JavaScript ES8 函数式编程实践入门

(第 2 版)

[印] 安托 • 阿拉文思(Anto Aravinth)
斯里坎特 • 马基拉朱(Srikanth Machiraju) 著
梁　平 译

清华大学出版社

北　京

北京市版权局著作权合同登记号　图字：01-2021-5978
First published in English under the title
Beginning Functional JavaScript: Uncover the Concepts of Functional Programming with EcmaScript 8, Second Edition
by Anto Aravinth, Srikanth Machiraju
Copyright © Anto Aravinth and Srikanth Machiraju, 2020
This edition has been translated and published under licence from Apress Media, LLC, part of Springer Nature.

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。

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。举报：010-62782989，beiqinquan@tup.tsinghua.edu.cn。

图书在版编目(CIP)数据

JavaScript ES8函数式编程实践入门：第2版 / (印)安托 • 阿拉文思(Anto Aravinth)，(印)斯里坎特 • 马基拉朱(Srikanth Machiraju) 著；梁平译. —北京：清华大学出版社，2022.1

书名原文：Beginning Functional JavaScript: Uncover the Concepts of Functional Programming with EcmaScript 8, Second Edition

ISBN 978-7-302-59777-3

Ⅰ. ①J… Ⅱ. ①安… ②斯… ③梁… Ⅲ. JAVA 语言—程序设计 Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字(2022)第 000538 号

责任编辑：王　军
装帧设计：孔祥峰
责任校对：成凤进
责任印制：杨　艳

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：010-83470000　　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：三河市东方印刷有限公司
经　　销：全国新华书店
开　　本：148mm×210mm　　**印　　张**：8　　**字　　数**：230 千字
版　　次：2022 年 3 月第 1 版　　**印　　次**：2022 年 3 月第 1 次印刷
定　　价：59.80 元

产品编号：092508-01

译 者 序

函数式编程是一种编程典范，它将计算机运算视为函数的计算。函数编程语言最重要的基础是λ演算(lambda calculus)。而且λ演算的函数能以函数作为输入(参数)和输出(返回值)。和指令式编程相比，函数式编程认为函数的计算比指令的执行重要。不同于过程化编程，在函数式编程中，函数的计算可随时调用。

在函数式编程中，编程人员可用一个天然框架开发更小、更简单和更一般化的模块，然后将它们组合在一起。函数式编程的一些基本特点包括：支持闭包和高阶函数，支持惰性计算、折叠递归、折叠引用透明性等。

JavaScript 是一种直译式脚本语言，是一种动态类型、弱类型、基于原型的语言，内置支持类型。它的解释器被称为 JavaScript 引擎，为浏览器的一部分，广泛用于客户端的脚本语言，最早应用于 HTML(标准通用标记语言下的一个应用)网页，目的是给 HTML 网页增加动态功能。

JavaScript 非常容易学。本书着力介绍 JavaScript 当今主要特性的运用技巧，从基本概念开始，逐步介绍按照当今 Web 标准编写 JavaScript 代码的方式。本书内容包括函数式编程简介、JavaScript 函数基础、高阶函数、闭包与高阶函数、数组的函数式编程、柯里化与偏应用、组合与管道、函子、Monad、Generator 的用法、小型库的构建、测试方法与技巧。

无论你是资深 JavaScript 编程人员，还是零基础新手，都可从本书中获益！本书是介绍 JavaScript 编程的经典教程。

在这里要感谢清华大学出版社的编辑们，他们为本书的出版投入了巨大的热情并付出了很多心血。没有他们的帮助和鼓励，本书不可能顺利付梓。

对于这本经典之作，译者本着“诚惶诚恐”的态度，在翻译过程中力求“信、达、雅”，但是鉴于译者水平有限，不足在所难免，如有任何意见和建议，请不吝指正。

梁平

作 者 简 介

Anto Aravinth 从事软件行业已经 6 年多了。他开发了许多用最新技术编写的系统。Anto 了解 JavaScript 的基础知识及其工作方式，并培训了许多人。Anto 在业余时间也做 OSS，他喜欢打乒乓球。

Srikanth Machiraju 作为开发人员、架构师、技术培训师和社区发言人，拥有超过 10 年的工作经验。他目前在 Microsoft Hyderabad 担任高级顾问，领导一个由 100 名开发人员和质量分析师组成的团队，为石油行业的科技巨头开发一个先进的云计算平台。他的目标是成为一名企业架构师，能够智能设计超大规模的现代应用程序，不断学习和分享使用前沿平台和技术的现代应用程序开发策略。在加入 Microsoft 前，他曾在 BrainScale 担任企业培训师和高级技术分析师，负责应用程序设计、开发，并使用 Azure 进行迁移。他是一名精通技术的开发人员，热衷于拥抱新技术，并通过博客和社区分享他的学习历程。他还撰写了题为“Learning Windows Server Containers”(学习 Windows 服务器容器)和“Developing Bots with Microsoft Bot Framework”(用 Microsoft 机器人框架开发机器人)的博客文章。

技术审校者简介

Sakib Shaikh 一直在一家大型科学出版社担任技术主管，拥有超过 10 年的 JavaScript 前端和后端系统全栈开发经验。在过去几年里，他一直在审阅技术书籍和文章，并以培训师、博客写手和导师的身份为开发者社区做出贡献。

致　　谢

我还记得在第一份实习工作中为 Juspay 系统编写的第一段代码。编程对我来说很有趣；有时，这也是一种挑战。现在，我有 6 年的软件开发经验，我想把我所有的知识都传递给社区。我喜欢教书，喜欢与社区分享我的想法，以获得反馈。这正是我撰写本书第 2 版的原因。

我不得不承认，在我经历的人生各个阶段一直站在我身边的人屈指可数：我的父亲 Belgin Rayen、母亲 Susila、姐夫 Kishore、姐姐 Ramya 和小侄子 Joshuwa，他们一直支持我，鼓励我更努力地实现目标。我想对 Divya 和本书的技术审校者说声谢谢，因为他们做得很好。幸运的是，我有一个很好的合作者 Srikanth，他的工作也非常出色。

最后，我要特别感谢 Bianaca、Deepak、Vishal、Arun、Vishwapriya 和 Shabala，是他们为我的生活增添了欢乐。

Anto Aravinth

我要感谢 Apress 给了我第二次写作的机会。我也要感谢我的家人，特别是我亲爱的妻子 Sonia Madan 和我 4 个月大的儿子 Reyanesh，感谢他们在这段时间里给予我的支持。

Srikanth Machiraju

前　言

一本书的第 2 版总是不同寻常的。当我写第 1 版时，我有大约两年的 IT 工作经验。读者对第 1 版的评价褒贬不一。我一直想在负面评价上下功夫，让内容更好，让本书物有所值。与此同时，JavaScript 得到了很大的发展。许多突破性的变化被添加到该语言中。Web 中充满了 JavaScript，请想象一个没有 Web 的世界。很困难，对吧？

第 2 版在第 1 版的基础上取得了很大改进，主要讲授 JavaScript 函数式编程的基础知识。其中增加了许多新内容，例如，如何用函数概念构建一个用于创建 Web 应用程序的库，还增加了测试部分。我们已重写本书以匹配最新的 ES8 语法，并使用了许多 async、await 模式和更多的示例!

你将从本书中获得许多知识，同时将在运行示例时获得乐趣。请开始阅读吧。

目　录

第 1 章

函数式编程简介

函数遵循两条原则：第一条原则是要小，第二条是要更小。

——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，或许还进行过一些思考。如果你已了解它的含义，非常好！但是那些不知道的人也不必担心。本章的目的就是：用通俗的语言介绍函数式编程。

本章将以一个简单的问题作为开端：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？它为何重要

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？数学中的函数可写成如下形式：

$$f(X) = Y$$

这条语句可被解读为：“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，但是其中包含几个关键点：

- 函数必须总是接收一个参数。
- 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*)运行，而不是依据外部环境运行。
- 对于一个给定的 *X*，只会输出一个 *Y*。

为什么我们要了解数学中的函数定义，而不是 JavaScript 中的函数定义？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 传授该思想。但是在整本书中，我们将看到数学函数的思想和用法，以便理解函数式编程。

有了数学函数的定义，下面看看 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？

我们可实现如代码清单 1-1 所示的函数。

代码清单 1-1　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。可用参数调用该函数，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，用数学的定义分析它。数学函数定义的一个关键点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。

修复方法非常简单：只需要移动 percentValue，并把它用作函数的参数。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 函数可被称为一个真正的函数。但是我们得到了什么？我们只是在该函数内部消除了对全局变量的访问。若移除一个函数内部对全局变量的访问，会使该函数的测试变得更容易(稍后将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，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，我们能以此创建仅依赖输入就可完成自身逻辑的函数。这可保证函数被多次调用时仍然返回相同的结果。函数不会改变外部环境的任何变量，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，必须理解函数和 JavaScript 方法之间的区别。

简言之，函数是一段可通过其名称被调用的代码。它可传递参数并返回值。

然而，方法是一段必须通过其名称及其关联对象来调用的代码。

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。

代码清单 1-3　一个简单函数

```
var simple = (a) => {return a}       // 一个简单函数
simple(5)                            // 用其名称调用
```

代码清单 1-4　一个简单方法

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

函数式编程的定义没有提及两个重要的特性。在深入研究函数式编程的好处之前，第 1.2 节和第 1.3 节将详细阐述这两个重要的特性。

1.2 引用透明性

根据函数的定义，可得出结论：所有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 }
```

上面的代码片段定义了一个简单的函数 identity。你以什么作为输入，该函数就会返回什么。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，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，在代码清单 1-2 中，我们从全局访问中移除了 percentValue，并将它用作传入的参数)。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，可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)，因为我们可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。想象一下，凭借该模型，可轻松地用多线程运行上面的代码，甚至不需要同步！为什么？同步的原因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依赖来自参数的输入。因此，线程可自由地运行，没有任何锁机制！

由于函数会为给定的输入返回相同的值，我们实际上可缓存它了！例如，假设用一个名为 factorial 的函数计算给定数值的阶乘。factorial

接收输入作为参数以计算其阶乘。我们都知道 5 的阶乘是 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，我们就能缓存 factorial 函数的值。因此，当 factorial(以 5 作为输入)被第二次调用时，就能返回已缓存的值，而不必再计算一次。

由此可见，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。稍后将带你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，如果用另一个表示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。

这就是本节定义的引用透明性。如果替换函数的值，并不会影响上下文。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。在进一步探讨之前，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。相关代码如代码清单 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？但是看起来我们似乎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。”我们称之为命令式解决方案。命令式编程主张告诉编译器“如何”做。

现在来看另一方面——声明式编程。在声明式编程中，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，后续章节将予以介绍)。现在，可用内置的 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
```

代码清单 1-7 打印了与代码清单 1-6 相同的输出。但是此处移除了“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”。这里使用了一个处理“如何”做的抽象函数，如此，开发者只需要关心手头的问题(做“什么”的部分)。在整本书中，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，这些函数可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。在此基础上，我们可以去研究函数式编程的好处了。

1.4　函数式编程的好处

现在，我们已经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问题：函数式编程的好处是什么？本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编写的纯函数。在此之前，我们将了解什么是纯函数。

1.5　纯函数

有了前面的定义，我们就能明确纯函数的含义。纯函数是对给定的输入返回相同输出的函数。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，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 5 作为输入调用 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　纯函数生成可测试的代码

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

```
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，主要是因为它依赖外部环境计算其逻辑。尽管该函数可运行，但非常难以测试！下面看看原因。

假设我们打算对 calculateTax 函数运行测试，分别执行 3 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。假设在你运行相同的测试用例时，外部环境正在改变变量 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。但这个问题很容易解决：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：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。通过改变任意一个外部变量，我们就能马上理解其中的含义。例如，思考一下代码清单 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，那么调用 badFunction 的动作会影响其他函数的行为。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，能非常轻易地实现该目标。为明确这一点，下面将使用一个简单的 double 函数(来自代码清单 1-8)：

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

通过函数的名称，能轻易地推理出：此函数使给定的数值翻倍，其他什么也没做！事实上，根据引用透明性概念，可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。如果代码库中包含具有副作用的函数，团队中的其他开发者将难以阅读此代码。包含纯函数的代码库更易于阅读、理解和测试。记住，函数(无论它是否为纯函数)必须始终具有一个有意义的名称。按照这种说法，在给定行为后不能将函数 double 命名为 dd。

脑力小游戏

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！这在理解函数思想的过程中是一个巨大的进步。我们取代函数值，并假定这是它要返回的结果！

为了快速练习你的脑力，下面用内置的 Math.max 函数测试你的推理能力。

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 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，你已经表现得很出色了！

1.6 并发代码

纯函数总是允许并发地执行代码。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来并发地执行函数，但是，如果项目使用了 WebWorker 并发地执行多任务，该怎么办呢？或者，如果 Node 环境中有一段服务端代码需要并发地执行函数，又该怎么办呢？

例如，假设代码清单 1-10 给出如下代码：

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，如果并发地执行这些函数，将会引起不良影响。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
    }
}
```

此处移动了 global 变量，将它用作两个函数的参数，使它们变成纯函数。现在可以并发地执行这两个函数了，不必担心任何问题。因为函数不依赖外部环境(global 变量)，所以我们不必像执行代码清单 1-10 时那样担心线程的执行顺序。

本节说明了纯函数如何使代码并发执行，而不会带来任何问题。

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，那么我们能缓存函数的输出。为了讲得更具体些，这里列举一个简单例子。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，那么可推知，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的做法不会改变上下文。)假设有一个记账对象，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，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！

本书随后将带你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，我们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的函数。UNIX 的理念是，只做一件事并把它做到完美，在实现纯函数时也要遵循这一原则。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，等等。这些命令的确一次只解决一个问题。但我们可用组合或管道来完成复杂的任务。假如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)。假如同样的命令行在

JavaScript 函数中已经实现了，我们就能根据同样的原则使用它们解决问题！

现在考虑用一种不同的方式解决另一个问题。假设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。它接收参数并向调用者返回输出，且不改变任何外部环境！

遵循一个简单的定义，我们就能收获很多好处。本章结束之前将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

在第 1.7 节中，我们见过如下一段代码(见代码清单 1-12)：

```
var longRunningFunction = (ip) => { //do long running tasks and
return }
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 的输入范围被限制为 1~11 的整数(正如例子所示)。因为我们已经为这个特别的范围构建了记账对象，所以只能参照 longRunningFnBookKeeper 为给定的输入返回输出。

下面分析该记账对象。该对象为我们清晰地描绘出，函数 longRunningFunction 接收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)。此外，key 中也不存在映射两个输出的输入。

经过上面的分析，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。函数的输入被称为参数，输出被称为值。对于一个给定的函数，所有被允许的输入集合被称为该函数的定义域，而被允许的输出集合被称为值域。

上面的定义与纯函数的定义完全一致！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，我们很容易发现数学函数的思想已被借鉴到函数式范式的世界(正如本章开始时阐述的那样)。

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 ES8-Functional 的函数式库。本书的各个章节将带你一步步地构建此库。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？可以说是，也可以说不是。本章的开头曾指出，函数式编程主张函数必须接收至少一个参数并返回一个值。不过坦率地讲，我们可在 JavaScript 创建一个不接收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应用于纯 JavaScript！这就是本书书名的由来！

JavaScript 语言支持将函数用作参数，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(后续章节将进一步讨论)。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

本章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，研究了简短而透彻的函数例子和 JavaScript 中的函数式编程。本章还介绍了什么是纯函数并详细讨论了它们的好处。本章结尾说明了纯函数和数学函数之间的关系，还讨论了 JavaScript 为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。

下一章将介绍如何使用 ES8 创建并执行函数。使用 ES8 创建函数的方式有很多种，我们将在下一章学习这些方式！