Broadview
www.broadview.com.cn

前端

函数式攻城指南

欧阳继超 / 著

作为Ramda.js的作者之一，我非常喜欢Braithwaite的*JavaScript Allonge*，喜欢Fogus的*Functional JavaScript*，我非常激动又有一本关于JavaScript的函数式书籍，希望尽快能见到这本书的英文版。

CrossEye

中国工信出版集团 電子工業出版社

内 容 简 介

在后端，函数式语言层出不穷。在前端，函数式最后的边界也已经被渐渐打破。Scala 的 Scala.js、Clojure 的 ClojureScript 都试图同构移向前端。然而，原生 JavaScript 其实也可以通过丰富的库让前端的函数式编程一样的舒适和优雅。

本书涵盖了大部分函数式编程思想，包括 JavaScript 的函数式支持，Clojure 风格的集合、递归、函数组合、宏、模式匹配、实用的 Monads，以及前端的并发编程。

本书适合想要了解函数式编程的 JavaScript 程序员或者想学习 JavaScript 的函数式程序员阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

前端函数式攻城指南 / 欧阳继超著. —北京：电子工业出版社，2016.10
ISBN 978-7-121-29724-3

Ⅰ. ①前… Ⅱ. ①欧… Ⅲ. ①函数－程序设计 Ⅳ. ①TP311.1

中国版本图书馆 CIP 数据核字（2016）第 200135 号

责任编辑：董　英
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：12　　字数：187 千字
版　　次：2016 年 10 月第 1 版
印　　次：2016 年 10 月第 1 次印刷
印　　数：3000 册　　定价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819　faq@phei.com.cn。

序

函数式编程可以说是非常古老的编程方式，但是近年来函数式编程越来越受到人们的关注。不管是 Google 力推的 Go、学术派的 Scala 与 Haskell，还是 Lisp 的新方言 Clojure，这些新的函数式编程语言都越来越受到人们的关注。

当然不仅是后端函数式编程语言层出不穷，前端也不甘示弱。虽然前端浏览器只支持一门语言——JavaScript，但是能支持函数式编程的 JavaScript 库越来越多，比如 Functional JavaScript[1]、Underscore、lodash 等。不仅如此，还有一些能编译成 JavaScript 的语言，能让前端的函数式编程发挥到极致，例如 Haskell 的 PureScript、Scala 的 Scalajs、Clojure 的 ClojureScript。

我两次都以 Clojure 结尾，是因为我喜欢把重点留到最后。Clojure 独特于其他语言，它既是一门新的语言、一门函数式编程范式的语言，又流淌着古老的血液——Lisp[2]。这是我选择用 Clojure 来诠释函数式编程的原因之一。

那么为什么我要选 JavaScript 作为函数式编程的目标呢？Michael Fogus 用 200 多页向大家展示了不一样的 *Functional JavaScript* 编程方式，可惜 Fogus 作为 ClojureScript 编译器的贡献者，竟然选择了 Underscore 作为函数式库，直接导致并不能完全展示 JavaScript 所能达到的函数式编程能力。有趣的是，ClojureScript 的作

---

1 没错，名字就叫这个。

2 还记得《计算机程序构造与解释》里面用的怪怪的括号语言 Scheme 吗？

者把 ClojureScript 的不可变（Immutable）数据结构移植到了 JavaScript，这彻底将 JavaScript 的函数式编程提升到了用其他库都完成不了的新高度[3]。不仅如此，Mozilla 的 Sweet.js（https://github.com/mozilla/sweet.js）更是完成了另一个突破——JavaScript 的 macro，它虽然不能算是函数式的概念，但也算是 Lisp 语言的一项独门绝技了[4]。

这一切的一切，都让我忍不住要帮 Fogus 出一本续集，用 JavaScript 实现其他函数式编程语言如 Clojure 甚至是 Haskell[5]的奇技淫巧，让大家进一步感受用 JavaScript 这门不完美的语言同样可以编写出优雅的函数式代码，以不一样的方式思考和解决问题。于是不管你是想转行 JavaScript 的 Clojure 开发者，还是想了解 Clojure 或函数式编程的 JavaScript 开发者，都可以在此找到一些启发。但这并不是一本 JavaScript 入门的好书[6]。

3 虽然 Facebook 也尝试实现自己的一个不可变数据结构 Immutable.js（https://facebook.github.io/immutable-js）。

4 虽然很多新的语言都在尝试实现 macro，比如 Scala、Rust 和 Julia，但是语法过于复杂。

5 就当是 bonus 吧。

6 Douglas 的《JavaScript 语言精粹》是个不错的选择。另外如果读者的英文好的话，还有一本可以在线免费阅读的 *JavaScript Allonge*（https://leanpub.com/javascriptallongesix/read）。

前言

1．看本书之前你要知道

1）最好能看懂 JavaScript 代码

这既不是一本介绍 Clojure 的书，也不是一本介绍 JavaScript 的书，而是一本介绍如何用 JavaScript 函数式编程的书。其中一些函数式的思想和表现形式都借用了 Clojure，因此叫作 Clojure 风格的函数式 JavaScript，但是并不要求在读本书前会 Clojure[1]，而只需要能阅读 JavaScript 代码。如果你会 Clojure，则可以完全忽略我解释 Clojure 代码的段落，当然 JavaScript 的部分才是重点。

2）你可能买错书了，如果你

- 想学 JavaScript

这不是一本 JavaScript 的教科书，这里只会介绍如何用 JavaScript 进行函数式编程，所以如果想要系统地学习 JavaScript，则学习《JavaScript 语言精粹》可能已经足够了。另外，如果读者的英文水平好的话，则还有一本可以在线免费阅读的 *JavaScript Allonge*。

- 想学 Clojure

同样，这也不是一本 Clojure 的教科书，只含有一些用于阐述函数式编程思想

---

1 就像计算机程序构造与解释中说的，Lisp 语言基本没有语法，就像学习象棋的规则只用花很少的时间，而如何下好棋，才是学习的关键，也是乐趣所在。

的 Clojure 代码。你确实可以学到一些 Clojure 编程知识，但很可能是非常零碎且不完整的知识。如果想要系统地了解和学习 Clojure，则非常推荐你阅读 *The Joy of Clojure*[2]，另外，如果读者英文比较好，还有一本可以免费在线阅读的 *CLOJURE for the BRAVE and TRUE* 也非常不错。

- 是函数式编程的专家

如果你已经在日常工作或学习中使用 Scala、Clojure 或者 Haskell 等函数式语言编程的话，那么本书对你在函数式编程上的帮助不会太大。**不过**：这本书对缓解你从函数式语言迁移到 JavaScript 编程的不适应该是非常有效的。这也正是本书的目的之一。

2．准备环境

在开始阅读本书之前，如果你希望能运行书中的代码，可能需要一些环境的配置。而且书中的所有源码和运行方式都可以在本书的 Github 仓库[3]中找到。当然如果你使用 Emacs（同时还配置了 org babel 的话）阅读本书的源码，对于大部分代码只需要光标放在代码处按 c-c c-c 即可。

- JavaScript

原生的 JavaScript 没有什么好准备的，可以通过 Node 或者 Firefox（推荐）的 Console 运行代码。当然第 5 章会有一些使用 Sweet.js 写的 Macro，这则需要安装 Sweet.js。

– 安装 Node/io.js

（1）下载 Node.js。

（2）如果使用 Mac，可以直接用 Brew 安装。

---

2 中文叫《Clojure 编程乐趣》，但是只有第一版的，原书已经第二版了。我刚好有幸翻译了作者 Michael Fogus 的另一本书《JavaScript 函数式编程》。

3 https://github.com/jcouyang/clojure-flavored-javascript/tree/source。

```
brew install node
# 或者
brew install iojs
```

– 安装 Sweet.js

在安装完 Node.js 之后在命令行输入：

```
npm install -g sweet.js
```

- Clojure

书中的 Clojure 代码大都用来描述函数式编程的概念，当然如果想要运行书中的 Clojure 代码，首先需要安装 JVM 或者 JDK[4]，至少需要 1.6 版本，推荐安装 1.8 版本。

– 安装 leiningen

leiningen 是 Clojure 的包管理工具，类似于 Node 的 Npm、Ruby 的 bundle、Python 的 pip。另外 leiningen 还提供脚手架的功能。可以通过官网的脚本安装[5]。Mac 用户可以简单地使用 brew install leiningen 安装。

安装完成之后，就可以运行 lein repl，打开 repl，试试输入下列 Clojure 代码，你将会立马看见结果。

```
(+ 1 1)
;=> 2
```

– 编辑器

如果更喜欢使用编辑器来编辑更长的一段代码，我推荐非 Emacs 用户使用 Light Table[6]，Intellij 用户则使用 cursive[7]。当然如果读者已经在使用 Emacs，那就更

4 http://www.oracle.com/technetwork/java/javase/downloads/index.html。

5 http://leiningen.org。

6 http://lighttable.com/。

7 https://cursive-ide.com/。

完美了，Emacs cider mode[8]是 Clojure 编程不错的选择。

3．本书中的代码

书中的所有源码和运行方式都可以在本书的 Github 仓库[9]中找到，书中几乎所有的例子都以测试的形式完成。

4．反馈

如果你是开源贡献者，那么应该很习惯通过 Github Issue[9] 提交任何反馈，如果是 Pull Request，那就更好了。当然如果没有使用过 Github Issue 也没有关系，这里[10]有非常详细的教程。

5．代码风格约定

本书的 JavaScript 代码都遵循 Airbnb JavaScript Style Guide[11] 中的 ES5 和 React 的风格约定。

6．本书的组织结构

第 1 章

将介绍 JavaScript 的基本函数式背景，简要地介绍为什么要关心函数式编程，为什么说 Underscore 不够函数式，JavaScript 要作为完整的函数式语言还缺些什么？

第 2 章

主要介绍 Clojure 的集合数据结构。这是个无聊但是又很重要的章节，可以说函数式编程最基本、最重要的就是集合操作。本章会涉及如何操作集合、惰性求值与惰性序列。

---

8 https://github.com/clojure-emacs/cider。

9 https://github.com/jcouyang/clojure-flavored-javascript/issues。

10 https://guides.github.com/features/issues/。

11 https://github.com/airbnb/javascript。

第 3 章

在了解了持久性数据结构后，我们可能会产生疑惑，如果数据结构都是不可变的，那么循环该怎么写呢？本章就是要解开各种使用不可变数据结构的疑惑，用这些不可变数据结构可以切换一种编程的思维方式。

第 4 章

Underscore 并不利于函数组合，但是函数组合其实是函数式编程最重要的思想。在这一章里面，我会详细介绍为什么说 Underscore 错了，而为什么要喜欢上柯里化，以及 Clojure 1.7 新推出的 Transducer 又是如何帮助我们更容易组合出更高效的函数的。

第 5 章

我特别不情愿把 Macro 翻译成宏。宏特别容易让人以为是 C 语言里面那个 #define 宏，虽然都是宏，但其实那里跟这里说的 Macro 不是一个级别的。Macro 是 Lisp 语言之所以特别的原因之一。本章我们就来看看到底什么是、为什么，以及如何在 JavaScript 中使用 Macro。

第 6 章

这里说的模式匹配包括两种：一种是按位置或者 key 匹配集合，取出相应数据。另一种是 Haskell 风格的对函数参数的模式匹配。本章篇幅比较小，因为模式匹配并不是 Clojure（也不是 JavaScript）的主要模式，尽管在一些有强大类型系统的函数式语言（Scala、Haskell）中比较重要。

第 7 章

Monad 这个范畴论里出来的神秘玩意，但你可能没有在意，其实这在前端世界早都被玩腻了。本章将会介绍 Monad 和它的朋友们，并且将带你体验 JavaScript 的 Promise，以及 Reactive 编程。

第 8 章

并发编程一直是令人头疼的编程方式，直到 Clojure 和 Go 的出现，彻底改变了我们并发编程的方式。而对于单线程的 JavaScript，基于事件循环的并发模型也一直困扰着我们，到底能从 Clojure 学些什么，可以使我们的前端并发编程之路更顺畅一些呢？本章将带你熟悉并发、JavaScript 的并发模型，以及 CSP 并发模型。

7．本书使用的约定

本书使用以下字体排版约定。

1）楷体

表示新的术语。

2）等宽字体

代码清单，出现在段落之内则表示变量、函数名、关键字等。

3）粗体

重点概念。

4）下画线

需要填入的词，我可能已经帮大家填上了。

5）横线

可以忽略的词。

目录

第 1 章　函数式 JavaScript.....1

1.1　JavaScript 也是函数式语言吗.....1

1.1.1　编程范式.....1

1.1.2　JavaScript 的函数式支持.....3

1.2　作为函数式语言，JavaScript 还差些什么.....10

1.2.1　不可变数据结构.....11

1.2.2　惰性求值.....11

1.2.3　函数组合.....12

1.2.4　尾递归优化.....13

1.3　Underscore 你错了.....14

1.3.1　跟大家都不一样的 map 函数.....14

1.3.2　ClojureScript.....16

1.3.3　Mori.....17

1.4　小结.....18

第 2 章　集合.....19

2.1　集合的使用.....20

2.1.1　向量（vector）.....20

2.1.2 Map 23
2.1.3 函数组合子 24

2.2 持久性数据结构 27

2.2.1 向量的持久性数据结构 28
2.2.2 最后一片叶子不完整 31
2.2.3 所有叶子完整且叶子个数不大于完全树的叶子个数 32
2.2.4 所有叶子完整且叶子个数大于完全树的叶子个数 34

2.3 不可变性 35

2.3.1 致命魔术 36
2.3.2 引用透明性 38
2.3.3 函数式 React 40
2.3.4 线程不安全 42

2.4 惰性序列 44

2.4.1 改良奥利奥吃法 44
2.4.2 惰性求值与及早求值 46
2.4.3 惰性求值的实现 48

2.5 小结 50

第 3 章 递归 51

3.1 不可变性与递归 51

3.1.1 真的需要循环吗 52
3.1.2 递归还是循环 54

3.2 柯里悖论 55

3.2.1 Y 组合子 57
3.2.2 栈是多么容易爆 60

3.3 尾递归优化 62

3.4 蹦跳乐园（Trampoline） 64
3.4.1 有穷状态机（DFA） 65
3.4.2 Trampoline 67
3.5 小结 69
第 4 章 函数组合 70
4.1 Underscore 到底做错了什么 70
4.1.1 自由（Free）变量与约束（Bound）变量 72
4.1.2 闭包 74
4.2 柯里化有什么用 75
4.3 Transducer 78
4.3.1 Reducer 79
4.3.2 来看看更好更快的解法 79
4.3.3 Reducer 80
4.3.4 Reducible 81
4.3.5 Transducer 详解 82
4.3.6 跟柯里化有什么区别 83
4.4 组合与管道 84
4.4.1 管道（Pipeline） 84
4.4.2 组合与管道 86
4.4.3 管道函数 87
4.5 小结 87
第 5 章 Macro 宏 89
5.1 什么是 REPL 89
5.1.1 宏扩展器（Macro Expander） 90

5.1.2 那么前端怎么办 92
5.2 为什么要语法糖 93
5.2.1 只是为了语法漂亮吗 94
5.3 Sweet.js 94
5.3.1 Rule Macro 95
5.3.2 Case Macro 97
5.4 Infix Macro 和 Operator 104
5.4.1 Infix Macro 104
5.4.2 自定义操作符 105
5.5 小结 106
第 6 章 模式匹配 107
6.1 Destructure 107
6.1.1 对象 109
6.1.2 数组 109
6.1.3 函数 109
6.2 Arity 函数 110
6.3 Fizz Buzz 111
6.3.1 字面匹配（Literal Matching） 111
6.3.2 绑定 113
6.3.3 Vector 与 Map 匹配 113
6.3.4 Guard 114
6.3.5 Sparkler 114
6.4 代数数据类型（ADT） 116
6.5 小结 118

第 7 章　Monadic 编程 119

7.1　链式调用 119

7.1.1　Promise 120

7.1.2　高阶 Promise 122

7.2　Monad 123

7.2.1　函子（Functor） 123

7.2.2　Applicative Functor 126

7.2.3　含幺半群（Monoid） 130

7.2.4　Monad 就是容器界的管道 132

7.2.5　Monad 就是自函子范畴上的一个幺半群 136

7.3　走钢丝 139

7.3.1　用 Monad 表示薛定谔猫 139

7.3.2　皮尔斯走钢丝 140

7.4　Monad 在 JavaScript 中的应用 143

7.4.1　Promise 版本的走钢丝 144

7.4.2　When 144

7.5　Reactive 编程 146

7.5.1　流（Stream） 146

7.5.2　Functor 147

7.5.3　Applicative 147

7.5.4　Monad 148

7.5.5　一个“简单”的 Reactive 实例 149

7.6　小结 153

第 8 章　并发编程 154

8.1　什么是并发 154

8.1.1　异步与多线程 155

8.1.2　JavaScript 的并发模型 157

8.2　通信顺序进程（CSP） 160

8.3　使用 Generator 实现 CSP 162

8.3.1　Generator 163

8.3.2　Go Block 163

8.3.3　timeout 164

8.3.4　take <! 165

8.3.5　put >! 165

8.3.6　JavaScript CSP 版本的例子 166

8.4　实战 CSP 168

8.4.1　使用移植的 core.async 168

8.4.2　使用 ES7 中的异步函数 169

8.4.3　try catch 170

8.5　小结 173

参考资料 174

第 7 章

Monadic 编程

这是范畴轮弄出来的神秘单词，但你可能没有在意，其实在前端世界早就被玩腻了。本章将会介绍 Monad 和它的朋友们，并且将带你体验 JavaScript 的 Promise，以及 Reactive 编程。

7.1 链式调用

说到**链式调用**，用过 jQuery 或 Underscore 的人再熟悉不过了。虽然我一直在强调函数组合如何复用性更好，更持续可组合，但是其实这两者之间并不对立，反而应该是可以互相结合的两种非常好的模式。

作为前端比较流行的模式，我们其实甚至不需要借助于任何库也可以进行简单的链式调用，比如 ES5 的 Array：

```
[1, 2, 3, 4]
.map(n => n + 1)
.filter(n => n % 2 == 0)
.reduce((acc, n) => acc + n);
```

由于 Array 的 map、filter 都仍然返回 Array，所以很自然，可以链式地继续调用 Array 的方法。

当然，如果做这种运算，我肯定更推荐使用 Transducer 来做，将会更高效而

且易于组合再利用。相比之下，链式调用其实有着更为适用的场景，那就是 ES6 标准中新加入的 Promise。

7.1.1 Promise

Promise 跟语义上一样，表示将来会发生但是还没有发生的事情。在不同的编程语言中 Promise 的意思略微有不同，或者是意思相近的单词，但基本上是同一个东西。只是，Promise 在 JavaScript 中是非阻塞的，也就是你不能直接 await 一个 Promise，这样会使得主线程被阻塞。

而在 Clojure 中，可以：

（1）定义一个 Promise。

（2）非阻塞地开启另一个线程给 Promise 递一个值。

（3）阻塞地从 Promise 把值取出来，@ 相当于 await。

```
(def fancy-promise (promise)) ; 1
(future (Thread/sleep 1000)
  (deliver fancy-promise (+ 1 1))) ;2
(print "waiting...")
@fancy-promise ;2
;=> waiting...
;一秒后 => 2
```

但是在资源有限的前端 JavaScript 中，宝贵的主线程只有一个，是不能让你能阻塞地去取一个 Promise 的值的。

```
const defered = when.defer();
setTimeout(_ => defered.resolve(1 + 1), 1000);
defered.promise.then(_ => console.log(_));
```

你会发现 JavaScript 的 Promise 略有不同。首先，在 JavaScript 中不能开一个线程去等上 1 秒，再给 Promise 递一个值。前端只能依靠浏览器的 timer 线程去计算时间，1 秒后回调递值的函数；其次，当然也不能阻塞主线程来等待 Promise 的值，只能通过传入回调函数给 Promise 的 then 方法，来接受 Promise 中的值并做

处理。因为 JavaScript 引擎只提供一个用户线程，这意味着如果我们阻塞地等待 Promise 的值：

```
// 伪代码
...
setTimeout(_ => defered.resolve(1 + 1), 1000);
await defered.promise
```

那么 `defered.resovle(1+1)` 永远不会被执行。因为 JavaScript 是事件循环的并发模型[1]，意味着事件循环会不停地从消息队列中拿任务。所以这几行代码大概的执行过程是这样的：

（1）setTimeout 的执行分派了浏览器 timer 去计时。

（2）promise.then 的执行给 Promise 挂了一个回调，当 Promise 满足时调用。

（3）代码都执行完了，消息队列里此时没有消息。

（4）1 秒后 timer 把 `defered.resovle(1+1)` 放入消息队列。

（5）`defered.resovle(1+1)` 随即触发 promise.then 上的回调。

所以如果（2）被阻塞，意味着（4）放回消息队列的任务永远也轮询不上，因为当 event loop 还正阻塞在（2），而（2）却在等（5）的结果，这时就出现了死锁。

因此，前后端 JavaScript 的所有 IO 操作（timer、XHR、worker）才被设计成为非阻塞的。

回到我们的问题上来，我们不知道 Promise 什么时候有值，也不能阻塞地等待 Promise 有值，所以只能把回调函数给 Promise，什么时候有值了再回调这个函数。而这个回调函数需要通过 Promise 的 then 方法传给 Promise。跟使用 jQuery 一样，then 返回的仍然是一个 Promise，因此我们可以继续调用 then 方法，绑定更

1　具体的并发模型会在第 8 章详细介绍。

多的回调函数。

当然，除了传入一个接收 Promise 内的值、返回另一个值（或不返回）的函数，还可以传入一个返回 Promise 的函数：

```
when.promise(resolve => setTimeout(_ => resolve(1), 1000))
  .then(value => (
    when.promise(resolve => setTimeout(_ => resolve(value + 1), 1000))
  ))
  .then(_ => console.log(_));
// => 2
```

两秒后打印出来的是 2，而不是一个 Promise。注意这里用的是更简捷的创建 Promise 的方法，直接给 Promise 的参数传入一个 resolve 函数，告诉 Promise 什么时候能 resolve。至于为什么最终打印的不是 Promise，而是 Promise 的值，我想看完 7.2 节的 Monad 自然会明白。

7.1.2 高阶 Promise

不像普通的操作值，可以通过高阶函数轻松地操作整个数组：

```
const data = [1, 2, 3, 4];

const double = data.map(_ => _ * 2);

console.log(double); // [2,4,6,8]
```

如果是一个 Promise 的数组，想用 map 或者 filter 之类的高阶函数来操作数组似乎就遇到了麻烦。

```
const data = [when(1), when(2), when(3), when(4)];

const double = data.map(_ => _ * 2);

console.log(double); // [NaN,NaN,NaN,NaN]
```

当然，想着都不可能会工作。这里期望的结果应该是 Promise 级别的 map，而不仅仅是把函数 map 到值上。选择使用 when 正是因为它提供更强大的 Promise，不但有操作单个 Promise 的方法，还提供通过 map、reduce 等高阶函数

轻松地操作 Promise 的数组。只需要做小小的改动，使用 `when.map` 方法就好了：

```
const double = when.map(data, _ => _ * 2);

double.then(_ => console.log(_)); // [2,4,6,8]
```

7.2　Monad

我为什么要在链式调用之后介绍 Monad 这个来自范畴论的看似令人费解的东西呢？当你问一个会 Haskell 的人如何理解 Monad 时，他们经常会让你先了解下**范畴论**（Category Theory），但其实并没有这个必要。Crockford 在解释 Monad 的时候就说过：

> **吃墨西哥鸡卷前要先学会墨西哥语吗？**—— Crockford

所以，我也并不打算在这里讲晦涩的范畴论，当然如果感兴趣网上有大把的资料。不过，你有可能会在本节结束时理解下面这句话：

> Monad **就是自函子范畴上的一个幺半群。**

那么，在了解 Monad 之前，我们必须先来聊聊函子（Functor）。

7.2.1　函子（Functor）

Functor 简单来说是可以被 map over 的类型。但是什么叫 map over？

比如 Array 就可以说是可以被 map over 的类型，来看看 Haskell 中是如何解释的：

```
ghci > :t fmap
fmap :: (a -> b) -> fa -> f b
```

这里的 `fmap` 就是 map over 在 Functor 上的函数。这个函数只干一件事情，`(a -> b) -> f a -> f b` 这个类型标记告诉了我们，一个 a 到 b 的映射在接收 a 的 Functor 之后，返回一个 b 的 Functor。

跟所有的数学定义一样，这个太抽象，我们来具体化一下以便理解。例如我现

在有一个 number 类型的 a，那么我可以定义一个由 number 到 string 的映射，也就是接受 a 为参数的函数 f，使得 f(a) 返回 string 类型 b。

```
const f = x => x + 'b';
const a = 2;
typeof a == 'number';
const b = f(a); // => 2b
typeof b == 'string'
```

那么如果 a 存在一个容器 fa 中，显然我们不能简单地通过 f(fa) 就能得到一个 b 类型的容器 fb。制作一个容器很简单，只需要创建一个对象就好了，这里我们把这个对象叫 Just 好了。

```
function Just(val){
  this.value = val;
}
const fa = new Just(a);
f(fa)
// => [object Object]b
```

显然这时候映射 f 是不工作的，输出的是一个奇怪的结果，而不是期望的包含 b 的容器 fb，也就是 Just(b)（图 7-1）。

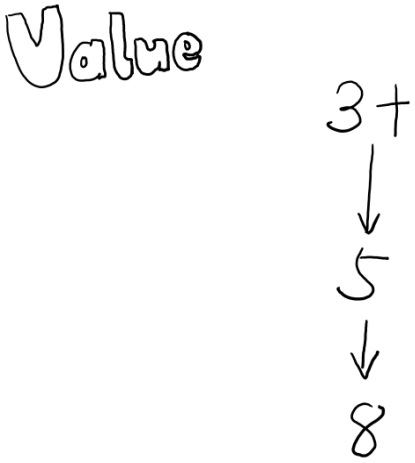

图 7-1　映射值的函数

要实现这种映射其实也非常简单，只需要容器 Just 能告诉映射 f 如何应用到容器内的值就好了，而这种如何将函数应用到自己内部值的方法，就是 fmap，而实现了 fmap 的容器，就叫作 Functor 的实例。不过为了遵循 JavaScript 的 fantasyland[2]标准，我们重命名 fmap 为 map：

```
Just.prototype.map = function(f) {
  return new Just(f(this.value));
}
Just.of = function(value) {
  return new Just(value)
}
const fa = Just.of(a);
const fb = fa.map(f); // => Object { value: "2b" }
fb instanceof Just; // => true
```

可以看到通过 map，得到了期望的在容器 Just 中的值 2b。其中的 `of` 方法是为了在构造一个 Just 容器时代码更可读一些，如图 7-2 所示。

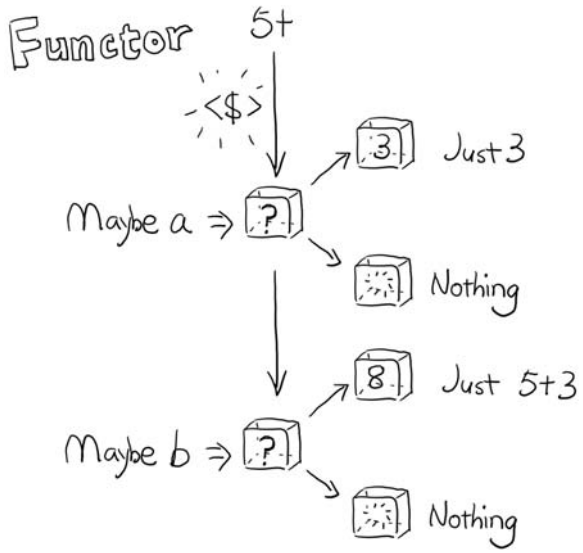

图 7-2　Functor 可以通过 map 把函数映射应用到容器内的值

[2] https://github.com/fantasyland/fantasy-land 所有这些代数结构（包括后面 Monad）的 JavaScript 实现都是有标准的，并不是我随便起的方法名。

其中，<$>代表 map，在 Haskell 中，$代表 apply。所以就像 apply 一个函数到值一样，Functor 使用<$>来 apply 一个函数到值，很形象，不是吗？apply（$）发生在容器（<>）中。

而刚刚的 Just 就是 Functor 的实例，由于它实现了 map，所以就可以说是 map over 了，自然就成为了 Functor 实例。

所以照这个逻辑，我只需要做一点点手脚，Array 由于已经实现了 map 方法，也就是一个 Functor 的实例。有意思的是，Array 本身是一个类型，但是又是 Functor 的实例，这么说来 Functor 是构造类型的类，所以 Functor 在 Haskell 中被称为——类型类。

等一下，出自范畴论，难道就没有些什么函子定律什么的？

当然，由于出自完整的理论，这里也顺便提一下 Functor 具有以下性质：

（1）`fa.map(x=>x)` 恒等于 `fa`。

（2）`fa.map(x=>f(g(x)))` 恒等于 `fa.map(g).map(f)`。

我们先来测试一下 Array：

```
[1, 2, 3].map(x => x); // => [1,2,3]
const f = x => x + 2
const g = x => x * 2
[1, 2, 3].map(x => f(g(x))); // => [4,6,8]
[1, 2, 3].map(g).map(f) // => [4,6,8]
```

看上去是没有什么问题的，我就不充分证明了，但是这个定理对后面的理解是非常有帮助的。

7.2.2 Applicative Functor

现在，我们已经有 map 可以把一个函数应用到一个 Functor 上，接着来看看下一个概念——Applicative Functor（下面简称 Applicative）。有了 Functor 之后，

我们可以直接给容器加一组映射（范畴论叫态射），但是，函数既然是一等的，那么值能放到容器中，函数是不是也可以放到容器里呢？

```
const fplus2 = Just.of(x => x + 2);
fplus2(Just.of(3));
```

显然一个含有函数的容器是不能直接应用到任何容器或者值的，map 显然也不能帮我们把装有函数的容器打开。这时候 Applicative 就出来解决这种问题了。

```
Just.prototype.ap = function(container){
  return container.map(this.value);
}
fplus2.ap(Just.of(3)) // => 5
```

没有想到，只是这么简单就又实现了 Applicative 吧。现在知道为什么我先介绍 Functor 了吧。现在看来，所有的 Applicative 也都是 Functor，因为需要实现 map 方法。

所以来看下 Applicative 的类型注解 `f (a -> b) -> f a -> f b`，一个 a 到 b 映射的容器，接收 a 的容器返回 b 的容器，那么这个容器就是 Applicative（图 7-3）。

图 7-3 中使用的`<*>`即 `ap` 方法，而`<$>`是 `map`，图中使用的是中缀表达式：

```
+ <$> Just 5 <*> Just 3
-- Just 8
```

如果改成前缀，并改写成 JavaScript：

```
plus = x => y => x + y
ap((map(plus, Just.of(5)), Just.of(3))
```

或者更像 JavaScript 习惯的那样：

```
Just.of(5).map(plus).ap(Just.of(3))
```

很有意思，不是吗？柯里化的 plus 让 Functor 与 Applicative 结合得如此巧妙。

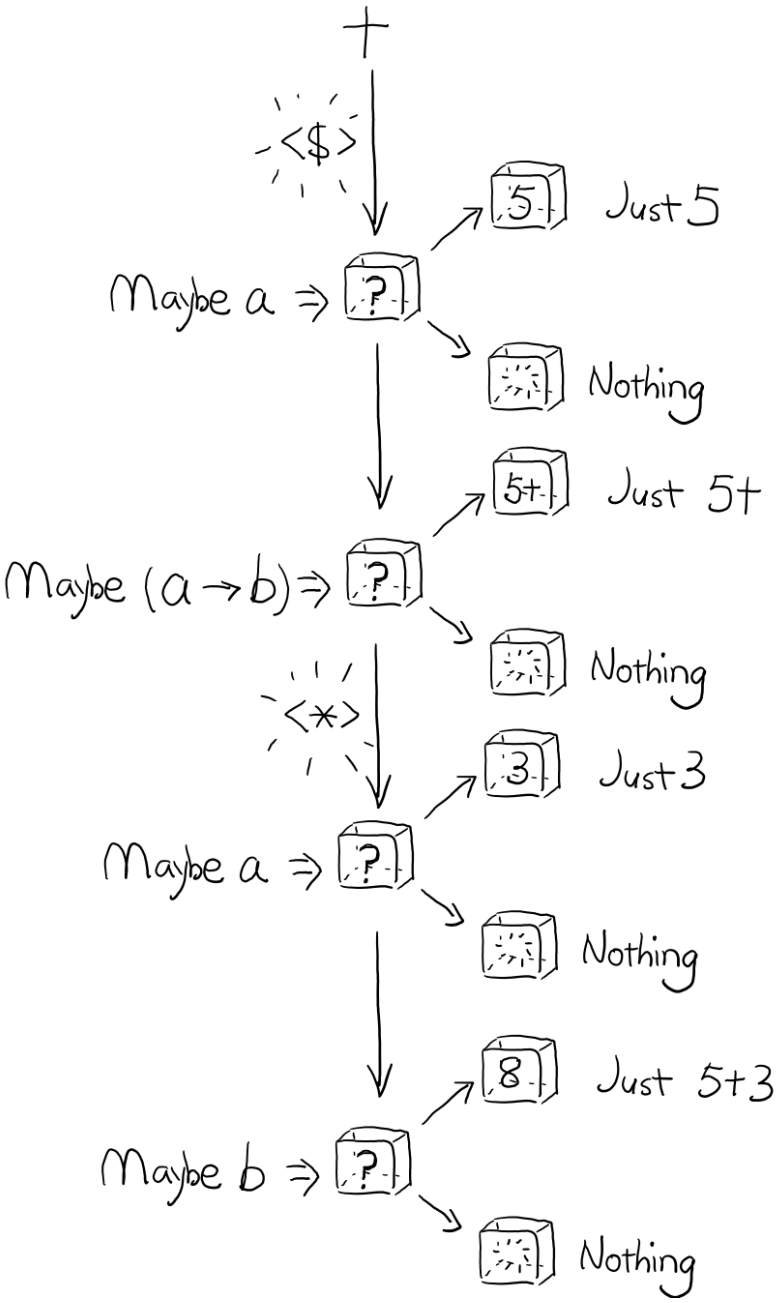

图 7-3　Applicative Functor

同样，Applicative 也具备一些性质。

（1）恒等性：`a.of(x=>x).ap(value)`恒等于 value。

（2）同态：`a.of(f).op(a.of(x))`等于 `a.of(f(x))`。

（3）交换律：`u.ap(a.of(y))`等于 `a.of(f=>f(y)).ap(u)`。

刚好试一下我们的新 Just 有没有符合这些性质：

```
Just.of(x => x).ap(Just.of(3)) //=> Just.of(3)
Just.of(x => x + 1).ap(Just.of(2))
  .equals(Just.of((x => x + 1)(2))) // true
Just.of(x => y).ap(Just.of(3))
  .equals(Just.of(f => f(3)).ap(Just.of(x => y))) //true
```

如果把容器看成电梯（lift），那么坐电梯下来（把容器剥开，这里即去掉所有的 Just.of 和 ap，如图 7-4 所示），它们其实保持着跟函数与值同样的关系。

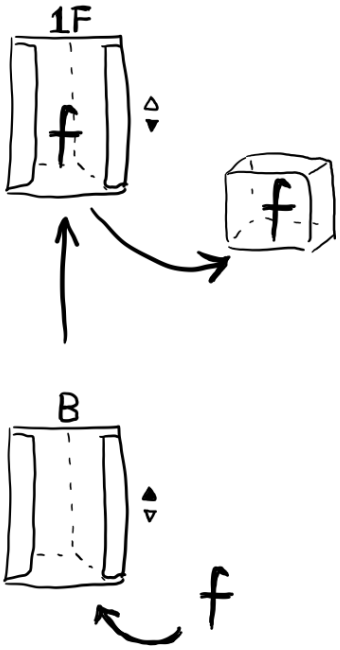

图 7-4　lift

```
(x=>x)(3) //=> 3
(x=>x+1)(2) == (x=>x+1)(2) // true
(x=>y)(3) == (f=>f(3))(x=>y) //true
```

7.2.3 含幺半群（Monoid）

前面说了两个抽象的 Functor 与 Applicative，分别是两种不同类型的 lift 操作，前者 lift 值，后者 lift 函数，但都是为了 lift 或者说抽象函数应用，例如 $f(x)$。

那么值之间的二元操作该如何抽象或者 lift 呢？比如简单的值之间的算术运算加法：

```
1 + 2 //=> 3
```

这并不是 $f(x)$的简单一元操作，这里如果把 + 理解成函数，这将是二元操作：

```
add(1, 2) //=> 3
```

那么这里需要抽象三个东西，或者说把三个值放到容器中，或者说如何把容器中的两个值叠加起来？

这样看来，无论是 Functor 或者 Applicative 都没有办法把这些盒子撮合到一起，变成一个含有 3 的盒子 `Just.of(3)`。

于是，我们需要把它们抽象成 Monoid，先别着急问 Monoid 的概念是什么，等我举完这个例子。这里用 Just 来做算术运算语义上不太合适，请允许我换一个语义更好一些的容器，比如 Sum。这里我们要做的大概是像这样叠加两个容器：

```
Sum.of(1).append(Sum.of(2)) // => Sum.of(3)
```

所以重点是这个 append 方法，实现上应该把两个盒子内的值做加法：

```
Sum.prototype.append = function(anotherSum){
  return anotherSum.map(value => this.value + value);
}
```

有意思的是，加法会存在一个类似 identity 的元素 empty，使得任何数加 empty 值完全不变，因此加法的 emtpy 就是 0。那么当我抽象到容器之后，应该符合同样的规则，Sum 应该也存在一个 empty，使得 `Sum.of(any).append`

`(Sum.empty)`仍然是 any。

```
Sum.empty = Sum.of(0);
```

没有错，那就是 Sum.of(0)，我们管这个 empty 叫作**幺元**（Identity Element）。好了，这个简单的例子讲完也实现了，现在可以来定义 Monoid 了，其实这个 Sum 就是 Monoid（如图 7-5 所示），因为它：

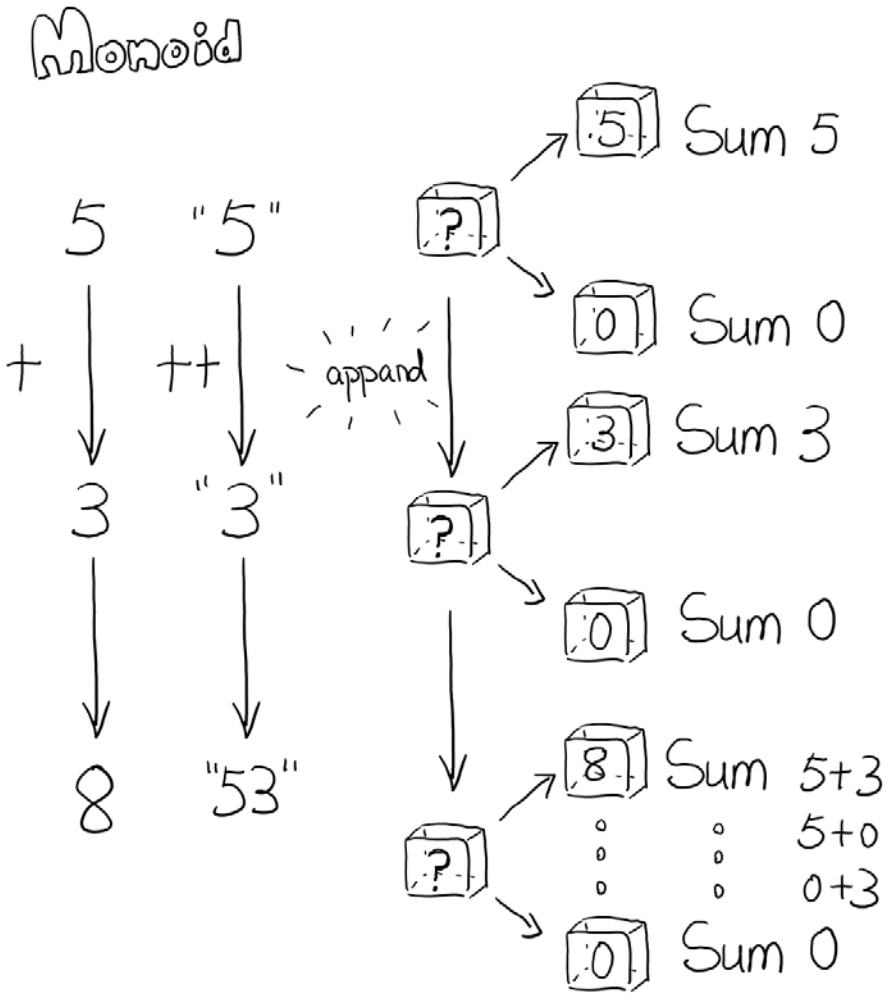

图 7-5　Monoid 只是抽象到容器的二元操作

（1）存在一个二元操作返回值仍然是 Sum。

（2）有一个幺元，满足 `Sum.empty.append(Sum.of(2)) == Sum.of(2)`。

（3）满足结合律 `Sum.of(1).append(Sum.of(2)).append(Sum.of(3)) == Sum.of(1).append(Sum.of(2).append(Sum.of(3)))`。

Monoid 的实现非常简单，可能并没有人会关心实现，而更重要的是，为什么要满足这些条件，这些公理能带来什么好处。

想一想我们要叠加多个数，可以这样：

```
1 + 2 + 3 + 4 + 5
```

当然有更高雅的 reduce 方式：

```
[1, 2, 3, 4, 5].reduce((_1, _2) => _1 + _2)
```

好了，那么抽象到容器，我们是不是可以干同样的事情？

```
[Sum.of(1), Sum.of(2), Sum.of(3), Sum.of(4), Sum.of(5)]
  .reduce((_1,_2) => _1.append(_2));
// Sum.of(15)
```

当然，除了简单的数组 reduce，想想稍微复杂的树，若是实现了 foldable 与 append，是否也同样可以简单地完成 Sum 运算呢？

7.2.4 Monad 就是容器界的管道

好了，有了包在容器中的函数或值之后，我们可以 map 一个函数到容器，还可以直接应用含有函数的容器到另一个容器，我们也可以像函数组合一样组合这些容器。但是，还记得与函数组合（Compose）一起介绍的管道（Pipeline）吗？通过管道，可以让函数组合的顺序更符合阅读习惯，更适合一眼就能看出数据的流向。激动人心的时刻到了，Monad 这个被解释得过于抽象的东西，其实就是容器界的管道而已。

如果我告诉你在前一节其实已经见过 Monad 了（如果不小心跳过了上一节，

赶紧回去补一补），你会信吗？

```
new Promise(resolve => resolve(0))
  .then(x => x + 1) // 1
  .then(x => 2 / x) // 2
```

当然，这个并不是 Monad，这是简单的 pipe，等同于：

```
mori.pipeline(0, x => x + 1, x => 2 / x)
```

正常的函数组合的顺序应该是这样的：

```
(x => 2 / x)((x => x + 1)(0))
```

运用我们前面学过的 Applicative Functor，来把这些都装容器里试试，首先，来把函数组合装容器中，继续使用刚刚实现的 Just：

```
Just.of(x => 2 / x).ap(Just.of(x => x + 1).ap(Just.of(0)))
```

本来看函数的嵌套调用都已经有些头晕了，这回全部包了层容器，更是摸不着头脑了。试试用 Functor 实现：

```
Just.of(0).map(x => x + 1).map(x => 2 / x)
```

似乎已经非常像 Pipeline，以及可以链式编程的 Promise 了，但是现在问题来了，如果 `map` 的过程有可能发生异常，比如：

```
Just.of(undefined)
  .map(x => x.value)
  .map(x => x - 1)
  .map(x => 2 / x)
```

你会看到直接抛出异常了，第一个 `map` 就会遇到 `undefined.value` 从而抛出 `Cannot read property 'value' of undefined` 异常。

那么如何能避免异常呢？例如可以在 `map` 里加 `try catch` 或者 `if else` 验证输入条件。但是，最后的那个 `map` 怎么办？如果是 try catch 没有返回，意味着后面的 `x - 1` 自然也会出错，如果是 `if else`，那么 `else` 里面应该返回什么，才能保证异常发生后后面的 `map` 不会被执行到呢？

如果能返回一个容器，map 任何函数到它都还是它自己，而不执行 map 中的函数，似乎就可以忽略掉一旦出错后面的一大堆 map 了。

```
function Nothing(){};
Nothing.prototype.map = function(){
  return this;
}
const nothing = new Nothing;
```

问题是，Just 不管怎么 map 都是 Just，如何能让它变成 Nothing 呢？

```
Just.of(undefined)
  .map(x => {
    if (x)
      return Just.of(x.value)
    else
      return nothing
  })
  .map(x => x.map(y => y + 1))
  .map(x => x.map(y => 2 / y))
// => Just.of(nothing)
```

没有错，这段代码返回的会是一个在容器里的容器，Just.of(nothing) 或者 Just.of(Just.of(2))，所以我们还需要修改后面 map 中的函数程序才能正常工作：

```
Just.of({value: 0})
  .map(x => {
    if (x)
      return Just.of(x.value)
    else
      return nothing
  })
  .map(x => x.map(y => y + 1))
  .map(x => x.map(y => 2 / y))
// => Just.of(Just.of(2))
```

这显然越改越复杂了，先是包一层，后面的 map 再拆掉前面包的这一层容器，我们何必要多包一层呢？不如直接不包，比如将包和拆包放到 flatmap 方法中：

```
Just.prototype.flat = function(){
  return this.value
```

```
}
Just.prototype.flatmap = function(f){
  return this.map(f).flat();
}
Nothing.prototype.flatmap = Nothing.prototype.map
```

当然，这样我们能写出一个更优雅的带容器的 Pipeline 了：

```
Just.of({value: 0})
  .flatmap(x => {
    if (x)
      return Just.of(x.value)
    else
      return nothing
  })
  .map(x => x + 1)
  .map(x => 2 / x)
// => Just.of(2)
```

到这里，我可以告诉你现在的 Just 就是 Monad 了，它是 Functor 的加强，把 map 的结果铺平（Flat）。同时又是 Applicative 的加强，Applicative 让我们可以用一般函数作用到在容器中的值，而 Monad 让我们可以把一个容器中的值传入一个接收值的函数中，并返回同样的容器，如图 7-6 所示。

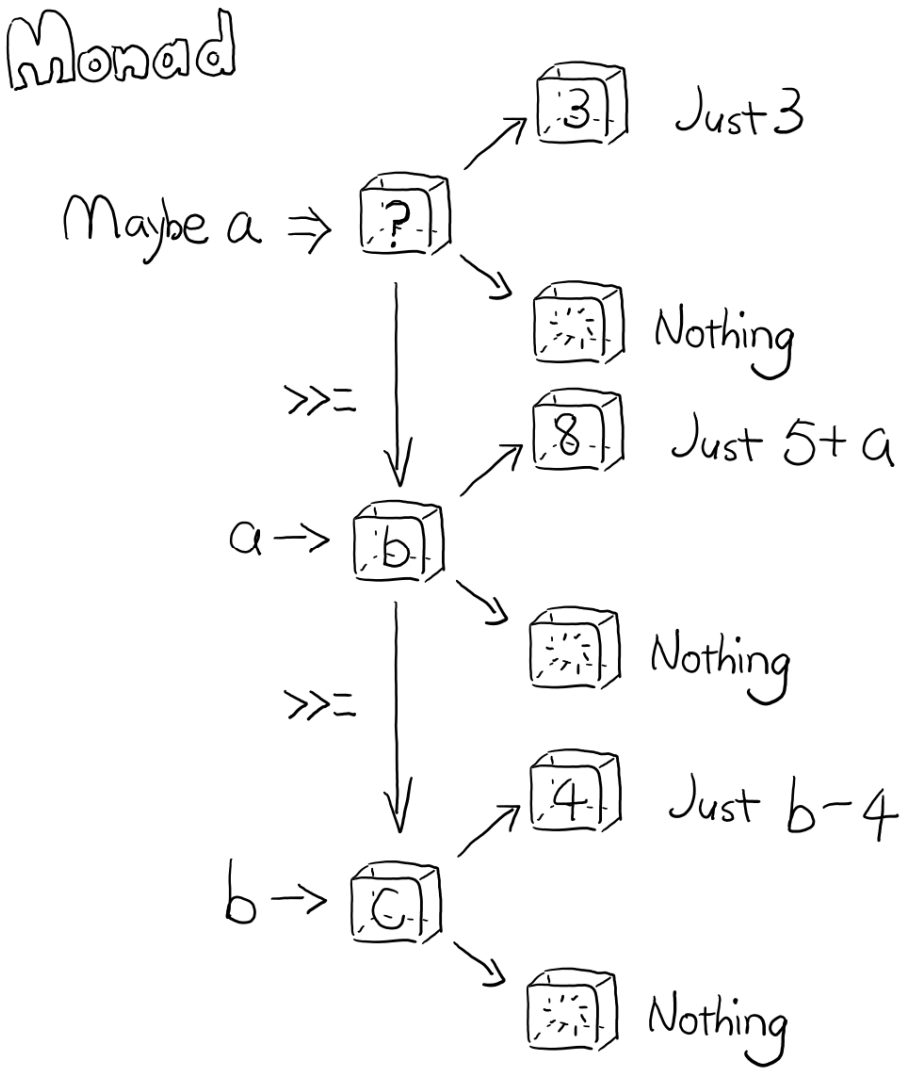

图 7-6　Monad

7.2.5　Monad 就是自函子范畴上的一个幺半群

但是举了这么多例子，怎么一点也看不出来 Monad 与 Monoid 的关系呢？倒

是很明确 Monad 是个比较特殊的 Functor。

我们需要进一步抽象才能解释这句话，首先，回顾前面 Monoid 的知识，比如那个 Sum 的 Monoid。

```
Sum.of(1).append(Sum.of(2)).append(Sum.of(0)) // => Sum.of(3)
```

这个 Monoid 很明显，它的二元操作是 `append`，幺元是 `Sum.of(0)`，范畴是 `Sum`。为了更明显，我们可以降一个维度（或者说范畴），把 `Sum.of(1)` 降成 `1`：

```
(1 + 2 + 0)
```

但是 Monad 哪来的二元操作啊？一个 `flat`，一个 `map`，都是一元操作啊？

如果我们降一个维度，到只有数字的维度上，用“.”代表二元操作的话：

```
(1.2).3 = 1.(2.3)
```

二元操作 `+` 可以使得这个等式成立。而幺元 0，又可以使得

```
0.2 = 2 = 2.0
```

在二元操作 `+` 上成立。

所以我们得到的 Monoid 为（数字集合，二元操作 `+`，幺元 0）。

同样，上升到函子的范畴上[3]，注意是自函子**范畴**上的幺半群，就代表的是函子范畴而不是函子实例是幺半群，所以以 Maybe 为例，就需要符合：

```
(Maybe.Maybe).Maybe = Maybe.(Maybe.Maybe)
```

能找到一个二元操作使得上式成立？而不是：

```
(Just.of(1).append(Just.of(2))).append(Just.of(3)) = Just.of(1).append
(Just.of(2).append(Just.of(3)))
```

这个二元操作就是我们刚实现的 Just Monad 的 flat。我们很容易可以把 flat 代入式子变换成以下格式：

3　基本上我们说的函子都是自函子，自函子是 map 到自己范畴上的函子。

```
flat(flat(Maybe.Maybe).Maybe) = flat(Maybe.flat(Maybe.Maybe))
```

其中的 . 是两个类型组合在一起。如同函数的组合是 `f.g x= f(g(x))`，在函子范畴上的组合就是 Maybe 类型的值，在一个 Maybe 类型的容器中：

```
Maybe.Maybe = Maybe[Maybe]
```

再代入一遍，就非常清晰结合律的等式是成立的了：

```
flat(flat(Maybe[Maybe])[Maybe]) = flat(Maybe[flat(Maybe[Maybe])])
```

大声念出来应该就是：flat 一个容器为 `flat(Maybe[Maybe])`类型，其值类型为`Maybe`得到的类型，flat 一个 Maybe 类型，其值类型为`flat(Maybe[Maybe])`得到的类型。

听起来比较像绕口令，看下图 7-7 会有更直观的理解。

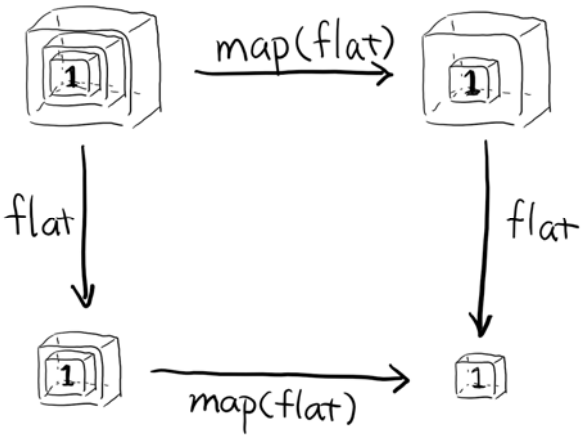

图 7-7　flat 嵌套

简单地说，就是不管 flat 内层的两个嵌套 Maybe 类型，还是 flat 外层的两个嵌套 Maybe 类型，效果是一样的。这就相当于一个 Monoid，不管是结合左边两个元素，还是结合右边两个元素，都一样。

另外一个 Monoid 的法则是需要有一个幺元，满足：

```
?.Maybe = Maybe = Maybe.?
```

我们很容易能猜到把 Maybe 代入就是我的解，因此任何 Maybe 类型都是幺元（如图 7-8 所示）。

```
flat(Maybe.Maybe) = Maybe = flat(Maybe.Maybe)
```

这样看起来比较怪异，左右式子看起来一样了。让我们用一个 pure 函数替换一下，假设 pure 函数可以：

```
pure(Maybe) = Maybe.Maybe
```

那么 pure 就可以替换之前的等式为：

```
flat(pure(Maybe)) = Maybe = flat(Maybe.map(pure))
```

所以，**flat** 就像 Moniod 里的 **append** 一样，但是它并不连接值或容器，而是连接函子组合，让函子在**不同范畴间变换**。

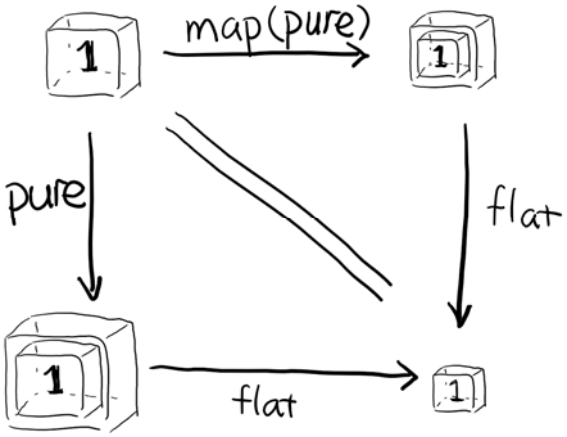

图 7-8　Monad 就是自函子范畴上的一个幺半群

7.3　走钢丝

7.3.1　用 Monad 表示薛定谔猫

如果用简单的 Monad 来表示薛定谔猫就再简单不过了，比如往盒子再加放射

性原子，如果猫活着，就是绿巨猫，如果猫是死的，那就是绿巨死猫。

```
Box.of(cat).flatmap(cat=>{
  const hulkcat = radioactive(cat)
  if(hulkcat=='dead')
    return BloodyBox.of(hulkcat)
  return Box.of(hulkcat)
}
```

所以可以说我们最终得到的不是（Either）含有绿巨猫就是含有一只绿巨死猫的血淋淋的盒子。这个例子过于简单，我们来一个更实际的例子。

7.3.2 皮尔斯走钢丝

皮尔斯决定要辞掉他的工作改行试着走钢丝。他对走钢丝还挺在行的。不过仍有个小问题，就是鸟会停在他拿的平衡竿上。它们会飞过来停一小会儿，然后再飞走。这样在两边的鸟的数量一样时并不是个太大的问题。但有时候，所有的鸟都会想要停在同一边，皮尔斯就失去了平衡，就会让他从钢丝上掉下去。

我们假设两边的鸟差异在 3 只之内的时候，皮尔斯仍能保持平衡。

- 一般解法

首先看看不用 Monad 怎么解：

```
const landLeft = function(n, pole){
   return [pole[0]+n, pole[1]];
}
const landRight = function(n, pole){
   return m.reverse(landLeft(n, m.reverse(pole)));
}
const result = m.pipeline([0,0],
               m.partial(landLeft, 1),
               m.partial(landRight,1),
               m.partial(landLeft, 2));
console.log(result);
// => [3 1]
```

对了，还差一个判断皮尔斯是否掉下来的操作：

```
const landLeft = n => pole => {
```

```
        if(pole=='dead') return pole;
        if(Math.abs(pole[0]-pole[1]) > 3)
          return 'dead';
        return [pole[0]+n, pole[1]];
    };
    const landRight = n => pole => {
        if(pole=='dead') return pole;
        return landLeft(n, eweda.reverse(pole));
    };
    const result = m.pipeline(landLeft(10), landRight(1),
landRight(8))([0,0]);
    console.log(result);
    // => dead
```

其中，`n => pole => ...` 的写法会得到一个简易的柯里化函数，接收 `n` 会返回函数 `pole => ...`

重要的是，注意看这两个函数的变化，`landLeft` 与 `landRight` 都需要加入异常判断 `if(pole=='dead') return pole`；这意味着 `pipeline` 一开始可能出现异常结果，随后的所有函数都必须处理这个异常，尽管它们的主要任务只是判断皮尔斯是否还活着而已。

- 现在来试试用 Either

我们先把皮尔斯放进 Either 容器里让他走钢丝，这样皮尔斯的状态只有打开 Either 容器才能看见。假设 Right 是装活着的皮尔斯的容器，Left 是装死了的皮尔斯的容器。

```
    const land = function(lr, n, pole){
        pole[lr] = pole[lr] + n;
        if(Math.abs(pole[0]-pole[1]) > 3) {
          return new Left("dead when land " + n + " became " + pole);
        }
        return new Right(pole);
    }

    const landLeft = n=>m.partial(land,0,n);
    const landRight = n=>m.partial(land,1,n);
```

现在落鸟后会返回一个 Either、Right 或者 Left。

偷看容器内东西的函数可以是这样的，类似于 map，但是这里可以分别对 Either 左右的值应用不同的函数：

```
const stillAlive = function(x){
   console.log(x)
}
const dead = function(x){
   console.log('皮尔斯 ' + x);
}
either(dead, stillAlive, landLeft(2, [0,0]))
```

好像越来越接近了，但是这里只落了一次鸟，如果要落好几次呢？这就需要实现 Either 的 flatmap 方法，让小鸟不停地落到皮尔斯的杆子上：

```
Left.prototype.flatmap = function(fn){return this;};
Right.prototype.flatmap = function(fn){
     return fn(this.value)
   }
```

当然，应用左右函数的 Either 函数实现起来也非常简单：

```
either = function(left, right, Either){
   if(Either.constructor.name == 'Right')
      return Either.map(right)
   else
      return Either.map(left)
}
```

我们来试试工作不工作：

```
const walkInLine = new Right([0,0]);
eitherDeadOrNot = walkInLine.flatmap(landLeft(2))
   .flatmap(landRight(5))
either(dead, stillAlive, eitherDeadOrNot)
// => [2,5]  <1>
eitherDeadOrNot = walkInLine.flatmap(landLeft(2))
  .flatmap(landRight(5))
  .flatmap(landLeft(3))
  .flatmap(landLeft(10)
  .flatmap(landRight(10)))

either(dead, stillAlive, eitherDeadOrNot)
// => "皮尔斯 dead when land 10 became 15,5" <2>
```

果然跟预期的一样：

（1）在所有小鸟的差值不大于 3 时，我们可以偷看到皮尔斯是死还是活着的。

（2）但是一旦大于 3，皮尔斯就已经死掉，不管之后再怎么落鸟，死的皮尔斯状态不会再变化。

7.4　Monad 在 JavaScript 中的应用

你知道 ES6 有个新的类型 Promise 吗？如果不知道，想必也听过 jQuery 的 `$.ajax` 吧，但如果你没听过 Promise，说明你没有认真看过它的返回值：

```
const aPromise = $.ajax({                          "
   url: https://api.github.com/users/jcouyang/gists
   dataType: 'jsonp'
   })
aPromise /***
=> Object { state: .Deferred/r.state(),
   always: .Deferred/r.always(),
   then: .Deferred/r.then(),
   promise: .Deferred/r.promise(),
   pipe: .Deferred/r.then(),
   done: b.Callbacks/p.add(),
   fail: b.Callbacks/p.add(),
   progress: b.Callbacks/p.add() }
***/
```

我们看到返回了好多 `Deferred` 类型的值，我们来试试它有什么用。

```
anotherPromise = aPromise.then(_ => _.data.forEach(y=>
console.log(y.description)))
/* =>
Object { state: .Deferred/r.state(),
   always: .Deferred/r.always(),
   then: .Deferred/r.then(),
   promise: .Deferred/r.promise(),
   pipe: .Deferred/r.then(),
   done: b.Callbacks/p.add(),
   fail: b.Callbacks/p.add(),
   progress: b.Callbacks/p.add() }

*/
```

看见没有，它又返回了同样一个东西，而且传给 then 的函数可以操作这个对象里面的值。这个对象其实就是 Promise 了。为什么说这是 Monad 呢？来试试再实现一次“走钢丝”。

7.4.1 Promise 版本的走钢丝

下面我们开始使用 ES6 标准的 Promise。同样，我们需要一个接收普通值返回容器的落鸟函数：

```
const land = function(lr, n, pole){
    pole[lr] = pole[lr] + n;
    if(Math.abs(pole[0]-pole[1]) > 3) {
      return new Promise((resovle,reject)=>reject("dead when land " + n
+ " became " + pole));
  }
    return new Promise((resolve,reject)=>resolve(pole));
 }
const landLeft = n=>m.partial(land,0,n)
const landRight = n=>m.partial(land,1,n)
```

然后就可以开始往皮尔斯的杆子上落鸟了。

```
Promise.all([0,0])
.then(landLeft(2))
.then(landRight(3)) // => Array [ 2, 3 ]
.then(landLeft(10))
.then(landRight(10))
.then(_=>console.log(_),_=>console.log(_))
// => "dead when land 10 became 12,3"
```

最后一个 then 相当于 either 函数，之前的每一个 then 相当于 flatmap。

7.4.2 When

虽然已经是 ES6 的标准，但是 Promise 只提供非常基本的支持，通常我们可以使用更实用的库 when 或者 q。这里我简单介绍一下更为丰富的 when。

- lift

还记得之前讲 Applicative 时提到的 lift 吗？可以把一个函数装到容器中，而

when.lift 就是把一般函数提升（lift）成一个含有函数的 Pormise。既然变成一个 lifted 的函数，当然，就像 Applicative 一样，可以应用到一个含有值的容器上，也就是类似 `when.lift(f).ap(aPromise)`。更方便的是，直接调用就等同于调用 ap 方法：

```
const readFile = function(){
   return when.promise(function(resolve){
   setTimeout(_=>resolve("i am a file"), 3000);
})
}

const print = function(text){
    console.log(text);
return "printed";
}
when.lift(print)(readFile())
 .then(_=>console.log(_))
// 三秒后 => "i am a file"
// => printed
```

跟 Applicative 一样，当把函数也提升到容器层面，那么含有函数的容器应用到一个含有值的容器，返回的当然是一个包含结果的容器。所以这里会看到应用函数 print 到 readFile() 会打印出 readFile() Promise 内的值，也就是三秒后 resolve 的 “i am a file”，同时，print 的返回值“printed”会被包在容器 Promise 中，从而还可以通过 then 打印出 print Promise 的值。

- when.map

这又是另一个特别方便的方法，在我们可以 lift 函数来操作 Promise 的时候，如果需要操作多个 Promise，很容易想到只要 map 这个 lift 的函数到 Promise 数组就好了。

```
[readFile(),readFile()].map(when.lift(print))
```

when 提供更便捷的方式来 map 一个普通函数到 Promise 数组，那就是 when.map：

```
when.map([readFile(),readFile()], print)
```

跟之前的代码效果完全相同，只是 when.map 自动把函数 lift 起来再 map 到数组中的 Promise 上。

当然 when 还有非常多强大的方法，这里就不一一介绍了，我的目的是通过比较有用的一些库中的 Monad 应用，说明 Monad 并非只是晦涩难懂的范畴论中的概念，而且在实际应用中其实是非常常用而且功能强大的。

7.5 Reactive 编程

上节提到了提供 Promise 的库 when，还有一些丰富的方法，当然 Promise 只能算是一种 Monad，在 JavaScript 的世界中还有非常多的 Monad，而我特别想要介绍跟 when 同样出自 cujojs 的 most。

most 的 Github 页面写得非常清楚，只有三个词——Monadic reactive streams，还是老习惯，如果看不懂这三个词，请先看完例子我再解释。

7.5.1 流（Stream）

如果说数组是空间维度，那么流则是时间维度版本的数组。比如我们有一个数组，需要 reduce 一下再打印出结果，是非常容易做到的：

```
[1,2,3,4].reduce((acc,x)=>acc+x)
```

那么问题是，我们操作在一个空间上已经存在的数组，是非常容易的，但是如果我们的输入是随着时间变化的，该如何处理？

在前端世界，这种情况非常常见，比如一个简单的用户输入框 input，同样，我想得到输入总和，似乎是有些棘手的一件事情，只是，对于函数式编程来说，对于状态的保存就非常头疼。当然如果不考虑函数式，弄一个全局变量来保存 acc 也是最直接的思路了。

```
var acc = 0;
$('input').onChange(_=>acc+=_)
```

这样每次在 input 中修改数字，都会加入 acc 中。

而旨在消除变量的函数式编程应该如何解决这种问题呢？

下面开始用 most：

```
most.fromEvent('input', document.querySelector('input'))
.reduce((acc,x)=>acc+x)
.then(_=>console.log(_))
```

这样的一组随时间变化的输入，就变成了输入流，使用 Reactive Programming 的技巧，我们可以像操作空间上的数组一样操作流，这就是 Reactive Programming，另外如果还符合 Monad 的一些公理，就会变成 Monadic Reactive Programming。

7.5.2　Functor

每个 most 的流都是一个 Functor，因此我们可以 map 一个函数到流上。

```
most.from([1,2,3,4]) //<1>
    .map(_=>_*2)
    .observe(_=>console.log(_)); //<2>
    // -2-4-6-8->
```

这段代码会依次输出 2 4 6 8。

1. most.from 会从一个数组生成一个 most 流，跟之前的 most.fromEvent 生成一个输入流一样。
2. observe 用于观察流内的数据，每次流的数据变化，都会触发 obse 上的回调。

7.5.3　Applicative

不仅如此，most 还是 Applicative Functor，希望之前的概念还记得，Applicative 可以把含有函数的容器应用到另一个含有值的容器上，所以上例可以用 Applicative 这样做：

```
most.of(_=>_*2)
  .ap(most.from([1,2,3,4]))
  .observe(_=>console.log(_))
```

除了使用 Applicative 之外，还可以把函数 lift 起来，这样在使用上跟一般的函数就没有什么区别了，只是现在已经 lift 的函数可以操作 most 流[4]。

```
const multiple2 = function(x){return x*2};
const lifedMultiple2 = most.lift(multiple2);
lifedMultiple2(most.of(3))
  .observe(_=>console.log(_))
```

7.5.4 Monad

当然，most 的流同时也是 Monad，因此可以方便地 flatmap 一个返回 stream 的函数。

```
most.from([1, 2])
    .flatMap(x=>most.periodic(x * 1000).take(5).constant(x))
    .observe(_=>console.log(_));
```

思考一下如果是一个数组 `[1,2]`，比如 flatMap `x=>[x*2]` 会得到一个展开的数组 `[2,4]`，而不是 `[[2],[4]]`。同样，flatMap 一个流，得到的应该是 flat 过的流，那么这里产生的两个流，`1-1-1-1-1` 和 `2---2---2---2---2`，想象一下要把两个流展开放到一个流里，空间的元素放到数组中可以按空间排列，那么元素放到流中则应该按照时间排列，我们做一个简单的对比：

```
1-1-1-1-1
2- -2- -2--2—2

1   1   1
2-1-2-1-2--2--2
```

其中每一个 - 代表 1 秒，所以输出会是 `12-1-12-1-12--2--2`。数字之间没有 - 代表会同时打印，因此有可能会出现 2 在 1 前的可能，其实应该是同时的，如图 7-9 所示。

正是因为流可以是 Monadic 的，又可以响应式地在流的内容变化时做出相应

4 虽然不知道为什么官网并没有推荐（Deprecated）使用 lift，我觉得反倒是用 lift 更适合函数的重用。

动作，因此叫作 Monadic Reactive Stream。

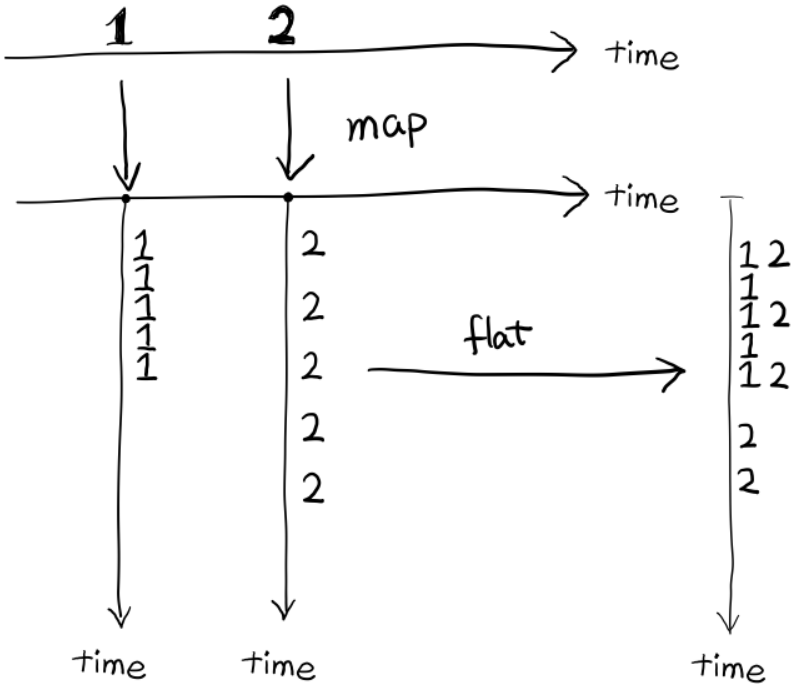

图 7-9　流的 flatMap

7.5.5　一个“简单”的 Reactive 实例

到底 Monadic Reactive 在前端是有多好用呢？我们用一个 React 的小例子来体现。当然 React 并不是 Reactive 的，因此我们需要使用另一个库 `react-most` 来将 most 的 reactive 技能融合到 React 中。

下面，我们来实现一个非常常见但有可能使用其他库实现比较复杂的功能。在输入的同时搜索并将结果更新到页面上。这么一个功能需要完成：

（1）响应用户（键盘输入）事件。

（2）但是又不能每次变动都响应，这样频繁请求会给服务器过大压力。我们设

置在 500ms 响应一次。

（3）500ms 后异步地发送到搜索的服务器（假设是 Github API）。

（4）数据返回后显示对应搜索结果的 repository 名字、作者名字及收藏数量。

完成后的效果应该是这样的[5]，如图 7-10 所示。

图 7-10　完成后的效果

- 一个简单的 Pure Component

React 什么的不是本书的重点，我们快快地过一遍就好，使用 React 创建一个 Component：

```
const TypeNsearch = (props)=>{
      let {search} = props.actions
      return <div>
         <input onChange={e=>search(e.target.value)}></input>
```

5　也可访问实例的网址 https://oyanglul.us/react-most/examples/type-n-search/public/。

```
            <ul>
            {
                props.results&&props.results.map(item=>{
                    return <li key={item.id}><a
href={item.html_url}>{item.full_name} ({item.stargazers_count})</a></li>
                })
            }
            </ul>
        </div>
    }
```

return 的地方就是 Virtual DOM 了，一个输入框 `input`，底下是一堆 `li`。当输入框内容发生改变，会调用 `search` 函数，该函数将输入框的值加入一个叫 *Intent Stream* 的流中。

好了，关于 React 要做的事情我们只需要了解这么多就好了。下面看看到底怎么 Reactive。

- Debounce

一旦把用户事件的值发送到 Intent Stream，响应事件的工作就算已经做完了。接下来实现第二步，至少间隔 500ms 才发送 API 请求。这意味着我们可以开始构建数据流了，让数据流中的类型为 `search` 的值 debounce 500ms 就好。

```
function(intent$){
  let updateSink$ = intent$.filter(i=>i.type=='search')
                   .debounce(500)
  ...
```

`debounce` 会把一个流转换成一个值间的间隔至少在给定时间的流。

```
--冷-冷笑--冷笑话-->

    --------冷笑话-->
```

- 发送 API 请求

接下来的事情就简单了，流上的值之间一定是间隔 500ms 以上，我们可以放心地直接通过这些值构造响应的 API 地址并发送请求。

```
  ...
```

```
.map(intent=>intent.value)
.filter(query=>query.length > 0)
.map(query=>GITHUB_SEARCH_API + query)
.map(rest)
...
```

rest 是一个 Isomophic（前后端同构） 的 JavaScript Restful 客户端。在拼好地址后可以简单地利用 rest 来发送请求，得到一个 Promise。

- 继续 flatMap 结果到流上

精彩的地方来了，当我们获得一个 Promise 后，如何将 Promise 内的一个异步的结果再作为流输出呢？这时候就可以派上我们的 Monad、Monadic 连接 Promise 和 Intent Stream。

```
.flatMap(request=>most.fromPromise(
                     request.then(resp=>({
                       type: 'dataUpdate',
                       value: resp.entity
                     }))))
```

其中的 `request` 是上一步 rest 返回的 Promise，在简单的格式转换后，使用 `most.fromPromise` 将其也转换成流。当得到 API 的返回组成的流之后，使用 `flatMap` 连到 Intent Stream 上。下面是 `flatMap` 两个 API 结果流的示意，以防读者忘了上一节的 `flatMap` 例子。

```
intentStream --urlA---urlB--->
rest(urlA)   -------respA---->
rest(urlB)   ---------respB-->
flatMap(rest)-------respA--respB--->
```

- Model

在得到一个里面都是 API 返回值的流之后，可以简单地模型化一下这条流的数据：

```
.filter(i=>i.type=='dataUpdate')
.map(data=>JSON.parse(data.value).items)
.map(items=>items.slice(0,10))
```

只取前 10 个结果作为例子。

最后，把结果映射成 state 到新 state 的函数：

```
.map(items=>state=>({results: items}))
modleStream ---mA---mB--->
stateStream ---state=>({results:mA})---state=>({results:mB})--->
```

接下来，react-most 会利用输出的 state 流中的函数调用 React 的 `setState` 方法[6]。

到这里，我们利用 Monadic Reactive Programming 的方式，Declarative 构造出了一整条从输入（用户事件）到该事件所产生的结果的数据流。其中没有一个变量与赋值操作，也没有任何状态和全局依赖，这样的数据流就跟纯函数一样，更易于推理和预测结果。而且由于 Promise 也是时间相关的容器，也可以轻松地转换成 Stream，因此无须关心异步编程，只需要掌控好数据流向与变换就好了。

7.6　小结

本章涉及许多有趣的概念，如果你在 Google 上搜索 Monad，会有大量的资料解释什么是 Monad，而且这些资料都有一个特点——很难看懂。但实际上，我们可能已经经常使用并乐在其中，因此我更想把重点放到如何应用 Monad 来进行 Monadic 编程与 Reactive 编程上。例如，Promise 和 Reactive 都已经是 JavaScript 编程中耳熟能详的概念。还是那句话，如果你只是想吃墨西哥鸡卷，只要有钱就好了，并不需要会说墨西哥语。

6　这不是本书的重点，但是我是受到 cycle.js 启发写了 react-most，从而可以描述式地写 react component。感兴趣可以到 react-most 的 Github 主页了解（https://github.com/jcouyang/react-most）。