Web开发经典丛书

JavaScript ES6 函数式编程入门经典

Beginning Functional JavaScript

[印] Anto Aravinth 著
梁宵 译

Apress®
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

北京市版权局著作权合同登记号　图字：01-2017-5756

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

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

前，我们需要理解这两个术语。我们都知道并使用过多种命令式范式。下面以一个问题为例，看看如何用命令式和声明式的方法解决它。

假设有一个数组，你想遍历它并把它打印到控制台。代码如代码清单 1-6 所示：

代码清单 1-6　用命令式方法遍历数组

```
var array = [1,2,3]
for(i=0;i<array.length;i++)
console.log(array[i]) // 打印 1, 2, 3
```

这段代码运行良好。但是为了解决问题，我们精确地告诉程序应该“如何”做。例如，我们用数组长度的索引计算结果编写了一个隐式的 for 循环并打印出数组项。在此暂停一下。我们的任务是什么？“打印数组的元素”，对不对？但是看起来我们像在告诉编译器该做什么。在本例中，我们在告诉编译器“获得数组长度，循环数组，用索引获取每一个数组元素，等等。”我们将之称为“命令式”解决方案。命令式编程主张告诉编译器“如何”做。

现在我们来看另一方面，声明式编程。在声明式编程中，我们要告诉编译器做“什么”，而不是“如何”做。“如何”做的部分将被抽象到普通函数中(这些函数被称为高阶函数，我们会在后续的章节中介绍)。现在我们可以用内置的 forEach 函数遍历数组并打印它。见代码清单 1-7。

代码清单 1-7　用声明式方法遍历数组

```
var array = [1,2,3]
array.forEach((element) => console.log(element))// 打印 1, 2, 3
```

上面的代码片段打印了与代码清单 1-6 相同的输出。但是我们移除了“如何”做的部分，比如“获得数组长度，循环数组，用索引获取每一个数组元素，等等。”我们使用了一个处理“如何”做的抽象函数，如此可以让开发者只需要关心手头的问题(做“什么”的部分)。这非常棒！贯穿本书，我们都将创建这样的内置函数。

函数式编程主张以抽象的方式创建函数，这些函数能够在代码的其

他部分被重用。现在我们对什么是函数式编程有了透彻的理解。基于这一点，我们就能够去研究函数式编程的好处了。

1.4　函数式编程的好处

我们了解了函数式编程的定义和一个非常简单的 JavaScript 函数。但是不得不回答一个简单的问题：“函数式编程的好处是什么？”这一节将帮助你透过现象看本质，了解函数式编程带给我们的巨大好处！大多数函数式编程的好处来自于编写纯函数。所以在此之前，我们将了解一下什么是纯函数。

1.5　纯函数

有了前面的定义，我们就能够定义纯函数的含义。纯函数是对给定的输入返回相同的输出的函数。举一个例子，见代码清单 1-8：

代码清单 1-8　一个简单的纯函数

```
var double = (value) => value * 2;
```

上面的函数“double”是一个纯函数，只因为给它一个输入，它总是返回相同的输出。你不妨自己试试。用输入 5 调用 double 函数总是返回结果 10！纯函数遵循引用透明性。因此，我们能够毫不犹豫地用 10 替换 double(5)。

所以，纯函数了不起的地方是什么？它能带给我们很多好处。下面依次讨论。

1.5.1　纯函数产生可测试的代码

不纯的函数具有副作用。下面以前面的计税函数为例进行说明(代码清单 1-1)：

```
var percentValue = 5;
var calculateTax = (value) => { return value/100 * (100 +
 percentValue) } //
 依赖外部环境的 percentValue 变量
```

函数 calculateTax 不是纯函数，主要因为它依赖外部环境计算其逻辑。尽管该函数可以运行，但非常难于测试！下面看看原因。

假设我们打算对 calculateTax 函数运行测试，分别执行三次不同的税值计算。按如下方式设置环境：

```
calculateTax(5) === 5.25

calculateTax(6) === 6.3

calculateTax(7) === 7.3500000000000005
```

整个测试通过了！但是别急，既然原始的 calculateTax 函数依赖外部环境变量 percentValue，就有可能出错。假设你在运行相同的测试用例时，外部环境也正在改变变量 percentValue：

```
calculateTax(5) === 5.25

// percentValue 被其他函数改成 2
calculateTax(6) === 6.3 // 这条测试能通过吗？

// percentValue 被其他函数改成 0
calculateTax(7) === 7.3500000000000005 // 这条测试能通过吗，还是
会抛出异常？
```

如你所见，此时的 calculateTax 函数很难测试。但是我们可以很容易地修复这个问题，从该函数中移除外部环境依赖，代码如下：

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

现在可以顺畅地测试 calculateTax 函数了！在结束本节前，我们需要提及纯函数的一个重要属性，即“纯函数不应改变任何外部环境的变量。”

换言之，纯函数不应依赖任何外部变量(就像例子中展示的那样)，也不应改变任何外部变量。我们通过改变任意一个外部变量就能马上理解其中的含义。例如，考虑代码清单 1-9：

代码清单 1-9　badFunction 例子

```
var global = "globalValue"
var badFunction = (value) => { global = "changed"; return value * 2 }
```

当 badFunction 函数被调用时，它将全局变量 global 的值改成 changed。需要担心这件事吗？是的！假设另一个函数的逻辑依赖 global 变量！因此，调用 badFunction 就影响了其他函数的行为。具有这种性质的函数(也就是具有副作用的函数)会使代码库变得难以测试。除了测试，在调试的时候这些副作用会使系统的行为变得非常难以预测！

至此，我们通过简单的示例了解到纯函数有助于我们更容易地测试代码。现在来看一下纯函数的其他好处——**合理的代码**。

1.5.2　合理的代码

作为开发者，我们应该善于推理代码或函数。通过创建和使用纯函数，能够非常简单地实现该目标。为了明确这一点，我们将使用一个简单的 double 函数(来自代码清单 1-8)：

```
var double = (value) => value * 2
```

通过函数的名称能够轻易地推理出：这个函数把给定的数值加倍，其他什么也没做！事实上，根据引用透明性概念，我们可以简单地用相应的结果替换 double 函数调用！开发者的大部分时间花在阅读他人的代码上。在代码库中包含具有副作用的函数对团队中的其他开发者来说是难以阅读的。包含纯函数的代码库会易于阅读、理解和测试。记住，函数(无论它是否为纯函数)必须总是具有一个有意义的名称。按照这种说法，在给定行为后你不能将函数“double”命名为“dd”。

小脑力游戏

我们只需要用值替换函数，就好像不看它的实现就知道结果一样！这是你在理解函数思想过程中的一个巨大进步。我们取代函数值，就好像这是它要返回的结果！

为了快速练习一下你的脑力，下面用内置的 Math.max 函数测试一

下你的推理能力。

给定函数调用：

```
Math.max(3,4,5,6)
```

结果是什么？

为了给出结果，你看了 max 的实现了吗？没有，对不对？为什么？答案是 Math.max 是纯函数。现在喝一杯咖啡吧，你已经完成了一项伟大的工作！

1.6 并发代码

纯函数总是允许我们并发地执行代码。因为纯函数不会改变它的环境，这意味着我们根本不需要担心同步问题！当然，JavaScript 并没有真正的多线程用来并发地执行函数，但是如果你的项目使用了 WebWorker 来并发地执行多任务，该怎么办呢？或者有一段 Node 环境中的服务端代码需要并发地执行函数，又该怎么办呢？

例如，假设我们有代码清单 1-10 给出的如下代码：

代码清单 1-10　非纯函数

```
let global = "something"
let function1 = (input) => {
// 处理 input
// 改变 global
global = "somethingElse"

}
let function2 = () => {
        if(global === "something")
        {
                // 业务逻辑
        }
}
```

如果我们需要并发地执行 function1 和 function2，该怎么办呢？假设线程一(T-1)选择 function1 执行，线程二(T-2)选择 function2 执行。现

在两个线程都准备好执行了，那么问题来了。如果 T-1 在 T-2 之前执行，情况会如何？由于两个函数(function1 和 function2)都依赖全局变量 global，并发地执行这些函数就会引起不良的影响。现在把这些函数改为纯函数，如代码清单 1-11 所示：

代码清单 1-11　纯函数

```
let function1 = (input,global) => {
        // 处理 input
        // 改变 global
        global = "somethingElse"
}
let function2 = (global) => {
        if(global === "something")
        {
                // 业务逻辑
        }
}
```

此处我们移动了 global 变量，把它作为两个函数的参数，使它们变成纯函数。现在可以并发地执行这两个函数了，不会带来任何问题。由于函数不依赖外部环境(global 变量)，因此我们不必再像代码清单 1-10 那样担心线程的执行顺序。

本节说明了纯函数是如何使代码并发执行的，你不必担心任何问题。

1.7　可缓存

既然纯函数总是为给定的输入返回相同的输出，那么我们就能够缓存函数的输出。讲得更具体些，请看下面的例子。假设有一个做耗时计算的函数，名为 longRunningFunction：

```
var longRunningFunction = (ip) => { //do long running tasks and return }
```

如果 longRunningFunction 函数是纯函数，我们知道对于给定的输入，它总会返回相同的输出！考虑到这一点，为什么要通过多次的输入来反复调用该函数呢？不能用函数的上一个结果代替函数调用吗？

(此处再次注意我们是如何使用引用透明性概念的，因此，用上一个结果值代替函数不会改变上下文)。假设我们有一个记账对象，它存储了longRunningFunction 函数的所有调用结果，如下所示：

```
var longRunningFnBookKeeper = { 2 : 3, 4 : 5 . . . }
```

longRunningFnBookKeeper 是一个简单的 JavaScript 对象，存储了所有的输入(key)和输出(value)，它是 longRunningFunction 函数的调用结果。现在使用纯函数的定义，我们能够在调用原始函数之前检查 key 是否在 longRunningFnBookKeeper 中，如代码清单 1-12 所示：

代码清单 1-12　通过纯函数缓存结果

```
var longRunningFnBookKeeper = { 2 : 3, 4 : 5 }
// 检查 key 是否在 longRunningFnBookKeeper 中
// 如果在，则返回结果，否则更新记账对象
longRunningFnBookKeeper.hasOwnProperty(ip) ?
      longRunningFnBookKeeper[ip] :
      longRunningFnBookKeeper[ip] = longRunningFunction(ip)
```

上面的代码相当直观。在调用真正的函数之前，我们用相应的 ip 检查函数的结果是否在记账对象中。如果在，则返回之，否则就调用原始函数并更新记账对象中的结果。看到了吗？用更少的代码很容易使函数调用可缓存。这就是纯函数的魅力！

在本书后面，我们将编写一个使用纯函数调用的用于处理缓存或技术性记忆(technical memorization)的函数库。

1.8　管道与组合

使用纯函数，我们只需要在函数中做一件事。纯函数能够自我理解，通过其名称就能知道它所做的事情。纯函数应该被设计为只做一件事。只做一件事并把它做到完美是 UNIX 的哲学，我们在实现纯函数时也将遵循这一原则。UNIX 和 LINUX 平台有很多用于日常任务的命令。例如，cat 用于打印文件内容，grep 用于搜索文件，wc 用于计算行数等。这些命令的确一次只解决一个问题。但是我们可以用组合或管道来完成

复杂的任务。假如我们要在一个文件中找到一个特定的名称并统计它的出现次数。在命令提示符中要如何做？命令如下：

```
cat jsBook | grep -i "composing" | wc
```

上面的命令通过组合多个函数解决了我们的问题。组合不是UNIX/LINUX 命令行独有的，但它们是函数式编程范式的核心。我们把它们称为函数式组合(Functional Composition)。假设同样的命令行在JavaScript 函数中已经实现了，我们就能够根据同样的原则使用它们来解决问题！

现在考虑用一种不同的方式解决另一个问题。你想计算文本中的行数。如何解决呢？你已经有了答案。不是吗？

根据我们的定义，命令实际上是一种纯函数。它接受参数并向调用者返回输出，不改变任何外部环境！

注意

也许你在想，JavaScript 支持用于组合函数的操作符“|”吗？答案是否定的，但是我们可以创建一个。后面的章节将创建相应的函数。

遵循一个简单的定义，我们收获了很多好处。在结束本章之前，我想说明纯函数与数学函数之间的关系。

1.9　纯函数是数学函数

在 1.7 节“可缓存”中我们见过如下一段代码(代码清单 1-12)：

```
var longRunningFunction = (ip) => { // 执行长时间运行的任务并返回
var longRunningFnBookKeeper = { 2 : 3, 4 : 5 }
// 检查 key 是否在 longRunningFnBookKeeper 中
// 如果在，则返回结果，否则更新记账对象
longRunningFnBookKeeper.hasOwnProperty(ip) ?
      longRunningFnBookKeeper[ip] :
      longRunningFnBookKeeper[ip] = longRunningFunction(ip)
```

这段代码的主要目的是缓存函数调用。我们通过记账对象实现了该功能。假设我们多次调用了 longRunningFunction，longRunningFnBook-Keeper 增长为如下的对象：

```
longRunningFnBookKeeper = {
   1 : 32,
   2 : 4,
   3 : 5,
   5 : 6,
   8 : 9,
   9 : 10,
   10 : 23,
   11 : 44
}
```

现在假设 longRunningFunction 的输入范围限制为 1-11 的整数(正如例子所示)。由于我们已经为这个特别的范围构建了记账对象，因此只能参照 longRunningFnBookKeeper 来为给定的输入返回输出。

下面分析一下该记账对象。该对象为我们清晰地描绘出，函数 longRunningFunction 接受一个输入并为给定的范围(在这个例子中，是 1-11)映射输出。此处的关键是，输入(在这个例子中，是 key)具有强制的、相应的输出(在这个例子中，是结果)。在 key 中也不存在映射两个输出的输入。

通过上面的分析，我们再看一下数学函数的定义(这次是来自维基百科的更具体的定义，网址为 https://en.wikipedia.org/wiki/Function_(mathematics))：

在数学中，函数是一种输入集合和可允许的输出集合之间的关系，具有如下属性: 每个输入都精确地关联一个输出。函数的输入称为参数，输出称为值。对于一个给定的函数，所有被允许的输入集合称为该函数的定义域，而被允许的输出集合称为值域。

上面的定义与纯函数完全一致！看一下 longRunningFnBookKeeper 对象。你能找到函数的定义域和值域吗？当然可以！通过这个非常简单的例子，很容易看到数学函数的思想被借鉴到函数式范式的世界(正如本章开始阐述的那样)。

1.10　我们要构建什么

本章介绍了很多关于函数和函数式编程的知识。有了这些基础知识，我们将构建一个名为 ES6-Functional 的函数式库。这个库将在全书中逐章地构建。通过构建这个函数式库，你将探索如何使用 JavaScript 函数，以及如何在日常工作中应用函数式编程(使用创建的函数解决代码库中的问题)！

1.11　JavaScript 是函数式编程语言吗

在结束本章之前，我们要回答一个基础的问题。JavaScript 是函数式编程语言吗？答案不置可否。在本章的开头，我们说函数式编程主张函数必须接受至少一个参数并返回一个值。不过坦率地讲，我们可以创建一个不接受参数并且实际上什么也不返回的函数。例如，下面的代码在 JavaScript 引擎中是一段有效的代码：

```
var useless = () => {}
```

上面的代码在 JavaScript 中执行时不会报错！原因是 JavaScript 不是一种纯函数语言(比如 Haskell)，而更像是一种多范式语言。但是如本章所讨论的，这门语言非常适合函数式编程范式。到目前为止，我们讨论的技术和好处都可以应用于纯 JavaScript！这就是书名的由来！

JavaScript 语言支持将函数作为参数，以及将函数传递给另一函数等特性——主要原因是 JavaScript 将函数视为一等公民(我们将在后续章节做更多的讨论)。由于函数定义的约束，开发者需要在创建 JavaScript 函数时将其考虑在内。如此，我们就能从函数式编程中获得很多优势，正如本章中讨论的一样。

1.12 小结

在本章中，我们介绍了在数学和编程世界中函数的定义。我们从数学函数的简单定义开始，研究了短小而透彻的函数例子和 JavaScript 中的函数式编程。还定义了什么是纯函数并详细讨论了它们的益处。在本章结尾，我们说明了纯函数和数学函数之间的关系。还讨论了 JavaScript 如何被视为一门函数式编程语言。通过本章的学习，你将收获颇丰。

在下一章中，我们将学习用 ES6 创建并执行函数。用 ES6 创建函数有多种方式，我们将在下一章中学习这些方式！