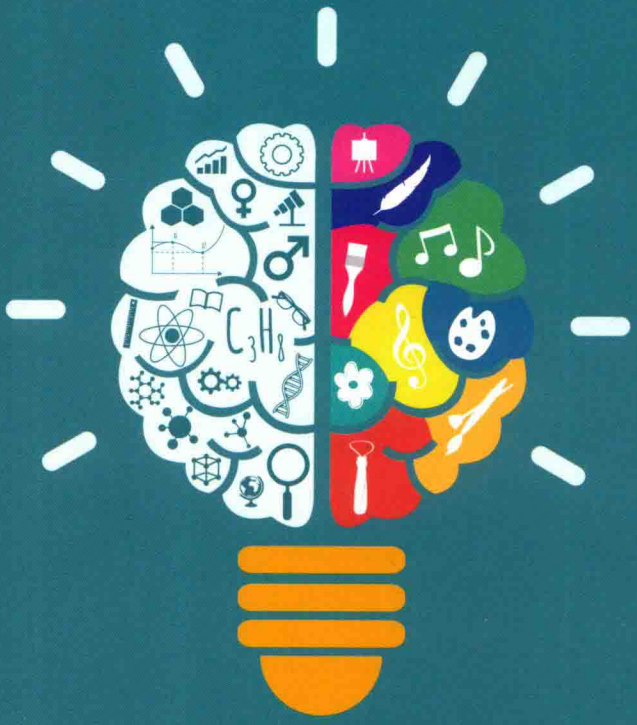
异步图书
www.epubit.com.cn



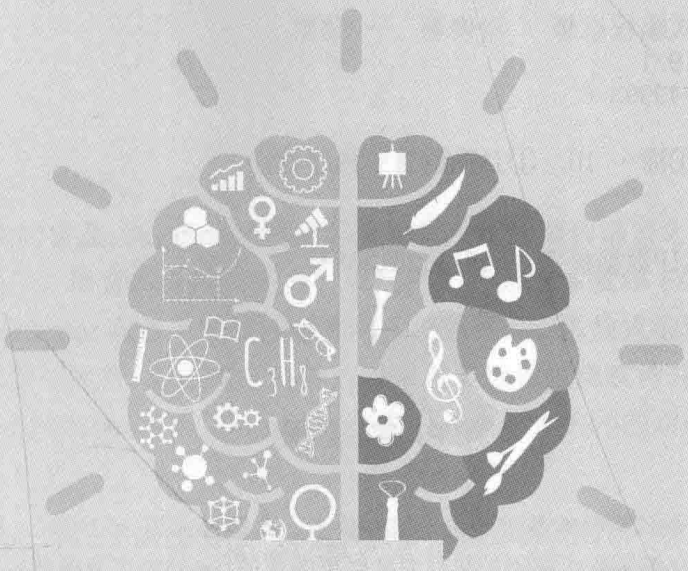
JavaScript 函数式编程思想

潘俊 著

涵盖函数式编程的理论基础、核心技术、典型特征。
介绍函数式编程中的递归模式和列表处理。
系统比较面向对象编程与函数式编程。

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

非外借



JavaScript 函数式编程思想

潘俊 著

人民邮电出版社
北京

图书在版编目（CIP）数据

JavaScript函数式编程思想 / 潘俊著. -- 北京 : 人民邮电出版社, 2019.1
ISBN 978-7-115-49993-6

Ⅰ. ①J… Ⅱ. ①潘… Ⅲ. ①JAVA语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2018)第251368号

内 容 提 要

本书主要介绍了函数式编程的基础理论、核心技术、典型特征和应用领域，以及它与面向对象编程的比较。本书既广泛介绍函数式编程的思想，也结合 JavaScript 的特点分析其应用和局限，注重从本质和内在逻辑的角度解释各个主题，并辅以相关的代码演示。对于函数式编程涉及的 JavaScript 语言本身的特性，以及与面向对象编程的比较，在书中也给予了重点讨论。

本书适合希望学习函数式编程的 JavaScript 程序员阅读，对一般的函数式编程理念感兴趣的读者也可以将本书作为参考。

◆ 著　　潘　俊
责任编辑　张　爽
责任印制　焦志炜

◆ 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
三河市祥达印刷包装有限公司印刷

◆ 开本：800×1000　1/16
印张：17.5
字数：371 千字　　2019 年 1 月第 1 版
印数：1－2 400 册　　2019 年 1 月河北第 1 次印刷

定价：59.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

前言

伴随着 Web 技术的普及，JavaScript 已成为应用最广泛的编程语言之一。由于其在 Web 前端编程中的统治地位、语言本身的表现力、灵活性、开源的本质和 ECMAScript 标准近年来的快速发展，JavaScript 向各个领域渗透的势头仍然强劲。函数式编程的思想和语言原来仅仅在计算机学术圈中流行，近年来它的魅力越来越多地被主流软件开发行业所认识到。Scala、Closure 等语言的出现，C#、Java 等语言中引入函数式编程的功能都是这一趋势的体现。

传统的 JavaScript 开发主要使用命令式和面向对象的编程范式，并零星地结合了一些函数式编程的技巧。通过系统地介绍函数式编程的思想和技术，展现它在提高代码的表现力、可读性和可维护性等方面的益处，本书希望能让更多的 JavaScript 程序员了解并喜欢上这种优美而高效的编程范式。

本书内容共分为 9 章。

第 1、2 章介绍了与 JavaScript 函数式编程所用技术紧密关联的名称和类型系统的理论。

第 3 章简要介绍了函数式编程的理论基础：lambda 演算和 JavaScript 中函数的相关知识。

第 4、5 章介绍了函数式编程的基础和核心技术：一等值的函数、部分应用和复合。

第 6 章介绍了函数式编程的典型特征：没有副作用的纯函数和不可变的数据。

第 7 章介绍了函数式编程中进行重复计算的递归模式。

第 8 章介绍了函数式编程的重要领域：列表处理。

第 9 章系统地比较了面向对象编程和函数式编程。

野人献曝，未免贻笑大方；愚者千虑，或有一得可鉴。书中的不足之处，敬请各位读者批评指正。

潘俊

2018 年 10 月

资源与支持

本书由异步社区出品，社区（https://www.epubit.com/）为您提供相关资源和后续服务。

提交勘误

作者和编辑尽最大努力来确保书中内容的准确性，但难免会存在疏漏。欢迎您将发现的问题反馈给我们，帮助我们提升图书的质量。

当您发现错误时，请登录异步社区，按书名搜索，进入本书页面，点击“提交勘误”，输入勘误信息，单击“提交”按钮即可。本书的作者和编辑会对您提交的勘误进行审核，确认并接受后，您将获赠异步社区的 100 积分。积分可用于在异步社区兑换优惠券、样书或奖品。

扫码关注本书

扫描下方二维码，您将会在异步社区微信服务号中看到本书信息及相关的服务提示。

与我们联系

我们的联系邮箱是 contact@epubit.com.cn。

如果您对本书有任何疑问或建议，请您发邮件给我们，并请在邮件标题中注明本书书名，以便我们更高效地做出反馈。

如果您有兴趣出版图书、录制教学视频，或者参与图书翻译、技术审校等工作，可以发邮件给我们；有意出版图书的作者也可以到异步社区在线提交投稿（直接访问 www.epubit.com/selfpublish/submission 即可）。

如果您是学校、培训机构或企业，想批量购买本书或异步社区出版的其他图书，也可以发邮件给我们。

如果您在网上发现有针对异步社区出品图书的各种形式的盗版行为，包括对图书全部或部分内容的非授权传播，请您将怀疑有侵权行为的链接发邮件给我们。您的这一举动是对作者权益的保护，也是我们持续为您提供有价值的内容的动力之源。

关于异步社区和异步图书

“异步社区”是人民邮电出版社旗下 IT 专业图书社区，致力于出版精品 IT 技术图书和相关学习产品，为作译者提供优质出版服务。异步社区创办于 2015 年 8 月，提供大量精品 IT 技术图书和电子书，以及高品质技术文章和视频课程。更多详情请访问异步社区官网 https://www.epubit.com。

“异步图书”是由异步社区编辑团队策划出版的精品 IT 专业图书的品牌，依托于人民邮电出版社近 30 年的计算机图书出版积累和专业编辑团队，相关图书在封面上印有异步图书的 LOGO。异步图书的出版领域包括软件开发、大数据、AI、测试、前端、网络技术等。

异步社区

微信服务号

目录

第1章

名称

一般对函数式编程的介绍都会从一等值和纯函数等概念开始，本书却准备在那之前先用一些篇幅讨论两个通常未得到足够重视的主题：名称和类型系统。前者包括名称绑定、作用域和闭包等内容；后者包括类型的含义和划分、强类型和弱类型、静态类型和动态类型，以及多态性的内容。理解这些概念对编程很有意义，无论是使用哪种语言，采用什么范式。具体到本书的核心，使用 JavaScript 进行函数式编程，在理解以上普适概念的基础上，掌握它们在 JavaScript 中的特定表现和行为，又具有格外的重要性。这一方面是因为 JavaScript 长期以来被认为是一种简单的脚本语言，缺少在通用知识背景下对其特性和行为的分析，以致对其行为的认识往往是零碎但实用的。另一方面是因为名称和类型系统与 JavaScript 的函数式编程有着紧密的关联。嵌套函数和闭包是 JavaScript 的函数式编程离不开的技术，鸭子类型是 JavaScript 借以实现函数式编程通常具备的参数多态性特征的机制。这些内容都将在下面两章中得到充分的讨论。

1.1 名称绑定

编程语言中有许多实体：常量、变量、函数、对象、类型、模块。从计算机的角度来看，所有这些都是用它们在存储器中的地址来代表的。要人们记住这些地址，并用它们来思考，当然是不可能的。就像在生活中和处理其他领域的问题一样，人们给编程语言中的实体以名称。所谓名称绑定（Name binding），是指将名称和它所要代表的实体联系在一起。编程语言中的名称通常又称为标识符（Identifier），它是字符序列，在许多语言中，能使用的字符种类会受到限制，例如 JavaScript 中的标识符只能由字母、数字、$和_组成，并且不能以数字开头。广义来说，编程语言中所有可用的名称都经过了绑定，包括在语言设计阶段绑定的关键字（如 `if`、`while`）和操作符（如+、.），我们这里关心的仅仅是程序员在代码中使用的标识符和它们所代表的实体之间的绑定。

名称绑定有 3 个要素：名称、实体和绑定。创建名称绑定因而也就包含 3 个动作：创建名称、创建实体和绑定。创建名称通过声明（Declaration）完成：声明变量、声明函数、

声明类型等。实体的创建方式随其类型而变化，数字、字符等原始数据类型的值只需写出其字面值（Literal），更复杂的数据类型值根据所用语言的语法创建。绑定则通过给名称赋值完成。在有些场景中，创建实体和绑定会在创建名称后自动完成。例如在许多静态类型的语言里，声明的数字变量若不赋值，会初始化为 0。在另一些场景中，创建名称、创建实体和绑定这 3 个动作是一并完成的。Java 中声明类型和 JavaScript 中声明函数都属于这种情况。

名称和实体是相互独立的存在，它们之间的绑定也不是一一对应的。名称可以不绑定任何实体，如 JavaScript 中已声明但未赋值的变量；也可以同时绑定多个实体，如操作符和方法重载（JavaScript 是弱类型的，所以没有方法重载的概念，目前也不支持自定义操作符重载，不过也用到了重载，+操作符就是如此）。反过来，实体可以不绑定任何名称，如表达式中未被赋值的对象；也可以同时绑定多个名称（这种现象称为别名 Aliases），如通过赋值多个变量都指向同一对象。

在程序运行过程中，名称和实体都有自己的生存期（Lifetime），从创建开始，到销毁结束，两者并不一定重合。实体可以延续比某个名称更长的时间。例如调用函数时传入某个对象参数，在该函数内参数名和对象之间构成绑定，到函数返回时名称失效，但在调用该函数的代码里对象仍然生存。在特殊情况下，名称的生存期也可能比其中的实体更长。例如在 C++中可以手工销毁对象，在以引用方式传递对象参数时，若在被调用者内销毁该对象，调用者内关于该对象的名称还依然存在，此时的名称就成为悬空指针（Dangling reference）。

常量和变量

通常名称所绑定的实体是可以被更换的，这样的名称被称为变量，其绑定的实体称为变量值，通过赋值来更换变量值。程序中经常会用到一些固定的值，如引力常数、颜色代码、给用户提示的字符串，将它们绑定到专门的名称有诸多好处：有意义的名称可以充当所绑定值的注释；多次使用某个值时，用名称不容易输入错误，即使发生错误，编程语言的编译或运行环境也能发现和报告；一旦需要更换，用名称时只需修改一处，否则需要找出代码中所有使用该值的地方并修改。对于绑定这些值的名称，在程序运行过程中更换它们的值是不需要也不应该的。所以许多编程语言引入了称为常量的名称，它们的值只能在声明时绑定，之后不允许更换。

需要注意的是，常量的值虽然不能被更换，但并不能保证它不会发生改变。假如常量值是简单数据类型的，如布尔值、数字，那么确实不可能改变；假如常量值是复合类型的，如 JavaScript 中的对象，那么虽然不能通过赋值更换为其他对象，但是可以修改它的属性

值。只有字符串等不可变的复合数据类型，才能够确保常量值不发生改变。关于简单和复合数据类型，将在第 2 章中介绍。关于数据的不变性，将在第 6 章中讨论。在 ECMAScript 2015 之前，JavaScript 只能通过 `var` 语句声明变量。ECMAScript 2015 新增了 const 语句以声明常量。

变量依据其与所绑定的值的关系，可以分为值模型（Value model）和引用模型（Reference model）。采用值模型的变量，可以看作值的容器，赋给变量的值就保存在容器中。变量值被更改，就在变量读写的位置就地完成。采用引用模型的变量，则是指向它所绑定的值的指针或者引用。变量值被更改，既有可能是变量被赋予新的指针，也有可能是指针不变，而它指向的数据发生变化。

变量采取的模型对其被赋值时的行为有很大影响。一个变量的值被赋予另一个变量，采取值类型时，值会被复制，副本保存在被赋值的变量中，两者彻底无关，不会相互影响；采用引用类型时，只是指针被复制，赋予第二个名称，数据仍只有一份，若是一方修改了指针指向的数据，另一方也能看到同样的变化。前者更安全，后者对于体量巨大的数据则节省了复制的时间和空间。

有些编程语言（如 Java、C#）针对不同的数据类型采取不同的模型：布尔值、数字等简单类型，占用的空间很小，采用值模型；动态数组和映射等复合类型，占用的空间可能很大，采用引用模型。在这些语言中，又可以把采用值模型的数据类型称为值类型（Value type），把采用引用模型的数据类型称为引用类型（Reference type）。有些编程语言则采取统一的变量模型，C 语言中所有的变量都是值模型的，不过可以通过指针来实现引用类型的行为。而动态类型的语言因为变量可以被赋予任何类型的值，大多采用单一的引用模型，如 Lisp、Smalltalk、JavaScript。

注意数据的值类型、引用类型和传递参数时的按值和按引用方式是不相干的概念，将在 3.3.3 节中阐释。

1.2 作用域

一个程序，可以从静态和动态两个角度来观察。前者是用空间的[①]维度，分析程序的代码；后者是以时间的维度，研究程序的运行。上一节提到名称的生存期，就是采用时间的维度考察名称有效的问题。程序运行的每一时刻都对应着代码中的某一语句，因而名称在

① 英文中常用 Lexical 一词，在编程语言的语境中通常译为“词法的”，指的是从字符、单词的角度来考察代码的文本，用在中文中仍显生硬。本书中采用“文本的”或“空间的”说法，与“时间的”一词分别对应“静态的”和“动态的”。

时间上的生存，也就对应着它在代码中空间上的有效。我们把名称在代码中有效的区域称为其作用域（Scope，该词在日常英语中的含义为范围，在这里也可用作动词，表示确定作用域的行为）。

一个自然的疑问就是，名称为什么会有有效性的问题？名称一旦声明，为什么不是在整个代码中都有效？这确实是一个选项，有些编程语言（如早期的BASIC）中的名称就是如此。但是可以想象，这种方案很快会给名称的创建和使用带来很大困扰。在整个程序中，任何名称的含义都是唯一的，在某处使用了num、i，在另一处就要使用number、j，在下一处就要使用number2、k。短小的名称很快就会用尽，代码里充斥的名称就会变得冗长难记。这不仅带来体力上的输入麻烦，更严重的是它要求程序员在脑力上维持对整个程序用到的所有名称的关注，从而大大限制了一个程序所能达到的规模。此外，在代码的不同区域重新使用名称又是完全可行的。不同函数中绑定的变量，即使名称相同，含义也是不同的。所以对以上方案初步的改进是，将名称分为全局的（Global）和函数局部的两类。前者在整个程序中都有效，后者仅仅在声明它们的函数内有效。这就是ECMAScript 2015标准之前，JavaScript中名称面临的状况：在某个函数内用var和function关键字声明的变量和函数的作用域是包围它们的函数，不在任何函数内声明的变量和函数的作用域是全局。

JavaScript在语法的外观上沿袭了很多C语言家族的惯例，最明显的就是用大括号包围的代码包块（Block）。if、while、for等控制流的结构，在形式上都与函数声明一样，有自己的包块。按照C语言家族的惯例，包块也是作用域的级别。也就是说，在包块中创建的名称只在该包块中有效，不同包块中可以使用相互独立的同样的名称，例如先后有两个for语句都声明i作循环变量而不会彼此干扰。ECMAScript 2015引入的分别用于声明变量、常量和类的let、const和class关键字，使得JavaScript也拥有了包块作用域。包块作用域背后的理念是将名称的有效范围局限在比函数更小的区域内，使得程序员在思考任何一段代码时，当前需要关注的名称所组成的集合尽可能的小。为了不影响历史代码，用var声明的变量作用域保持不变，而用function关键字声明的函数，作用域则发生变化。在ECMAScript 2015之前，在严格模式下，不允许在包块内声明函数；在非严格模式下，标准未规定包括内声明函数的意义，实际行为取决于具体的JavaScript引擎。在ECMAScript 2015之后，在严格模式下，包块内声明的函数的作用域是该包块；在非严格模式下，包块内声明的函数的作用域仍然是该包块所在的函数。ECMAScript 2015还为JavaScript增加了模块的语法，一个模块中声明的实体只在该模块中可见，需要通过导出导入语句才能为其他模块使用，因此在函数与全局之间有了模块层级的作用域。

1.2.1 包块作用域与就近声明

有些编程风格提倡将一个函数内所有的变量声明都集中于函数顶部，显然这种做法享受不到包块作用域的好处。即使在没有包块作用域的语言中，就近声明变量也是更好的习惯，它使得变量的使用更靠近声明，从而令代码更易于理解。而且在顶部集中声明变量，要求程序员在编写函数的一开始就在脑海中列出所有用到的变量的清单，实行起来的难度也更高。

采用就近声明的风格时，用 `let` 取代 `var` 来定义变量，有可能会遇到一些小问题，见如下代码。

```
var condition = true;
if (condition) {
    var foo = 1;
    // ...
} else {
    // ...
}
foo = 2;
```

进入 `if` 语句后，发觉要声明一个变量 `foo`，在 `if` 语句之后还会用到这个变量。这一段代码在语法上没有错误，现在改用 `let` 来声明变量。

```
if (condition) {
    let bar = 1;
    // ...
} else {
    // ...
}
bar = 2;
```

`if` 语句之后使用的 `bar` 是未声明的变量，因为它已经超出了 `if` 语句中所声明 `bar` 的包块作用域，要令它可以使用，须将 `bar` 的声明上移到 `if` 语句之前。

```
let bar;
if (condition) {
    bar = 1;
    // ...
} else {
    // ...
```

```
}
bar = 2;
```

对于只有一个层次的 if 语句，预想到其中可能用到的变量提前声明，或许不是什么难事。但当编写嵌套的多个层次的 if 语句时，就近声明似乎显得更方便。比较下面两段代码。

```
var condition1, condition2, condition3;
if (condition1) {
    var foo1 = 1;
    // ...
    if (condition2) {
        var foo2 = 2;
        // ...
        if (condition3) {
            var foo3 = 3;
            // ...
        }
    } else {
        // ...
    }
} else {
    // ...
}
foo1 = 2, foo2 = 3, foo3 = 4;


let bar1, bar2, bar3;
if (condition1) {
    bar1 = 1;
    // ...
    if (condition2) {
        bar2 = 2;
        // ...
        if (condition3) {
            bar3 = 3;
            // ...
        }
    } else {
        // ...
```

```
    }
} else {
    // ...
}
bar1 = 2, bar2 = 3, bar3 = 4;
```

第二段代码的变量声明集中于最外层的 `if` 语句之前，看上去与在函数顶部声明变量的风格接近。如此比较，在有些场合，似乎沿用没有包块作用域的 `var` 变量声明更方便。细究其实不然。在第二段代码中，因为变量的有效范围限定在声明它的包块内，理解其中的任何一部分时，需要关心的变量要么声明在当前包块内，要么在包围它的上一级包块内，依次类推，不在这个上溯链条中的其他包块就可以被快速忽略。而第一段代码，在任何地方声明的变量都在整个函数内有效，要查找某个变量的声明和经历的变化，就要遍历所有的角落。习惯运用包块作用域，对于编写和理解代码都有益处。

1.2.2 静态作用域和动态作用域

前面讨论的确定作用域的方式称为静态作用域（Static scoping），静态指的是在程序运行前通过分析名称在代码中的相对位置，就能确定它的作用域。一般应用的规则为，名称在代码中有效的区域，就是它在其中被声明的包块。这个包块内可能有内嵌的包块，它们也都属于该名称的作用域。包块是一种通用的代码层次，模块、类型、对象、函数、控制结构等都能使用，甚至还可以创建不依附于这些实体的、仅仅用于分隔代码或获得独立作用域的包块。内嵌包块中如果声明了一个在外套包块中已声明的名称，则在该内嵌包块以及它可能包围的更深层次的包块中，该名称使用的是内嵌包块中的声明。这种现象称为隐藏（Hide）或遮盖（Shadow，内嵌包块中的声明遮盖了外套包块中的声明）。

从静态作用域简明的规则，可以反向推导出对于代码中任何地方出现的某个名称，如何确定其含义（即它在何处被声明）的规则：首先在该名称所处的包块内查找其声明，若未找到，则在上一级外套包块中继续，直到最外层包块之外的全局代码区域；假如还未找到，就发生了引用未声明名称的错误。简而言之，名称使用的是最靠近它的包块中的声明。几乎所有编程语言都有一些内置的（Built-in）实体，例如用于输入输出的例程、数学计算的函数以及基本的数据类型。这些实体同样被绑定到对应的名称上，只不过给它们命名的不是使用这些语言的程序员，而是发明这些语言的程序员。我们可以想象全局代码区域和其中内嵌的包块一样，外面还套有一个虚拟的包块，在这个包块中声明了编程语言内置的实体。这样静态作用域的规则就不仅适用于代码中自定义的名称，还扩展到能覆盖编程语言内置的名称。并且在有些语言中，程序员还能重新声明这些内置的名称，将它们在虚拟包块中的声明覆盖。

与静态作用域相对的是动态作用域（Dynamic scoping）。采用这种方式时，名称的作用域取决于程序的控制流。原则上来说，某个名称的作用域依然是它在其中被声明的区域，如包块、函数，不过当这个区域中有函数调用时，作用域将扩展到该函数的代码，对任何进一步的函数调用也是如此。假如被调用的函数中声明了同一个名称，在该声明的区域以及其中进一步调用的函数内，将使用这一较新的声明。对照静态作用域的规则，动态作用域将包块嵌套换成了函数调用，视角从空间的代码结构变成了时间的程序运行。

同样，从动态作用域的定义，可以反向推导出在代码中任何地方出现的某个名称，如何确定其含义（即它在何处被声明）的规则。名称的含义就是程序运行中最近一次对它做出的并且此时尚未失效（超出包块范围或从函数返回）的声明。由于在阅读代码时无法预知函数的调用顺序，所以对于那些不在当前函数内声明的名称，不能确定其含义。这不仅给理解代码带来困难，而且容易产生错误——非局部声明的名称在程序运行时读取或写入的值出乎程序员的预料，而这类错误是很难纠正的。正因为如此，很少有编程语言会采用动态作用域。

JavaScript 使用的是静态作用域，下面的代码既可以验证这一点，也可以展现两种作用域方式的区别。

```
let x = 1;

function f() {
    let x = 3;
    g();
}

function g() {
    //读取变量 x。
    console.log(x);
    //写入变量 x。
    x = 2;
}

f();
//=> 1
console.log(x);
//=> 2
```

变量 x 分别在全局和函数 f 内声明，f 调用函数 g，其中分别读取和写入 x。编写这段逻辑的语言若是采用静态作用域，则函数 g 内读写的 x 指向的都是最靠近它的包块（此