TURING 图灵程序设计丛书

你不知道的JavaScript（上卷）

Scope & Closures
this & Object Prototypes

[美] Kyle Simpson 著
赵望野 梁杰 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北 京

图书在版编目（CIP）数据

你不知道的JavaScript. 上卷 / (美) 辛普森
(Simpson,K.) 著 ; 赵望野, 梁杰译. -- 北京 : 人民邮
电出版社, 2015.4
（图灵程序设计丛书）
ISBN 978-7-115-38573-4

Ⅰ. ①你… Ⅱ. ①辛… ②赵… ③梁… Ⅲ. ①JAVA语
言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2015)第033934号

内 容 提 要

很多人对 JavaScript 这门语言的印象都是简单易学，很容易上手。JavaScript 语言本身有很多复杂的概念，语言的使用者不必深入理解这些概念也可以编写出功能全面的应用。殊不知，这些复杂精妙的概念才是语言的精髓，即使是经验丰富的 JavaScript 开发人员，如果没有认真学习的话也无法真正理解它们。在本书中，我们要直面当前 JavaScript 开发者不求甚解的大趋势，深入理解语言内部的机制。

本书既适合 JavaScript 语言初学者阅读，又适合经验丰富的 JavaScript 开发人员深入学习。

◆ 著 [美] Kyle Simpson
译 赵望野 梁 杰
责任编辑 李松峰
执行编辑 李 静 曹静雯 魏 然
责任印制 杨林杰

◆ 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
北京 印刷

◆ 开本：800×1000 1/16
印张：13
字数：270千字 2015年 4 月第 1 版
印数：1－3 500册 2015年 4 月北京第 1次印刷
著作权合同登记号 图字：01-2014-7511号

定价：49.00元

读者服务热线：(010)51095186转600 印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第 0021 号

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

第二部分　this 和对象原型

前言

在互联网发展的早期，JavaScript 就已经成为了支撑网页内容交互体验的基础技术。那时 JavaScript 的作用可能仅仅是生成一些闪烁的鼠标轨迹或者烦人的弹出窗口，但是经过了大约 20 年的发展，JavaScript 的技术和能力都发生了天翻地覆的变化，现在的 JavaScript 毫无疑问已经成为了世界上使用范围最广的软件平台——互联网——的核心技术。

但是作为一个语言来说，它总是成为大家批评的对象，部分原因是它有很多历史遗留问题，但主要原因是它的设计哲学有问题。就像 Brendan Eich 曾经说过的，JavaScript 甚至连名字都给人一种“蠢弟弟”的感觉，就像是它更成熟的大哥 Java 的不完整版本。不过名字只不过是营销策略上的一个意外，这两个语言有许多本质上的区别。JavaScript 和 Java 的关系，就像 Carnival（嘉年华）和 Car（汽车）的关系一样，八竿子打不着。

JavaScript 借鉴了许多语言的概念和语法，比如 C 风格的过程式编程以及不太明显的 Scheme/List 风格的函数式编程，因此吸引了许多开发者，甚至是那些不会编程的新手。用 JavaScript 来编写“Hello World”是非常简单的，因此这门语言很有吸引力并且很好上手。

虽然 JavaScript 可能是最早出现的语言之一，但是由于其本身的特殊性，相比其他语言，能真正掌握 JavaScript 的人比较少。如果想用 C、C++ 这样的语言编写功能全面的程序，那需要对语言有很深的了解。但是对于 JavaScript 来说，编写功能全面的程序仅仅是冰山一角。

JavaScript 语言本质上有许多复杂的概念，但是却用一种*看起来*比较简单的方式体现出来，比如回调函数，因此 JavaScript 开发者通常只是简单地使用这些特性，并不会关心语言内部的实现原理。

JavaScript 既是一门充满吸引力、简单易用的语言，又是一门具有许多复杂微妙技术的语言，即使是经验丰富的 JavaScript 开发者，如果没有认真学习的话也无法真正理解它们。

这就是 JavaScript 的矛盾之处，也是这门语言的阿喀琉斯之踵[1]。由于 JavaScript 不必理解就可以使用，因此通常来说很难真正理解语言本身，这就是我们面临的挑战。

使命

如果每次遇到 JavaScript 中出乎意料的行为时，你的反应就是把它加入黑名单（很多人都是这么做的），那用不了多久你就会把 JavaScript 语言真正的多样性全部排除。

剩下的部分就是非常著名的“好的部分”（Good Parts），但是亲爱的读者们，我恳请你们把它称作“简单的部分”、“安全的部分”甚至“不完整的部分”。

“你不知道的 JavaScript”系列丛书要做的事恰好相反：学习并且深入理解整个 JavaScript，尤其是那些“难的部分”。

在本书中，我们要直面当前 JavaScript 开发者不求甚解的大趋势，他们往往不会深入理解语言内部的机制，遇到困难就会退缩。我们要做的恰好相反，不是退缩，而是继续前进。

你们应当像我一样，不满足于只是让代码正常工作，而是想要弄清楚“为什么”。我希望你能勇于挑战这条崎岖颠簸的“少有人走的路”，拥抱整个 JavaScript。掌握了这些知识之后，无论什么技术、框架和流行词语你都能轻松理解。

这个系列中的每本书专注于语言中一个最容易被误解或者最难理解的核心部分，进行深入、详尽的介绍。在阅读本书时，你应当审视自己对于 JavaScript 的理解，仔细思考书中讲解的理论和那些“你需要知道”的东西。

现在你所理解的 JavaScript 很可能是从别人那里学来的不完整版。这样的 JavaScript 只是真正的 JavaScript 的影子。学完这个系列之后，你就会掌握真正的 JavaScript。读下去吧，我的朋友，JavaScript 恭候你的光临。

小结

JavaScript 非常特殊，只学一部分的话非常简单，但是想要完整地学习会很难（就算学到够用也不容易）。当开发者感到迷惑时，他们通常会责怪语言本身，而不是怪自己对语言缺乏了解。这个系列就是为了解决这个问题，让你打心眼儿里欣赏这门语言。

本书中的许多例子都需要运行在即将到来的现代 JavaScript 引擎环境中，比如 ES6。部分代码在旧（ES6 之前的）引擎上可能无法正常运行。

注 1：指某人或某事物的最大或者唯一弱点，即罩门关键所在。——译者注

本书排版约定

本书中使用以下排版约定。

- 楷体
  表示新的术语。
- 等宽字体
  表示代码段以及段落中的程序元素，比如变量、函数名、数据库、数据类型、环境变量、语句以及关键字。
- 等宽粗体
  表示命令中不可改动的部分。
- 等宽斜体
  表示将由用户提供的值（或由上下文确定的值）替换的文本。

这个图标表示提示或建议。

这个图标表示重要说明。

这个图标表示警告或提醒。

使用代码示例

可以在这里下载本书第一部分“作用域和闭包”随附的资料（代码示例、练习题等）：http://bit.ly/1c8HEWF。

可以在这里下载本书第二部分“`this` 和对象原型”随附的资料（代码示例、练习题等）：http://bit.ly/ydkjs-this-code

让本书助你一臂之力。也许你需要在自己的程序或文档中用到本书中的代码。除非大段大

段地使用，否则不必与我们联系取得授权。例如，无需请求许可，就可以用本书中的几段代码写成一个程序。但是销售或者发布 O'Reilly 图书中代码的光盘则必须事先获得授权。引用书中的代码来回答问题也无需授权。将大段的示例代码整合到你自己的产品文档中则必须经过许可。

使用我们的代码时，希望你能标明它的出处，但不强求。出处信息一般包括书名、作者、出版商和书号，例如：*Scope and Closures*，Kyle Simpson 著（O'Reilly，2014）。版权所有，978-1-491-33558-8。

如果还有关于使用代码的未尽事宜，可以随时与我们联系：permissions@oreilly.com。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应需而变的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

Safari Books Online 是技术专家、软件开发人员、Web 设计师、商务人士和创意人士开展调研、解决问题、学习和认证培训的第一手资料。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O’Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书第一部分“作用域和闭包”的网址是 http://oreil.ly/JS_scope_closures。本书第二部分“this 和对象原型”的网址是 http://bit.ly/ydk-js-this-object-prototypes。

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

要查看“你不知道的 JavaScript”系列丛书中的全部图书，请访问：

http://YouDontKnowJS.com

第一部分

作用域和闭包

[美] Kyle Simpson 著
赵望野 译

序

小时候，我特别喜欢把东西拆成零部件，然后再重新装回去——旧的移动电话、立体声音响等我能拿到的一切物件都没能幸免。对于年幼的我来说，使用这些东西还为时过早，但是一旦它们坏了，我就立刻想弄清楚它们的工作原理。

记得有一次，我看见一个老式收音机的电路板，其中有一个缠满铜线的奇怪长管。我不知道这个长管的用途，所以立刻开始研究它。它有什么用？为什么出现在收音机里？为什么它看起来和电路板的其他部分不一样？为什么会有铜线缠绕着它？如果我把铜线拆下来会发生什么？现在我知道了，这是一个在晶体管收音机中很常见的、由缠绕着铜线的铁氧体棒制成的环形天线。

你是否也曾对解答各种各样的为什么很上瘾？大多数孩子都会。事实上，这可能是孩子身上我最喜欢的地方——求知欲很强。

很遗憾，现在我从事着一份专业性的工作，并以制作一些东西来度日。而我儿时的梦想是有一天能够制作那些被我拆开过的东西。当然，现在我所制作的大部分东西都是用 JavaScript 做成的，而不是铁氧体棒……但它们很相似！尽管我曾经一度非常热爱制作东西，但是现在却更渴望了解事物的运行原理。我经常寻找解决问题或修复 bug 的最佳方法，却很少花时间来研究我所使用的工具。

这也是为什么我一看到“你不知道的 JavaScript”系列图书就很激动，因为 JavaScript 的确有很多我不了解的地方。我每天从早到晚都在使用 JavaScript，并且已经持续了好几年，但我真的了解它了吗？答案是否定的。当然，我了解它的很多细节，并且经常阅读标准文档和邮件列表中的内容，但是了解的程度低于我内心那个六岁的孩子希望我达到的水平。

第一部分“作用域和闭包”是一个非常好的切入点。它对于如我一般的受众来说非常有用（希望对你也同样有用）。这本书并不会教你如何使用 JavaScript，但是它会让你意识到对于其内部的运行原理你可能了解得并不多。同时这本书出现的时机也非常巧：ES6 终于稳定下来了，并且各家浏览器的实现工作也正在逐步展开。如果你还没有学习其中的新功能（比如 `let` 和 `const`），这本书将起到很好的介绍作用。

所以我希望你能喜欢这本书，尤其希望 Kyle 对 JavaScript 工作原理每一个细节的批判性思考会渗透到你的思考过程和日常工作中。知其然，也要知其所以然。

Shane Hudson
www.shanehudson.net

第 1 章

作用域是什么

几乎所有编程语言最基本的功能之一，就是能够储存变量当中的值，并且能在之后对这个值进行访问或修改。事实上，正是这种储存和访问变量的值的能力将状态带给了程序。

若没有了状态这个概念，程序虽然也能够执行一些简单的任务，但它会受到高度限制，做不到非常有趣。

但是将变量引入程序会引起几个很有意思的问题，也正是我们将要讨论的：这些变量住在哪里？换句话说，它们储存在哪里？最重要的是，程序需要时如何找到它们？

这些问题说明需要一套设计良好的规则来存储变量，并且之后可以方便地找到这些变量。这套规则被称为作用域。

但是，究竟在哪里而且怎样设置这些作用域的规则呢？

1.1 编译原理

尽管通常将 JavaScript 归类为“动态”或“解释执行”语言，但事实上它是一门编译语言。这个事实对你来说可能显而易见，也可能你闻所未闻，取决于你接触过多少编程语言，具有多少经验。但与传统的编译语言不同，它不是提前编译的，编译结果也不能在分布式系统中进行移植。

尽管如此，JavaScript 引擎进行编译的步骤和传统的编译语言非常相似，在某些环节可能比预想的要复杂。

在传统编译语言的流程中，程序中的一段源代码在执行之前会经历三个步骤，统称为“编译”。

- *分词/词法分析*（Tokenizing/Lexing）
  这个过程会将由字符组成的字符串分解成（对编程语言来说）有意义的代码块，这些代码块被称为词法单元（token）。例如，考虑程序 `var a = 2;`。这段程序通常会被分解成为下面这些词法单元：`var`、`a`、`=`、`2` 、`;`。空格是否会被当作词法单元，取决于空格在这门语言中是否具有意义。

分词（tokenizing）和词法分析（Lexing）之间的区别是非常微妙、晦涩的，主要差异在于词法单元的识别是通过有状态还是无状态的方式进行的。简单来说，如果词法单元生成器在判断 `a` 是一个独立的词法单元还是其他词法单元的一部分时，调用的是有状态的解析规则，那么这个过程就被称为*词法分析*。

- *解析/语法分析*（Parsing）
  这个过程是将词法单元流（数组）转换成一个由元素逐级嵌套所组成的代表了程序语法结构的树。这个树被称为“抽象语法树”（Abstract Syntax Tree，AST）。
  `var a = 2;` 的抽象语法树中可能会有一个叫作 `VariableDeclaration` 的顶级节点，接下来是一个叫作 `Identifier`（它的值是 `a`）的子节点，以及一个叫作 `AssignmentExpression` 的子节点。`AssignmentExpression` 节点有一个叫作 `NumericLiteral`（它的值是 `2`）的子节点。

- *代码生成*
  将 AST 转换为可执行代码的过程称被称为代码生成。这个过程与语言、目标平台等息息相关。
  抛开具体细节，简单来说就是有某种方法可以将 `var a = 2;` 的 AST 转化为一组机器指令，用来*创建*一个叫作 `a` 的变量（包括分配内存等），并将一个值储存在 `a` 中。

关于引擎如何管理系统资源超出了我们的讨论范围，因此只需要简单地了解引擎可以根据需要创建并储存变量即可。

比起那些编译过程只有三个步骤的语言的编译器，JavaScript 引擎要复杂得多。例如，在语法分析和代码生成阶段有特定的步骤来对运行性能进行优化，包括对冗余元素进行优化等。

因此在这里只进行宏观、简单的介绍，接下来你就会发现我们介绍的这些看起来有点高深的内容与所要讨论的事情有什么关联。

首先，JavaScript 引擎不会有大量的（像其他语言编译器那么多的）时间用来进行优化，因为与其他语言不同，JavaScript 的编译过程不是发生在构建之前的。

对于 JavaScript 来说，大部分情况下编译发生在代码执行前的几微秒（甚至更短！）的时间内。在我们所要讨论的作用域背后，JavaScript 引擎用尽了各种办法（比如 JIT，可以延迟编译甚至实施重编译）来保证性能最佳。

简单地说，任何 JavaScript 代码片段在执行前都要进行编译（通常就在执行前）。因此，JavaScript 编译器首先会对 `var a = 2;` 这段程序进行编译，然后做好执行它的准备，并且通常马上就会执行它。

1.2 理解作用域

我们学习作用域的方式是将这个过程模拟成几个人物之间的对话。那么，由谁进行这场对话呢？

1.2.1 演员表

首先介绍将要参与到对程序 `var a = 2;` 进行处理的过程中的演员们，这样才能理解接下来将要听到的对话。

- 引擎
  从头到尾负责整个 JavaScript 程序的编译及执行过程。
- 编译器
  引擎的好朋友之一，负责语法分析及代码生成等脏活累活（详见前一节的内容）。
- 作用域
  引擎的另一位好朋友，负责收集并维护由所有声明的标识符（变量）组成的一系列查询，并实施一套非常严格的规则，确定当前执行的代码对这些标识符的访问权限。

为了能够完全理解 JavaScript 的工作原理，你需要开始像引擎（和它的朋友们）一样思考，从它们的角度提出问题，并从它们的角度回答这些问题。

1.2.2 对话

当你看见 `var a = 2;` 这段程序时，很可能认为这是一句声明。但我们的新朋友引擎却不这么看。事实上，引擎认为这里有两个完全不同的声明，一个由编译器在编译时处理，另一

个则由引擎在运行时处理。

下面我们将 `var a = 2;` 分解，看看引擎和它的朋友们是如何协同工作的。

编译器首先会将这段程序分解成词法单元，然后将词法单元解析成一个树结构。但是当编译器开始进行代码生成时，它对这段程序的处理方式会和预期的有所不同。

可以合理地假设编译器所产生的代码能够用下面的伪代码进行概括："为一个变量分配内存，将其命名为 a，然后将值 2 保存进这个变量。"然而，这并不完全正确。

事实上编译器会进行如下处理。

1. 遇到 `var a`，编译器会询问作用域是否已经有一个该名称的变量存在于同一个作用域的集合中。如果是，编译器会忽略该声明，继续进行编译；否则它会要求作用域在当前作用域的集合中声明一个新的变量，并命名为 a。
2. 接下来编译器会为引擎生成运行时所需的代码，这些代码被用来处理 `a = 2` 这个赋值操作。引擎运行时会首先询问作用域，在当前的作用域集合中是否存在一个叫作 a 的变量。如果是，引擎就会使用这个变量；如果否，引擎会继续查找该变量（查看 1.3 节）。

如果引擎最终找到了 a 变量，就会将 2 赋值给它。否则引擎就会举手示意并抛出一个异常！

总结：变量的赋值操作会执行两个动作，首先编译器会在当前作用域中声明一个变量（如果之前没有声明过），然后在运行时引擎会在作用域中查找该变量，如果能够找到就会对它赋值。

1.2.3 编译器有话说

为了进一步理解，我们需要多介绍一点编译器的术语。

编译器在编译过程的第二步中生成了代码，引擎执行它时，会通过查找变量 a 来判断它是否已声明过。查找的过程由作用域进行协助，但是引擎执行怎样的查找，会影响最终的查找结果。

在我们的例子中，引擎会为变量 a 进行 LHS 查询。另外一个查找的类型叫作 RHS。

我打赌你一定能猜到"L"和"R"的含义，它们分别代表左侧和右侧。

什么东西的左侧和右侧？是一个赋值操作的左侧和右侧。

换句话说，当变量出现在赋值操作的左侧时进行 LHS 查询，出现在右侧时进行 RHS 查询。

讲得更准确一点，RHS 查询与简单地查找某个变量的值别无二致，而 LHS 查询则是试图找到变量的容器本身，从而可以对其赋值。从这个角度说，RHS 并不是真正意义上的“赋值操作的右侧”，更准确地说是“非左侧”。

你可以将 RHS 理解成 retrieve his source value（取到它的源值），这意味着“得到某某的值”。

让我们继续深入研究。

考虑以下代码：

```
console.log( a );
```

其中对 a 的引用是一个 RHS 引用，因为这里 a 并没有赋予任何值。相应地，需要查找并取得 a 的值，这样才能将值传递给 console.log(..)。

相比之下，例如：

```
a = 2;
```

这里对 a 的引用则是 LHS 引用，因为实际上我们并不关心当前的值是什么，只是想要为 = 2 这个赋值操作找到一个目标。

LHS 和 RHS 的含义是“赋值操作的左侧或右侧”并不一定意味着就是“=赋值操作符的左侧或右侧”。赋值操作还有其他几种形式，因此在概念上最好将其理解为“赋值操作的目标是谁（LHS）”以及“谁是赋值操作的源头（RHS）”。

考虑下面的程序，其中既有 LHS 也有 RHS 引用：

```
function foo(a)  {
    console.log( a ); // 2
}

foo( 2 );
```

最后一行 foo(..) 函数的调用需要对 foo 进行 RHS 引用，意味着“去找到 foo 的值，并把它给我”。并且 (..) 意味着 foo 的值需要被执行，因此它最好真的是一个函数类型的值！

这里还有一个容易被忽略却非常重要的细节。

代码中隐式的 a = 2 操作可能很容易被你忽略掉。这个操作发生在 2 被当作参数传递给 foo(..) 函数时，2 会被分配给参数 a。为了给参数 a（隐式地）分配值，需要进行一次 LHS 查询。

这里还有对 a 进行的 RHS 引用，并且将得到的值传给了 console.log(..)。console.log(..) 本身也需要一个引用才能执行，因此会对 console 对象进行 RHS 查询，并且检查得到的值中是否有一个叫作 log 的方法。

最后，在概念上可以理解为在 LHS 和 RHS 之间通过对值 2 进行交互来将其传递进 log(..)（通过变量 a 的 RHS 查询）。假设在 log(..) 函数的原生实现中它可以接受参数，在将 2 赋值给其中第一个（也许叫作 arg1）参数之前，这个参数需要进行 LHS 引用查询。

你可能会倾向于将函数声明 function foo(a) {... 概念化为普通的变量声明和赋值，比如 var foo、foo = function(a) {...。如果这样理解的话，这个函数声明将需要进行 LHS 查询。

然而还有一个重要的细微差别，编译器可以在代码生成的同时处理声明和值的定义，比如在引擎执行代码时，并不会有线程专门用来将一个函数值“分配给”foo。因此，将函数声明理解成前面讨论的 LHS 查询和赋值的形式并不合适。

1.2.4 引擎和作用域的对话

```
function foo(a) {
    console.log( a ); // 2
}

foo( 2 );
```

让我们把上面这段代码的处理过程想象成一段对话，这段对话可能是下面这样的。

引擎：我说作用域，我需要为 foo 进行 RHS 引用。你见过它吗？

作用域：别说，我还真见过，编译器那小子刚刚声明了它。它是一个函数，给你。

引擎：哥们太够意思了！好吧，我来执行一下 foo。

引擎：作用域，还有个事儿。我需要为 a 进行 LHS 引用，这个你见过吗？

作用域：这个也见过，编译器最近把它声名为 foo 的一个形式参数了，拿去吧。

引擎：大恩不言谢，你总是这么棒。现在我要把 2 赋值给 a。

引擎：哥们，不好意思又来打扰你。我要为 console 进行 RHS 引用，你见过它吗？

作用域：咱俩谁跟谁啊，再说我就是干这个。这个我也有，console 是个内置对象。给你。

引擎：么么哒。我得看看这里面是不是有 log(..)。太好了，找到了，是一个函数。

引擎：哥们，能帮我再找一下对 a 的 RHS 引用吗？虽然我记得它，但想再确认一次。

作用域：放心吧，这个变量没有变动过，拿走，不谢。

引擎：真棒。我来把 a 的值，也就是 2，传递进 log(..)。

……

1.2.5 小测验

检验一下到目前的理解程度。把自己当作引擎，并同作用域进行一次“对话”：

```
function foo(a) {
    var b = a;
    return a + b;
}

var c = foo( 2 );
```

1. 找到其中所有的 LHS 查询。（这里有 3 处！）
2. 找到其中所有的 RHS 查询。（这里有 4 处！）

查看本章小结中的参考答案。

1.3 作用域嵌套

我们说过，作用域是根据名称查找变量的一套规则。实际情况中，通常需要同时顾及几个作用域。

当一个块或函数嵌套在另一个块或函数中时，就发生了作用域的嵌套。因此，在当前作用域中无法找到某个变量时，引擎就会在外层嵌套的作用域中继续查找，直到找到该变量，或抵达最外层的作用域（也就是全局作用域）为止。

考虑以下代码：

```
function foo(a) {
    console.log( a + b );
}

var b = 2;

foo( 2 ); // 4
```

对 b 进行的 RHS 引用无法在函数 foo 内部完成，但可以在上一级作用域（在这个例子中就是全局作用域）中完成。

因此，回顾一下引擎和作用域之间的对话，会进一步听到：

> 引擎：foo 的作用域兄弟，你见过 b 吗？我需要对它进行 RHS 引用。
> 作用域：听都没听过，走开。

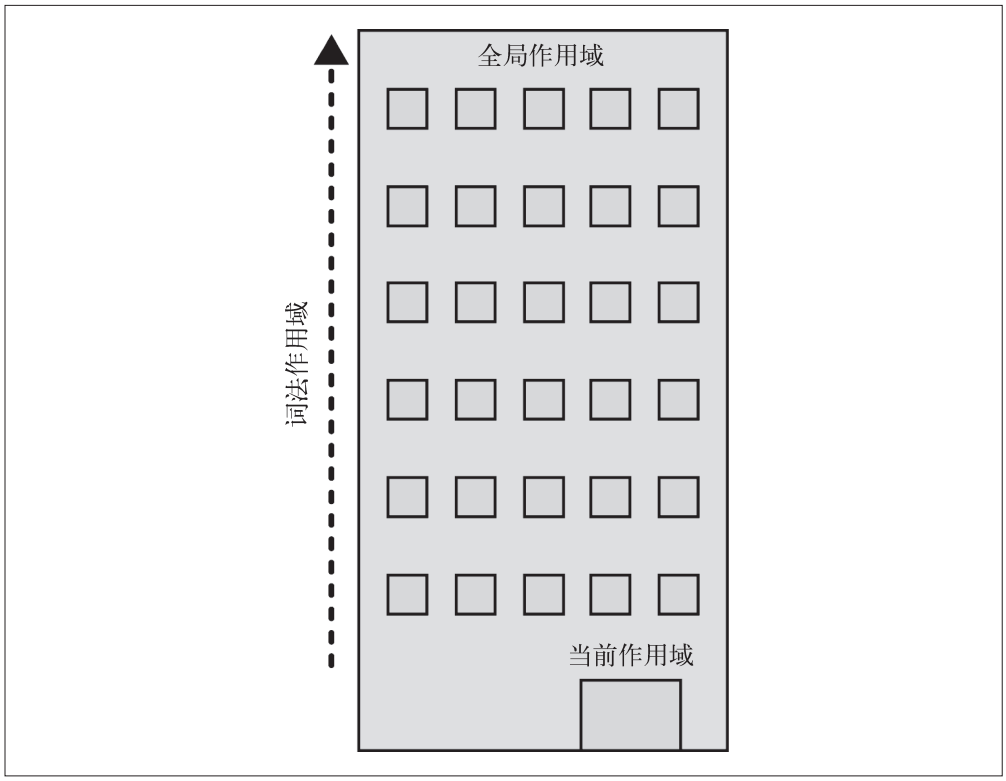
引擎：foo 的上级作用域兄弟，咦？有眼不识泰山，原来你是全局作用域大哥，太好了。你见过 b 吗？我需要对它进行 RHS 引用。

作用域：当然了，给你吧。

遍历嵌套作用域链的规则很简单：引擎从当前的执行作用域开始查找变量，如果找不到，就向上一级继续查找。当抵达最外层的全局作用域时，无论找到还是没找到，查找过程都会停止。

把作用域链比喻成一个建筑

为了将作用域处理的过程可视化，我希望你在脑中想象下面这个高大的建筑：



这个建筑代表程序中的嵌套作用域链。第一层楼代表当前的执行作用域，也就是你所处的位置。建筑的顶层代表全局作用域。

LHS 和 RHS 引用都会在当前楼层进行查找，如果没有找到，就会坐电梯前往上一层楼，如果还是没有找到就继续向上，以此类推。一旦抵达顶层（全局作用域），可能找到了你所需的变量，也可能没找到，但无论如何查找过程都将停止。

1.4 异常

为什么区分 LHS 和 RHS 是一件重要的事情？

因为在变量还没有声明（在任何作用域中都无法找到该变量）的情况下，这两种查询的行为是不一样的。

考虑如下代码：

```
function foo(a) {
    console.log( a + b );
    b = a;
}

foo( 2 );
```

第一次对 b 进行 RHS 查询时是无法找到该变量的。也就是说，这是一个“未声明”的变量，因为在任何相关的作用域中都无法找到它。

如果 RHS 查询在所有嵌套的作用域中遍寻不到所需的变量，引擎就会抛出 ReferenceError 异常。值得注意的是，ReferenceError 是非常重要的异常类型。

相较之下，当引擎执行 LHS 查询时，如果在顶层（全局作用域）中也无法找到目标变量，全局作用域中就会创建一个具有该名称的变量，并将其返还给引擎，前提是程序运行在非“严格模式”下。

“不，这个变量之前并不存在，但是我很热心地帮你创建了一个。”

ES5 中引入了“严格模式”。同正常模式，或者说宽松 / 懒惰模式相比，严格模式在行为上有很多不同。其中一个不同的行为是严格模式禁止自动或隐式地创建全局变量。因此，在严格模式中 LHS 查询失败时，并不会创建并返回一个全局变量，引擎会抛出同 RHS 查询失败时类似的 ReferenceError 异常。

接下来，如果 RHS 查询找到了一个变量，但是你尝试对这个变量的值进行不合理的操作，比如试图对一个非函数类型的值进行函数调用，或着引用 null 或 undefined 类型的值中的属性，那么引擎会抛出另外一种类型的异常，叫作 TypeError。

ReferenceError 同作用域判别失败相关，而 TypeError 则代表作用域判别成功了，但是对结果的操作是非法或不合理的。

1.5 小结

作用域是一套规则，用于确定在何处以及如何查找变量（标识符）。如果查找的目的是对变量进行赋值，那么就会使用 LHS 查询；如果目的是获取变量的值，就会使用 RHS 查询。

赋值操作符会导致 LHS 查询。=操作符或调用函数时传入参数的操作都会导致关联作用域的赋值操作。

JavaScript 引擎首先会在代码执行前对其进行编译，在这个过程中，像 `var a = 2` 这样的声明会被分解成两个独立的步骤：

1. 首先，`var a` 在其作用域中声明新变量。这会在最开始的阶段，也就是代码执行前进行。
2. 接下来，`a = 2` 会查询（LHS 查询）变量 `a` 并对其进行赋值。

LHS 和 RHS 查询都会在当前执行作用域中开始，如果有需要（也就是说它们没有找到所需的标识符），就会向上级作用域继续查找目标标识符，这样每次上升一级作用域（一层楼），最后抵达全局作用域（顶层），无论找到或没找到都将停止。

不成功的 RHS 引用会导致抛出 `ReferenceError` 异常。不成功的 LHS 引用会导致自动隐式地创建一个全局变量（非严格模式下），该变量使用 LHS 引用的目标作为标识符，或者抛出 `ReferenceError` 异常（严格模式下）。

小测验答案

```
function foo(a) {
    var b = a;
    return a + b;
}

var c = foo( 2 );
```

1. 找出所有的 LHS 查询（这里有 3 处！）
   c = ..;、*a* = 2（隐式变量分配）、*b* = ..
2. 找出所有的 RHS 查询（这里有 4 处！）
   foo(2..、= *a*;、*a* ..、.. *b*