

Web开发经典丛书

JavaScript ES6 函数式编程入门经典

Beginning Functional JavaScript

[印] Anto Aravinth

梁宵

著
译



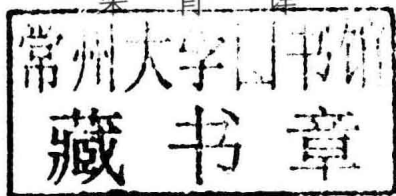
清华大学出版社

Web 开发经典丛书

JavaScript ES6 函数式 编程入门经典

[印] Anto Aravinth 著

梁 宵 译



清华大学出版社

北 京

Anto Aravinth

Beginning Functional JavaScript

EISBN: 978-1-4842-2655-1

Original English language edition published by Apress Media. Copyright © 2017 by Anto Aravinth. Simplified Chinese-Language edition copyright © 2017 by Tsinghua University Press. All rights reserved.

本书中文简体字版由 Apress 出版公司授权清华大学出版社出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字：01-2017-5756

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

图书在版编目(CIP)数据

JavaScript ES6 函数式编程入门经典/ (印) 安东尼奥·阿内维斯(Anto Aravinth) 著；梁宵 译。—北京：清华大学出版社，2018

(Web 开发经典丛书)

书名原文：Beginning Functional JavaScript

ISBN 978-7-302-48714-2

I. ①J… II. ①安… ②梁… III. ①JAVA 语言—程序设计 IV. ①TP312.8

中国版本图书馆 CIP 数据核字(2017)第 271335 号

责任编辑：王 军 于 平

封面设计：牛艳敏

版式设计：思创景点

责任校对：曹 阳

责任印制：刘海龙

出版发行：清华大学出版社

网 址：<http://www.tup.com.cn>, <http://www.wqbook.com>

地 址：北京清华大学学研大厦 A 座 邮 编：100084

社 总 机：010-62770175

邮 购：010-62786544

投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈：010-62772015, zhiliang@tup.tsinghua.edu.cn

印 装 者：北京嘉实印刷有限公司

经 销：全国新华书店

开 本：148mm×210mm

印 张：5.875

字 数：158 千字

版 次：2018 年 1 月第 1 版

印 次：2018 年 1 月第 1 次印刷

印 数：1~3000

定 价：49.80 元

产品编号：076421-01

译者序

函数式编程是一种古老的编程范式。近些年来，随着 RxJS 等函数式框架的流行，它焕发了青春，再次进入了我们的视野。与 Haskell 等语言相比，JavaScript 虽然不是一种纯函数语言，但它将函数视为一等公民，非常适合函数式编程范式。函数式编程为应用带来的可维护性、可测试性和可扩展性是不言而喻的，而纯函数、高阶函数、柯里化、组合、Monad 等诸多概念往往令刚刚接触它的人无从下手。

快速掌握一个知识体系的秘诀是抓住概念并理清概念之间的关系。本书将函数式编程中那些抽象的原理分解为一个个简单的概念，娓娓道来，并配以丰富的实战案例，逐步带你领略函数式编程的魅力。掌握函数式编程思想对开发与理解单数据流应用非常有帮助，愿本书带你开启这段非凡的旅程！

本译作能够顺利完成，首先感谢清华大学出版社李阳老师的推荐与信任，提供的非常有价值的建议使我在翻译的过程中受益良多。感谢我的妻子对我的理解与支持。感谢如天使般可爱的女儿 Eva，你是上天赐给我最好的礼物。本书全部内容由梁宵翻译，参与翻译的还有腾讯高级工程师王志寿和 Uber 高级工程师罗誉家。

在翻译过程中我尽力修正了一些原作的小错误，但由于水平有限，难免存在不足之处，恳请广大读者不吝惠赐。

梁 宵

作者简介

Anto Aravinth 是来自 VisualBI Chennai 研发中心的高级商业智能开发工程师。在过去的五年中，他曾使用 Java、JavaScript 语言以及 ReactJs、Angular 等框架开发 Web 应用。他对 Web 和 Web 标准有透彻的理解。他也是流行框架 ReactJs、Selenium 和 Groovy 的开源贡献者。

Anto Aravinth 在业余时间喜欢打乒乓球。他很有幽默感！他也是 *React Quickly* 一书的技术开发编辑，此书在 2017 年由 Manning 出版社出版。

致 谢

撰写一本书没有我想象的那么简单，整个过程几乎像拍电影一样。要根据书的目录仔细推敲每一个单元。目录就像电影的脚本，它需要一个震撼的开场，然后吊住观众的胃口，最后呈现出一个完美的结局。一本优秀的剧本要通过生动的文字传达出来。当编辑团队认可目录时，写书的过程就开始了。为此，我要感谢 Pramila，她在本书的开始阶段帮助了我。当然，写一本技术书就需要技术纠错。为此，特别感谢技术编辑团队！他们非常善于在书写中找出技术问题。特别感谢 Anila，她检查了所有章节并找到了语法错误——确保了将优质的内容呈现给读者。所有上述过程由经理 Prachi 管理，感谢 Prachi 让这一切变为现实！

我要将本书献给已故的父亲 Belgin Rayen 和挚爱的母亲 Susila。我也要感谢姐夫 Kishore，他在生活和事业上一直支持我。我从未告诉唯一的同胞姐姐 Ramya 我在写一本书。我只是无法预料她对此事的反应。也特别感谢她。

特别感谢所有在职业生涯中给予我支持的朋友和同事：Deepak、Vishal、Shiva、Mustafa、Anand、Ram (Juspay)、Vimal (Juspay)、Lalitha、Swetha、Vishwapriya。最后感谢我亲密的兄弟姐妹：Bianca、Jennifer、Amara、Arun、Clinton、Shiny、Sanju。

我在书写、内容、行文等方面还有待改进。如果你愿意分享你的想法，
请通过 antoaravinthrayen@gmail.com 联系我。我的 twitter 是 @antoaravinth。
感谢你购买本书！希望你能喜欢它。祝你好运！

Anto Aravinth, 于印度

目 录

第 1 章 函数式编程简介	1
1.1 什么是函数式编程？为何它重要	1
1.2 引用透明性	4
1.3 命令式、声明式与抽象	5
1.4 函数式编程的好处	7
1.5 纯函数	7
1.5.1 纯函数产生可测试的代码	7
1.5.2 合理的代码	9
1.6 并发代码	10
1.7 可缓存	11
1.8 管道与组合	12
1.9 纯函数是数学函数	13
1.10 我们要构建什么	15
1.11 JavaScript 是函数式编程语言吗	15
1.12 小结	16
第 2 章 JavaScript 函数基础	17
2.1 ECMAScript 历史	18
2.2 创建并执行函数	19

2.2.1	第一个函数	19
2.2.2	严格模式	21
2.2.3	return 语句是可选的	22
2.2.4	多语句函数	22
2.2.5	函数参数	24
2.2.6	ES5 函数在 ES6 中是有效的	24
2.3	设置项目	24
2.3.1	初始设置	24
2.3.2	用第一个函数式方法处理循环问题	26
2.3.3	export 要点	28
2.3.4	import 要点	28
2.3.5	使用 babel-node 运行代码	29
2.3.6	在 npm 中创建脚本	30
2.3.7	从 git 上运行源代码	31
2.4	小结	31
第 3 章	高阶函数	33
3.1	理解数据	34
3.1.1	理解 JavaScript 数据类型	34
3.1.2	存储函数	35
3.1.3	传递函数	35
3.1.4	返回函数	37
3.2	抽象和高阶函数	38
3.2.1	抽象的定义	38
3.2.2	通过高阶函数实现抽象	39
3.3	真实的高阶函数	42
3.3.1	every 函数	42
3.3.2	some 函数	44
3.3.3	sort 函数	44
3.4	小结	48
第 4 章	闭包与高阶函数	49
4.1	理解闭包	50

4.1.1	什么是闭包	50
4.1.2	记住闭包生成的位置	52
4.1.3	回顾 sortBy 函数	53
4.2	真实的高阶函数 (续)	54
4.2.1	tap 函数	54
4.2.2	unary 函数	56
4.2.3	once 函数	57
4.2.4	memoized 函数	58
4.3	小结	60
第 5 章	数组的函数式编程	61
5.1	数组的函数式方法	62
5.1.1	map	62
5.1.2	filter	65
5.2	连接操作	67
5.3	reduce 函数	71
5.4	zip 数组	77
5.5	小结	81
第 6 章	柯里化与偏应用	83
6.1	一些术语	84
6.1.1	一元函数	84
6.1.2	二元函数	84
6.1.3	变参函数	84
6.2	柯里化	86
6.2.1	柯里化用例	87
6.2.2	日志函数——应用柯里化	89
6.2.3	回顾 curry	90
6.2.4	回顾日志函数	93
6.3	柯里化实战	94
6.3.1	在数组内容中查找数字	94
6.3.2	求数组的平方	95
6.4	数据流	96

6.4.1	偏应用	96
6.4.2	实现偏函数	97
6.4.3	柯里化与偏应用	99
6.5	小结	100
第7章	组合与管道	101
7.1	组合的概念	102
7.2	函数式组合	104
7.2.1	回顾 map 与 filter	104
7.2.2	compose 函数	106
7.3	应用 compose 函数	106
7.3.1	引入 curry 与 partial	108
7.3.2	组合多个函数	111
7.4	管道/序列	113
7.5	组合的优势	114
7.5.1	组合满足结合律	114
7.5.2	使用 tap 函数调试	115
7.6	小结	116
第8章	函子	117
8.1	什么是函子	118
8.1.1	函子是容器	118
8.1.2	函子实现了 map 方法	120
8.2	Maybe 函子	121
8.2.1	实现 Maybe 函子	122
8.2.2	简单用例	123
8.2.3	真实用例	125
8.3	Either 函子	129
8.3.1	实现 Either 函子	130
8.3.2	reddit 例子的 Either 版本	131
8.4	Pointed 函子	134
8.5	小结	134

第 9 章 深入理解 Monad	135
9.1 根据搜索词条获取 Reddit 评论	136
9.2 问题描述	136
9.2.1 实现第一步	138
9.2.2 合并 Reddit 调用	141
9.2.3 多个 map 的问题	144
9.3 通过 join 解决问题	146
9.3.1 实现 join	146
9.3.2 实现 chain	148
9.4 小结	151
第 10 章 使用 Generator	153
10.1 异步代码及其问题	154
10.2 Generator 基础	156
10.2.1 创建 Generator	156
10.2.2 Generator 的注意事项	157
10.2.3 yield 关键字	158
10.2.4 done 属性	160
10.2.5 向 Generator 传递数据	162
10.3 使用 Generator 处理异步调用	164
10.3.1 一个简单的案例	164
10.3.2 一个真实的案例	169
10.4 小结	172
附录	173

第 1 章



函数式编程简介

函数的第一条原则是要小。函数的第二条原则是要更小。

—ROBERT C. MARTIN

欢迎来到函数式编程的世界。在这个只有函数的世界中，我们愉快地生活着，没有任何外部环境的依赖，没有状态，没有突变——永远没有。函数式编程是最近的一个热点。你可能在团队中和小组会议中听说过这个术语，或许还做过一些思考。如果你已经了解了它的含义，非常好！但是那些不知道的人也不必担心。本章的目的就是：用通俗的语言为你介绍函数式编程。

我们将以一个简单的问题开始本章：数学中的函数是什么？随后给出函数的定义并用其创建一个简单的 JavaScript 函数示例。本章结尾将说明函数式编程带给开发者的好处。

1.1 什么是函数式编程？为何它重要

在开始了解函数式编程这个术语的含义之前，我们要回答另一个问题：数学中的函数是什么？数学中的函数可以写成如下形式：

$$f(x) = y$$

这条语句可以被解读为“一个函数 F ，以 x 作为参数，并返回输出

Y。”例如，X 和 Y 可以是任意的数字。这是一个非常简单的定义，但是其中包含了几个关键点：

- 函数必须总是接受一个参数。
- 函数必须总是返回一个值。
- 函数应该依据接收到的参数(例如 X)而不是外部环境运行。
- 对于一个给定的 X，只会输出唯一的一个 Y。

你可能想知道为什么我们要了解数学中的函数定义而不是 JavaScript 中的。你是这样想的吗？对于我来说这是一个值得思考的问题。答案非常简单：函数式编程技术主要基于数学函数和它的思想。但是等等——我们并不是要在数学中教你函数式编程，而是使用 JavaScript 来传授该思想。但是贯穿全书，我们将看到数学函数的思想和用法，以便能够理解函数式编程。

有了数学函数的定义，下面来看看 JavaScript 函数的例子。

假设我们要编写一个计税函数。在 JavaScript 中你会如何做？

注意

本书的所有例子都用 ES6 编写。书中的代码片段是独立的，所以你可以复制并把它们粘贴到任意喜欢的支持 ES6 的浏览器中。所有的例子可以在 Chrome 浏览器的 51.0.2704.84 版本中运行。ES6 的规范请参见：<http://www.ecma-international.org/ecma-262/6.0/>。

我们可以实现如代码清单 1-1 所示的函数。

代码清单 1-1 用 ES6 编写的计税函数

```
var percentValue = 5;  
var calculateTax = (value) => { return value/100 * (100 + percentValue) }
```

上面的 `calculateTax` 函数准确地实现了我们的想法。你可以用参数调用该函数，它会在控制台中返回计算后的税值。该函数看上去很整洁，不是吗？让我们暂停一下，用数学的定义分析一下它。数学函数定义的关键是函数逻辑不应依赖外部环境。在 `calculateTax` 函数中，我们让函数依赖全局变量 `percentValue`。因此，该函数在数学意义上就不能被称

为一个真正的函数。下面将修复该问题。请思考一下，为什么不能在模板中改变字体？

修复方法非常简单：我们只需要移动 `percentValue`，把它作为函数的参数。见代码清单 1-2。

代码清单 1-2 重写计税函数

```
var calculateTax = (value, percentValue) => { return value/100 * (100 + percentValue) }
```

现在 `calculateTax` 函数可以被称为一个真正的函数了。但是我们得到了什么？我们只是在其内部消除了对全局变量的访问。移除一个函数内部对全局变量的访问会使该函数的测试更容易(我们将在本章的稍后部分讨论函数式编程的好处)。

现在我们了解了数学函数与 JavaScript 函数的关系。通过这个简单的练习，我们就能用简单的技术术语定义函数式编程。函数式编程是一种范式，我们能够以此创建仅依赖输入就可以完成自身逻辑的函数。这保证了当函数被多次调用时仍然返回相同的结果。函数不会改变任何外部环境的变量，这将产生可缓存的、可测试的代码库。

函数与 JavaScript 方法

前面介绍了很多有关“函数”的内容。在继续之前，我想确保你理解了函数和 JavaScript 方法之间的区别。

简言之，**函数**是一段可以通过其名称被调用的代码。它可以传递参数并返回值。

然而，**方法**是一段必须通过其名称及其关联对象的名称被调用的代码。

下面快速看一下函数和方法的例子，如代码清单 1-3 和代码清单 1-4 所示。

函数

代码清单 1-3 一个简单的函数

```
var simple = (a) => {return a} // 一个简单的函数
```

```
simple(5) // 用其名称调用
```

方法

代码清单 1-4 一个简单的方法

```
var obj = {simple : (a) => {return a} }  
obj.simple(5) // 用其名称及其关联对象调用
```

在函数式编程的定义中还有两个重要的特性并未提及。在研究函数式编程的好处之前，我们将在下一节详细阐述。

1.2 引用透明性

根据函数的定义，我们可以得出结论：所有的函数对于相同的输入都将返回相同的值。函数的这一属性被称为引用透明性(Referential Transparency)。下面举一个简单的例子，如代码清单 1-5 所示：

代码清单 1-5 引用透明性的例子

```
var identity = (i) => { return i }
```

在上面的代码片段中，我们定义了一个简单的函数 `identity`。无论传入什么作为输入，该函数都会把它返回。也就是说，如果你传入 5，它就会返回 5(换言之，该函数就像一面镜子或一个恒等式)。注意，我们的函数只根据传入的参数“i”进行操作，在函数内部没有全局引用(记住代码清单 1-2，我们从全局访问中移除了“percentValue”并把它作为一个传入的参数)。该函数满足了引用透明性条件。现在假设该函数被用于其他函数调用之间，如下所示：

```
sum(4,5) + identity(1)
```

根据引用透明性的定义，我们可以把上面的语句转换为：

```
sum(4,5) + 1
```

该过程被称为替换模型(Substitution Model)，因为你可以直接替换

函数的结果(主要因为函数的逻辑不依赖其他全局变量),这与它的值是一样的。这使并发代码和缓存成为可能。根据该模型想象一下,你可以轻松地用多线程运行上面的代码,甚至不需要同步!为什么?同步的问题在于线程不应该在并发运行的时候依赖全局数据。遵循引用透明性的函数只能依赖来自参数的输入。因此,线程可以自由地运行,没有任何锁机制!

由于函数会为给定的输入返回相同的值,实际上我们就可以缓存它了!例如,假设有一个函数“factorial”计算给定数值的阶乘。“factorial”接受输入作为参数以计算其阶乘。我们都知道“5”的“factorial”是“120”。如果用户第二次调用“5”的“factorial”,情况会如何呢?如果“factorial”函数遵循引用透明性,我们知道结果将依然是“120”(并且它只依赖输入参数)。记住这个特性后,就能够缓存“factorial”函数的值。因此,如果“factorial”以“5”作为输入被第二次调用,就能够返回已缓存的值,而不必再计算一次。

在此可以看到,引用透明性在并发代码和可缓存代码中发挥着重要的作用。本章的稍后部分将编写一个用于缓存函数结果的库函数。

引用透明性是一种哲学

“引用透明性”一词来自分析哲学(https://en.wikipedia.org/wiki/Analytical_philosophy)。该哲学分支研究自然语言的语义及其含义。单词“Referential”或“Referent”意指表达式引用的事物。句子中的上下文是“引用透明的”,如果用另一个引用相同实体的词语替换上下文中的一个词语,并不会改变句子的含义。

这就是我们在本节定义的引用透明性。替换函数的值并不影响上下文。这就是函数式编程的哲学!

1.3 命令式、声明式与抽象

函数式编程主张声明式编程和编写抽象的代码。在更进一步介绍之