需要过分担心实际的类型。
- 我可以返回适合特定用例的类型。例如，小的数组在建模小 map时相当快，但作为map增长，用一个对象可能更合适。这种切换可以根据程序中的使用出现。
- 如果类型或方法改变，那么我只需要更改函数，而不是每一个使用的地方。
- 我可以对函数添加前置和后置条件。
- 该函数可组合。

以这种方式使用的函数并不是告别面向对象编程（事实上，它们是互补的）。相反，它推动你进入实现细节领域。这使你 and 用户都工作在函数式的抽象，同时允许实施者集中于改变底层机制而不破坏现有代码。

7.3 控制变化的政策

让我们面对现实。虽然能消除代码中所有不必要的突变和副作用是极好的，但是也有些时候，你需要改变一些状态。我的目标是帮助你思考如何减少这种变化的足迹。例如，假设有这样一个程序，一个对象被改变，它所有引起的变化如图7-3所示。

图7-3看起来很眼熟，因为我在本章的前面讲到了它。当你将一个可变对象传来传去，并在某些函数中修改它时，你已经将这些修改提升到了对整个程序的影响。如果你添加一个接收特定值的函数，会发生什么？如果你把一个会造成微妙突变的函数删掉，会发生什么？如果你通过JavaScript的异步操作引入并发性会发生什么？所有这些因素都将颠覆你在以后做变更的能力。然而，如果改变只发生在一个单一的点，如图7-4所示，会是什么样子？

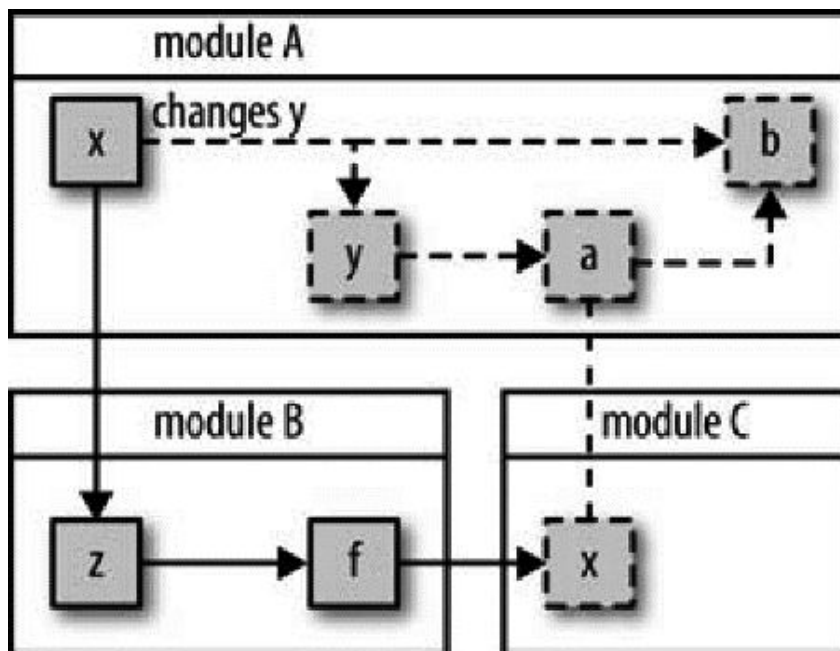


图7-3 突变的网络，任何潜在的变化对全局的影响

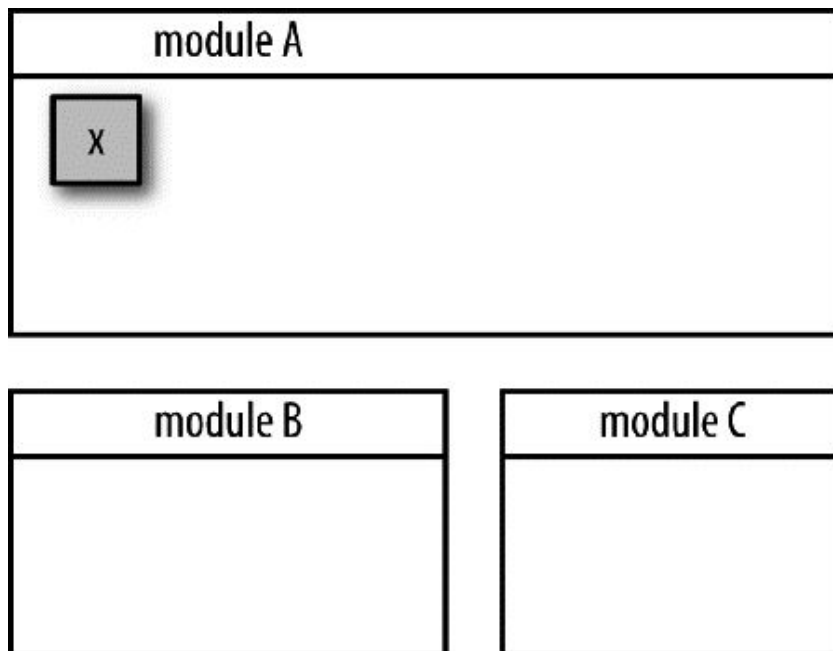


图7-4 如果你绝对需要管理状态，那么理想的情况是将它隔离到一个地方

本节是关于隔离变化到单个点并实现需要保持状态的折中策略。控制变化的范围的方法是隔离可变的東西。也就是说，比起随便拿到一个对象并改变它，更好的策略是把对象放入容器中，并更改该容器，即：

```
var container = contain({name: "Lemonjon"});
```

```
container.set({name: "Lemongrab"});
```

对比：

```
var being = {name: "Lemonjon"};
```

```
being.name = "Lemongrab";
```

这个简单的间接层可以让你更容易找到其中一个给定值的变化，其他倒也没有什么优点了。不过，我可以采取这种思路并更进一步，限制变化为函数调用的结果：

```
var container = contain({name: "Lemonjon"});
```

```
container.update(merge, {name: "Lemongrab"});
```

这个思想背后的概念是双重的。首先，比起用虚构的container#set方法来替代直接变更值，改成函数结果的方式可以得到容器以及一些参数。其次，通过增加这一函数式的中间层，可以改变任何函数，即使是在那些特定域。与此相反，想想对于确认随处突变的值是有多么困难，特别是如果你的程序中有多处突变时。

我可以展示一个容器类型的简单实现：

```
function Container(init) {  
  this._value = init;  
};
```

使用Container方法如下：

```
var aNumber = new Container(42);  
  
aNumber;  
//=> {_value: 42}
```

然而，还有更多的需要实现，如 update 方法：

```
Container.prototype = {  
  update: function(fun /*, args */) {  
    var args = _.rest(arguments);  
    var oldValue = this._value;  
  
    this._value = fun.apply(this, construct(oldValue, args));  
  
    return this._value;  
  }  
};
```

Container#update方法很简单：接收一个函数和一些参数，然后设置基于旧值调用的函数结果为新值。下面来观察具体如何操作：

```
var aNumber = new Container(42);
```

```
aNumber.update(function(n) { return n + 1 });  
//=> 43
```

```
aNumber;  
//=> {_value: 43}
```

一个采用多个参数的例子：

```
aNumber.update(function(n, x, y, z) { return n / x / y / z }, 1, 2, 3);  
//=> 7.166666666666667
```

一个使用限制函数的例子：

```
aNumber.update(_.compose(megaCheckedSqr, always(0)));  
// Error: arg must not be zero
```

这仅仅是个开始。事实上，在第9章我将使用“基于协议扩展”的思想扩展Container的实现。然而，就目前而言，减少突变的种子已经播下。

7.4 总结

本章深入介绍了函数纯度，可以概括为一个不改变、返回或依赖任何超出本身控制范围之外的变量的函数。虽然我花了很多时间谈论不改变参数的函数，我还提到，如果你需要改变一个内部变量，只要没有人知道你已改变了变量，就没有什么关系。

本章后面的部分谈到不变性的相关话题。不可变数据在JavaScript中是不现实的，因为变量可变是默认的。然而，通过观察程序中的变化模式，你可以得到尽可能接近的不可变性。同样，如果你的调用者不知道，是不会有影响的。察觉不变性和纯度，不仅在你的程序会很大的时候，而且在单元测试的级别也可以帮助你。如果你能清楚地推理和隔离函数，那么你就可以更容易地推出组合的函数。

在下一章中，我将介绍函数式“管道”的概念。这与函数组合`_.compose`关系紧密，但是更深入到抽象以及安全性，还是值得单独拿出来作为一章。

[1] Jasmine是一个非常好用的测试框架，我个人使用，并强烈推荐。

[2] 这是不完全正确的，因为`nth`检查数组边界，当索引超过数组的长度时抛出一个错误。当改变底层实现，做到心中有数，保证不偏离原始行为。

[3] Ruby编程语言允许字符串突变，之前的1.9版本就是这个陷阱的牺牲品。然而，Ruby 1.9的哈希对象复制字符串键，因此屏蔽了突变。不幸的是，它仍然允许可变对象作为键，因此这些突变能够破坏哈希查找。

[4] 我被Underscore的`extend`函数愚弄过一次，但实际上是我自己认为它是一个纯函数。当我得知不是纯函数时，我意识到一个有趣的方式来利用这一事实的优势，你将会在第9章中看到。

[5] 这并不是说所有的JavaScript程序都是这样。在过去的几年里，出现了越来越注重规范的设计。特别值得注意的沿着这个主题的一篇文章是“不要修改不属于你的对象”（Zakas 2010）。

[6] 函数可以作为隐藏突变边界，但它不是唯一的方式。我将在第8章中展示，有可以隐藏突变的更大的边界。对象一直以来都是隐藏突变很好的边界，甚至是库和系统都利用对象加快突变速度并保持很好的函数式接口。

[7] 递归函数的状态变化与第3章中介绍的用栈动态改变值的方式很像。

[8] 我用这个方法在第8章实现了createPerson。

[9] 请注意，我省略了constructor属性，如Point.prototype = {constructor: Point, ...}。虽然在这个例子中没有严格要求，但最好坚持在产品代码中保持这样的最佳实践。

[10] 有一些方法可以从一个实例中创建并共享元素到另一个，以避免复制较大的结构。然而，这种方法超出了本书的范围。

[11] 同样，我特意省略了constructor，以避免弄乱例子。

[12] 还有，由于JavaScript使用习惯和规范不一致，越来越多的JavaScript.next语言发展起来。

第8章 基于流的编程

本章将继续讨论函数式风格，展示如何将函数结合纯度以及隔离突变，组合成相当流畅的编程风格。随后讨论如何将函数式块结合在一起的想法，并展示相关的例子。

8.1 链接

如果你还记得，在第5章 的condition1中，我使用了下面的代码：

```
// ...
    var errors = mapcat(function(isValid) {
        return isValid(arg) ? [] : [isValid.message];
    }, validators);
// ...
```

这样做的原因是，最终的结果必须是一个错误字符串的数组，每个中间阶段返回子错误信息数组或是空数组。另一个原因是，我想结合不同的行为，返回不同的类型。如果返回值的格式刚好可作为另一个函数的参数，将更容易编写这些行为。例如下面这段代码：

```
function createPerson() {
    var firstName = "";
    var lastName = "";
    var age = 0;

    return {
        setFirstName: function(fn) {
            firstName = fn;
            return this;
        },
```

```

    setLastName: function(ln) {
        lastName = ln;
        return this;
    },
    setAge: function(a) {
        age = a;
        return this;
    },
    toString: function() {
        return [firstName, lastName, age].join(' ');
    }
};
}

```

```

createPerson()
    .setFirstName("Mike")
    .setLastName("Fogus")
    .setAge(108)
    .toString();

```

//=> "Mike Fogus 108"

可以链接方法的“魔法”在于，每个链接的方法都返回同样的宿主对象引用（Stefanov, 2010）。方法链返回共通的值的方式实际上是JavaScript中一个常见的设计模式，如jQuery和Underscore。事实上，Underscore有3个有用的函数：`_.tap`，`_.chain`和`_.value`。在第2章中，我用这些函数实现了建立“99瓶啤酒”歌词生成器的函数`lyricSegment`。然而，在实现中我掩盖了这些函数式是如何工作的。

`_.chain`是这三个函数中最酷的，它允许你指定一个对象作为一个隐式的目标，可以重复使用Underscore的函数对其操作。从`_.chain`的简单例子开始介绍可能比较好理解：

```
_.chain(library)
  .pluck('title')
  .sort();
```

```
//=> _
```

什么^[1]？

返回Underscore对象好像完全可以解释得通。_.chain函数接收对象，并把它封装在一个包含所有的修改过的Underscore函数。也就是说，像_.pluck这样的函数有着类似function pluck(array, propertyName)的默认调用签名，而函数_.chain使用的包装对象调用的是function pluck(propertyName)。Underscore使用了很多这种有趣的小伎俩，这样做的结果是，通过从上一个函数传入下一个的是包装对象而不是目标对象本身。因此，当想要结束_.chain并提取最终值时，需要使用_.value函数：

```
_.chain(library)
  .pluck('title')
  .sort()
  .value();
```

```
//=> ["Joy of Clojure", "SICP", "SICP"]
```

使用_.value可以将包装对象里的值取出来。这一概念还会在未来几节中出现。当使用_.chain函数时，你可能会收到各种错误结果。想象一下以下情形：

```
var TITLE_KEY = 'titel';

// ... a whole bunch of code later

_.chain(library)
  .pluck(TITLE_KEY)
  .sort()
  .value();

//=> [undefined, undefined, undefined]
```

因为代码紧凑，问题很明显，我拼错了“title”。然而，在一个庞大的代码库中，你很可能需要调试并慢慢靠近故障点。不幸的是，当调用`_.chain`时，貌似没有简单的方法可以接入（tap into）并检查中间值。然而事实并非如此。实际上，Underscore提供了`_.tap`函数，给定一个对象和一个函数，调用该函数仍然返回给定对象：

```
_.tap([1,2,3], note);
;; NOTE: 1,2,3
//=> [1, 2, 3]
```

传入`note`函数^[2]到Underscore的`tap`函数表明，`note`确实被调用了，而且能返回该数组。你可能怀疑，`_.chain`包装的对象同样可以使用`_.tap`，因此可以用来检查中间值，例如

```
_.chain(library)
  .tap(function(o) {console.log(o)})
  .pluck(TITLE_KEY)
  .sort()
```

```
.value();
```

```
// [{title: "SICP" ...  
//=> [undefined, undefined, undefined]
```

library表的内容似乎没有什么不对劲，但如果我把tap放到别的位置会怎么样？

```
_.chain(library)  
  .pluck(TITLE_KEY)  
  .tap(note)  
  .sort()  
  .value();
```

```
// NOTE: ,,  
//=> [undefined, undefined, undefined]
```

现在，可以看出pluck的结果数组看起来很奇怪。此时，tap已经发现了问题的所在：_.pluck调用。是TITLE_KEY或者_.pluck本身有问题。值得庆幸的是，问题处在我所控制的代码中。

_.chain是非常强大的，尤其是当你想流利地描述对单一目标发生的一系列动作时。然而，_chain也有一个局限，它不是惰性的。可以从下列代码中看出我的意思：

```
_.chain(library)  
  .pluck('title')  
  .tap(note)  
  .sort();
```

```
// NOTE: SICP, SICP, Joy of Clojure  
//=> _
```

虽然我在整个过程中从来没有显式地用_.value函数进行求值，但函数链中的所有调用都会执行求值。如果_.chain是惰性的，那么在调

用`_.value`之前什么都不会发生。

8.1.1 惰性链

出于第6章 实现trampoline时的教训，我可以实现`_.chain`的一个惰性变种，使其在调用`_.value`之前不会运行任何目标函数：

```
function LazyChain(obj) {  
  this._calls = [];  
  this._target = obj;  
}
```

LazyChain对象的构造函数很简单，它像`_.chain`一样接收一个目标对象并建立一个空的调用数组。第6章 的trampoline操作的是一个隐式调用链式，而LazyChain操作一个显式的数组。所以，问题仍然是，`_calls`数组中是什么。最合乎逻辑的答案，当然是函数，如下所示：

```
LazyChain.prototype.invoke = function(methodName /*, args */) {  
  var args = _.rest(arguments);  
  
  this._calls.push(function(target) {  
    var meth = target[methodName];  
  
    return meth.apply(target, args);  
  });  
  
  return this;  
};
```

LazyChain#invoke方法相当简单，但我想还是有必要简单过一遍。LazyChain#invoke的参数是一个字符串形式的方法名，以及一些额外的参数。LazyChain#invoke是将实际方法调用“包装”在一个闭包内，并将其推入`_calls`数组。注意看调用LazyChain#invoke后的`_calls`数组：


```
new LazyChain([2,1,3]).invoke('sort')._calls;  
//=> [function (target) { ... }]
```

在_calls数组中的元素是对[2, 1, 3]数组Array#sort方法延时调用的引用。

封装了一些行为的函数通常被称为thunk^[3]，如图8-1 所示。存储在_calls中的thunk期待将作为接收最终方法调用的对象的中间目标。



图8-1 thunk是等待被调用的函数

虽然并不是所有的编程语言都支持thunks，但由于JavaScript的函数实现比较容易，所以支持thunks便是自然而然的事情。

由于thunk等待被调用，那么调用它会发生什么？

```
new LazyChain([2,1,3]).invoke('sort')._calls[0]();
```

```
// TypeError: Cannot read property 'sort' of undefined
```

结果并不令人满意。但问题是，直接调用thunk是不能使其正确执行的。如果你还记得，在thunk预期目标对象上执行它闭合（closed-over）的方法。因此要使它工作，我需要以某种方式将原始数组作为参数传递给thunk，如图8-2 所示。

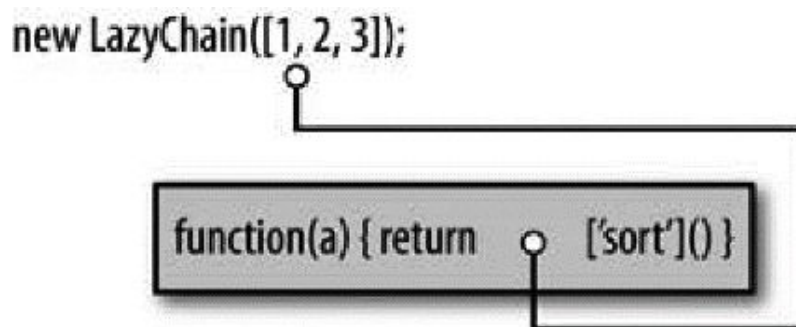


图8-2 为了使LazyChain正常工作，我们需要找到再次调用原始对象的方式

我可以通过复制并粘贴数组到thunk调用：

```
new LazyChain([2,1,3]).invoke('sort')._calls[0]([2,1,3]);
```

```
//=> [1, 2, 3]
```

直接将目标对象作参数传入thunk调用似乎是在作弊，但更像一个难看的API。相反，我可以用`_.reduce`函数，不仅可以重复调用初始thunk的参数，还可以调用任何`_calls`数组中的中间调用：

```
LazyChain.prototype.force = function() {  
  return _.reduce(this._calls, function(target, thunk) {  
    return thunk(target);  
  }, this._target);  
};
```

函数`LazyChain#force`是惰性链接逻辑的引擎。如图8-3 所示，`_.reduce`提供了类似第6章 的蹦床逻辑。从最初的目标对象入手，一个接一个调用thunk。

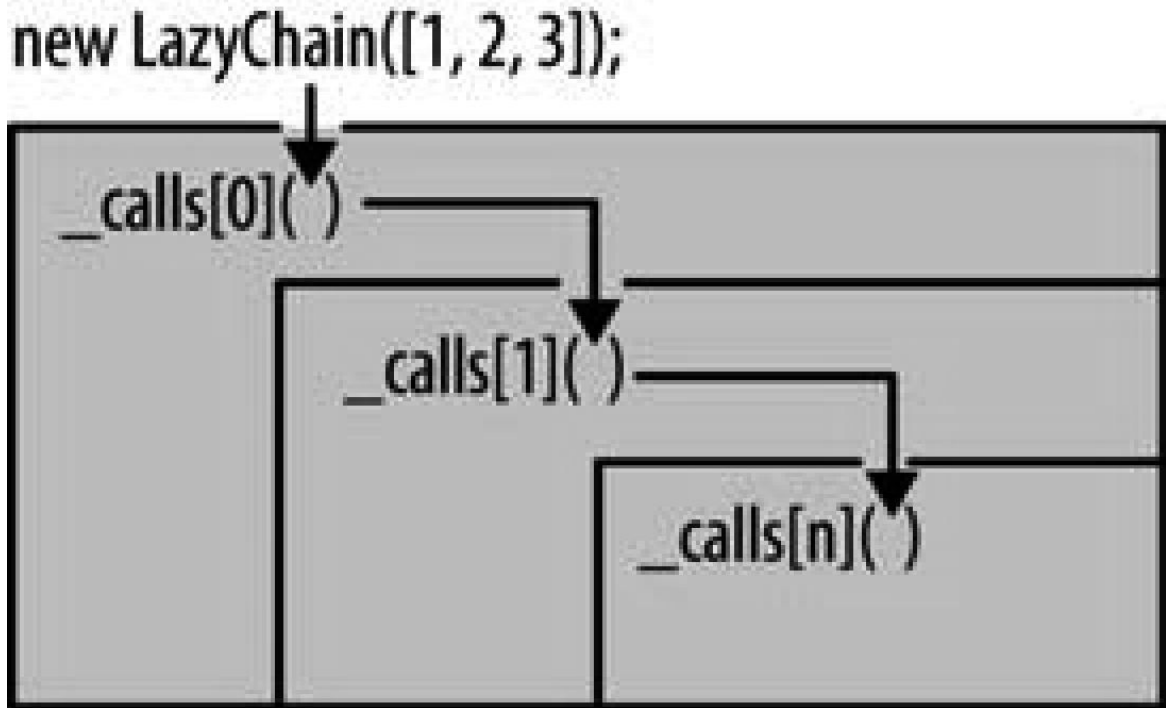


图8-3 使用reduce可以把中间结果传到各个thunk

既然有了LazyChain#force方法，来看看“终止”惰性链时会发生什么：

```
new LazyChain([2,1,3]).invoke('sort').force();
```

```
//=> [1, 2, 3]
```

太棒了！逻辑似乎是合理的，但若加入更多的thunk会发生什么？注意：

```
new LazyChain([2,1,3])  
  .invoke('concat', [8,5,7,6])  
  .invoke('sort')  
  .invoke('join', ' ')  
  .force();
```

```
//=> "1 2 3 5 6 7 8"
```

只要注意传递的类型，可以一直这样链下去。我提到过LazyChain的执行是懒惰的。因为thunks存储在_calls数组内并未曾执行，直到执行LazyChain#force时，为了更好地说明它的惰性，首先，我实现了一个惰性版本的_.tap与LazyChain的例子：

```
LazyChain.prototype.tap = function(fun) {  
  this._calls.push(function(target) {  
    fun(target);  
    return target;  
  });  
  
  return this;  
}
```

LazyChain#tap与LazyChain#invoke类似，因为实际的工作（调用一个函数，并返回目标）是包裹在一个thunk里的。我将展示tap是如何工作的：

```
new LazyChain([2,1,3])  
  .invoke('sort')  
  .tap(alert)  
  .force();
```

```
// alert box pops up  
//=> "1,2,3"
```

但是，如果不调用LazyChain#force会发生什么？

```
var deferredSort = new LazyChain([2,1,3])  
  .invoke('sort')  
  .tap(alert);
```

```
deferredSort;  
//=> LazyChain
```

什么也不会发生！deferredSort一直不会被调用，直到我显式地调用它：

```
// ... in the not too distant future  
  
deferredSort.force();  
  
// alert box pops up  
//=> [1, 2, 3]
```

这与jQuery的promises很类似。在讨论promises之前，我想再简单地扩展一下LazyChain，让它能支持链接其他的惰性链。请记住，LazyChain其实是一个thunk数组，我们只需要改变其构造函数使其可以接收LazyChain作为参数：

```

function LazyChainChainChain(obj) {
    var isLC = (obj instanceof LazyChain);

    this._calls = isLC ? cat(obj._calls, []) : [];
    this._target = isLC ? obj._target : obj;
}

```

也就是说，如果该参数的构造函数是另一个LazyChain实例，那么就抽取它的调用链和目标对象：

```

new LazyChainChainChain(deferredSort)
    .invoke('toString')
    .force();

// an alert box pops up
//=> "1,2,3"

```

这种可以组合链接的概念非常强大。可以建立一个离散行为库，而不用担心最终结果。还有很多其他方法可以加强LazyChain，比如缓存结果，提供不依赖于字符串的接口，这些给读者留作练习。

8.1.2 Promises

用LazyChain和LazyChainChainChain包装计算的描述供以后执行非常有用，jQuery^[4]也提供了类似功能——Promises，但略有区别。也就是说，jQuery的promises是为了给与主程序并发执行的异步操作提供一个流畅的API。

首先，可以简单地把promise理解为一个未完成的活动。如下面的代码，jQuery可以通过\$.Deferred创建promises：

```

var longing = $.Deferred();

```

我可以取到Deferred的promise：

```

longing.promise();
//=> Object

```

返回的对象是未完成动作的句柄：

```
longing.promise().state();  
//=> "pending"
```

正如结果显示，promise还是等待模式。原因当然是这个promise还未满足。只需要resolve就可以满足该promise：

```
longing.resolve("<3");
```

```
longing.promise().state();  
//=> "resolved"
```

此时，promise的状态显示为已满足，也可访问其值：

```
longing.promise().done(note);  
// NOTE: <3  
//=> <the promise itself>
```

Deferred#done方法仅仅是promise API提供的许多有用的链接方法之一。我不打算深入介绍jQuery的promise，而是用一个更复杂的例子来帮助理解promise与惰性链的区别。jQuery中的构建promise的一种方式是使用\$.when函数开始一个promise链，如下所示：

```
function go() {  
    var d = $.Deferred();  
  
    $.when("")  
        .then(function() {  
            setTimeout(function() {  
                console.log("sub-task 1");  
            }, 5000)  
        })  
        .then(function() {  
            setTimeout(function() {  
                console.log("sub-task 2");  
            }, 10000)  
        })  
        .then(function() {  
            setTimeout(function() {  
                d.resolve("done done done done");  
            }, 15000)  
        })  
  
    return d.promise();  
}
```

go函数中的promise链非常简单。我需要做的只是让jQuery开始三个异步任务，其中每个任务需要的时间都比前一个长。Deferred#then方法接收

一个函数并立即执行该函数。只有在需要最长时间的任务中解决Deferred。运行go可以揭开谜底：

```
var yearning = go().done(note);
```

done中的方法会等待go返回的promise被满足时才促发。但是，运行go后什么也没有发生。这是因为子任务需要等待超时，控制台日志尚未记录。我可以使用方法检查promise的状态：

```
yearning.state();  
//=> "pending"
```

正如你所期望的，状态依然未解决。几秒钟后：

```
// (console) sub-task 1
```

第一子任务的timeout被触发并打印到控制台。

```
yearning.state();  
//=> "pending"
```

当然，因为当初的promise链中的其他两个动作还在等待超时，状态仍是待定。再等待几秒钟，控制台会显示如下信息：

```
// (console) sub-task 2
```

```
// ... ~5 seconds later
```

```
// NOTE: done done done done
```

最终，延迟链中的最后一个环节被调用，并且由note函数打印出来。再次检查状态：

```
yearning.state();  
//=> "resolved"
```

当然，promise已经解决了，因为最后的子任务运行，调用了Deferred实例的resolve方法。这一系列事件与之前的LazyChain截然不同。也就是

说，一个LazyChain表示计算值的固定的顺序的调用。Promises虽然也表示调用序列，但一旦被执行，则随时可以得到值。

此外，jQuery的promise API是为了定义由多个异步子任务组合成的聚合任务。子任务可以尽可能同时进行。但是，总的任务不被视为完成，需要等待所有子任务完成并且通过resolve方法传入结果值。

惰性链也代表子任务组成的聚合任务，但它们总是一个接一个运行。两者之间的区别可以概括为高度聚合关联的任务（LazyChain）与低耦合（Deferred）的任务。

大多数jQuery的异步API调用都返回promise，因此异步调用的结果是可以链接的。然而，这种API的完整规范则是本书的范围之外。

8.2 管道

链接模式有利于给对象的方法调用创建流畅的API，但是对于函数式API则未必。Underscore提供的`_.chain`函数可实现链接，跟大多数函数一样，需要一个集合作为第一个参数。与此相反，本书中的函数将函数作为其第一个参数。这样做的目的非常明确，就是为了促进局部应用和柯里化。

方法链接有各种各样的缺点，包括紧耦合对象的`set`和`get`逻辑（Fowler称它们为命令/查询分离[2010]）和尴尬的命名问题。但主要的问题是，函数链经常会在调用之间改变传递的共同引用，如图8-4所示。函数式API重点在操作值而不是引用，而且还能微妙地（有时不那么微妙）转换数据，返回新的结果。

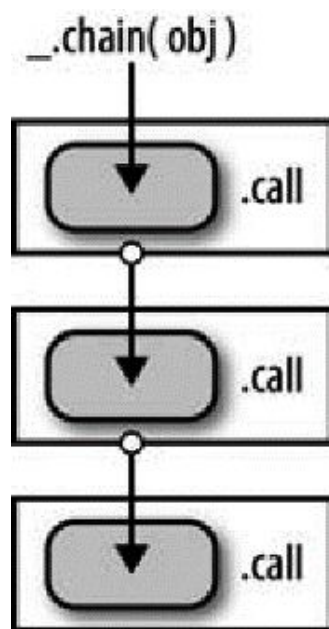


图8-4 链式方法调用改变共同引用

在本节中，我将讨论函数管道，以及如何最大化利用它。在一个理想的世界中，提供给函数的原始数据应该在调用后保持不变。函数式代码中的调用链应该接受期望的数据，对其进行非破坏性的修改，然后返回新的数据，如图8-5所示。

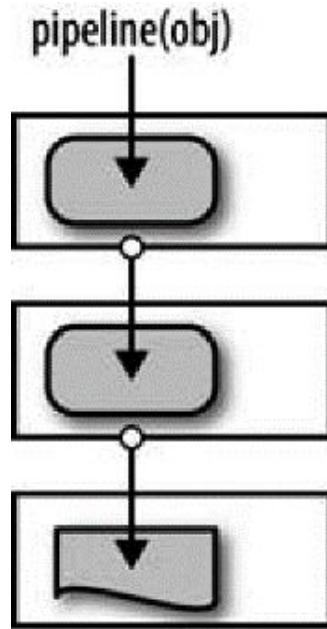


图8-5 用流水线函数转换数据

一个变换管道的“伪”API大概是这样的：

```
pipeline([2, 3, null, 1, 42, false]
, _.compact
, _.initial
, _.rest
, rev);
```

```
//=> [1, 3]
```

这条管道的调用顺序可以描述为。

1. 把数组[2, 3, null, 1, 42, false]传入_.compact函数。
2. 把_.compact返回的结果传给_.initial。
3. 把_.initial返回的结果传入_.rest。

4. 最后把`_.rest`返回的结果传入`rev`。

换句话说，如果用嵌套调用写出来的话，管道会是这样的：

```
rev(_.rest(_.initial(_.compact([2, 3, null, 1, 42, false]))));  
  
//=> [1, 3]
```

看到这个描述，你心里应该开始响起警钟。因为这基本与`LazyChain#force`是相同的。两种算法都使用相同的结果/调用编织。因此，`pipeline`的实现也应该与`LazyChain#force`很相似：

```
function pipeline(seed /*, args */) {  
  return _.reduce(_.rest(arguments),  
    function(l,r) { return r(l); },  
    seed);  
};
```

在`pipeline`中使用`_.reduce`几乎微不足道，然而，一个看似少量的代码却别有内涵。在继续深入之前，先来看看几个`pipeline`实战的例子：

```
pipeline();  
//=> undefined
```

```
pipeline(42);  
//=> 42
```

```
pipeline(42, function(n) { return -n });  
//=> -42
```

第一个传入`pipeline`的参数为种子值，或者称为启动函数的原始值。以后每次函数调用的结果都会被传入下一个函数，直到所有函数被执行[\[5\]](#)。

这有点类似于惰性链，只是它不是惰性的，而且重点在于值而不是可变引用^[6]。相反，管道更类似于使用`_.compose`创建的函数。想要惰性管道，只需要将其封装在函数内（或者`thunk`）：

```
function fifth(a) {  
  return pipeline(a  
    , _.rest  
    , _.rest  
    , _.rest  
    , _.rest  
    , _.first);  
}
```

想要管道工作，只需要传入数据：

```
fifth([1,2,3,4,5]);  
//=> 5
```

通过管道构建抽象并将其插入另一个管道是一个非常强大的技术。因此，它们可以组合成：

```
function negativeFifth(a) {
  return pipeline(a
    , fifth
    , function(n) { return -n });
}
```

```
negativeFifth([1,2,3,4,5,6,7,8,9]);
//=> -5
```

这个例子看上去很有意思，但如果能创建流畅的API则会更吸引眼球。回想第2章 中的关系代数运算符as, project和restrict的实现。每个函数的第一个参数用来生成一个新的表，以某种方式被“修改”。这些函数似乎非常适合于管道，例如查询表中版本是初版的书：

```
function firstEditions(table) {
  return pipeline(table
    , function(t) { return as(t, {ed: 'edition'}) }
    , function(t) { return project(t, ['title', 'edition', 'isbn']) }
    , function(t) { return restrict(t, function(book) {
      return book.edition === 1;
    });
  });
}
```

然后使用firstEditions：

```
firstEditions(library);

//=> [{title: "SICP", isbn: "0262010771", edition: 1},
//    {title: "Joy of Clojure", isbn: "1935182641", edition: 1}]
```

对于处理或提取表中元素，关系运算符效果很好，有了管道能让它更好地处理。

问题是，管道内的函数只接受一个参数。由于关系运算符预期两个参数，需要一个适配器函数来包装它们，才能在管道内正常工作。然而，关系运算符的设计上要求符合一致的接口：该表是第一个参数，“变”的规范是第二个。以这种一致性的优势，我可以用curry2建立更流畅的柯里化版的关系运算符：

```
var RQL = {  
  select: curry2(project),  
  as: curry2(as),  
  where: curry2(restrict)  
};
```

我决定将柯里化函数放到RQL（关系查询语言）对象的命名空间里，并在不同情况下用不同名字，以模仿SQL运算符。现在它们是柯里化的版本，而新的firstEditions更可读：

```
function allFirstEditions(table) {  
  return pipeline(table  
    , RQL.as({ed: 'edition'})  
    , RQL.select(['title', 'edition', 'isbn'])  
    , RQL.where(function(book) {  
      return book.edition === 1;  
    }));  
}
```

新的allFirstEditions函数不仅更容易阅读^[7]，而且能一样正常工作：

```
allFirstEditions(table);  
  
//=> [{title: "SICP", isbn: "0262010771", edition: 1},  
//    {title: "Joy of Clojure", isbn: "1935182641", edition: 1}]
```


在JavaScript中使用管道，再加上柯里化与部分应用，提供了强有力的方式来组合流畅的函数。事实上，本书中创建的函数都适合在管道中工作。作为一个额外的好处，函数式编程的重点是数据的转化，但当数据从一个函数流到了下一个，经常会被间接和深嵌套函数阻碍。用管道可以使得数据流更加明确。但是，管道并不适合于所有情况。事实上，我很少会使用管道来执行有副作用的I/O操作、Ajax调用或突变，因为它们往往不返回值。

传入管道的数据自始至终都应该是相同的。这样做有助于确保该管道是可组合的。是不是有方法可以让我编写管道状结构的纯函数？在下一节中，我将基于之前探索链接与管道时的教训，讨论如何以流畅的方式组合具有副作用的操作。

8.3 数据流与控制流

在lazyChain的例子中，我从执行逻辑（.force）分离出来执行规范（.invoke）。同样，用pipeline函数，我可以并列多个纯函数来实现串行处理管道。以lazyChain和pipeline为例，从调用序列中的一个节点移动到下一个时值是稳定的。具体来说，直到force被调用之前，lazyChain总是返回一些LazyChain对象。同样，虽然在任何时候pipeline的中间类型可以改变，但这种变化的优先级比组合更高，以保证阶段之间的传值正确。但是，如果组合了不应该被组合的函数会发生什么？

在这一章的最后一节中，我将讨论一种新的使用惰性管道actions（Stan，2011）组合所谓不协调返回类型函数的新技术。

想象函数是接收某些“形状”的输入数据而输出另一形状（也可能相同）的数据的盒子的话，如图8-6 所示[8]。

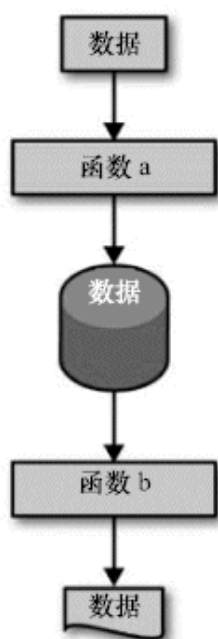


图8-6 函数a接收一个长方形的物体，并返回一个数据库形状的物体；
函数b需要一个数据库状物体，并返回一个文件形状的数据

因此，惰性链能正常工作的原因是，链接里的方法调用之间的形状都是一致的，只有当force被调用时才起变化[9]。图8-7 说明了这一事实。

类似地，管道节点之间或组合函数之间的图形，并不像一般对象那样稳定，需要根据下一节点的需求发生变化，如图8-8 所示。

问题在于，如果形状不匹配，那么`pipeline`，`_.compose`和`lazyChain`将不会像预期一样正常工作：

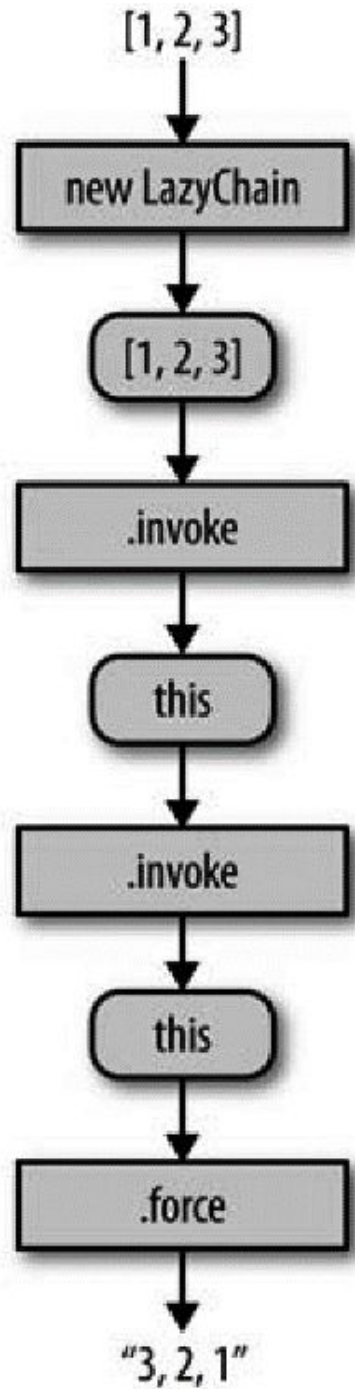


图8-7 在一个惰性的调用链之间流动的图形是稳定的，只有调用force会发生（潜在的）变化

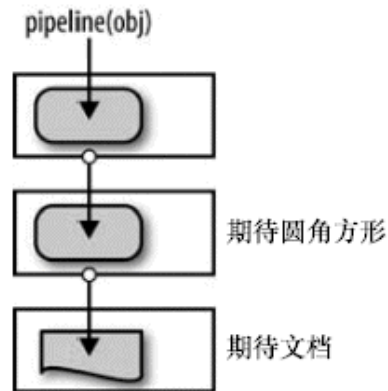


图8-8 在管道调用或组合之间的图形需要根据预期的方式改变

```
pipeline(42
  , sqr
  , note
  , function(n) { return -n }));
```

```
// NOTE: 1764
//=> NaN
```

导致失败的原因是中途的形状（从note）变成了undefined。

事实上，如果你想达到满意的效果，那么需要手动执行此操作：

```
function negativeSqr(n) {
  var s = sqr(n);
  note(n);
  return -s;
}
```

```
negativeSqr(42);
// NOTE: 1764
//=> -1764
```

有了样板可以满足快速增涨的要求。同样，可以只改变note函数，使其返回任何参数。这可能是一个好主意，但这里这样做只解决了一种症状，而不能根治不相容的中间形状的问题。有的函数返回不兼容的形状，甚至没有返回，则要求控制流能微妙地编排组合代码。取得这种微妙的平衡的关键在于寻找一种方式来组合函数，能让值从一个函数传到下一个。

现在你可能以为要解决这个问题的办法就是找到稳定的节点之间的形状——确实是这样的。

8.3.1 找个一般的形状

确定一般形状难在放置什么进去，而不是类型选择（普通对象即可）。一种可行的方法是选定一种数据能在流之间传递，例如negativeSqr：

```
{values: [42, 1764, undefined, -1764]}
```

还需要什么吗？能够保存数据的状态，或者目标对象会有些用处。图8-9 可视化了不同的输入和输出形状时的组合动作。

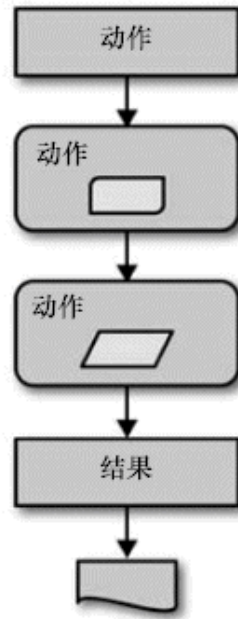


图8-9 动作之间流动的形状被制成稳定使用的上下文对象

最后一个节点（例如结果）类似于force操作，会把答案提取出来。例如negativeSqr函数，获取最终结果的方式是取values的最后一个元素：

```
{values: [42, 1764, undefined, -1764],  
state: -1764}
```

现在，actions函数的实现是用pipeline和lazyChain的组合来管理这些中间状态，如下所示：

```

function actions(acts, done) {
  return function (seed) {
    var init = { values: [], state: seed };

    var intermediate = _.reduce(acts, function (stateObj, action) {
      var result = action(stateObj.state);
      var values = cat(stateObj.values, [result.answer]);

      return { values: values, state: result.state };
    }, init);

    var keep = _.filter(intermediate.values, existy);

    return done(keep, intermediate.state);
  };
};

```

actions函数需要一个函数数组，其中每个函数接收一个值并返回中间状态对象的函数。actions函数进而规约所有函数到数组中，并建立一个中间状态的对象：

```

...
var intermediate = _.reduce(acts, function (stateObj, action) {
  var result = action(stateObj.state);
  var values = cat(stateObj.values, [result.answer]);

  return { values: values, state: result.state };
}, init);
...

```

在此过程中，actions期望每个函数得到的结果是包含两个键（answer和state）的对象。answer的值对应于调用函数的结果，state值代表执行之后的新状态。对于函数note，其状态不会改变。intermediate状态对象可能有一些伪答案（如answer的note是undefined），可以用actions将它筛选出来：

```

...
var keep = _.filter(intermediate.values, existy);

return done(keep, intermediate.state);
...

```

最后，actions将过滤后的values (keep) 以及state传入done函数，以取得最后的结果。我完全可以只传state或values至done函数，但我想通过传这两个参数以最大化其灵活性，而且这样也有助于说明问题。

为了演示actions的工作原理，我需要拆解negativeSqr，并重新组合这一系列的动作。首先，sqr函数显然不知道状态对象，所以我需要创建一个适配器的函数mSqr^[10]：

```
function mSqr() {  
  return function(state) {  
    var ans = sqr(state);  
    return {answer: ans, state: ans};  
  }  
}
```

现在可以使用actions进行双平方操作：

```
var doubleSquareAction = actions(  
  [mSqr(),  
    mSqr()],  
  function(values) {  
    return values;  
  }  
);
```

```
doubleSquareAction(10);  
//=> [100, 10000]
```


因为直接返回values数组，doubleSquareAction的结果都是中间状态（10的平方以及10的平方的平方）。这几乎与pipeline一模一样。真正的魔力在于混合了不同形状的函数：

```
function mNote() {  
  return function(state) {  
    note(state);  
    return {answer: undefined, state: state};  
  }  
}
```

mNote的answer当然是undefined，因为它是用于打印的函数；然而，state可以继续传递下去。mNeg函数现在似乎就很清楚了：

```
function mNeg() {  
  return function(state) {  
    return {answer: -state, state: -state};  
  }  
}
```

现在可以组合这些新函数到actions了：

```
var negativeSqrAction = actions([mSqr(), mNote(), mNeg()],  
  function(_, state) {  
    return state;  
  }));
```

用法如下：

```
negativeSqrAction(9);  
// NOTE: 81  
//=> -81
```

使用actions的组合范式是组成不同形状的函数的一般方法。可悲的是，之前的代码好像用了很多繁琐的过程来达到所需要的效果。幸运的

是，有一个更好的方式来定义动作，无须知道状态对象是如何创建的细节，并避免随之而来的样板。

8.3.2 函数可以简化创建action

在本节中，我将定义一个函数lift，它接收两个函数：一个提供给出结果值，一个提供新状态的函数。lift函数将用于抽象actions中间的状态管理。lift的实现很简单：

```
function lift(answerFun, stateFun) {  
  return function(/* args */) {  
    var args = _.toArray(arguments);  
  
    return function(state) {  
      var ans = answerFun.apply(null, construct(state, args));  
      var s = stateFun ? stateFun(state) : ans;  
  
      return {answer: ans, state: s};  
    };  
  };  
};
```

lift看起来像柯里化（如它返回一个函数），事实就是这样的。除非能提供一个更好的接口，否则没有理由柯里化lift，我将会举例说明。事实上，使用lift，我可以更好地重新定义mSqr，mNote和mNeg：

```
var mSqr2  = lift(sqr);  
var mNote2 = lift(note, _.identity);  
var mNeg2  = lift(function(n) { return -n });
```

sqr和负数函数的answer和state是相同的值，所以我只需要提供answer函数。而note得到的答案（undefined）显然不是状态值，所以使用_.identity可以指定其为传递动作。

用actions组合的新actions：

```

var negativeSqrAction2 = actions([mSqr2(), mNote2(), mNeg2()],
  function(_, state) {
    return state;
  });

```

用法和以前一样：

```

negativeSqrAction(100);
// NOTE: 10000
//=> -10000

```

如果想使用lift和actions来实现stackAction，可以：

```

var push = lift(function(stack, e) { return construct(e, stack) });

```

push函数返回一个新数组，伪装成一个栈，新元素排在前面。由于中间状态也是答案，就没有必要提供一个状态函数。pop的实现需要两个函数：

```

var pop = lift(_.first, _.rest);

```

栈是由数组模拟的，pop的结果是第一个元素。相反，状态函数_.rest返回新的已删除栈顶元素的栈。我现在可以使用这两个函数来编写两次入栈和一次出栈操作，如下所示：

```
var stackAction = actions([
  push(1),
  push(2),
  pop()
],
function(values, state) {
  return values;
});
```

令人惊讶的是，通过使用actions函数，我抓到事件栈顺序过程中的值。

```
stackAction([]);
```

```
//=> [[1], [2, 1], 2]
```

如上所示，stackAction仅仅是一个函数，现在可以与其他函数组合成高阶行为。既然已经决定返回所有的中间结果，由此产生的返回值也在其中：

```
pipeline(  
  []  
  , stackAction  
  , _.chain)  
  .each(function(elem) {  
    console.log(polyToString(elem))  
  });  
  
  // (console) [[1],      // the stack after push(1)  
  // (console) [2, 1],    // the stack after push(2)  
  // (console) 2]         // the result of pop([2, 1])
```

这几乎像魔术一样，但通过解构它，可以证明它确实没有神奇可言。相反，使用一般的中间型和以及lift和actions函数的管理可以组合不同形状的函数。这种管理可以将需要保持控制流类型的问题，转化为数据流的问题（Piponi, 2010）。

8.4 总结

本章着重探讨将行为看作离散步骤序列的可能性。在本章的第一部分中，我们讨论了链接。方法链在JavaScript库如流行的jQuery中广泛使用。总之，方法链是让对象的方法返回一个一般的this引用，以便一般方法可以在序列中调用。我还通过使用jQuery的promises和Underscore的`_.chain`函数进一步介绍链接，接着探索了“惰性链”的点子，将一些方法串到目标后面待以后执行。

对于链的另一个想法是“管道”，或可以说是接收一个数据块，返回转换后的数据块的函数序列。管道不像链接，它操作数组或对象而不是引用。另外，流过管道的数据类型可以改变，只要是在管道中的下一个步骤预期的类型。管道不会对流过的数据造成损伤。

虽然链和管道分别操作于引用和数据类型，但动作序列的想法并不局限于此。actions类型的实现隐藏了管理用于混合不同返回和参数类型的函数使用的数据的结构细节。

在接下来的最后一章，将讨论如何用函数式编程进行“无类”风格的编程。

[1] 如果你正在使用Underscore的压缩（minified）版，可能会在这里看到一个不同的对象名。那是由压缩工具造成的。

[2] `note`是在第1章中定义的。

[3] “thunk”与ALGOL有很深的渊源。

[4] 还有很多JavaScript库提供类似jQuery的 `promises` 库，如：`Q`，`RSVP.js`，`when.js`和`node-promises`。

[5] 如果你想更花哨一点，可以把pipeline叫作thrush连接符。

[6] 基于这些巨大的差异，你可以说它们一点也不像。

[7] 至少我认为代码对读者应该很轻松。

[8] 我只想说明，字符串数组与浮点数有不同的形状。你可以用“类型”或“结构”来替换形状，但我会坚持用形状，因为图片看起来更好。这个可视化图形的灵感来自于Alan Dipert (Dipert, 2012)。

[9] 虽然force的结果可能也是一个惰性链。

[10] action其实可以称为Monad，但我却选择前者，因为我认为Monad在缺乏一个强类型以及返回型多态性的系统中会被大大削弱。然而，这并不是说，Monad不能给我们在JavaScript中使用解构的宝贵经验。

第9章 无类编程

许多人第一次接触到JavaScript及其工具（函数、对象、原型和数组）时都印象深刻。因此，为了“修改”JavaScript以符合软件建模解决方案，他们往往寻求或重新创建基于类的系统。这种愿望是完全可以理解的，一般人们都会寻求熟悉的方案。不过，既然你已经在JavaScript中的函数式编程的探索中走到这一步，就值得将前面章节中的所有思路连贯起来，探索如何具体化函数式和面向对象思想。

本章将首先回顾对数据和函数的思考。然而，虽然函数式的思考很重要，但也可能仍然需要建立自定义的抽象。因此，我将介绍一种方法来“混合”离散行为，组合成更复杂的行为。我还将讨论如何用函数式的API隐藏这些自定义设置。

9.1 数据导向

在这本书中，我故意根据JavaScript的基本类型、数组和对象定义数据来建模。也就是说，为了组合简单数据，以形成更高层次的概念，如表（第8章）和命令（第4章），我故意避免创建层级结构的类型。坚持把重点放在函数而不是方法上，让我能提供不依赖于对象思考与方法论的API。相反，通过秉承的函数式接口，真正实现数据抽象的具体类型变得不那么重要。这带来了改变数据的实现细节的灵活性，同时还能保持一致的函数式接口。

图9-1 说明了使用函数式API时，并不需要担心调用链中的节点之间流动的类型。

当然，函数本身应当能够处理流之间的类型，但精心设计的API是为了组合并抽象中间类型的细节。然而，有些时候以对象为中心的思想是至关重要的。例如，第8章中实现的LazyChain专门处理目标对象上的方法的惰性执行。显然，这个问题的答案便是供对象在何处调用方法的解决方案。然而，实现要求LazyChain的用户直接创建该类型的实例。由于JavaScript极大的灵活性，没有必要专门建立LazyChain类型。惰性链是从函数lazyChain返回的能响应.invoke和.force调用的对象。

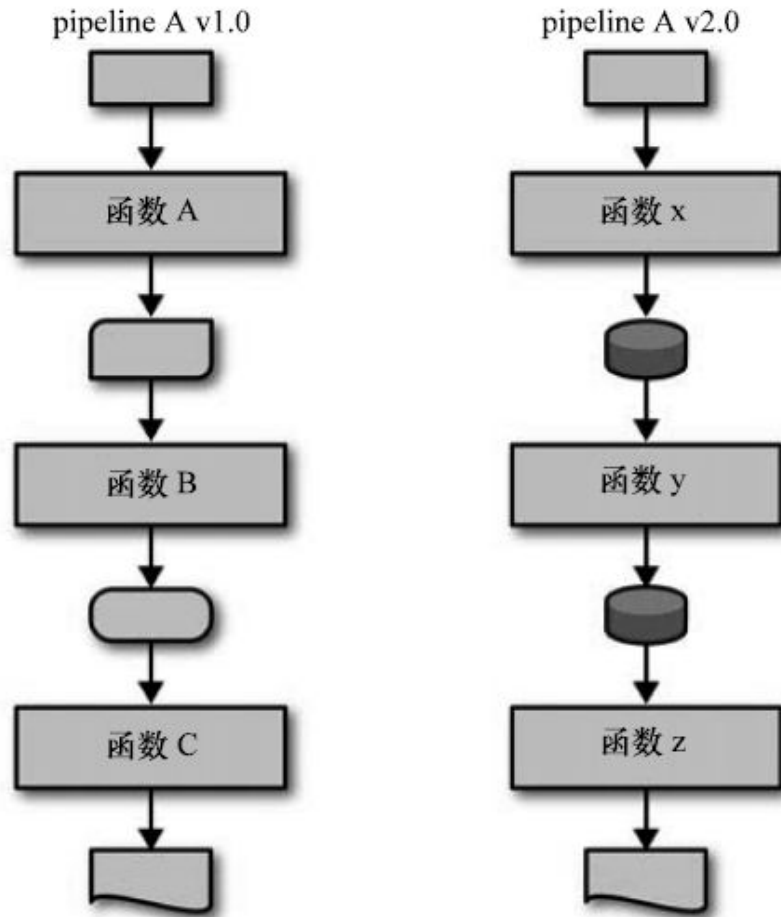


图9-1 当坚持一个函数式接口时，中间数据的类型就不那么重要了，而且可以根据需要进化（或退化），特别是如果关心的主要是计算的开始和结束

```
function lazyChain(obj) {  
  var calls = [];  
  
  return {  
    invoke: function(methodName /* args */) {  
      var args = _.rest(arguments);
```

```

        calls.push(function(target) {
            var meth = target[methodName];

            return meth.apply(target, args);
        });

        return this;
    },
    force: function() {
        return _.reduce(calls, function(ret, thunk) {
            return thunk(ret);
        }, obj);
    }
};
}

```

这几乎是LazyChain实现的所有代码，除了以下几种情况。

- 惰性链是通过函数调用启动的。
- （calls内的）调用链是私有数据^[1]。
- 没有明确的LazyChain类型。

lazyChain的实现如下所示：

```

var lazyOp = lazyChain([2,1,3])
    .invoke('concat', [7,7,8,9,0])
    .invoke('sort');

lazyOp.force();
//=> [0, 1, 2, 3, 7, 7, 8, 9]

```

当然，也可以建立明确的数据类型。我将在下一节中展示，而且会等到必要时再定义它们。相反，我们优先选择抽象。因为与惰性链交互比指定LazyChain类型更重要。

- JavaScript提供了大量有效的方法来延缓或消除创建具名类型和类型层级结构的必要，包括如下几种。
- 可用的原始数据类型。
- 可用的聚合数据类型（例如数组和对象）。
- 操作于内置数据类型的函数。
- 匿名对象包含方法。
- 类型对象。
- 类。

上述几点可以作为一个实现JavaScript API的清单，如图9-2 所示。

JavaScript开发人员经常反转图9-2 所示的层次结构，直接去构造类，这样做使得他们一开始就可以和抽象说再见了。如果你转而选择从内置类型开始，再加上流畅的函数式的API，这样可以让自己有很大的灵活扩展空间。

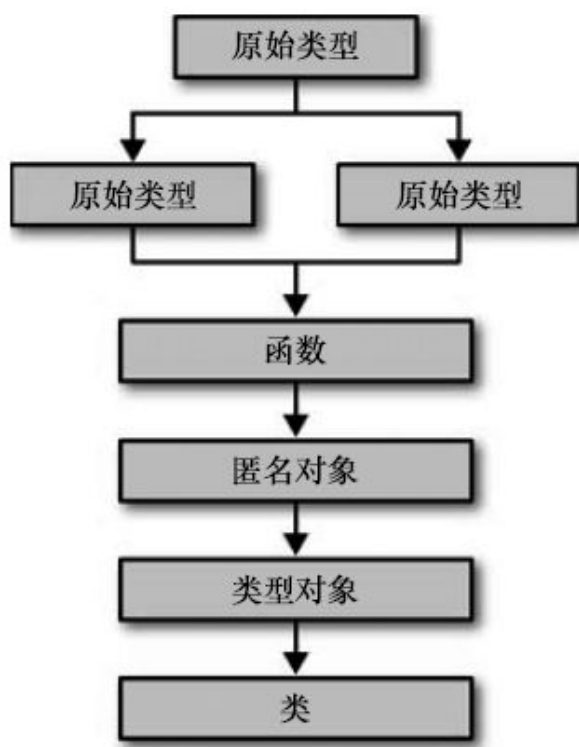


图9-2 数据思维的“层级”

通过函数来构建

对于大多数编程任务，发生在一些计算中间的活动是最重要的（Elliott, 2010）。例如，读取表单的值，并验证它，然后对新类型进行

操作，最终发送新值作为字符串发往别处。相比于验证和处理步骤，获得字符串的行为显得不是那么重要。

我创建过的一些做这种任务的工具像是函数式和基于对象思想的混合体。然而，如果只是用函数的话，那么会得到更流畅的解决方案。

首先，惰性链显然是以对象为中心的，事实上需要穿帙方法来进行操作。然而，惰性链可以被解构为三个阶段。

1. 获得一些对象。
2. 定义一个与对象相关的链。
3. 执行该链。

获取一个对象的行为是微不足道的；它只作为JavaScript代码运行的一部分。定义一个链，则有趣多了。而一个惰性链被限制在一个特定的实例上执行，通过提升成对函数的操作，我可以让惰性链对跨类型的对象操作：

```
function deferredSort(ary) {  
    return lazyChain(ary).invoke('sort');  
}
```

这让我可以通过一个普通的函数调用创建各种数组的惰性排序：

```
var deferredSorts = _.map([[2,1,3], [7,7,1], [0,9,5]], deferredSort);  
  
//=> [<thunk>, <thunk>, <thunk>]
```

当然，我想执行每一个thunk，但因为要分解函数，我宁愿封装方法调用：

```
function force(thunk) {
  return thunk.force();
}
```

现在我可以执行任意惰性链：

```
_.map(deferredSorts, force);
```

```
//=> [[1,2,3], [1, 7, 7], [0, 5, 9]]
```

我已经将方法调用“提升”到函数应用的境界，现在我可以定义相应的数据处理的原子功能离散块：

```
var validateTriples = validator(
  "Each array should have three elements",
  function (arrays) {
    return _.every(arrays, function(a) {
      return a.length === 3;
    });
  });
```

```
var validateTripleStore = partial1(condition1(validateTriples), _.identity);
```

聚合验证到它自己的函数（或许更多函数）可以让我验证独立于任何活动中的其他步骤，而且可以在类似活动中重用该验证。

再检查看验证是否按预期工作：

```
validateTripleStore([[2,1,3], [7,7,1], [0,9,5]]);
//=> [[2,1,3], [7,7,1], [0,9,5]]
```

```
validateTripleStore([[2,1,3], [7,7,1], [0,9,5,7,7,7,7,7,7]]);
// Error: Each array should have three elements
```

现在可以定义其他（不一定）懒惰的处理步骤：

```
function postProcess(arrays) {  
    return _.map(arrays, second);  
}
```

现在，我可以定义将这些碎片聚合到一个特定域的更高级别的活动：

```
function processTriples(data) {  
    return pipeline(data  
        , JSON.parse  
        , validateTripleStore  
        , deferredSort  
        , force  
        , postProcess  
        , invoker('sort', Array.prototype.sort)  
        , str);  
}
```

processTriples用法如下：

```
processTriples("[[2,1,3], [7,7,1], [0,9,5]]");  
  
//=> "1,7,9"
```

将验证加到管道中的好处是，当给定的是坏数据时就会提前终止：

```
processTriples("[[2,1,3], [7,7,1], [0,9,5,7,7,7,7,7,7]]");  
  
// Error: Each array should have three elements
```

这样我就可以随意使用该函数：

```
$.get("http://djhkjhkdj.com", function(data) {  
    $('#result').text(processTriples(data));  
});
```

还可以通过抽象报告逻辑让这一过程更通用：

```
var reportDataPackets = _.compose(  
  function(s) { $('#result').text(s) },  
  processTriples);
```

继续探索reportDataPackets：

```
reportDataPackets("[[2,1,3], [7,7,1], [0,9,5]]");  
// a page element changes
```

现在你可以将这个离散行为添加到你的应用程序，以实现预期的效果：

```
$.get("http://djhkjhkdj.com", reportDataPackets);
```

创建于一般函数可以让你将问题看成数据从管道一端到另一端的逐步转型。你应该还记得图9-1 中，每个变换管道本身可以看作一个离散活动，以预期方式处理已知数据类型。如图9-3 所示，兼容管道可以串成端到端的前馈方式，而不相容的管道则可以通过适配器链接。

从程序的角度来看，管道与适配器可以加到输入和输出源。这样的想法让你从较小的、已知的部分开始组合一个系统，同时允许根据需要灵活地互换部件和中间数据表现。通过适配器链接数据流的想法是可扩展的概念，从一个单一的功能到整个系统。

然而，有些时候，对象层次的思考是合适的，特别是当具体类型秉承通用mixin这样的抽象。在下一节中，我将谈论一个mixin的想法，以及如何使用它对函数建立抽象。

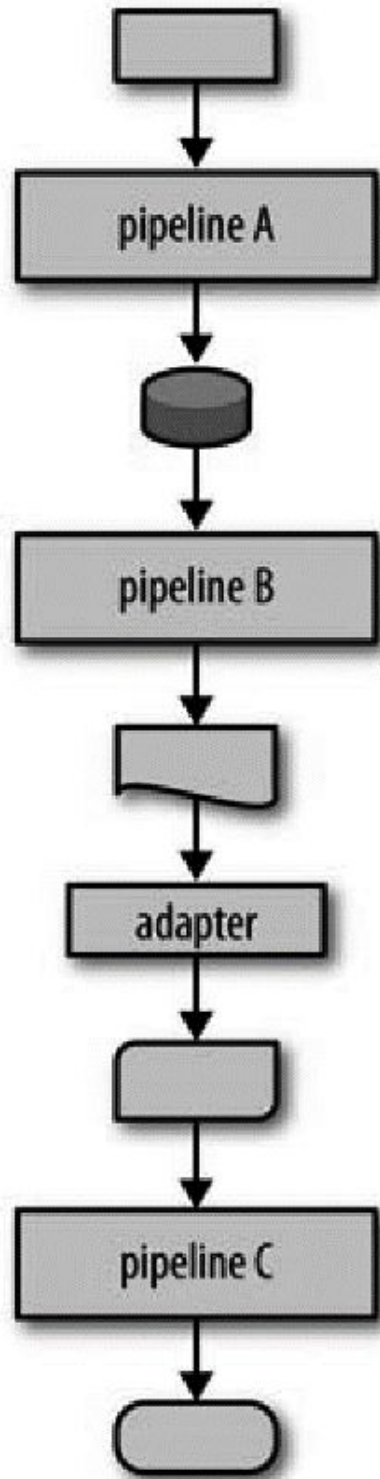


图9-3 通过适配器链接管道

9.2 Mixins

虽然我已经花了大量的时间和篇幅概括函数式编程的风格，但有些时候，对象和方法也是很合适的解决方案。在本节中，我将概括基于mixin的扩展方法，类似于基于类的系统的构建，但却有意加以限制。在深入mixin之前，让我花点时间来加强一下对象思想。想象一下，一个函数polyToString接受一个对象，并返回它的字符串的表现形式。一个原生的polyToString实现可以是这样的：

```
function polyToString(obj) {  
  if (obj instanceof String)  
    return obj;  
  else if (obj instanceof Array)  
    return stringifyArray(obj);  
  
  return obj.toString();  
}  
  
function stringifyArray(ary) {  
  return "[" + ary.map(polyToString).join(",") + "]".join('');  
}
```

如上述代码所示，polyToString初期的实现可以写成嵌套if语句，其中每个分支进行类型检查。加入stringifyArray可以创建更好看的格式的字符串。我们来测试一下polyToString的行为：

```
polyToString([1,2,3]);  
//=> "[1,2,3]"
```

```
polyToString([1,2,[3,4]]);  
//=> "[1,2,[3,4]]"
```

这似乎是合理的，不是吗？如果试图建立新的字符串格式，则需要添加一个新的if分支到polyToString。这样看起来很愚蠢。更好的办法是使用类似第5章 的dispatch。该函数接收一些函数，并尝试执行每一个函数，返回第一个非undefined的值：

```
var polyToString = dispatch(  
  function(s) { return _.isString(s) ? s : undefined },  
  function(s) { return _.isArray(s) ? stringifyArray(s) : undefined },  
  function(s) { return s.toString() });
```

这次是通过使用dispatch进行类型检查的，我还将每个检查都抽象成单独的函数，方便以后的组合与扩展。当然，使用 dispatch还是可以按预期方式正常工作的：

```
polyToString(42);  
//=> "42"
```

```
polyToString([1,2,[3,4]]);  
//=> "[1, 2, [3, 4]]"
```

```
polyToString('a');  
//=> "a"
```

你可能已经想到，新类型将依然存在问题，如果它们还没有一个好的#toString实现：

```
polyToString({a: 1, b: 2});  
//=> "[object Object]"
```

然而，比起痛苦地修改嵌套if语句，用dispatch能让我简单地组合出另一个函数：

```
var polyToString = dispatch(  
  function(s) { return _.isString(s) ? s : undefined },  
  function(s) { return _.isArray(s) ? stringifyArray(s) : undefined },  
  function(s) { return _.isObject(s) ? JSON.stringify(s) : undefined },  
  function(s) { return s.toString() });
```

再次，polyToString新的实现也如预期的正常工作：

```
polyToString([1,2,{a: 42, b: [4,5,6]}, 77]);
```

```
//=> '[1,2,{"a":42,"b":[4,5,6]},77]'
```

这样使用dispatch的方式看起来相当优雅^[2]，但我还是不禁觉得它有点怪异。增加对另一种类型如第7章 中Container的支持，就可以看出我所说的怪异：

```
polyToString(new Container(_.range(5)));
```

```
//=> '{"_value":[0,1,2,3,4]}'
```

当然，可以通过往组成dispatch的调用链中添加另一个链接，使其看起来也更顺眼，像下面这样：

```
...  
  return ["@", polyToString(s._value)].join('');  
...
```

但问题是，dispatch以一种极其简单的方式工作，即它从第一个函数开始，不断尝试直到其中一个有返回值。超出单一层级的编码类型信息最终将使它变得复杂。相反，如定制toString操作则会是一个很好的方法论。然而，实现这一目标的典型做法是违背我在前言 中概括的JavaScript使用政策。

- 修改了核心原型。
- 构建了类层次结构。

在讲基于mixin的扩展之前，我先谈谈这两项。

9.2.1 修改核心原型

很多时候，用JavaScript创建新的类型时，可能需要组合或扩展之外的特定行为。Container类型是一个很好的例子：

```
(new Container(42)).toString();  
//=> "[object Object]"
```

这是不行的。显而易见的选择是，我可以将Container专用的toString方法添加到prototype：

```
Container.prototype.toString = function() {  
  return ["@<", polyToString(this._value), ">"].join('');  
}
```

现在Container的所有实例将具有相同的toString行为：

```
(new Container(42)).toString();  
//=> "@<42>"  
  
(new Container({a: 42, b: [1,2,3]})).toString();  
//=> "@<{"a":42,"b":[1,2,3]}>"
```

当然，Container是我控制的类型，所以修改其prototype便是理所当然的，下面重担就落在文档化期望接口和用例上。不过，如果我想补充一些核心对象能力怎么办？唯一的选择就是修改核心原型：

```
Array.prototype.toString = function() {  
  return "DON'T DO THIS";  
}
```

```
[1,2,3].toString();  
//=> "DON'T DO THIS"
```

但问题是，如果任何人使用你的库，他创建的任何数组都会受到Array#toString方法的污染。因此，对于核心类型如Array和Object，最好

还是保持自定义行为与代理到自定义类型的函数分离。在 `Container#toString` 中，我们则代理给 `polyToString`。我会在讨论 `mixin` 时详细介绍这种做法。

9.2.2 类层次结构

在 `Smalltalk` 中，一切都发生在其他地方。

——阿黛尔·戈德堡

当使用面向对象的方法定义一个系统时，你通常试图枚举所有组成该系统的类型，以及它们之间的关联。当通过一个面向对象的眼镜来看待问题时，通常会发现类之间的关系是层次结构。比如员工类型可以分为会计师、托管人或CEO。经常用这种层次关系来描述系统的组成。

想象一下实现 `Container` 类型的类型层次结构（见图9-4）。

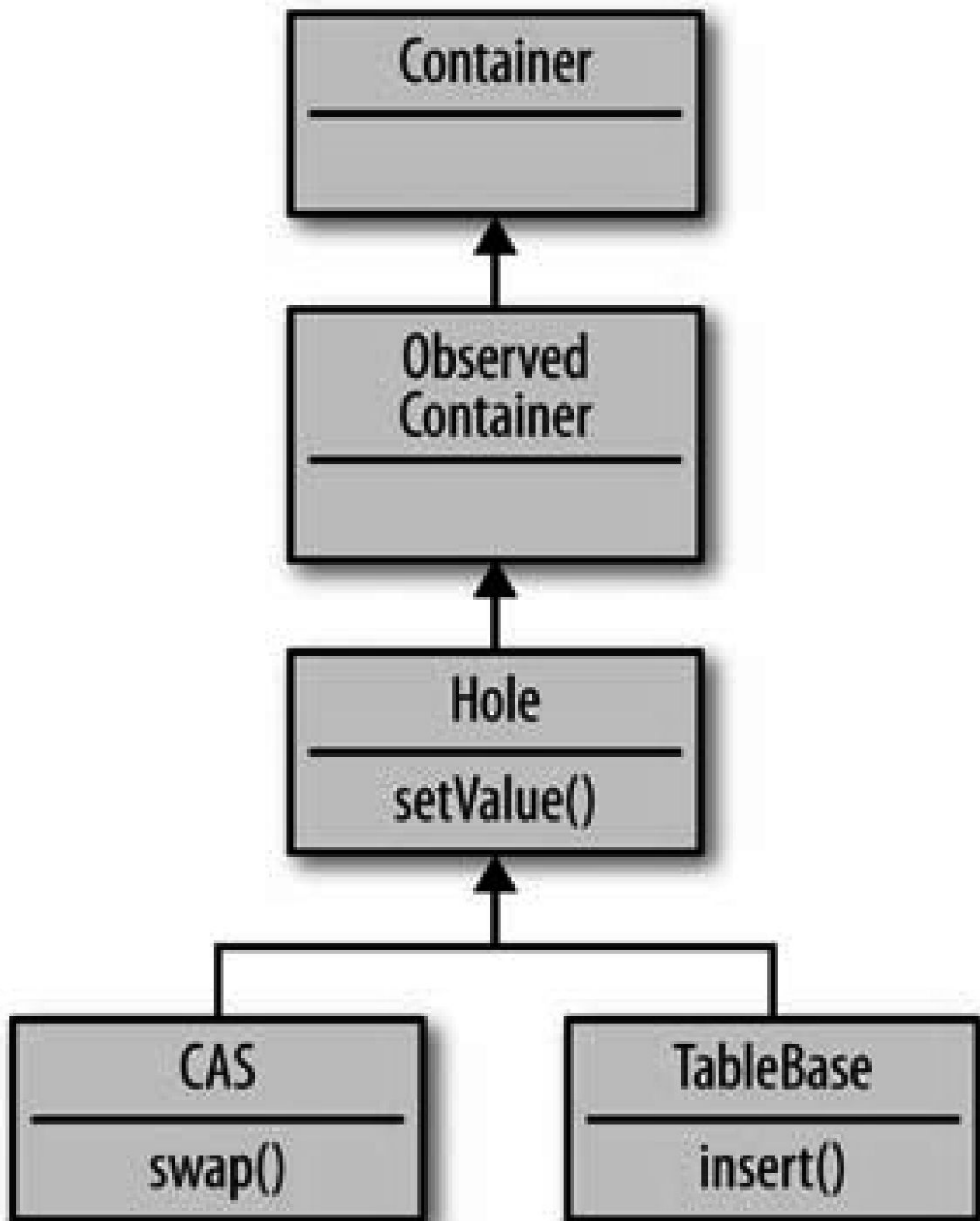


图9-4 容器类型的层次结构

图9-4 中指出，层次结构的根是类Container和派生类ObservedContainer，该类用于连接接收状态变化信息的函数。从ObservedContainer中，我得出一个Hole类型，它是“set-able”。最后，我定义了两个不同的Hole类型，对如何赋值具有不同的语义。

使用基于John Resig创造类库，我可以大概画出层次结构（Resig, 2008）：

```
function ContainerClass() {}  
function ObservedContainerClass() {}  
function HoleClass() {}  
function CASClass() {}  
function TableBaseClass() {}
```

```
ObservedContainerClass.prototype = new ContainerClass();  
HoleClass.prototype = new ObservedContainerClass();  
CASClass.prototype = new HoleClass();  
TableBaseClass.prototype = new HoleClass();
```

现在，所有的层次关系都联系在一起，可以试试是否如我所料：

```
(new CASClass()) instanceof HoleClass;  
//=> true
```

```
(new TableBaseClass()) instanceof HoleClass;  
//=> true
```

```
(new HoleClass()) instanceof CASClass;  
//=> false
```

这是我所期望的继承沿层次往上走，而不是往下。现在，先在实现中放一些stub：


```
var ContainerClass = Class.extend({  
  init: function(val) {  
    this._value = val;  
  },  
});
```

```
var c = new ContainerClass(42);
```

```
c;  
//=> {_value: 42 ...}
```

```
c instanceof Class;  
//=> true
```

ContainerClass只有一个值。然而，ObservedContainerClass提供了一些额外的功能：

```
var ObservedContainerClass = ContainerClass.extend({  
  observe: function(f) { note("set observer") },  
  notify: function() { note("notifying observers") }  
});
```

当然，ObservedContainerClass自身不会做太多事情。相反，需要一种方法来设置一个值，并通知值发生变化：

```

var HoleClass = ObservedContainerClass.extend({
  init: function(val) { this.setValue(val) },
  setValue: function(val) {
    this._value = val;
    this.notify();
    return val;
  }
});

```

正如你所期望的，新HoleClass实例就有层次结构：

```

var h = new HoleClass(42);
// NOTE: notifying observers

h.observe(null);
// NOTE: set observer

h.setValue(108);
// NOTE: notifying observers
//=> 108

```

现在，在层次结构的底部添加新的行为：

```

var CASClass = HoleClass.extend({
  swap: function(oldVal, newVal) {
    if (!_.isEqual(oldVal, this._value)) fail("No match");

    return this.setValue(newVal);
  }
});

```

CASClass实例中加入了额外的“比较并交换”语义，即“提供你所认为的旧值和新值，只有当预期与实际旧值匹配时才会设置新值。”这种语义的变化特别利于异步编程，因为它提供了一种检查旧值是否是期望值的

方法。将JavaScript的“运行直到结束”保障与“比较与交换”耦合，是确保异步变化的一致性的强有力的方式^[3]。

可以看看实际表现：

```
var c = new CASClass(42);  
// NOTE: notifying observers
```

```
c.swap(42, 43);  
// NOTE: notifying observers  
//=> 43
```

```
c.swap('not the value', 44);  
// Error: No match
```

因此，用基于类的层次结构，可以通过实现一些小的行为，用继承来建立更大的抽象。

9.2.3 改变层级结构

然而，还有一个潜在的问题。如果我想在层次结构中间添加一个新的类型，比如叫ValidatedContainer，该类型可用于添加验证函数，应该添加到哪儿呢？

如图9-5 所示，把ValidatedContainer放在与ObservedContainer平行的地方似乎是合乎逻辑的。

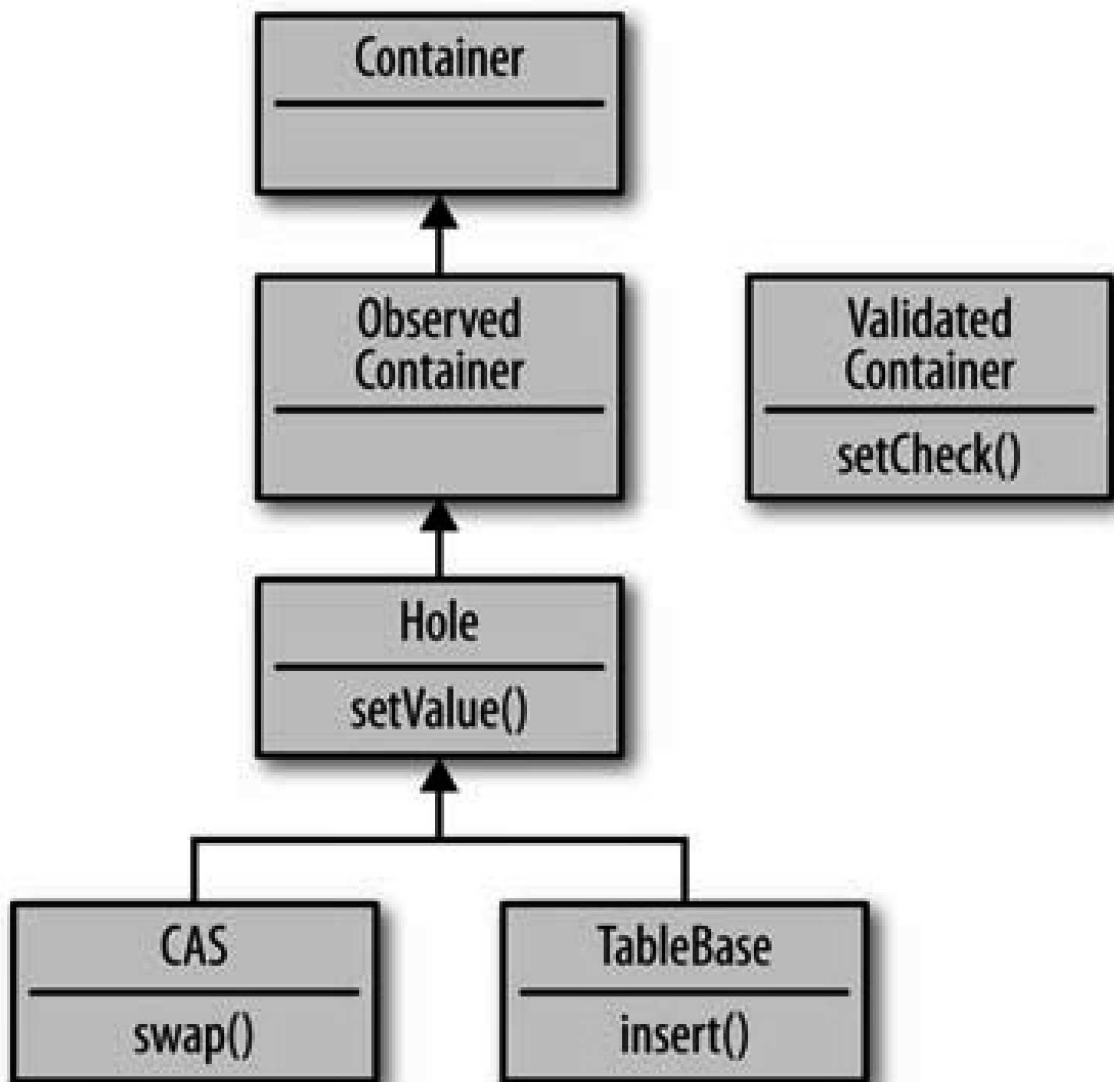


图9-5 扩展层次结构

如果可以给所有的Hole都加上验证，那将再方便不过了，但我并不确定能否这样做（更别提多重继承的问题了）。我当然不想假设用户会希望这样的行为，最好是随时能扩展其功能。例如，CAS类需要校验器，那么我可以把ValidatedContainer放到上层，并从它开始扩展，如图9-6 所示。

但是，如果新的类型需要“比较并交换”语义，但并不需要验证，那么层次结构图（见图9-6 ）是有问题的。绝对不应该强制实现继承自CAS。

类层次结构最大的问题是建立在我们一开始对需要行为的假设上的。也就是说，面向对象技术决定了我们先声明行为的层次结构，然后将我们的类融合到声明中。然而，如ValidatedContainer所示，某些行为很难按逻辑分类。有时候行为只是单纯的行为。

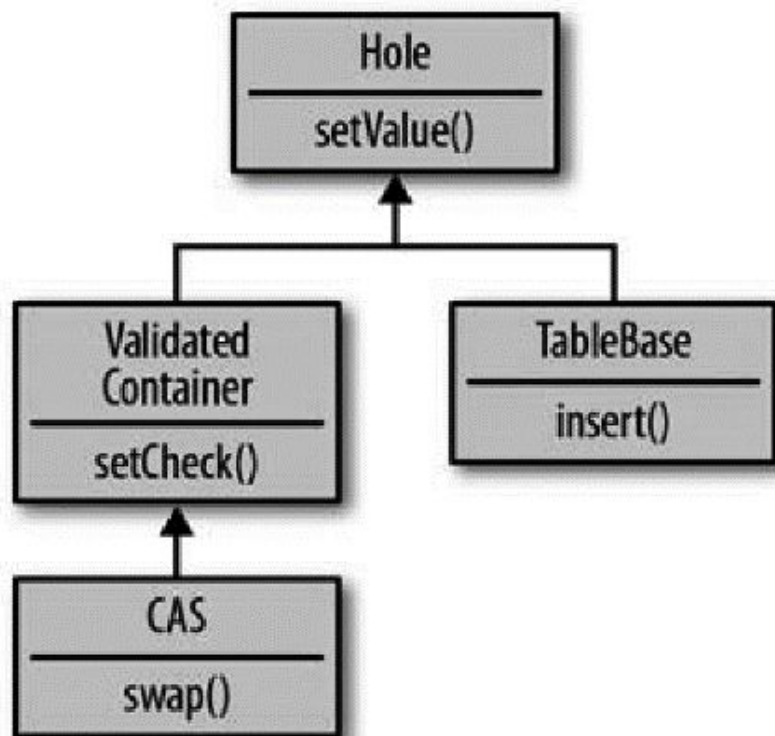


图9-6 将特殊情况的类移动到层次结构下层是很棘手的

9.2.4 用Mixin扁平化层级结构

让我尝试到这里简化一下问题。试想一下，如果我可以把Container基本的功能ObservedContainer，ValidatedContainer和Hole，放在同一层次（见图9-7）。

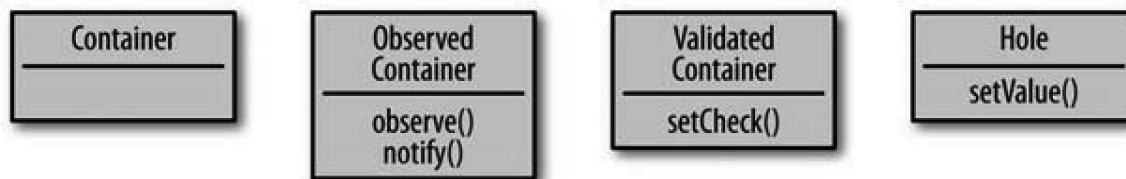


图9-7 扁平化的层次结构

如图9-7 所示，当我们扁平化层次结构时，会导致它们之间没有明显的关系。事实上，这些方框并不真正定义类型。事实上，这里定义的是一组离散的行为，或者说mixins。如果我们只有行为，然后构造新的行为方式是，要么重新定义它们，要么将现成的行为“混合”起来^[4]。这又让人回想起组合现有的函数创建新的函数的想法。

让我们重新实现一遍Container:

```
function Container(val) {  
  this._value = val;  
  this.init(val);  
}
```

```
Container.prototype.init = _.identity;
```

除了调用init方法，这个实现的Container构造函数看起来很像第7章中的。init调用的出现其实定义了一个mixin，该mixin扩展了Container，使客户端与它进行交互。具体而言，Container的mixin协议如下：

(1) 扩展协议

必须提供init方法。

(2) 接口协议

只限构造函数。

通过mixin扩展设计API时，你会经常需要委托给未知函数。这不仅提供了与该类型交互的标准，而且提供了扩展点。以Container为例，init调用委托给Underscore的`_`.identity。之后我会覆盖init，但是现在，先看看如何使用Container：

```
var c = new Container(42);  
  
c;  
//=> {_value: 42}
```

因此，新的Container行为还跟老的一样。但是，我想要做的是创建一个类似但是是新的类型。我想到的类型称为Hole，具有以下语义。

- 持有值。
- 委托一个验证函数来检查设置的值。

- 委托给一个通知功能，如果值变动则发出通知。

我可以将这些语义直接映射到代码中：

```
var HoleMixin = {  
    setValue: function(newValue) {  
        var oldVal  = this._value;  
  
        this.validate(newValue);  
        this._value = newValue;  
        this.notify(oldVal, newValue);  
        return this._value;  
    }  
};
```

HoleMixin#setValue方法定义了一组必须满足的情况，用来保证可以作为Hole的资格。任何Hole的扩展类型都应提供notify和validate方法。实际上并没有Hole类型，只有描述“Hole属性（holiness）”^[5]的mixin。Hole的实现相当简单：

```
var Hole = function(val) {  
    Container.call(this, val);  
}
```

对于Hole构造函数的签名跟Container是一样的；事实上，使用Container.call方法，使得Hole实例的this指针可以确保Container在构造函数中做的事情，都发生在Hole实例上下文中。

HoleMixin的mixin协议规范如下。

(1) 扩展协议

必须提供notify, validate和init方法。

(2) 接口协议

构造函数和setValue。

由于直接使用构造函数中的Container，不需要再使用init方法了。而不符合特定的mixin，尤其是Container的mixin，有可能造成可怕后果：

```
var h = new Hole(42);  
//TypeError: Object [object Object] has no method 'init'
```

Container扩展接口并没有达成一致，意味着任何时候使用Hole都将会失败。但不要绝望；有趣的是，无论是直接还是通过扩展，任何给定的类型都是由现有的mixin组合成的。

根据图9-8 所示，满足Hole类型需要实现或者混合ObserverMixin及ValidateMixin。

既然这些mixin都不存在，那么需要创建它们，从ObserverMixin开始：

```
var ObserverMixin = (function()) {  
    var _watchers = [];  
  
    return {  
        watch: function(fun) {  
            _watchers.push(fun);  
            return _.size(_watchers);  
        },  
        notify: function(oldVal, newVal) {
```



```
_.each(_watchers, function(watcher) {  
    watcher.call(this, oldVal, newVal);  
});  
  
    return _.size(_watchers);  
}  
};  
}());
```

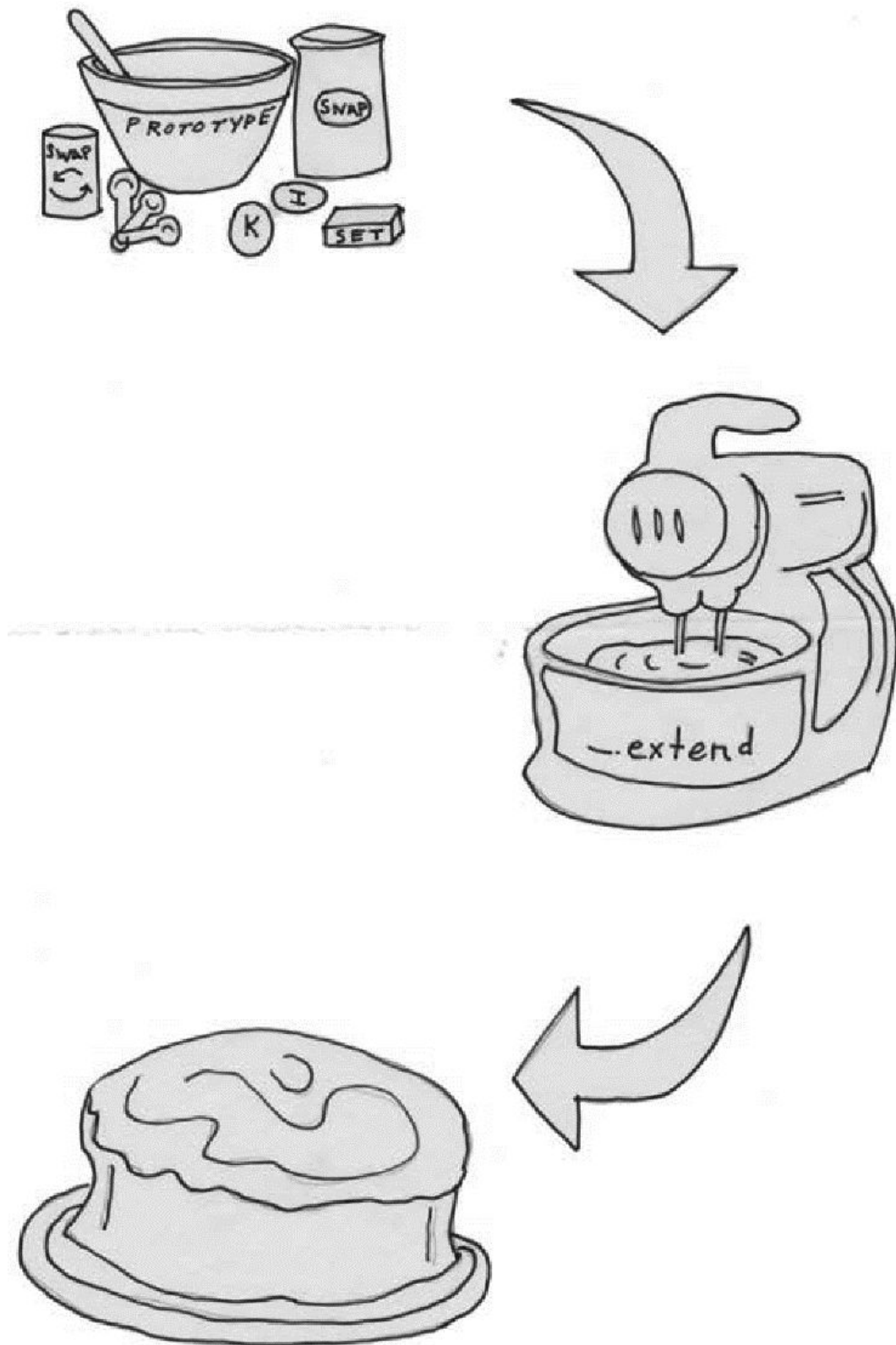


图9-8 使用mixin“混合”行为

使用JavaScript闭包(function() {...}())的魔力来封装_watchers是常用的隐藏数据的方式，因此它也是隐藏mixin状态的首选方式。watch函数接收含有两个值的函数：一个旧值和一个新值，并将其添加到_watchers数组。该watch方法同时也返回存储的watcher的数量。notify方法通过遍历_watchers并调用每个函数，最终返回通知watcher的数量。通过实现ObserverMixin可以加强watch函数的鲁棒性，同时也允许拆除watcher。这部分就当作练习留给读者^[6]。

第二个mixin是ValidateMixin，实现如下：

```
var ValidateMixin = {
  addValidator: function(fun) {
    this._validator = fun;
  },
  init: function(val) {
    this.validate(val);
  },
  validate: function(val) {
    if (existy(this._validator) &&
        !this._validator(val))
      fail("Attempted to set invalid value " + polyToString(val));
  }
};
```

如上所示，ValidateMixin终于满足了init扩展的要求。这样做比较合理，因为有效的初始化步骤是验证容器的起始值。另外两个函数addValidator和validate，设置验证功能，并分别调用（如果已设置）。

既然已经有了mixin，是时候将它们混合在一起以满足Hole类型的要求了：

```
_.extend(Hole.prototype
          , HoleMixin
```

, ValidateMixin , ObserverMixin);

第7章 中提到的Underscore的`_.extend`函数比较棘手，因为它会修改目标对象。然而，在mixin扩展的情况下，这种行为正是我想要的。也就是说，通过使用`_.extend`，我可以将所有的方法拷贝到`Hole.mixin`。那么，完全混合的实现是如何工作的？注意：

```
var h = new Hole(42);
```

构造函数仍然能正常工作。如果我加一个肯定会失败的验证器，会怎么样呢？

```
h.addValidator(always(false));
```

```
h.setValue(9);
```

```
// Error: Attempted to set invalid value 9
```

由于添加了一个对所有情况都返回的`false`的验证器，无法再次设置其他值，除非删除验证功能。然而，让我创建一个验证限制较少的新`Hole`实例：

```
var h = new Hole(42);
```

```
h.addValidator(isEven);
```

新的实例只允许偶数的值：

```
h.setValue(9);  
// Error: Attempted to set invalid value 9
```

```
h.setValue(108);  
//=> 108
```

```
h;  
//=> {_validator: function isEven(n) {...},  
//   _value: 108}
```

Hole的实例h只允许设置偶数的值。下面我用watch方法添加watcher:

```
h.watch(function(old, nu) {  
    note(["Changing", old, "to", nu].join(' '));  
});  
//=> 1
```

```
h.setValue(42);  
// NOTE: Changing 108 to 42  
//=> 42
```

传入偶数42表明watcher被调用，所以再增加一个也一样可以:

```
h.watch(function(old, nu) {  
    note(["Veranderende", old, "tot", nu].join(' '));  
});  
//=> 2
```

```
h.setValue(36);  
// NOTE: Changing 42 to 36  
// NOTE: Veranderende 42 tot 36  
//=> 36
```

所以我已经成功通过使用两个构造函数调用创建了一个新的JavaScript类型，并通过混合离散的数据包合成一个连贯的hole……我的意思是whole^[7]。在下一节中，我将讨论如何使用mixin扩展现有类型新的功能。

9.2.5 通过Mixin扩展新的语义

添加新的功能到现有的JavaScript类型实在不能太简单了；只需要污染prototypeKABOOM，就可以附加上新的行为。KABOOM这里其实代表操作，因为实际上没有这么简单。扩展现有类型并不总是这么简单直白，因为你永远不知道是否会打破一些微妙的内部平衡。请牢记这一点，我将探讨如何扩展Hole类型的功能，包括新的语义变化。首先，我喜欢将setValue方法作为一种低层次的方式来插入变化机制的想法。不过，我想提出另一个方法swap，它接收一个函数和一些参数，并基于调用结果设置新值。解释该想法的最好办法是给出实现的例子：

```
var SwapMixin = {
  swap: function(fun /* , args... */) {
    var args = _.rest(arguments)
    var newValue = fun.apply(this, construct(this._value, args));

    return this.setValue(newValue);
  }
};
```

SwapMixin的swap方法确实需要一个函数和一些参数。然后用给出的函数计算_value和其他参数得出新的值。SwapMixin的mixin协议规范如下。

(1) 扩展协议

必须提供setValue方法和_value属性。

(2) 接口协议

swap方法。

其实我可以分开测试SwapMixin：

```
var o = {_value: 0, setValue: _.identity};

_.extend(o, SwapMixin);

o.swap(construct, [1,2,3]);
//=> [0, 1, 2, 3]
```

如上所示，该swap mixin似乎符合逻辑。在我用它来增强Hole之前，我想实现另一个mixin: SnapshotMixin，它提供一种从Hole实例取值的安全方法：

```
var SnapshotMixin = {
  snapshot: function() {
    return deepClone(this._value);
  }
};
```

SnapshotMixin提供了一个新的名为snapshot的方法，该方法克隆其所在对象。现在，Hole新的规范为：

```
_.extend(Hole.prototype
, HoleMixin
, ValidateMixin
, ObserverMixin
, SwapMixin
, SnapshotMixin);
```

从现在开始，任何新的Hole实例都将具有增强的行为：

```
var h = new Hole(42);
```

```
h.snapshot();
```

```
//=> 42
```

```
h.swap(always(99));
```

```
//=> 99
```

```
h.snapshot();
```

```
//=> 99
```

Mixin扩展不仅是定义新类型强有力的方式，同时也可以增强现有类型。请记住它并不总是直接扩展现有的类型，而且额外的任何扩展都是对全局起作用。

9.2.6 通过Mixin混合出新的类型

现在，我已经展示了如何定义两个基本类型（Container和Hole），让我实现一个叫CAS的类型，提供“比较和交换”语义。也就是说，任何类型

的变化都发生在你已知道现有的值会发生什么的前提下。从Hole的构造函数开始定义：

```
var CAS = function(val) {  
    Hole.call(this, val);  
}
```

CASMixin定义有趣的部分是，它覆盖了SwapMixin的swap方法：

```
var CASMixin = {  
    swap: function(oldVal, f) {  
        if (this._value === oldVal) {  
            this.setValue(f(this._value));  
            return this._value;  
        }  
        else {  
            return undefined;  
        }  
    }  
};
```

CASMixin#swap方法接收两个参数，而SwapMixin接收一个。此外，如果预期值与实际_value不符，CASMixin#swap方法将返回undefined。有两种方法来混合CAS类型的实现。首先，我只需要放弃SwapMixin而用CASMixin代替，因为我知道，swap方法是唯一的替代品。不过，我会修改_.extend顺序来处理重载：

```
_.extend(CAS.prototype  
        , HoleMixin  
        , ValidateMixin  
        , ObserverMixin  
        , SwapMixin  
        , CASMixin  
        , SnapshotMixin);
```

虽然我知道SwapMixin完全包括在CASMixin内，但保留下来也不完全是坏事。原因是，如果不控制SwapMixin，那么以后想扩展增强swap方法就变得容易了。将其留在扩展链，我可以在未来随意扩展。如果我不喜欢未来“增强”功能，那么我可以选择以后删除SwapMixin。作为本节的总结，展示一下CAS类型的使用方式：

```
var c = new CAS(42);
```

```
c.swap(42, always(-1));
```

```
//=> -1
```

```
c.snapshot();
```

```
//=> -1
```

```
c.swap('not the value', always(100000));
```

```
//=> undefined
```

```
c.snapshot();  
//=> -1
```

这总结了mixin扩展的讨论。不过，这里还要声明一点：如果处理得当，mixin扩展是一个实现细节。事实上，基于mixin编程仍然会碰到像基本类型、数组和对象（如map）的简单数据。具体来说，我发现在处理大量的数据元素，那么简单的数据是最好的，因为你可以使用普通的工具和函数来处理它，用越通用的数据处理工具则越好。另外，你一定会找到一个需要建立高度特化类型，以及良好定义的接口驱动的类型语义^[8]。我也是通过这些特化类型的用例发现基于mixin开发的真正优势。

简单的数据是最好的。特化数据类型就应该特别。

9.2.7 方法是低级别操作

这在上一节中创建的类型是以对象/方法为中心的技术细节，不是函数式API。正如我在本书中强调的，如果创建得好，函数式API是可组合的，而且不需要了解组合的中间类型。因此，通过简单地创建用于访问和操作的容器类型的函数的API，我可以隐藏大部分执行细节。

首先，从容器开始：

```
function contain(value) {  
    return new Container(value);  
}
```

简单吧？如果我提供了一个容器库，那么我会提供contain函数作为面向用户的API：

```
contain(42);  
//=> {_value: 42} (of type Container, but who cares?)
```

对于开发者，可能还需要提供mixin定义以便扩展用。

Hole的函数式API没有什么变化，但比较强大：

```

function hole(val /*, validator */) {
  var h = new Hole();
  var v = _.toArray(arguments)[1];
  if (v) h.addValidator(v);

  h.setValue(val);

  return h;
}

```

我已经成功封装了大量hole函数的验证逻辑。这是理想的情况，因为我可以随意编写下层的方法。使用hole函数的约定比使用Hole构造函数和addValidator方法的组合要简单得多：

```

var x = hole(42, always(false));
// Error: Attempted to set invalid value 42

```

同样，虽然setValue是该类型的方法，但没有理由暴露其功能，而只需要swap和snapshot函数来替代：

```

var swap = invoker('swap', Hole.prototype.swap);

swap函数跟很多invoker绑定方法一样，用目标对象作为第一个参数：

var x = hole(42);

swap(x, sqr);
//=> 1764

```

暴露的CAS类型的功能与Hole非常相似。

```
function cas(val /*, args */) {  
  var h = hole.apply(this, arguments);  
  var c = new CAS(val);  
  c._validator = h._validator;  
  
  return c;  
}
```

```
var compareAndSwap = invoker('swap', CAS.prototype.swap);
```

我使用（可能是滥用）Hole类型的私有细节来实现大部分CAS函数的功能，但因为我可以控制这两种类型的代码，强耦合好像也是合理的。一般情况下，我会避免这种情况发生，尤其是如果滥用类型不在控制范围内。

最后，我现在用泛型委托实现余下的容器功能：

```
function snapshot(o) { return o.snapshot() }  
function addWatcher(o, fun) { o.watch(fun) }
```

这些函数都可以正常工作：

```
var x = hole(42);

addWatcher(x, note);

swap(x, sqr);
// NOTE: 42 chapter01.js:38
//=> 1764

var y = cas(9, isOdd);

compareAndSwap(y, 9, always(1));
//=> 1

snapshot(y);
//=> 1
```

我相信，通过给容器类型加上函数式的外壳，我已经获得了对象/方法模式所达不到的灵活性。

9.3 总结

本章总结了如何用JavaScript函数式编程方式构建软件。即使有些看似是对象或类的问题，很多时候也可以用函数式的方式来达到同样的目标。不仅可以用函数式方式来构建系统的一部分，还可以通过不捆绑你的用户到以对象为中心的API，以建立功能更加灵活的系统。

同样，即使当一个问题需要对象的思想，从函数式的角度导致的解决方案会完全不同于面向对象的程序设计。如果函数式组合被证明为更有用，对象组合怎么办？在本章中，我讨论了基于mixin的设计以及它如何将对象组合转化为函数式风格。

写这本书，对我来说一直很有趣。我希望整个过程都对你富有启发。不应该只把函数式编程当作学习目标，而是作为实现自己的目标的技术。可能有的时候，它不是最佳选择，但即使这样，函数式的思维还是有助于改变你的通用构建软件的方式。

[1] 私有化的chain数组导致链接多个惰性链稍微更复杂。然而，要处理这种情况，需要让force识别并传递结果给下一惰性链。

[2] 为了方便，我已经委派给了JSON.stringify，因为本节重点不是如何将对象转换为字符串。

[3] 简言之，run-to-completion是指JavaScript的事件循环的一个属性。也就是说，在事件循环的某个特定“tick”运行时的调用都会保证在下个“tick”之前结束。这本书不是关于事件循环的。我推荐大卫·弗拉纳根的《JavaScript权威指南》（第6版），其中进行了全面潜入JavaScript事件系统（或其他JavaScript内容）。

[4] 本章提到的mixin是“协议”以及模板方法的设计模式的叠加，不包含继承。

[5] 译者注：作者用holiness双关，这里代表Hole的属性，但holiness原意为神圣。

[6] ECMAScript.next中描述的Object.observe方法，其工作方式类似于本文中所描述的这个特征。这个规范应该会在太阳燃尽前成为

JavaScript的核心实现。更多信息可参考

<http://wiki.ecmascript.org/doku.php?id=harmony:observe> 。

[7] 译者注：作者的双关，合成一个整体（Whole），又指合成一个新类型Hole。

[8] 如果你有Scala的背景，那么这里介绍的基于mixin的开发还远未实现知名蛋糕模式（Wampler 2009）。然而，通过运行时mixin检查，可以大概达到大型模块定义的功能。

附录A 更多函数式JavaScript

本书并不能代表JavaScript函数式编程原始的思想。多年以来，自从有了JavaScript，人们已经将其推向函数式风格。在本附录中，我将试图简单地总结一下函数式JavaScript的用例以及库。出现顺序不代表排名。

A.1 JavaScript的函数式库

其实有无数值得注意的JavaScript库。我将展示一些高级特征，并提供几个例子。

A.1.1 Functional JavaScript

Oliver Steele的Functional JavaScript库，是我发现的第一个函数式库。它提供了常用的高阶函数如map，但它提供的是非常有趣的基于字符串的简短形式的函数格式。也就是说，要对数组中的数字做乘方，通常会这样写：

```
map(function(n) { return n * n }, [1, 2, 3, 4]);  
//=> [2, 4, 9, 16]
```

然而，Functional JavaScript库的函数可以写成字符串：

```
map('n*n', [1, 2, 3, 4]);  
//=> [2, 4, 9, 16]
```

Functional JavaScript还可以柯里化这种字符函数：

```
var lessThan5 = rcurry('<', 5);  
lessThan5(4);  
//=> true  
  
lessThan5(44);  
//=> false
```

Functional JavaScript利用高超的JavaScript元编程技术，非常值得探索。

A.1.2 Underscore-contrib

很久以前我写了一个叫Doris的函数式JavaScript库，其深受Steele的Functional JavaScript和Clojure语言的影响。我自己用了Doris一段时间，但最终转向Underscore和ClojureScript。在写这本书时，我又重新翻出Doris的源代码，移植到Underscore，并清理了代码，更名为Lemonad（读作lemonade），然后将大部分的功能集成到Underscore-contrib库。

Underscore-contrib为Underscore提供数十种有益的应用性、高阶和monadic函数。当导入Underscore-contrib时，所有函数会混入到Underscore的_对象，将Underscore发挥到极致。除了核心函数，我已经实现了一些“额外”的 Underscore-contrib，包括以下内容。

(1) Codd

一个关系代数库。

(2) Friebyrd

提供嵌入式逻辑系统的库。

(3) Minker

提供嵌入式数据记录的库。

```
var a = ['a', 'a', 'b', 'a'];  
var m = _.explode("mississippi");
```

```
_.frequencies(a)  
//=> {a: 3, b: 1}
```

```
_.frequencies(m)  
//=> {p: 2, s: 4, i: 4, m: 1}
```

还有许多好用的函数。幸运的是，大多数在这本书中定义的函数都能在Lemonad或Underscore-contrib中找到，可以把本身看成非官方的说明书。

A. 1.3 RxJS

微软的Reactive Extensions for JavaScript (RxJS) 是异步事件驱动编程模型库。RxJS使用Observable抽象，它允许你通过一个丰富的如LINQ般的功能查询模型处理异步数据流。

当我还年轻的时候，我花了很多时间玩任天堂NES系统。日本的Konami公司创造了许多有趣的游戏，但其中我最喜欢的要属魂斗罗。魂斗罗的目标是……好吧，没人关心。有趣的是，你可以输入获得30个额外的生命作弊代码。作弊代码描述如下：

```
var codes = [  
    38, // up  
    38, // up  
    40, // down  
    40, // down  
    37, // left  
    39, // right  
    37, // left  
    39, // right  
    66, // b  
    65  // a  
];
```

这段金手指代码可以进入游戏开始前输入，这也是我能玩通关魂斗罗的唯一原因。如果你想把这段Konami码添加到一个网页，那么你可以用RxJS来做。RxJS中有能方便比较序列的方法sequenceEqual，可以用来检查Konami码相符：

```
function isKonamiCode(seq) {  
    return seq.sequenceEqual(codes);  
}
```

RxJS可以让你挖掘到异步事件，包括页面文档的按键、许多来源，如下所示：

```
var keyPressStream = $(document).keyupAsObservable()  
    .select(function (e) { return e.keyCode })  
    .windowWithCount(10, 10);
```

keyPressStream代表按键事件建立的键码流。比起观察每一次按键，RxJS可以用windowWithCount方法把流截成块。还有一点是，RxJS点缀了jQuery本身有关Observable创建的方法，并且应用到其他各种JavaScript框架中，能与现有的库无缝集成。

现在，有了键码流，我可以声明告诉RxJS要做什么：

```
keyPressStream  
    .selectMany(isKonamiCode)  
    .where(_.identity)  
    .subscribe(function () {  
        alert("You now have thirty lives!");  
    });
```

值得注意的是，where方法可以改变前进的道路上的数据值，但我选择让数据经过_.identity函数。subscribe^[1]方法中的函数会在selectMany返回truthy时执行。如果我要将前面的代码加载到一个网页，并推动其次是字符“a”和方向键的正确顺序“B”，那么警告框将推出。

RxJS是很好的库，它提供了将异步流捕获成值的方式——一个真正令人费解的范式。

A. 1.4 Bilby

Bilby将Lemonad升级到了新的高度。Brian McKenna的自包含函数式库Bilby扩展了JavaScript中函数式风格的可能性。Bilby的所有函数集都十分值得通读，其中多重方法的实现非常好。

Bilby的多重方法与第5章中定义的函数dispatch非常类似，但更为强大和灵活。通过Bilby，可以定义任意数量的对应条件分发的函数。Bilby还提供聚合相关的方法和属性的模块系统environment：

```
var animals = bilby.environment();
```

加入多重方法之前，我可以定义一些辅助函数：

```
function voice(type, sound) {  
    return ["The", type, "says", sound].join(' ');  
}
```

```
function isA(thing) {  
    return function(obj) {  
        return obj.type == thing;  
    }  
}
```

```
function say(sound) {  
    return function(obj) {  
        console.log(voice(obj.type, sound));  
    }  
}
```

使用这些辅助函数，我可以告诉Bilby：

- 该方法的名称；
- 检查参数的谓词函数；
- 一个动作函数用来执行该方法的行为。

Environment#method只需三个参数：

```
var animals = animals.method('speak', isA('cat'), say("mew"));
```

装饰好的environment会返回一个新的环境。现在可以调用speak：

```
animals.speak({type: 'cat'});  
// The cat says mew
```

添加一个新的多态行为很简单：

```
var animals = animals.method('speak', isA('dog'), say("woof"));
```

对狗类型调用speak也按预期工作：

```
animals.speak({type: 'cat'});  
// The cat says mew
```

```
animals.speak({type: 'dog'});  
// The dog says woof
```

当然，我可以给谓词函数加任意条件：

```
var animals = animals.method('speak',  
  function(obj) {  
    return (isA('frog')(obj) && (obj.status == 'dead'))  
  },  
  say('Hello ma, baby!'));
```

所以传进去一个死的青蛙也同样可以工作：

```
animals.speak({type: 'frog', status: 'dead'});  
// The frog says Hello ma, baby!
```

Bilby 提供的远比多重方法要多，包括可以返回函数的蹦床、monadic 结构、验证器等。

A.1.5 allong.es

Reginald Braithwaite的allong.es库有一堆有用的函数组合子。但是，从我（可能不是很深入）的角度看，有趣的方面是它对有状态的迭代器的支持：

```
var iterators = require('./allong.es').iterators  
var take = iterators.take,  
    map = iterators.map,  
    drop = iterators.drop;  
  
var ints = iterators.numbers();
```

除了引入allong.es的迭代函数，我还定义了一个数字迭代器ints。然后就可以“操作”ints迭代器：

```
var squares = take(drop(map(ints, function(n) {  
    return n * n;  
})), 100000), 100);
```

这里我对所有整数取平方，去掉前100 000个结果，只取之后的100个。allong.es迭代器的神奇之处在于，我实际上还没有执行任何计算。只有当我用外部迭代器去查询时（查询迭代器for将触发所有计算）：

```
var coll = [];  
for (var i = 0; i < 100; i++ ) {  
    coll.push(squares())  
}
```

```
coll;  
//=> [10000200001,  
//     10000400004,  
//     10000600009,  
//     10000800016,  
//     10001000025,  
//     10001200036,  
//     ...  
//     10020010000]
```

我可以通过手动取100 001的平方（还记得我去掉了前100 000吗？）：

```
100001 * 100001  
//=> 10000200001
```

手动计算的答案与coll阵列第一个元素是匹配的。allong.es（和对一般迭代器）能做的实在太多了，我建议读者继续探索。

A. 1.6 其他函数式库

有越来越多的不同程度支持函数式编程的JavaScript库。jQuery一直有函数式的部分，而且引入了promises。我一直关注的Reduces项目，实现了一个广义的可reduce的集合API，其灵感来自Clojure的reducer。Lo-Dash项目是Underscore的一个主要分支，尝试用简洁的内核并提供更高的性能。David Nolen的Mori项目模拟ClojureScript的核心库，包括它的持久数据结构。Udon是类似于Lemonad的简单函数式库。最后，prelude.ls也是一个简单的函数式库。然而，prelude.ls的不同之处在于，它起初是由TypeScript编译成JavaScript^[2]。

A.2 能编译成JavaScript的函数式语言

如果函数式库都不能使之简单，越来越多的程序员在使用JavaScript作为其编制目标的新语言。我只列出很少一部分我熟悉的语言。下面这些基本是我用在真正的项目或贡献过的，或是平时闲暇时自学的语言。

A.2.1 ClojureScript

ClojureScript 编程语言是能编译为 JavaScript 的 Clojure 的一个变体。它有许多Clojure的特性，包括但不限于。

- 持久化数据结构。
- 引用类型。
- 命名空间。
- 强大的JavaScript的互操作。
- 惰性。
- 解构赋值。
- 协议，类型及记录。

ClojureScript大概是这样的：

```
(defn hi [name]
  (.log js/console (str "Hello " name "!")))

(hi "ClojureScript")

;; (console) Hello ClojureScript
```

ClojureScript针对大型JavaScript应用来说是门很好的语言。事实上，我已经开始用Pedestal web框架来编写健壮的单页面应用程序^[3]。读者可以通过我的另一本书*The Joy of Clojure*第2版发掘ClojureScript的乐趣。

A.2.2 CoffeeScript

CoffeeScript是一种流行的编程语言，它用简明的语法阐明了“JavaScript精粹”。这个“Hello World”的例子简直微不足道：

```
hi = (name) ->
  console.log ['Hello ', name, '!'].join ' '

hi 'CoffeeScript'

# (console) Hello CoffeeScript
```

它对于JavaScript的附加功能包括。

- 文学编程（Literate programming）支持（我非常喜欢）。
- 可变参数。
- 列表解析。
- 解构赋值。

它的函数式编程的支持比JavaScript更好，能写出函数式风格的更简洁的代码。

A. 2. 3 Roy

Roy是一个静态类型的函数式编程语言，早期阶段的灵感来自ML。Roy提供了许多ML家族语言常见的特性，包括模式匹配、结构类型，以及标记并集（tagged unions）。我最感兴趣的是它的类型系统。如果我试着去连接JavaScript的字符，会得到意想不到的惊喜：

```
let hi name: String =
  alert "Hello " + name + "!"

// Error: Type error: String is not Number
```

Roy保留了+运算符只做数学运算，不允许串联操作。然而，Roy提供了++运算符：

```
let hi name: String =
  console.log "Hello " ++ name ++ "!"
```

调用hi函数非常简单：

```
hi "Roy"
```

```
// Hello Roy!
```

我将持续关注Roy的进展，并希望看到更多好的东西。

A.2.4 Elm

跟Roy一样，Elm是一种静态类型的语言，能编译为JavaScript。另外，Elm不允许使用+进行字符串连接，如下所示：

```
hi name = plainText ("Hello " + name + "!")
```

```
-- Type error (Line 1, Column 11):  
-- String is not a {Float,Int}  
-- In context: + "Hello "
```

还有，Elm保留了++函数用于这样的用途：

```
hi name = plainText ("Hello " ++ name ++ "!")
```

```
main = hi "Elm"
```

```
-- (page text) Hello Elm!
```

然而，Elm跟Roy不同的是，它不仅仅是一种编程语言，也是用于开发的系统。也就是说，Elm是以函数式响应式编程（Functional Reactive Programming, FRP）范式为中心的语言。简而言之，FPR为达到构建健壮的响应系统范围的变化而集成了时间模型以及事件系统。单单用这一页永远无法充分覆盖FRP，估计要写一本书才足够。如果你想扩展一下思路，Elm是个不错的选择，即使只是练习。

[1] 译者注：作者原文笔误写成了select方法。

[2] 译者注：其实是LiveScript的函数式库，参见<http://livescript.net>。

[3] 前身叫Pedestal，官网是<http://pedestal.io/>。

作者简介

Michael Fogus在分布式仿真、机器视觉和专家系统建设方面经验丰富。他主要活跃于Clojure和Scala社区。他是《Clojure编程乐趣》一书的作者。

封面介绍

函数式JavaScript封面的动物是绒鸭（*Somateria mollissima*），长度在50~70厘米之间的海鸭。绒鸭活动在欧洲、北美海岸和西伯利亚的东海岸。他们会在冬天迁移到北方温带地区。绒鸭会以每小时113千米（70英里）的速度飞行。

雌性绒鸭会用胸部的羽毛筑巢，通常建在海洋附近。绒鸭毛通常会用作枕头和被子填充物。为了可持续利用，通常会在绒鸭放弃巢之后收集这些鸭绒。近年鸭绒被合成替代品和农产品鹅绒取代。

雄性绒鸭的特点是黑白羽毛相间，绿色的脖子；雌性绒鸭则是棕色的。在一般情况下，绒鸭体型笨重、大，有着楔形的鸟喙。它们以甲壳类动物和软体动物为食。对于它们青睐的食物如贻贝，是整个吞下，在胃部将壳压碎。

该物种在北美和欧洲的数量在1.5万~200万只；而在西伯利亚东部的数量很大，但具体数目尚不清楚。在英格兰诺森伯兰郡的法尔群岛，绒鸭在676年成为第一个鸟类保护法的保护对象。该法律由诺森伯兰郡的守护神Saint Cuthber建立，并且给绒鸭命名为“Cuddy的鸭子”（“Cuddy”是“Cuthbert”的昵称）。

在20世纪90年代，由于不断变化的冰川，绒鸭在加拿大的哈德逊湾相继死亡。据加拿大野生生物服务机构收集的数据，之后数量有所回升。

附录B 推荐书目

论文/书籍/博客文章/会谈

计算机程序的构造和解释, Harold Abelson, Gerald Jay Sussman, Julie Sussman (MIT Press, 1996)

这本书是最有影响力的编程书籍。每一页是一个宝石, 所有句子都是亮点。非常值得关注和研究。

解析极限编程: 拥抱变化, Kent Beck (Addison-Wesley, 1999)

引人入胜的书, 阐明了革命性的编程原则。

函数式编程简介, Richard J. Bird and Philip Wadler (Prentice Hall, 1998)

我更喜欢第一个版本。

Closure: 权威指南, Michael Bolin (O'Reilly, 2010)

Bolin对JavaScript的伪类的继承的思想对我影响深刻。

JavaScript Allongé, Reginald Braithwaite (Leanpub, 2013)

我有幸读到这本伟大的书的初稿, 这将是本书一个不错的后续读物。将函数式JavaScript推到了极致。

JavaScript精粹, Douglas Crockford (O'Reilly, 2008)

Crockford的书就像是一个写得很好恐怖电影。它是编程书籍中的《阴风阵阵》。它会让你做恶梦, 但你无法把目光移开。

数据库系统导论, C. J. Date (Addison-Wesley, 2003)

一定要读。

SQL与关系理论：如何编写准确SQL代码, C. J. Date (O'Reilly, 2011)

一个惊人的书，真正了解关系代数以及我们写的查询为何如此之慢的原因。

JavaScript：权威指南，第6版 David Flanagan (O'Reilly, 2011)

在我看来是JavaScript的终极书。

领域特定语言, Martin Fowler (Addison-Wesley, 2010)

一个深邃的作家和思想家的深刻的话题。

设计模式：可复用面向对象软件, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995)

深受喜爱和嘲笑，这本书原本是寻找描述系统的建设的通用语言，值得一读。

Java并发实践, Brian Goetz等。(Addison-Wesley, 2005)

如果你曾经打算写Java代码，那绝对的必读。

On Lisp , Paul Graham (Prentice Hall, 1993)

被许多人认为是权威Lisp书籍。

Effective JavaScript：编写高质量JavaScript代码的68个有效方法, David Herman (Addison-Wesley, 2012)

像JavaScript Allongé, Herman的书也是本书的扩展读物。

Clojure编程乐趣, Chris Houser, Michael Fogus第二版 (Manning, 2013)

函数式JavaScript的另一个目标是提供能平滑的过渡到理解Clojure乐趣的知识。

Hints for Computer System Design {0}, Butler W. Lampson
(Xerox Palo Alto Research Center, 1983)

Lampson很大的影响了现代编程，即使你可能从来没有听说过他的名字。

ML程序设计教程，第二版，L. C. Paulson (剑桥大学出版社，1996)

通过阅读ML能为你理解函数式JavaScript带来什么呢？事实证明有很多益处。

Applicative High Order Programming: Standard ML in Practice , Stefan Sokolowski (Chapman & Hall Computing, 1991)

一个被遗忘的宝藏。

JavaScript模式，Stoyan Stefanov (O’Reilly, 2010)

这里的模式不是“设计模式”中的概念，而是JavaScript程序常用结构的模式。一个很不错的读物。

Common Lisp: A Gentle Introduction to Symbolic Computation, David S. Touretzky (Addison-Wesley/Benjamin Cummings, 1990)

通过阅读Lisp能为你理解函数式JavaScript带来什么呢？事实证明有很多益处。

Programming Scala, Dean Wampler and Alex Payne
(O’Reilly, 2009)

一个写得很好的Scala书，提供免费在线阅读。

高性能JavaScript，Nicolas Zakas (O’Reilly, 2010)

加速你的功能抽象的基本阅读。

演讲

“Pushing The Limits of Web Browsers...or Why Speed Matters” , *Lars Bak*

2012年Strange Loop大会的特邀主题演讲。Bak是几十年来语言速度优化的驱动力。

“*Programming with Values in Clojure*” , *Alan Dipert*

2012年的Clojure / West会议的专题介绍。

“*The Next Mainstream Programming Language: A Game Developer's Perspective*” , *Tim Sweeney*

2006年Symposium on Principles of Programming Languages会议的专题介绍。

博客文章

Can functional programming be liberated from the von Neumann paradigm?, *Conal Elliott*

在探索如何以及为什么I / O破坏函数式的理想的描述性。

Markdown, John Gruber

Markdown越来越普及了。

Rich Hickey Q&A, Rich Hickey, Michael Fogus. Code Quarterly 2011.

充满了关于编程，设计和语言和系统的宝石。

Monads are Tress with Grafting, *Dan Piponi*

这篇文章帮助了我理解Monad。不过情况因人而异。

Simple JavaScript Inheritance, John Resig

虽然我不喜欢层次建设，Resig的实现是非常干净和有启发。

Understanding Monads With JavaScript, Ionut G. Stan

Stan的Monad实现帮助了我对Monad的理解。此外，actions的实现是从他的代码派生的。

Execution in the Kingdom of Nouns, Steve Yegge

Yegge将面向对象vs函数式编程比作动词vs名词。虽然他的观点是值得商榷的，但是还是很形象的。

Maintainable JavaScript: Don't modify objects you don't own, Nicholas Zakas

Zakas对JavaScript风格的思考了相当长一段时间。

杂志文章

“*Why functional programming matters*”，John Hughes. The Computer Journal (1984)

在明确的论述函数式编程。虽然给出的例子不怎么好，散文倒是非常值得一读。

欢迎来到异步社区！

异步社区的来历

异步社区(www.epubit.com.cn)是人民邮电出版社旗下IT专业图书旗舰社区，于2015年8月上线运营。

异步社区依托于人民邮电出版社20余年的IT专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与POD按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。

异步社区 技术圈 图书 电子书 文章 图书/作者/类别

我们一岁啦

异步社区成立一周年大型活动开启

周年庆满减促销 | 满100元减20元、满150元减35元、满200元减50元

CCIE路由和交换认证考试指南 (第5版) (第1卷)

数据科学实战手册 (R+Python)

软技能: 代码之外的生存指南

Python密码学编程

Python游戏编程快速上手

机器学习项目开发实战

树莓派Python编程入门与实践 (第2版)

像计算机科学家一样思考Python (第2版)

我要写书

近期活动

异步社区成立一周年大型赠书活动开启！
异步社区的来历 异步社区是人民邮电出版社旗下IT专，业图书旗舰社区，于2015年8月上线运营。异步社区依托于人民邮电出版社20余年的IT专业...

猫叔郭志敬 2016-08-02
阅读 575 推荐 2 收藏 0 评论 8

2016 iWeb峰会北京站即将开启，为HTML5喝彩！
每一次振臂高呼辐射行业的影响，每一天无数人兢兢业业的勤奋，2016雄起！未吧，8月27日，HTML5峰会北京站，我在这里，等你来，为HTML5喝彩！...

猫叔郭志敬 2016-07-29
阅读 60 推荐 1 收藏 0 评论 0

每周半价电子书

树莓派Python编程入门与实践 (第2版)
[美] Richard Blum 勃鲁姆, Christine Bresnahan 布莱斯纳罕 (作者) 陈晓明 马立新 (译者)

社区里都有什么？

购买图书

我们出版的图书涵盖主流IT技术，在编程语言、Web技术、数据科学等领域有众多经典畅销图书。社区现已上线图书1000余种，电子书400多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

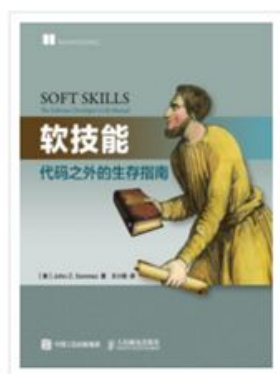
用户帐户中的积分可以用于购书优惠。100积分=1元，购买图书时，在 里填入可使用的积分数值，即可扣减相应金额。

特别优惠

购买本电子书的读者专享异步社区优惠券。使用方法：注册成为社区用户，在下单购书时输入“57AWG”，然后点击“使用优惠码”，即可享受电子书8折优惠（本优惠券只可使用一次）。

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。



软技能：代码之外的生存指南

[美]约翰 Z. 森梅兹 (John Z. Sonmez) (作者)

王小刚 (译者)

杨海玲 (责任编辑)



分享

6

推荐



想读

9.0K

阅读

这是一本真正从“人”（而非技术也非管理）的角度关注软件开发人员自身发展的书。书中论述的内容既涉及生活习惯，又包括思维方式，凸显技术中“人”的因素，全面讲解软件行业从业人员所需知道的所有“软技能”。

本书聚焦于软件开发人员生活的方方面面，从揭秘面试的流程到精耕细作出一份杀手级简历，从创建大受欢迎的博客到打造你的个人品牌，从提高工作效率到与“拖延症”做斗争，甚至包括如何投资不动产，如何关注自己的健康。

本书共分为职业篇、自我营销篇、学习篇、生产力篇、理财篇、健身篇、精神篇等七篇，概括了软件行业从业人员所需的“软技能”。

● 纸质版 ¥59.00 **¥46.02 (7.8折)**

○ 电子版 **¥35.00**

○ 电子版 + 纸质版 **¥59.00**

现在购买

下载PDF样章

配套文件下载

社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得100积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于Markdown的写作环境，喜欢写作的您可以在此一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握IT圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：



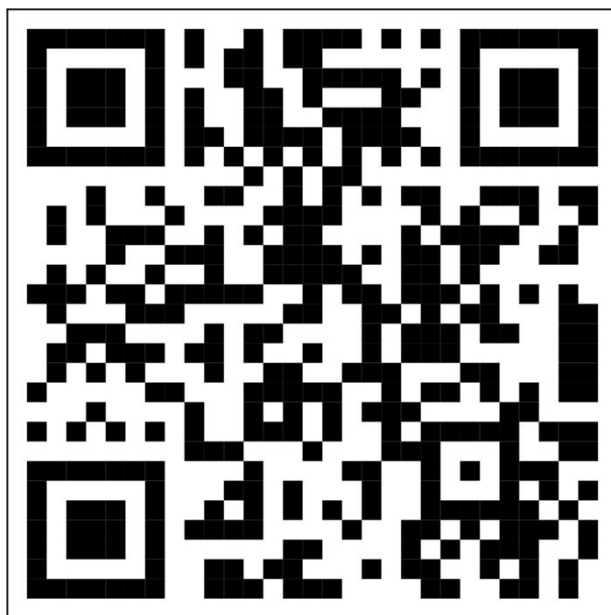
异步社区



微信订阅号



微信服务号



官方微博



QQ群: 368449889

社区网址: www.epubit.com.cn

官方微信: 异步社区

官方微博: @人邮异步社区, @人民邮电出版社-信息技术分社

投稿&咨询: contact@epubit.com.cn