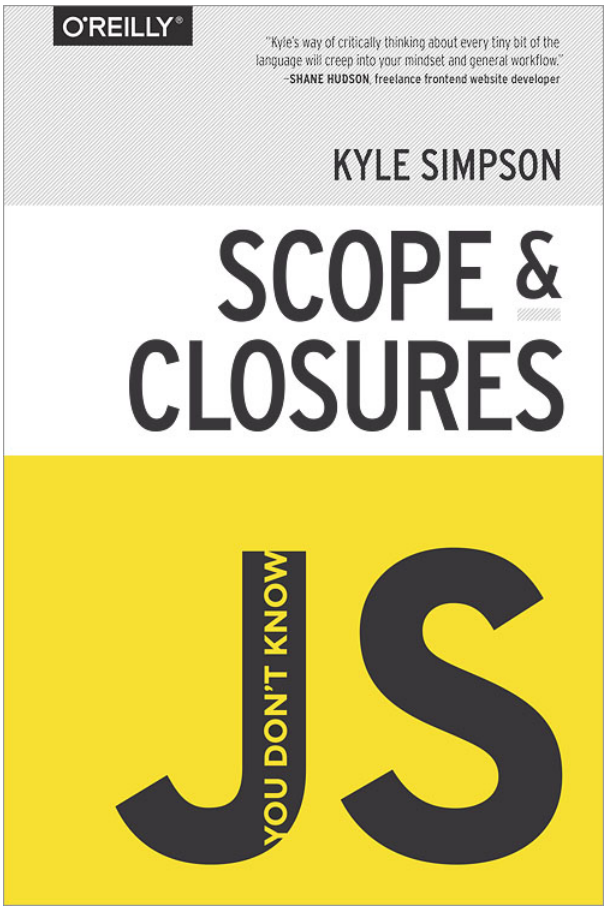



---

**从 O'Reilly 购买数字/印刷版**

---

目录

# 第一章：什么是作用域?

几乎所有语言的最基础模型之一就是在变量中存储值，并且在稍后取出或修改这些值的能力。事实上，在变量中存储值和取出值的能力，给程序赋予了 *状态*。

如果没有这样的概念，一个程序虽然可以执行一些任务，但是它们将会受到极大的限制而且不会非常有趣。

但是在我们的程序中纳入变量，引出了我们现在将要解决的最有趣的问题：这些变量 *存活* 在哪里? 换句话说，它们被存储在哪儿? 而且，最重要的是，我们的程序如何在需要它们的时候找到它们?

回答这些问题需要一组明确定义的规则，它定义如何在某些位置存储变量，以及如何在稍后找到这些变量。我们称这组规则为：*作用域*。

但是，这些 *作用域*规则是在哪里、如何被设置的?

## 编译器理论

根据你与各种编程语言打交道的水平不同，这也许是不证自明的，或者这也许令人吃惊，尽管 JavaScript 一般被划分到 “动态” 或者 “解释型” 语言的范畴，但是其实它是一个编译型语言。它 *不是*像许多传统意义上的编译型语言那样预先被编译好，编译的结果也不能在各种不同的分布式系统间移植。

但是无论如何，JavaScript 引擎在实施许多与传统的语言编译器相同的步骤，虽然是以一种我们不易察觉的更精巧的方式。

在传统的编译型语言处理中，一块儿源代码，你的程序，在它被执行 *之前*通常将会经历三个步骤，大致被称为 “编译”：

1. **分词/词法分析**：将一连串字符打断成（对于语言来说）有意义的片段，称为 token（记号）。举例来说，考虑这段程序：`var a = 2;`。这段程序很可能会被打断成如下 token：`var`，`a`，`=`，`2`，和 `;`。空格也许会被保留为一个 token，这要看它是否是有意义的。

   **注意**：分词和词法分析之间的区别是微妙和学术上的，其中心在于这些 token 是否以 *无状态*或 *有状态*的方式被识别。简而言之，如果分词器去调用有状态的解析规则来弄清 `a` 是否应当被考虑为一个不同的 token，还是只是其他 token 的一部分，那么这就是 **词法分析**。

2. **解析**：将一个 token 的流（数组）转换为一个嵌套元素的树，它综合地表示了程序的语法结构。这棵树称为 “抽象语法树”（AST —— Abstract Syntax Tree）。

   `var a = 2;` 的树也许开始于称为 `VariableDeclaration`（变量声明）顶层节点，带有一个称为 `Identifier`（标识符）的子节点（它的值为 `a`），和另一个称为 `AssignmentExpression`（赋值表达式）的子节点，而这个子节点本身带有一个称为 `NumericLiteral`（数字字面量）的子节点（它的值为 `2`）。

3. **代码生成**：这个处理将抽象语法树转换为可执行的代码。这一部分将根据语言，它的目标平台等因素有很大的不同。

   所以，与其深陷细节，我们不如笼统地说，有一种方法将我们上面描述的 `var a = 2;` 的抽象语法树转换为机器指令，来实际上 *创建*一个称为 `a` 的变量（包括分配内存等等），然后在 `a` 中存入一个值。

   **注意**：引擎如何管理系统资源的细节远比我们要挖掘的东西深刻，所以我们将理所当然地认为引擎有能力按其需要创建和存储变量。

和大多数其他语言的编译器一样，JavaScript 引擎要比这区区三步复杂太多了。例如，在解析和代码生成的处理中，一定会存在优化执行效率的步骤，包括压缩冗余元素，等等。

所以，我在此描绘的只是大框架。但是我想你很快就会明白为什么我们涵盖的这些细节是重要的，虽然是在很高的层次上。

其一，JavaScript 引擎没有（像其他语言的编译器那样）大把的时间去优化，因为 JavaScript 的编译和其他语言不同，不是提前发生在一个构建的步骤中。

对 JavaScript 来说，在许多情况下，编译发生在代码被执行前的仅仅几微秒之内（或更少!）。为了确保最快的性能，JS 引擎将使用所有的招数（比如 JIT，它可以懒编译甚至是热编译，等等），而这远超出了我们关于“作用域”的讨论。

为了简单起见，我们可以说，任何 JavaScript 代码段在它执行之前（通常是 *刚好*在它执行之前!）都必须被编译。所以，JS 编译器将把程序 `var a = 2;` 拿过来，并首先编译它，然后准备运行它，通常是立即的。

## 理解作用域

我们将采用的学习作用域的方法，是将这个处理过程想象为一场对话。但是，*谁*在进行这场对话呢？

### 演员

让我们见一见处理程序 `var a = 2;` 时进行互动的演员吧，这样我们就能理解稍后将要听到的它们的对话：

1. *引擎*：负责从始至终的编译和执行我们的 JavaScript 程序。
2. *编译器*：*引擎*的朋友之一；处理所有的解析和代码生成的重活儿（见前一节）。
3. *作用域*：*引擎*的另一个朋友；收集并维护一张所有被声明的标识符（变量）的列表，并对当前执行中的代码如何访问这些变量强制实施一组严格的规则。

为了 *全面理解* JavaScript 是如何工作的，你需要开始像 *引擎*（和它的朋友们）那样 *思考*，问它们问的问题，并像它们一样回答。

## 反复

当你看到程序 `var a = 2;` 时，你很可能认为它是一个语句。但这不是我们的新朋友 引擎 所看到的。事实上，引擎看到两个不同的语句，一个是 编译器将在编译期间处理的，一个是 引擎将在执行期间处理的。

那么，让我们来分析 引擎和它的朋友们将如何处理程序 `var a = 2;`。

编译器将对这个程序做的第一件事情，是进行词法分析来将它分解为一系列 token，然后这些 token 被解析为一棵树。但是当 编译器到了代码生成阶段时，它会以一种与我们可能想象的不同的方式来对待这段程序。

一个合理的假设是，编译器将产生的代码可以用这种假想代码概括："为一个变量分配内存，将它标记为 a，然后将值 2 贴在这个变量里"。不幸的是，这不是十分准确。

编译器将会这样处理：

1. 遇到 `var a`，编译器让 作用域去查看对于这个特定的作用域集合，变量 a 是否已经存在了。如果是，编译器就忽略这个声明并继续前进。否则，编译器就让 作用域去为这个作用域集合声明一个称为 a 的新变量。

2. 然后 编译器为 引擎生成稍后要执行的代码，来处理赋值 `a = 2`。引擎运行的代码首先让 作用域去查看在当前的作用域集合中是否有一个称为 a 的变量可以访问。如果有，引擎就使用这个变量。如果没有，引擎就查看 其他地方（参见下面的嵌套 作用域一节）。

如果 引擎最终找到一个变量，它就将值 2 赋予它。如果没有，引擎将会举起它的手并喊出一个错误！

总结来说：对于一个变量赋值，发生了两个不同的动作：第一，编译器声明一个变量（如果先前没有在当前作用域中声明过），第二，当执行时，引擎在 作用域中查询这个变量并给它赋值，如果找到的话。

## 编译器术语

为了继续更深入地理解，我们需要一点儿更多的编译器术语。

当 引擎执行 编译器在第二步为它产生的代码时，它必须查询变量 a 来看它是否已经被声明过了，而且这个查询是咨询 作用域的。但是 引擎所实施的查询的类型会影响查询的结果。

在我们这个例子中，引擎将会对变量 a 实施一个 "LHS" 查询。另一种类型的查询称为 "RHS"。

我打赌你能猜出 "L" 和 "R" 是什么意思。这两个术语表示 "Left-hand Side（左手边）" 和 "Right-hand Side（右手边）"

什么的……边？**赋值操作的**。

换言之，当一个变量出现在赋值操作的左手边时，会进行 LHS 查询，当一个变量出现在赋值操作的右手边时，会进行 RHS 查询。

实际上，我们可以表述得更准确一点儿。对于我们的目的来说，一个 RHS 是难以察觉的，因为它简单地查询某个变量的值，而 LHS 查询是试着找到变量容器本身，以便它可以赋值。从

这种意义上说，RHS 的含义实质上不是 *真正的* “一个赋值的右手边”，更准确地说，它只是意味着 “不是左手边”。

在这一番油腔滑调之后，你也可以认为 “RHS” 意味着 “取得他/她的源（值）”，暗示着 RHS 的意思是 “去取……的值”。

让我们挖掘得更深一些。

当我说：

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

这个指向 a 的引用是一个 RHS 引用，因为这里没有东西被赋值给 a。而是我们在查询 a 并取得它的值，这样这个值可以被传递进 console.log(..)。

作为对比：

```
a = 2;
```

这里指向 a 的引用是一个 LHS 引用，因为我们实际上不关心当前的值是什么，我们只是想找到这个变量，将它作为 = 2 赋值操作的目标。

**注意：** LHS 和 RHS 意味着 “赋值的左/右手边” 未必像字面上那样意味着 “ = 赋值操作符的左/右边”。赋值有几种其他的发生形式，所以最好在概念上将它考虑为：“赋值的目标（LHS）” 和 “赋值的源（RHS）”。

考虑这段程序，它既有 LHS 引用又有 RHS 引用：

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

foo( 2 );
```

调用 foo(..) 的最后一行作为一个函数调用要求一个指向 foo 的 RHS 引用，意味着，“去查询 foo 的值，并把它交给我”。另外，(..) 意味着 foo 的值应当被执行，所以它最好实际上是一个函数！

这里有一个微妙但重要的赋值。**你发现了吗？**

你可能错过了这个代码段隐含的 a = 2。它发生在当值 2 作为参数值传递给 foo(..) 函数时，值 2 **被赋值**给了参数 a。为了（隐含地）给参数 a 赋值，进行了一个 LHS 查询。

这里还有一个 a 的值的 RHS 引用，它的结果值被传入 console.log(..)。console.log(..) 需要一个引用来执行。它为 console 对象进行一个 RHS 查询，然后发生一个属性解析来看它是否拥有一个称为 log 的方法。

最后，我们可以将这一过程概念化为，在将值 2（通过变量 a 的 RHS 查询得到的）传入 log(..) 时发生了一次 LHS/RHS 的交换。在 log(..) 的原生实现内部，我们可以假定它拥有参数，其中的第一个（也许被称为 arg1）在 2 被赋值给它之前，进行了一次 LHS 引用查询。

**注意**：你可能会试图将函数声明 `function foo(a) {...` 概念化为一个普通的变量声明和赋值，比如 `var foo` 和 `foo = function(a){...`。这样做会诱使你认为函数声明涉及了一次 LHS 查询。

然而，一个微妙但重要的不同是，在这种情况下 *编译器*在代码生成期间同时处理声明和值的定义，如此当 *引擎*执行代码时，没有必要将一个函数值“赋予” `foo`。因此，将函数声明考虑为一个我们在这里讨论的 LHS 查询赋值是不太合适的。

### 引擎/作用域对话

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

foo( 2 );
```

让我们将上面的（处理这个代码段的）交互想象为一场对话。这场对话将会有点儿像这样进行：

> **引擎**：嘿 *作用域*，我有一个 `foo` 的 RHS 引用。听说过它吗？
>
> **作用域**：啊，是的，听说过。*编译器*刚在一秒钟之前声明了它。它是一个函数。给你。
>
> **引擎**：太棒了，谢谢！好的，我要执行 `foo` 了。
>
> **引擎**：嘿，*作用域*，我得到了一个 `a` 的 LHS 引用，听说过它吗？
>
> **作用域**：啊，是的，听说过。*编译器*刚才将它声明为 `foo` 的一个正式参数了。给你。
>
> **引擎**：一如既往的给力，*作用域*。再次感谢你。现在，该把 2 赋值给 `a` 了。
>
> **引擎**：嘿，*作用域*，很抱歉又一次打扰你。我需要 RHS 查询 `console`。听说过它吗？
>
> **作用域**：没关系，*引擎*，这是我一天到晚的工作。是的，我得到 `console` 了。它是一个内建对象。给你。
>
> **引擎**：完美。查找 `log(..)`。好的，很好，它是一个函数。
>
> **引擎**：嘿，*作用域*。你能帮我查一下 `a` 的 RHS 引用吗？我想我记得它，但只是想再次确认一下。
>
> **作用域**：你是对的，*引擎*。同一个家伙，没变。给你。
>
> **引擎**：酷。传递 `a` 的值，也就是 2，给 `log(..)`。
>
> ...

### 小测验

检查你到目前为止的理解。确保你扮演 *引擎*，并与 *作用域* “对话”：

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

var c = foo( 2 );
```

1. 找到所有的 LHS 查询（有 3 处！）。
2. 找到所有的 RHS 查询（有 4 处！）。

**注意**：小测验答案参见本章的复习部分！

## 嵌套的作用域

我们说过 *作用域*是通过标识符名称查询变量的一组规则。但是，通常会有多于一个的 *作用域*需要考虑。

就像一个代码块儿或函数被嵌套在另一个代码块儿或函数中一样，作用域被嵌套在其他的作用域中。所以，如果在直接作用域中找不到一个变量的话，*引擎*就会咨询下一个外层作用域，如此继续直到找到这个变量或者到达最外层作用域（也就是全局作用域）。

考虑这段代码：

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

var b = 2;

foo( 2 ); // 4
```

`b` 的 RHS 引用不能在函数 `foo` 的内部被解析，但是可以在它的外围 *作用域*（这个例子中是全局作用域）中解析。

所以，重返 *引擎*和 *作用域*的对话，我们会听到：

**引擎**：“嘿，`foo` 的 *作用域*，听说过 `b` 吗？我得到一个它的 RHS 引用。”

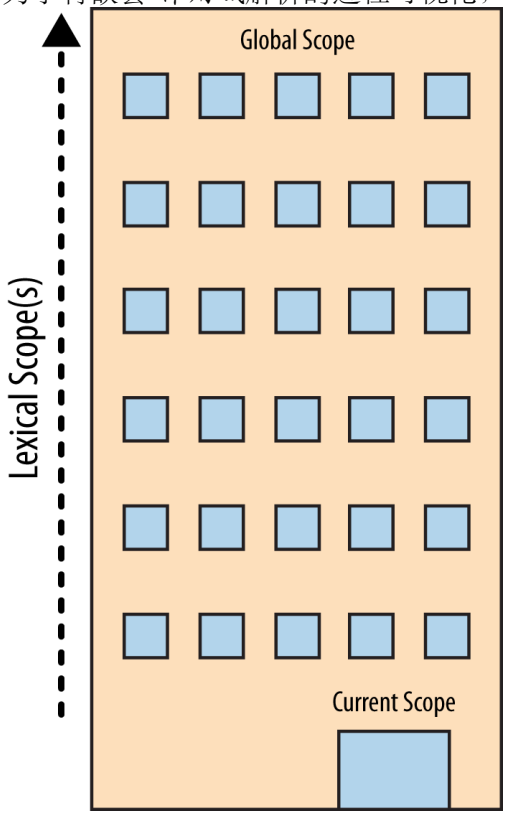
**作用域**：“没有，从没听说过。问问别人吧。”

> **引擎**：“嘿，foo 外面的 作用域，哦，你是全局 作用域，好吧，酷。听说过 b 吗？我得到一个它的 RHS 引用。”
>
> **作用域**：“是的，当然有。给你。”

遍历嵌套 作用域的简单规则：引擎从当前执行的 作用域开始，在那里查找变量，如果没有找到，就向上走一级继续查找，如此类推。如果到了最外层的全局作用域，那么查找就会停止，无论它是否找到了变量。

### 建筑的隐喻

为了将嵌套 作用域解析的过程可视化，我想让你考虑一下这个高层建筑。




这个建筑物表示我们程序的嵌套 作用域规则集合。无论你在哪里，建筑的第一层表示你当前执行的 作用域。建筑的顶层表示全局 作用域。

你通过在你当前的楼层中查找来解析 LHS 和 RHS 引用，如果你没有找到它，就坐电梯到上一层楼，在那里寻找，然后再上一层，如此类推。一旦你到了顶层（全局 作用域），你要么找到了你想要的东西，要么没有。但是不管怎样你都不得不停止了。

### 错误

为什么我们区别 LHS 和 RHS 那么重要？

因为在变量还没有被声明（在所有被查询的 作用域中都没找到）的情况下，这两种类型的查询的行为不同。

考虑如下代码：

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

foo( 2 );
```

当 b 的 RHS 查询第一次发生时，它是找不到的。它被说成是一个“未声明”的变量，因为它在作用域中找不到。

如果 RHS 查询在嵌套的 作用域的任何地方都找不到一个值，这会导致 引擎抛出一个 ReferenceError。必须要注意的是这个错误的类型是 ReferenceError。

相比之下，如果 引擎在进行一个 LHS 查询，但到达了顶层（全局 作用域）都没有找到它，而且如果程序没有运行在“Strict 模式”[1]下，那么这个全局 作用域将会在 **全局作用域中**创建一个同名的新变量，并把它交还给 引擎。

“不，之前没有这样的东西，但是我可以帮忙给你创建一个。”

在 ES5 中被加入的“Strict 模式”[2]，有许多与一般/宽松/懒惰模式不同的行为。其中之一就是不允许自动/隐含的全局变量创建。在这种情况下，将不会有全局 作用域的变量交回给 LHS 查询，并且类似于 RHS 的情况，引擎将抛出一个 ReferenceError。

现在，如果一个 RHS 查询的变量被找到了，但是你试着去做一些这个值不可能做到的事，比如将一个非函数的值作为函数运行，或者引用 null 或者 undefined 值的属性，那么 引擎就会抛出一个不同种类的错误，称为 TypeError。

ReferenceError 是关于 作用域解析失败的，而 TypeError 暗示着 作用域解析成功了，但是试图对这个结果进行了一个非法/不可能的动作。

## 复习

作用域是一组规则，它决定了一个变量（标识符）在哪里和如何被查找。这种查询也许是为了向这个变量赋值，这时变量是一个 LHS（左手边）引用，或者是为取得它的值，这时变量是一个 RHS（右手边）引用。

LHS 引用得自赋值操作。作用域相关的赋值可以通过 = 操作符发生，也可以通过向函数参数传递（赋予）参数值发生。

JavaScript 引擎在执行代码之前首先会编译它，因此，它将 var a = 2; 这样的语句分割为两个分离的步骤：

1. 首先，var a 在当前 作用域中声明。这是在最开始，代码执行之前实施的。

[1] MDN: Strict Mode

[2] MDN: Strict Mode

2. 稍后，`a = 2` 查找这个变量（LHS 引用），并且如果找到就向它赋值。

LHS 和 RHS 引用查询都从当前执行中的 *作用域*开始，如果有需要（也就是，它们在这里没能找到它们要找的东西），它们会在嵌套的 *作用域*中一路向上，一次一个作用域（层）地查找这个标识符，直到它们到达全局作用域（顶层）并停止，既可能找到也可能没找到。

未被满足的 RHS 引用会导致 `ReferenceError` 被抛出。未被满足的 LHS 引用会导致一个自动的，隐含地创建的同名全局变量（如果不是 “Strict 模式”[3]），或者一个 `ReferenceError`（如果是 “Strict 模式”[4]）。

### 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（有 3 处!）。

   **`c = ..`, `a = 2`（隐含的参数赋值）和 `b = ..`**

2. 找出所有的 RHS 查询（有 4 处!）。

   **`foo(2..`, `= a;`, `a + ..` 和 `.. + b`**

## 第二章：词法作用域

在第一章中，我们将 “作用域” 定义为一组规则，它主宰着 *引擎*如何通过标识符名称在当前的 *作用域*，或者在包含它的任意 *嵌套作用域*中来查询一个变量，

作用域的工作方式有两种占统治地位的模型。其中的第一种是最最常见，在绝大多数的编程语言中被使用的。它称为 **词法作用域**，我们将深入检视它。另一种仍然被一些语言（比如 Bash 脚本，Perl 中的一些模式，等等）使用的模型，称为 **动态作用域**。

动态作用域在附录 A 中讲解。我在这里提到它仅仅是为词法作用域提供一个对比，而词法作用域是 JavaScript 所采用的作用域模型。

### 词法分析时

正如我们在第一章中讨论的，标准语言编译器的第一个传统步骤称为词法分析（也就是分词）。如果你回忆一下，词法分析处理是检查一串源代码字符，并给 token 赋予语法含义作为某种有状态解析的输出。

[3] MDN: Strict Mode

[4] MDN: Strict Mode

正是这个概念给理解词法作用域是什么提供了基础，它也是这个名字的渊源。

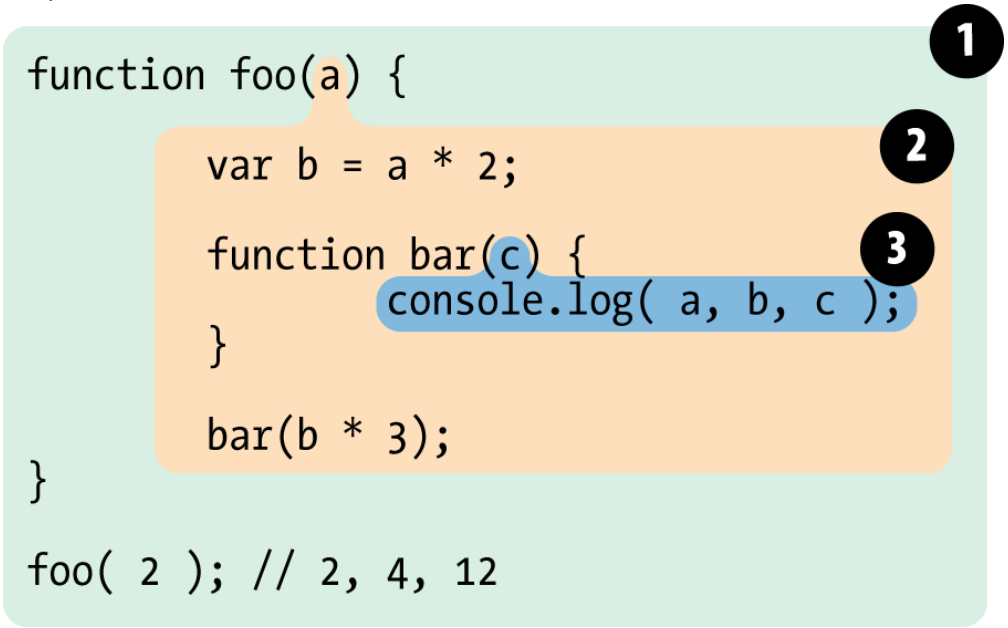
要定义它有点儿兜圈子，词法作用域是在词法分析时被定义的作用域。换句话说，词法作用域是基于，你，在写程序时，变量和作用域的块儿在何处被编写决定的，因此它在词法分析器处理你的代码时（基本上）是固定不变的。

**注意**：我们将会稍稍看到有一些方法可以骗过词法作用域，从而在词法分析器处理过后改变它，但是这些方法都是使人皱眉头的。事实上公认的最佳实践是，将词法作用域看作是仅仅依靠词法的，因此自然而然地完全是编写时决定的。

让我们考虑这段代码：

```
function foo(a) {

    var b = a * 2;

    function bar(c) {
        console.log( a, b, c );
    }

    bar(b * 3);
}

foo( 2 ); // 2 4 12
```

在这个代码实例中有三个固有的嵌套作用域。将这些作用域考虑为套在一起的气泡可能有助于思考。




**气泡 1** 包围着全局作用域，它里面只有一个标识符：`foo`。

**气泡 2** 包围着作用域 foo，它含有三个标识符：a，bar 和 b。

**气泡 3** 包围着作用域 bar，它里面只包含一个标识符：c。

作用域气泡是根据作用域的块儿被写在何处定义的，一个嵌套在另一个内部，等等。在下一章中，我们将讨论作用域的不同单位，但是就现在来说，让我们认为每一个函数创建了一个新的作用域气泡。

bar 的气泡完全被包含在 foo 的气泡中，因为（而且只因为）这就是我们选择定义函数 bar 的位置。

注意这些嵌套的气泡是严格嵌套的。我们没有讨论气泡可以跨越边界的维恩图（Venn diagrams）。换句话说，没有那个函数的气泡可以同时（部分地）存在于另外两个外部的作用域气泡中，就像没有函数可以部分地存在于它的两个父函数中一样。

### 查询

这些作用域气泡的结构和相对位置完全解释了 *引擎*在查找一个标识符时，它需要查看的所有地方。

在上面的代码段中，*引擎*执行语句 console.log(..) 并开始查找三个被引用的变量 a，b 和 c。它首先从最内部的作用域气泡开始，也就是 bar(..) 函数的作用域。在这里它找不到 a，所以它向上走一层，到外面下一个最近的作用域气泡，foo(..) 的作用域。它在这里找到了 a，于是它就使用这个 a。同样的事情也发生在 b 身上。但是对于 c，它在 bar(..) 内部就找到了。

如果在 bar(..) 内部和 foo(..) 内部都有一个 c，那么 console.log(..) 语句将会找到并使用 bar(..) 中的那一个，绝不会到达 foo(..) 中的那一个。

**一旦找到第一个匹配，作用域查询就停止了**。相同的标识符名称可以在嵌套作用域的多个层中被指定，这称为“遮蔽（shadowing）”（内部的标识符“遮蔽”了外部的标识符）。无论如何遮蔽，作用域查询总是从当前被执行的最内侧的作用域开始，向外/向上不断查找，直到第一个匹配才停止。

**注意**：全局变量也自动地是全局对象（在浏览器中是 window，等等）的属性，所以不直接通过全局变量的词法名称，而通过将它作为全局对象的一个属性引用来间接地引用，是可能的。

```
window.a
```

这种技术给出了访问全局变量的方法，没有它全局变量将因为被遮蔽而不可访问。然而，被遮蔽的非全局变量是无法访问的。

不管函数是从 *哪里*被调用的，也不论它是 *如何*被调用的，它的词法作用域是由这个函数被声明的位置 **唯**一定义的。

词法作用域查询 *仅仅*在处理头等标识符时实施，比如 a，b，和 c。如果你在一段代码中拥有一个 foo.bar.baz 的引用，词法作用域查询将在查找 foo 标识符时实施，但一旦定位这个变量，对象属性访问规则将会分别接管 bar 和 baz 属性的解析。

## 欺骗词法作用域

如果词法作用域是由函数被声明的位置唯一定义的，而且这个位置完全是一个编写时的决定，那么怎么可能有办法在运行时“修改”（也就是，作弊欺骗）词法作用域呢？

JavaScript 有两种这样的机制。在广大的社区中它们都等同地被认为是让人皱眉头的，在你代码中使用它们是一种差劲儿的做法。但是关于它们的常见的争论经常错过了最重要的一点：**欺骗词法作用域会导致更低下的性能**。

在我讲解性能的问题以前，先让我们看看这两种机制是如何工作的。

### eval

JavaScript 中的 `eval(..)` 函数接收一个字符串作为参数值，并将这个字符串的内容看作是好像它已经被实际编写在程序的那个位置上。换句话说，你可以用编程的方式在你编写好的代码内部生成代码，而且你可以运行这个生成的代码，就好像它在编写时就已经在那里了一样。

如果以这种观点来评价 `eval(..)`，那么 `eval(..)` 是如何允许你修改词法作用域环境应当是很清楚的：欺骗并假装这个编写时（也就是，词法）代码一直就在那里。

在 `eval(..)` 被执行的后续代码行中，引擎将不会“知道”或“关心”前面的代码是被动态翻译的，而且因此修改了词法作用域环境。引擎将会像它一直做的那样，简单地进行词法作用域查询。

考虑如下代码：

```
function foo(str, a) {
    eval( str ); // 作弊!
    console.log( a, b );
}

var b = 2;

foo( "var b = 3;", 1 ); // 1 3
```

在 `eval(..)` 调用的位置上，字符串 `"var b = 3"` 被看作是一直就存在在那里的代码。因为这个代码恰巧声明了一个新的变量 `b`，它就修改了现存的 `foo(..)` 的词法作用域。事实上，就像上面提到的那样，这个代码实际上在 `foo(..)` 内部创建了变量 `b`，它遮蔽了声明在外部（全局）作用域中的 `b`。

当 `console.log(..)` 调用发生时，它会在 `foo(..)` 的作用域中找到 `a` 和 `b`，而且绝不会找到外部的 `b`。这样，我们就打印出“1 3”而不是一般情况下的“1 2”。

**注意**：在这个例子中，为了简单起见，我们传入的“代码”字符串是固定的文字。但是它可以通过根据你的程序逻辑将字符拼接在一起，很容易地以编程方式创建。`eval(..)` 通常被用于执行动态创建的代码，因为动态地对一段实质上源自字符串字面值的静态代码进行求值，并不会比直接编写这样的代码带来更多真正的好处。

默认情况下，如果 eval(..) 执行的代码字符串包含一个或多个声明（变量或函数）的话，这个动作就会修改这个 eval(..) 所在的词法作用域。技术上讲，eval(..) 可以通过种种技巧（超出了我们这里的讨论范围）被“间接”调用，而使它在全局作用域的上下文中执行，以此修改全局作用域。但不论那种情况，eval(..) 都可以在运行时修改一个编写时的词法作用域。

**注意**：当 eval(..) 被用于一个操作它自己的词法作用域的 strict 模式程序时，在 eval(..) 内部做出的声明不会实际上修改包围它的作用域。

```
function foo(str) {
   "use strict";
   eval( str );
   console.log( a ); // ReferenceError: a is not defined
}

foo( "var a = 2" );
```

在 JavaScript 中还有其他的工具拥有与 eval(..) 非常类似的效果。setTimeout(..) 和 setInterval(..) 可以为它们各自的第一个参数值接收一个字符串，其内容将会被 eval 为一个动态生成的函数的代码。这种老旧的，遗产行为早就被废弃了。别这么做！

new Function(..) 函数构造器类似地为它的 **最后**一个参数值接收一个代码字符串，来把它转换为一个动态生成的函数（前面的参数值，如果有的话，将作为新函数的形式参数）。这种函数构造器语法要比 eval(..) 稍稍安全一些，但在你的代码中它仍然应当被避免。

在你的代码中动态生成代码的用例少的不可思议，因为在性能上的倒退使得这种能力几乎总是得不偿失。

### with

JavaScript 的另一个使人皱眉头（而且现在被废弃了！），而且可以欺骗词法作用域的特性是 with 关键字。有许多种合法的方式可以讲解 with，但是我在此选择从它如何与词法作用域互动并影响词法作用域的角度来讲解它。

讲解 with 的常见方式是作为一种缩写，来引用一个对象的多个属性，而 不必每次都重复对象引用本身。

例如：

```
var obj = {
    a: 1,
    b: 2,
    c: 3
};

//  重复“obj”显得更“繁冗”
```

```
obj.a = 2;
obj.b = 3;
obj.c = 4;

// “更简单”的缩写
with (obj) {
    a = 3;
    b = 4;
    c = 5;
}
```

然而，这里发生的事情要比只是一个对象属性访问的便捷缩写要多得多。考虑如下代码：

```
function foo(obj) {
    with (obj) {
        a = 2;
    }
}

var o1 = {
    a: 3
};

var o2 = {
    b: 3
};

foo( o1 );
console.log( o1.a ); // 2

foo( o2 );
console.log( o2.a ); // undefined
console.log( a ); // 2 -- 哦，全局作用域被泄漏了！
```

在这个代码示例中，创建了两个对象 o1 和 o2。一个有 a 属性，而另一个没有。foo(..) 函数接收一个对象引用 obj 作为参数值，并在这个引用上调用 with (obj) {..}。在 with 块儿内部，我们制造了一个变量 a 的看似是普通词法引用的东西，实际上是一个 LHS 引用（见第一章），并将值 2 赋予它。

当我们传入 o1 时，赋值 a = 2 找到属性 o1.a 并赋予它值 2，正如在后续的 console.log(o1.a) 语句中反映出的那样。然而，当我们传入 o2，因为它没有 a 属性，没有这样的属性被创建，所以 o2.a 还是 undefined。

但是之后我们注意到一个特别的副作用，赋值 a = 2 创建了一个全局变量 a。这怎么可能?

with 语句接收一个对象，这个对象有 0 个或多个属性，并 **将这个对象视为好像它是一个完全隔离的词法作用域**，因此这个对象的属性被视为在这个 “作用域” 中词法定义的标识符。

**注意：**尽管一个 with 块儿将一个对象视为一个词法作用域，但是在 with 块儿内部的一个普通 var 声明将不会归于这个 with 块儿的作用域，而是归于包含它的函数作用域。

如果 eval(..) 函数接收一个含有一个或多个声明的代码字符串，它就会修改现存的词法作用域，而 with 语句实际上是从你传递给它的对象中凭空制造了一个 **全新的词法作用域**。

以这种方式理解的话，当我们传入 o1 时 with 语句声明的 “作用域” 就是 o1，而且这个 “作用域” 拥有一个对应于 o1.a 属性的 “标识符”。但当我们使用 o2 作为 “作用域” 时，它里面没有这样的 a “标识符”，于是 LHS 标识符查询（见第一章）的普通规则发生了。

“作用域” o2 中没有，foo(..) 的作用域中也没有，甚至连全局作用域中都没有找到标识符 a，所以当 a = 2 被执行时，其结果就是自动全局变量被创建（因为我们没有在 strict 模式下）。

with 在运行时将一个对象和它的属性转换为一个带有 “标识符” 的 “作用域”，这个奇怪想法有些烧脑。但是对于我们看到的结果来说，这是我能给出的最清晰的解释。

**注意：**除了使用它们是个坏主意以外，eval(..) 和 with 都受 Strict 模式的影响（制约）。with 干脆就不允许使用，而虽然 eval(..) 还保有其核心功能，但各种间接形式的或不安全的 eval(..) 是不允许的。

### 性能

通过在运行时修改，或创建新的词法作用域，eval(..) 和 with 都可以欺骗编写时定义的词法作用域。

你可能会问，那又有什么大不了的? 如果它们提供了更精巧的功能和编码灵活性，那它们不是 好的特性吗? **不**。

JavaScript 引擎在编译阶段期行许多性能优化工作。其中的一些优化原理都归结为实质上在进行词法分析时可以静态地分析代码，并提前决定所有的变量和函数声明都在什么位置，这样在执行期间就可以少花些力气来解析标识符。

但如果 引擎在代码中发现一个 eval(..) 或 with，它实质上就不得不 假定自己知道的所有的标识符的位置可能是无效的，因为它不可能在词法分析时就知道你将会向 eval(..) 传递什么样的代码来修改词法作用域，或者你可能会向 with 传递的对象有什么样的内容来创建一个新的将被查询的词法作用域。

换句话说，悲观地看，如果 eval(..) 或 with 出现，那么它 将做的几乎所有的优化都会变得没有意义，所以它就会简单地根本不做任何优化。

你的代码几乎肯定会趋于运行的更慢，只因为你在代码的任何地方引入了一个了 eval(..) 或 with。无论 引擎将在努力限制这些悲观臆测的副作用上表现得多么聪明，**都没有任何办法可**

**以绕过这个事实：没有优化，代码就运行的更慢。**

### 复习

词法作用域意味着作用域是由编写时函数被声明的位置的决策定义的。编译器的词法分析阶段实质上可以知道所有的标识符是在哪里和如何声明的，并如此在执行期间预测它们将如何被查询。

在 JavaScript 中有两种机制可以“欺骗”词法作用域：`eval(..)` 和 `with`。前者可以通过对一个拥有一个或多个声明的“代码”字符串进行求值，来（在运行时）修改现存的词法作用域。后者实质上是通过将一个对象引用看作一个“作用域”，并将这个对象的属性看作作用域中的标识符，（同样，也是在运行时）创建一个全新的词法作用域。

这些机制的缺点是，它压制了 引擎在作用域查询上进行编译期优化的能力，因为 引擎不得不悲观地假定这样的优化是无效的。这两种特性的结果就是代码 将会运行的更慢。**不要使用它们。**

## 第三章：函数与块儿作用域

正如我们在第二章中探索的，作用域由一系列“气泡”组成，这些“气泡”的每一个就像一个容器或篮子，标识符（变量，函数）就在它里面被声明。这些气泡整齐地互相嵌套在一起，而且这种嵌套是在编写时定义的。

但是到底是什么才能制造一个新气泡？只能是函数吗？JavaScript 中的其他结构可以创建作用域的气泡吗？

### 函数中的作用域

对这些问题的最常见的回答是，JavaScript 拥有基于函数的作用域。也就是，你声明的每一个函数都为自己创建了一个气泡，而且没有其他的结构可以创建它们自己的作用域气泡。但是就像我们一会儿将会看到的，这不完全正确。

但首先，让我们探索一下函数作用域和它的含义。

考虑这段代码：

```
function foo(a) {
    var b = 2;

    // 一些代码

    function bar() {
        // ...
    }
```

```
    // 更多代码

    var c = 3;
}
```

在这个代码段中，foo(..) 的作用域气泡包含标识符 a，b，c 和 bar。一个声明出现在作用域 何处是 **无关紧要的**，不管怎样，变量和函数属于包含它们的作用域气泡。在下一章中我们将会探索这到底是如何工作的。

bar(..) 拥有它自己的作用域气泡。全局作用域也一样，它仅含有一个标识符：foo。

因为 a，b，c，和 bar 都属于 foo(..) 的作用域气泡，所以它们在 foo(..) 外部是不可访问的。也就是，接下来的代码都会得到 ReferenceError 错误，因为这些标识符在全局作用域中都不可用：

```
bar(); // 失败

console.log( a, b, c ); // 3 个都失败
```

然而，所有这些标识符（a，b，c，和 bar）在 foo(..) 内部都是可以访问的，而且在 bar(..) 内部也都是可用的（假定在 bar(..) 内部没有遮蔽标识符的声明）。

函数作用域支持着这样的想法：所有变量都属于函数，而且贯穿整个函数始终都可以使用和重用（而且甚至可以在嵌套的作用域中访问）。这种设计方式可以十分有用，而且肯定可以完全利用 JavaScript 的 “动态” 性质 —— 变量可以根据需要接受不同种类型的值。

另一方面，如果你不小心提防，跨越整个作用域存在的变量可能会导致一些意料之外的陷阱。

### 隐藏于普通作用域

考虑一个函数的传统方式是，你声明一个函数，并在它内部添加代码。但是相反的想法也同样强大和有用：拿你所编写的代码的任意一部分，在它周围包装一个函数声明，这实质上 “隐藏” 了这段代码。

其实际结果是在这段代码周围创建了一个作用域气泡，这意味着现在在这段代码中的任何声明都将绑在这个新的包装函数的作用域上，而不是前一个包含它们的作用域。换句话说，你可以通过将变量和函数围在一个函数的作用域中来 “隐藏” 它们。

为什么 “隐藏” 变量和函数是一种有用的技术?

有多种原因驱使着这种基于作用域的隐藏。它们主要是由一种称为 “最低权限原则” 的软件设计原则引起的[5]，有时也被称为 “最低授权” 或 “最少曝光”。这个原则规定，在软件设计中，比如一个模块/对象的 API，你应当只暴露所需要的最低限度的东西，而 “隐藏” 其他的一切。

这个原则可以扩展到用哪个作用域来包含变量和函数的选择。如果所有的变量和函数都在全局作用域中，它们将理所当然地对任何嵌套的作用域来说都是可访问的。但这回违背 “最少……”

[5] Principle of Least Privilege

原则，因为你（很可能）暴露了许多你本应当保持为私有的变量和函数，而这些代码的恰当用法是不鼓励访问这些变量/函数的。

例如：

```
function doSomething(a) {
    b = a + doSomethingElse( a * 2 );

    console.log( b * 3 );
}

function doSomethingElse(a) {
    return a - 1;
}

var b;

doSomething( 2 ); // 15
```

在这个代码段中，变量 b 和函数 doSomethingElse(..) 很可能是 doSomething(..) 如何工作的“私有”细节。允许外围的作用域“访问” b 和 doSomethingElse(..) 不仅没必要而且可能是“危险的”，因为它们可能会以种种意外的方式，有意或无意地被使用，而这也许违背了 doSomething(..) 假设的前提条件。

一个更“恰当”的设计是讲这些私有细节隐藏在 doSomething(..) 的作用域内部，比如：

```
function doSomething(a) {
    function doSomethingElse(a) {
        return a - 1;
    }

    var b;

    b = a + doSomethingElse( a * 2 );

    console.log( b * 3 );
}

doSomething( 2 ); // 15
```

现在，b 和 doSomethingElse(..) 对任何外界影响都是不可访问的，而是仅仅由

doSomething(..) 控制。它的功能和最终结果不受影响，但是这种设计将私有细节保持为私有的，这通常被认为是好的软件。

### 避免冲突

将变量和函数“隐藏”在一个作用域内部的另一个好处是，避免两个同名但用处不同的标识符之间发生无意的冲突。冲突经常导致值被意外地覆盖。

例如：

```
function foo() {
    function bar(a) {
        i = 3; // 在外围的 for 循环的作用域中改变 `i`
        console.log( a + i );
    }

    for (var i=0; i<10; i++) {
        bar( i * 2 ); // 噢，无限循环！
    }
}

foo();
```

bar(..) 内部的赋值 i = 3 意外地覆盖了在 foo(..) 的 for 循环中声明的 i。在这个例子中，这将导致一个无限循环，因为 i 被设定为固定的值 3，而它将永远 < 10。

bar(..) 内部的赋值需要声明一个本地变量来使用，不论选用什么样的标识符名称。var i = 3; 将修复这个问题（并将为 i 创建一个前面提到的“遮蔽变量”声明）。一个*另外的*选项，不是代替的选项，是完全选择另外一个标识符名称，比如 var j = 3;。但是你的软件设计也许会自然而然地使用相同的标识符名称，所以在这种情况下利用作用域来“隐藏”你的内部声明是你最好/唯一的选择。

### 全局“名称空间”

变量冲突（很可能）发生的一个特别强有力的例子是在全局作用域中。当多个库被加载到你的程序中时，如果它们没有适当地隐藏它们的内部/私有函数和变量，那么它们可以十分容易地互相冲突。

这样的库通常会在全局作用域中使用一个足够独特的名称来创建一个单独的变量声明，它经常是一个对象。然后这个对象被用作这个库的一个“名称空间”，所有要明确暴露出来的功能都被作为属性挂在这个对象（名称空间）上，而不是将它们自身作为顶层词法作用域的标识符。

例如：

```
var MyReallyCoolLibrary = {
    awesome: "stuff",
    doSomething: function() {
        // ...
    },
    doAnotherThing: function() {
        // ...
    }
};
```

**模块管理**

另一种回避冲突的选择是通过任意一种依赖管理器，使用更加现代的“模块”方式。使用这些工具，没有库可以向全局作用域添加任何标识符，取而代之的是使用依赖管理器的各种机制，要求库的标识符被明确地导入到另一个指定的作用域中。

应该可以看到，这些工具并不拥有可以豁免于词法作用域规则的“魔法”功能。它们简单地使用这里讲解的作用域规则，来强制标识符不会被注入任何共享的作用域，而是保持在私有的，不易冲突的作用域中，这防止了任何意外的作用域冲突。

因此，如果你选择这样做的话，你可以防御性地编码，并在实际上不使用依赖管理器的情况下，取得与使用它们相同的结果。关于模块模式的更多信息参见第五章。

### 函数作为作用域

我们已经看到，我们可以拿来一段代码并在它周围包装一个函数，而这实质上对外部作用域“隐藏”了这个函数内部作用域包含的任何变量或函数声明。

例如：

```
var a = 2;

function foo() { // <-- 插入这个

    var a = 3;
    console.log( a ); // 3

} // <-- 和这个
foo(); // <-- 还有这个

console.log( a ); // 2
```

虽然这种技术“可以工作”，但它不一定非常理想。它引入了几个问题。首先是我们不得不声明一个命名函数 `foo()`，这意味着这个标识符名称 `foo` 本身就“污染”了外围作用域（在这个

例子中是全局）。我们要不得不通过名称（foo()）明确地调用这个函数来使被包装的代码真正运行。

如果这个函数不需要名称（或者，这个名称不污染外围作用域），而且如果这个函数能自动地被执行就更理想了。

幸运的是，JavaScript 给这两个问题提供了一个解决方法。

```
var a = 2;

(function foo(){ // <-- 插入这个

    var a = 3;
    console.log( a ); // 3

})(); // <-- 和这个

console.log( a ); // 2
```

让我们分析一下这里发生了什么。

首先注意，与仅仅是 function... 相对，这个包装函数语句以 (function... 开头。虽然这看起来像是一个微小的细节，但实际上这是一个重大改变。与将这个函数视为一个标准的声明不同的是，这个函数被视为一个函数表达式。

**注意**：区分声明与表达式的最简单的方法是，这个语句中（不仅仅是一行，而是一个独立的语句）“function” 一词的位置。如果 “function” 是这个语句中的第一个东西，那么它就是一个函数声明。否则，它就是一个函数表达式。

这里我们可以观察到一个函数声明和一个函数表达式之间的关键不同是，它的名称作为一个标识符被绑定在何处。

比较这前两个代码段。在第一个代码段中，名称 foo 被绑定在外围作用域中，我们用 foo() 直接调用它。在第二个代码段中，名称 foo 没有被绑定在外围作用域中，而是被绑定在它自己的函数内部。

换句话说，(function foo(){ .. }) 作为一个表达式意味着标识符 foo 仅能在 .. 代表的作用域中被找到，而不是在外部作用域中。将名称 foo 隐藏在它自己内部意味着它不会没必要地污染外围作用域。

### 匿名与命名

你可能对函数表达式作为回调参数再熟悉不过了，比如：

```
setTimeout( function(){
    console.log("I waited 1 second!");
}, 1000 );
```

这称为一个“匿名函数表达式”，因为 `function()...` 上没有名称标识符。函数表达式可以是匿名的，但是函数声明不能省略名称 —— 那将是不合法的 JS 程序。

匿名函数表达式可以快速和很容易地键入，而且许多库和工具往往鼓励使用这种代码惯用风格。然而，它们有几个缺点需要考虑：

1. 在栈轨迹上匿名函数没有有用的名称可以表示，这可能会使得调试更加困难。
2. 没有名称的情况下，如果这个函数需要为了递归等目的引用它自己，那么就需要很不幸地使用 **被废弃的** `arguments.callee` 引用。另一个需要自引用的例子是，当一个事件处理器函数在被触发后想要把自己解除绑定。
3. 匿名函数省略的名称经常对提供更易读/易懂的代码很有帮助。一个描述性的名称可以帮助代码自解释。

**内联函数表达式**很强大且很有用 —— 匿名和命名的问题并不会贬损这一点。给你的函数表达式提供一个名称就可以十分有效地解决这些缺陷，而且没有实际的坏处。最佳的方法是总是命名你的函数表达式：

```
setTimeout( function timeoutHandler(){ // <-- 看，我有一个名字!
    console.log( "I waited 1 second!" );
}, 1000 );
```

#### 立即调用函数表达式

```
var a = 2;

(function foo(){

    var a = 3;
    console.log( a ); // 3

})();

console.log( a ); // 2
```

得益于包装在一个 `()` 中，我们有了一个作为表达式的函数，我们可以通过在末尾加入另一个 `()` 来执行这个函数，就像 `(function foo(){ .. })()`。第一个外围的 `( )` 使这个函数变成表达式，而第二个 `()` 执行这个函数。

这个模式是如此常见，以至于几年前开发者社区一致同意给它一个术语：**IIFE**，它表示“立即被调用的函数表达式”（**I**mmediately **I**nvoked **F**unction **E**xpression）。

当然，IIFE 不一定需要一个名称 —— IIFE 的最常见形式是使用一个匿名函数表达式。虽然少见一些，但与匿名函数表达式相比，命名的 IIFE 拥有前述所有的好处，所以它是一个可以采用的好方式。

```
var a = 2;

(function IIFE(){

    var a = 3;
    console.log( a ); // 3

})();

console.log( a ); // 2
```

传统的 IIFE 有一种稍稍变化的形式，一些人偏好这样：`(function(){ .. }())`。仔细观察不同之处。在第一种形式中，函数表达式被包在 ( ) 中，然后用于调用的 () 出现在它的外侧。在第二种形式中，用于调用的 () 被移动到用于包装的 ( ) 内侧。

这两种形式在功能上完全相同。**这纯粹是一个你偏好的风格的选择**。

IIFE 的另一种十分常见的变种是，利用它们实际上只是函数调用的事实，来传入参数值。

例如：

```
var a = 2;

(function IIFE( global ){

    var a = 3;
    console.log( a ); // 3
    console.log( global.a ); // 2

})( window );

console.log( a ); // 2
```

我们传入 `window` 对象引用，但是我们将参数命名为 `global`，这样我们对于全局和非全局引用就有了一个清晰的文体上的划分。当然，你可以从外围作用域传入任何你想要的东西，而且你可以将参数命名为任何适合你的名称。这几乎仅仅是文体上的选择。

这种模式的另一种应用解决了一个小问题：默认的 `undefined` 标识符的值也许会被不正确地覆盖掉，而导致意外的结果。通过将参数命名为 `undefined`，同时不为它传递任何参数值，我们就可以保证在一个代码块中 `undefined` 标识符确实是是一个未定义的值。

```
undefined = true; // 给其他的代码埋地雷！别这么干！

(function IIFE( undefined ){

    var a;
    if (a === undefined) {
        console.log( "Undefined is safe here!" );
    }

})();
```

IIFE 还有另一种变种，它将事情的顺序倒了过来，要被执行的函数在调用和传递给它的参数 *之后*给出。这种模式被用于 UMD（Universal Module Definition —— 统一模块定义）项目。一些人发现它更干净和易懂一些，虽然有点儿繁冗。

```
var a = 2;

(function IIFE( def ){
    def( window );
})(function def( global ){

    var a = 3;
    console.log( a ); // 3
    console.log( global.a ); // 2

});
```

`def` 函数表达式在这个代码段的后半部分被定义，然后作为一个参数（也叫 `def`）被传递给在代码段前半部分定义的 `IIFE` 函数。最后，参数 `def`（函数）被调用，并将 `window` 作为 `global` 参数传入。

### 块儿作为作用域

虽然函数是最常见的作用域单位，而且当然也是在世面上流通的绝大多数 JS 中最为广泛传播的设计方式，但是其他的作用域单位也是可能的，而且使用这些作用域单位可以导致更好、对于维护来说更干净的代码。

JavaScript 之外的许多其他语言都支持块儿作用域，所以有这些语言背景的开发者习惯于这种思维模式，然而那些主要在 JavaScript 中工作的开发者可能会发现这个概念有些陌生。

但即使你从没用块儿作用域的方式写过一行代码，你可能依然对 JavaScript 中这种极其常见的惯用法很熟悉：

```
for (var i=0; i<10; i++) {
    console.log( i );
}
```

我们在 for 循环头的内部直接声明了变量 i，因为我们意图很可能是仅在这个 for 循环内部的上下文环境中使用 i，而实质上忽略了这个变量实际上将自己划入了外围作用域中（函数或全局）的事实。

这就是有关块儿作用域的一切。尽可能靠近地，尽可能局部地，在变量将被使用的位置声明它。另一个例子是：

```
var foo = true;

if (foo) {
    var bar = foo * 2;
    bar = something( bar );
    console.log( bar );
}
```

我们仅在 if 语句的上下文环境中使用变量 bar，所以我们将它声明在 if 块儿的内部是有些道理的。然而，当使用 var 时，我们在何处声明变量是无关紧要的，因为它们将总是属于外围作用域。这个代码段实质上为了代码风格的原因“假冒”了块儿作用域，并依赖于我们要管好自己，不要在这个作用域的其他地方意外地使用 bar。

从将信息隐藏在函数中，到将信息隐藏在我们代码的块儿中，块儿作用域是一种扩展了早先的“最低 ~~权限~~ 暴露原则”[6]的工具。

再次考虑这个 for 循环的例子：

```
for (var i=0; i<10; i++) {
    console.log( i );
}
```

为什么要用仅将（或者至少是，仅 *应当*）在这个 for 循环中使用的变量 i 去污染一个函数的整个作用域呢？

但更重要的是，开发者们也许偏好于 *检查* 他们自己来防止在变量预期的目的之外意外地（重）使用它们，例如如果你试着在错误的地方使用变量会导致一个未知变量的错误。对于变量 i 的块儿作用域（如果它是可能的话）将使 i 仅在 for 循环内部可用，使得如果在函数的其他地方访问 i 将导致一个错误。这有助于保证变量不会被糊涂地重用或者难于维护。

但是，悲惨的现实是，表面上看来，JavaScript 没有块儿作用域的能力。

更确切地说，直到你再深入一些才有。

[6] Principle of Least Privilege

### with

我们在第二章中学习了 `with`。虽然它是一个使人皱眉头的结构，但它确实是一个（一种形式的）块儿作用域的例子，它从对象中创建的作用域仅存在于这个 `with` 语句的生命周期中，而不在外围作用域中。

### try/catch

一个鲜为人知的事实是，JavaScript 在 ES3 中明确指出在 `try/catch` 的 `catch` 子句中声明的变量，是属于 `catch` 块儿的块儿作用域的。

例如：

```
try {
    undefined(); //用非法的操作强制产生一个异常!
}
catch (err) {
    console.log( err ); // 好用!
}

console.log( err ); // ReferenceError: `err` not found
```

如你所见，`err` 仅存在于 `catch` 子句中，并且在你试着从其他地方引用它时抛出一个错误。

**注意**：虽然这种行为已经被明确规定，而且对于几乎所有的标准 JS 环境（也许除了老 IE）来说都是成立的，但是如果你在同一个作用域中有两个或多个 `catch` 子句，而它们又各自用相同的标识符名称声明了它们表示错误的变量时，许多 linter 依然会报警。实际上这不是重定义，因为这些变量都安全地位于块儿作用域中，但是 linter 看起来依然会恼人地抱怨这个事实。

为了避免这些不必要的警告，一些开发者将他们的 `catch` 变量命名为 `err1`，`err2`，等等。另一些开发者干脆关闭 linter 对重复变量名的检查。

`catch` 的块儿作用域性质看起来像是一个没用的，只有学院派意义的事实，但是参看附录 B 来了解更多它如何有用的信息。

### let

至此，我们看到 JavaScript 仅仅有一些奇怪的小众行为暴露了块儿作用域功能。如果这就是我们拥有的一切，而且许多许多年以来这 *确实就是* 我们拥有的一切，那么块作用域对 JavaScript 开发者来说就不是非常有用。

幸运的是，ES6 改变了这种状态，并引入了一个新的关键字 `let`，作为另一种声明变量的方式伴随着 `var`。

`let` 关键字将变量声明附着在它所在的任何块儿（通常是一个 `{ .. }`）的作用域中。换句话说，`let` 为它的变量声明隐含地劫持了任意块儿的作用域。

```
var foo = true;

if (foo) {
    let bar = foo * 2;
    bar = something( bar );
    console.log( bar );
}

console.log( bar ); // ReferenceError
```

使用 let 将一个变量附着在一个现存的块儿上有些隐晦。它可能会使人困惑 —— 在你开发和设计代码时，如果你不仔细注意哪些块儿的作用域包含了变量，并且习惯于将块儿四处移动，将它们包进其他的块儿中，等等。

为块儿作用域创建明确的块儿可以解决这些问题中的一些，使变量附着在何处更加明显。通常来说，明确的代码要比隐晦或微妙的代码好。这种明确的块儿作用域风格很容易达成，而且它与块儿作用域在其他语言中的工作方式匹配得更自然：

```
var foo = true;

if (foo) {
    { // <-- 明确的块儿
        let bar = foo * 2;
        bar = something( bar );
        console.log( bar );
    }
}

console.log( bar ); // ReferenceError
```

我们可以在一个语句是合法文法的任何地方，通过简单地引入一个 { .. } 来为 let 创建一个任意的可以绑定的块儿。在这个例子中，我们在 if 语句内部制造了一个明确的块儿，在以后的重构中将整个块儿四处移动可能会更容易，而且不会影响外围的 if 语句的位置和语义。

**注意**：另一个明确表达块儿作用域的方法，参见附录 B。

在第四章中，我们将讲解提升（hoisting），它讲述关于声明在它们所出现的整个作用域中都被认为是存在的。

然而，使用 let 做出的声明将 不会在它们所出现的整个块儿的作用域中提升。如此，直到声明语句为止，声明将不会 “存在” 于块儿中。

```
{
   console.log( bar ); // ReferenceError!
```

```
    let bar = 2;
}
```

**垃圾回收**

块儿作用域的另一个有用之处是关于闭包和释放内存的垃圾回收。我们将简单地在这里展示一下，但是闭包机制将在第五章中详细讲解。

考虑这段代码：

```
function process(data) {
    // 做些有趣的事
}

var someReallyBigData = { .. };

process( someReallyBigData );

var btn = document.getElementById( "my_button" );

btn.addEventListener( "click", function click(evt){
    console.log("button clicked");
}, /*capturingPhase=*/false );
```

点击事件的处理器回调函数 `click` 根本不 *需要* `someReallyBigData` 变量。这意味着从理论上讲，在 `process(..)` 运行之后，这个消耗巨大内存的数据结构可以被作为垃圾回收。然而，JS 引擎很可能（虽然这要看具体实现）仍会将这个结构保持一段时间，因为 `click` 函数在整个作用域上拥有一个闭包。

块儿作用域可以解决这个问题，使引擎清楚地知道它不必再保持 `someReallyBigData` 了：

```
function process(data) {
    // 做些有趣的事
}

// 运行过后，任何定义在这个块中的东西都可以消失了
{
    let someReallyBigData = { .. };

    process( someReallyBigData );
}
```

```
var btn = document.getElementById( "my_button" );

btn.addEventListener( "click", function click(evt){
    console.log("button clicked");
}, /*capturingPhase=*/false );
```

声明可以将变量绑定在本地的明确的块儿是一种强大的工具，你可以把它加入你的工具箱。

**let 循环**

一个使 let 闪光的特殊例子是我们先前讨论的 for 循环。

```
for (let i=0; i<10; i++) {
    console.log( i );
}

console.log( i ); // ReferenceError
```

在 for 循环头部的 let 不仅将 i 绑定在 for 循环体中，而且实际上，它会对每一次循环的迭代 **重新绑定** i，确保它被赋予来自上一次循环迭代末尾的值。

这是描绘这种为每次迭代进行绑定的行为的另一种方式：

```
{
    let j;
    for (j=0; j<10; j++) {
        let i = j; // 每次迭代都重新绑定
        console.log( i );
    }
}
```

这种为每次迭代进行的绑定有趣的原因将在第五章中我们讨论闭包时变得明朗。

因为 let 声明附着于任意的块儿，而不是外围的函数作用域（或全局），所以在重构代码时可能会有一些坑需要额外小心：现存的代码拥有对函数作用域的 var 声明有隐藏的依赖，但你想要用 let 来取代 var。

考虑如下代码：

```
var foo = true, baz = 10;

if (foo) {
    var bar = 3;
```

```
    if (baz > bar) {
        console.log( baz );
    }

    // ...
}
```

这段代码可以相当容易地重构为：

```
var foo = true, baz = 10;

if (foo) {
    var bar = 3;

    // ...
}

if (baz > bar) {
    console.log( baz );
}
```

但是，当使用块儿作用域变量时要小心这样的变化：

```
var foo = true, baz = 10;

if (foo) {
    let bar = 3;

    if (baz > bar) { // <-- 移动时不要忘了 `bar`
        console.log( baz );
    }
}
```

附录 B 介绍了一种块作用域的（更加明确的）替代形式，它可能会在这些场景下提供更易于维护/重构的更健壮的代码。

#### const

除了 `let` 之外，ES6 还引入了 `const`，它也创建一个块儿作用域变量，但是它的值是固定的（常量）。任何稍后改变它的企图都将导致错误。

```
var foo = true;

if (foo) {
    var a = 2;
    const b = 3; // 存在于包含它的 `if`作用域中

    a = 3; // 没问题!
    b = 4; // 错误!
}

console.log( a ); // 3
console.log( b ); // ReferenceError!
```

### 复习

在 JavaScript 中函数是最常见的作用域单位。在另一个函数内部声明的变量和函数，实质上对任何外围 “作用域” 都是 “隐藏的”，这是优秀软件的一个有意的设计原则。

但是函数绝不是唯一的作用域单位。块儿作用域指的是这样一种想法：变量和函数可以属于任意代码块儿（一般来说，就是任意的 `{ .. }`），而不是仅属于外围的函数。

从 ES3 开始，`try/catch` 结构在 `catch` 子句上拥有块儿作用域。

在 ES6 中，引入了 `let` 关键字（`var` 关键字的表兄弟）允许在任意代码块中声明变量。`if (..) { let a = 2; }` 将会声明变量 `a`，而它实质上劫持了 `if` 的 `{ .. }` 块儿的作用域，并将自己附着在这里。

虽然有些人对此深信不疑，但是块儿作用域不应当被认为是 `var` 函数作用域的一个彻头彻尾的替代品。两种机能是共存的，而且开发者们可以并且应当同时使用函数作用域和块儿作用域技术 —— 在它们各自可以产生更好，更易读/易维护代码的地方。

## 第四章：提升

至此，你应当对作用域的想法，以及变量如何根据它们被声明的方式和位置附着在不同的作用域层级上感到相当适应了。函数作用域和块儿作用域的行为都是依赖于这个相同规则的：在一个作用域中声明的任何变量都附着在这个作用域上。

但是关于出现在一个作用域内各种位置的声明如何附着在作用域上，有一个微妙的细节，而这个细节正是我们要在这里检视的。

### 先有鸡还是先有蛋?

有一种倾向认为你在 JavaScript 程序中看到的所有代码，在程序执行的过程中都是从上到下一行一行地被解释执行的。虽然这大致上是对的，但是这种猜测中的一个部分可能会导致你

错误地考虑你的程序。

考虑这段代码：

```
a = 2;

var a;

console.log( a );
```

你觉得在 `console.log(..)` 语句中会打印出什么？

许多开发者会期望 `undefined`，因为语句 `var a` 出现在 `a = 2` 之后，这很自然地看起来像是这个变量被重定义了，并因此被赋予了默认的 `undefined`。然而，输出将是 `2`。

考虑另一个代码段：

```
console.log( a );

var a = 2;
```

你可能会被诱导而这样认为：因为上一个代码段展示了一种看起来不是从上到下的行为，也许在这个代码段中，也会打印 `2`。另一些人认为，因为变量 `a` 在它被声明之前就被使用了，所以这一定会导致一个 `ReferenceError` 被抛出。

不幸的是，两种猜测都不正确。输出是 `undefined`。

**那么。这里发生了什么？** 看起来我们遇到了一个先有鸡还是先有蛋的问题。哪一个先有？声明（“蛋”），还是赋值（“鸡”）？

### 编译器再次袭来

要回答这个问题，我们需要回头引用第一章关于编译器的讨论。回忆一下，引擎实际上将会在它解释执行你的 JavaScript 代码之前编译它。编译过程的一部分就是找到所有的声明，并将它们关联在合适的作用域上。第二章向我们展示了这是词法作用域的核心。

所以，考虑这件事情的最佳方式是，在你的代码的任何部分被执行之前，所有的声明，变量和函数，都会首先被处理。

当你看到 `var a = 2;` 时，你可能认为这是一个语句。但是 JavaScript 实际上认为这是两个语句：`var a;` 和 `a = 2;`。第一个语句，声明，是在编译阶段被处理的。第二个语句，赋值，为了执行阶段而留在 **原处**。

于是我们的第一个代码段应当被认为是这样被处理的：

```
var a;

a = 2;

console.log( a );
```

……这里的第一部分是编译，而第二部分是执行。

相似地，我们的第二个代码段实际上被处理为：

```
var a;

console.log( a );

a = 2;
```

所以，关于这种处理的一个有些隐喻的考虑方式是，变量和函数声明被从它们在代码流中出现的位置“移动”到代码的顶端。这就产生了“提升”这个名字。

换句话说，**先有蛋（声明），后有鸡（赋值）**。

**注意**：只有声明本身被提升了，而任何赋值或者其他的执行逻辑都被留在 *原处*。如果提升会重新安排我们代码的可执行逻辑，那就会是一场灾难了。

```
foo();

function foo() {
    console.log( a ); // undefined

    var a = 2;
}
```

函数 `foo` 的声明（在这个例子中它还 *包含*一个隐含的、实际为函数的值）被提升了，因此第一行的调用是可以执行的。

还需要注意的是，提升是 **以作用域为单位的**。所以虽然我们的前一个代码段被简化为仅含有全局作用域，但是我们现在检视的函数 `foo(..)` 本身展示了，`var a` 被提升至 `foo(..)` 的顶端（很明显，不是程序的顶端）。所以这个程序也许可以更准确地解释为：

```
function foo() {
    var a;

    console.log( a ); // undefined

    a = 2;
}

foo();
```

函数声明会被提升，就像我们看到的。但是函数表达式不会。

```
foo(); // 不是 ReferenceError，而是 TypeError!

var foo = function bar() {
    // ...
};
```

变量标识符 foo 被提升并被附着在这个程序的外围作用域（全局），所以 foo() 不会作为一个 ReferenceError 而失败。但 foo 还没有值（如果它不是函数表达式，而是一个函数声明，那么它就会有值）。所以，foo() 就是试图调用一个 undefined 值，这是一个 TypeError —— 非法操作。

同时回想一下，即使它是一个命名的函数表达式，这个名称标识符在外围作用域中也是不可用的：

```
foo(); // TypeError
bar(); // ReferenceError

var foo = function bar() {
    // ...
};
```

这个代码段可以（使用提升）更准确地解释为：

```
var foo;

foo(); // TypeError
bar(); // ReferenceError

foo = function() {
    var bar = ...self...
    // ...
}
```

### 函数优先

函数声明和变量声明都会被提升。但一个微妙的细节（可以在拥有多个“重复的”声明的代码中出现）是，函数会首先被提升，然后才是变量。

考虑这段代码：

```
foo(); // 1
```

```
var foo;

function foo() {
    console.log( 1 );
}

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

1 被打印了，而不是 2！这个代码段被 引擎解释执行为：

```
function foo() {
    console.log( 1 );
}

foo(); // 1

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

注意那个 var foo 是一个重复（因此被无视）的声明，即便它出现在 function foo()... 声明之前，因为函数声明是在普通变量之前被提升的。

虽然多个/重复的 var 声明实质上是被忽略的，但是后续的函数声明确实会覆盖前一个。

```
foo(); // 3

function foo() {
    console.log( 1 );
}

var foo = function() {
    console.log( 2 );
};

function foo() {
    console.log( 3 );
}
```

虽然这一切听起来不过是一些有趣的学院派细节，但是它强调了一个事实：在同一个作用域内的重复定义是一个十分差劲儿的主意，而且经常会导致令人困惑的结果。

在普通的块儿内部出现的函数声明一般会被提升至外围的作用域，而不是像这段代码暗示的那样有条件地被定义：

```
foo(); // "b"

var a = true;
if (a) {
   function foo() { console.log( "a" ); }
}
else {
   function foo() { console.log( "b" ); }
}
```

然而，重要的是要注意这种行为是不可靠的，而且是未来版本的 JavaScript 将要改变的对象，所以避免在块儿中声明函数可能是最好的做法。

### 复习

我们可能被诱导而将 `var a = 2` 看作是一个语句，但是 JavaScript 引擎可不这么看。它将 `var a` 和 `a = 2` 看作两个分离的语句，第一个是编译期的任务，而第二个是执行时的任务。

这将导致在一个作用域内的所有声明，不论它们出现在何处，都会在代码本身被执行前 *首先*被处理。你可以将它可视化为声明（变量与函数）被“移动”到它们各自的作用域顶部，这就是我们所说的“提升”。

声明本身会被提升，但不是赋值，即便是函数表达式的赋值，也 *不会*被提升。

要小心重复声明，特别是将一般的变量声明和函数声明混在一起 —— 如果你这么做的话，危险就在眼前！

## 第五章：作用域闭包

希望我们是带着对作用域工作方式的健全，坚实的理解来到这里的。

我们将我们的注意力转向这个语言中一个重要到不可思议，但是一直难以捉摸的、*几乎是神话般的*部分：**闭包**。如果你至此一直跟随着我们关于词法作用域的讨论，那么你会感觉闭包将在很大程度上没那么令人激动，几乎是显而易见的。*有一个魔法师坐在幕后，现在我们即将见到他*。不，他的名字不是 Crockford！

如果你还对词法作用域感到不安，那么现在就是在继续之前回过头去再复习一下第二章的好时机。

## 启蒙

对于那些对 JavaScript 有些经验，但是也许从没全面掌握闭包概念的人来说，理解闭包看起来就像是必须努力并作出牺牲才能到达的涅槃状态。

回想几年前我对 JavaScript 有了牢固的掌握，但是不知道闭包是什么。它暗示着这种语言有着另外的一面，它许诺了甚至比我已经拥有的还多的力量，它取笑并嘲弄我。我记得我通读早期框架的源代码试图搞懂它到底是如何工作的。我记得第一次“模块模式”的某些东西融入我的大脑。我记得那依然栩栩如生的啊哈！一刻。

那时我不明白的东西，那个花了我好几年时间才搞懂的东西，那个我即将传授给你的东西，是这个秘密：**在 JavaScript 中闭包无所不在，你只是必须认出它并接纳它**。闭包不是你必须学习新的语法和模式才能使用的特殊的可选的工具。不，闭包甚至不是你必须像卢克在原力中修炼那样，一定要学会使用并掌握的武器。

闭包是依赖于词法作用域编写代码而产生的结果。它们就这么发生了。要利用它们你甚至不需要有意地创建闭包。闭包在你的代码中一直在被创建和使用。你缺少的是恰当的思维环境，来识别，接纳，并以自己的意志利用闭包。

启蒙的时刻应该是：**哦，闭包已经在我的代码中到处发生了，现在我终于看到它们了**。理解闭包就像是尼欧第一次见到母体。

## 事实真相

好了，夸张和对电影的无耻引用够多了。

为了理解和识别闭包，这里有一个你需要知道的简单粗暴的定义：

> 闭包就是函数能够记住并访问它的词法作用域，即使当这个函数在它的词法作用域之外执行时。

让我们跳进代码来说明这个定义：

```
function foo() {
    var a = 2;

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

    bar();
}

foo();
```

根据我们对嵌套作用域的讨论，这段代码应当看起来很熟悉。由于词法作用域查询规则（在这个例子中，是一个 RHS 引用查询），函数 bar() 可以 *访问*外围作用域的变量 a。

这是“闭包”吗?

好吧，从技术上讲…… *也许是*。但是根据我们上面的“你需要知道”的定义…… *不确切*。我认为解释 bar() 引用 a 的最准确的方式是根据词法作用域查询规则，但是那些规则 *仅仅*是闭包的（一个很重要的!）**一部分**。

从纯粹的学院派角度讲，上面的代码段被认为是函数 bar() 在函数 foo() 的作用域上有一个 *闭包*（而且实际上，它甚至对其他的作用域也可以访问，比如这个例子中的全局作用域）。换一种略有不同的说法是，bar() 闭住了 foo() 的作用域。为什么? 因为 bar() 嵌套地出现在 foo() 内部。就这么简单。

但是，这样一来闭包的定义就是不能直接 *观察到*的了，我们也不能看到闭包在这个代码段中 *被行使*。我们清楚地看到词法作用域，但是闭包仍然像代码后面谜一般的模糊阴影。

让我们考虑这段将闭包完全带到聚光灯下的代码:

```
function foo() {
    var a = 2;

    function bar() {
        console.log( a );
    }

    return bar;
}

var baz = foo();

baz(); // 2 -- 哇噢，看到闭包了，伙计。
```

函数 bar() 对于 foo() 内的作用域拥有词法作用域访问权。但是之后，我们拿起 bar()，这个函数本身，将它像 *值*一样传递。在这个例子中，我们 return bar 引用的函数对象本身。

在执行 foo() 之后，我们将它返回的值（我们的内部 bar() 函数）赋予一个称为 baz 的变量，然后我们实际地调用 baz()，这将理所当然地调用我们内部的函数 bar()，只不过是通过一个不同的标识符引用。

bar() 被执行了，必然的。但是在这个例子中，它是在它被声明的词法作用域 *外部*被执行的。

foo() 被执行之后，一般说来我们会期望 foo() 的整个内部作用域都将消失，因为我们知道 *引擎*启用了 *垃圾回收器*在内存不再被使用时来回收它们。因为很显然 foo() 的内容不再被使用了，所以看起来它们很自然地应该被认为是 *消失了*。

但是闭包的“魔法”不会让这发生。内部的作用域实际上 *依然* “在使用”，因此将不会消失。谁在使用它？**函数 bar() 本身**。

有赖于它被声明的位置，bar() 拥有一个词法作用域闭包覆盖着 foo() 的内部作用域，闭包为了能使 bar() 在以后任意的时刻可以引用这个作用域而保持它的存在。

**bar() 依然拥有对那个作用域的引用，而这个引用称为闭包**。

所以，在几微秒之后，当变量 baz 被调用时（调用我们最开始标记为 bar 的内部函数），它理所应当地对编写时的词法作用域拥有 *访问*权，所以它可以如我们所愿地访问变量 a。

这个函数在它被编写时的词法作用域之外被调用。**闭包**使这个函数可以继续访问它在编写时被定义的词法作用域。

当然，函数可以被作为值传递，而且实际上在其他位置被调用的所有各种方式，都是观察/行使闭包的例子。

```
function foo() {
    var a = 2;

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

    bar( baz );
}

function bar(fn) {
    fn(); // 看妈妈，我看到闭包了！
}
```

我们将内部函数 baz 传递给 bar，并调用这个内部函数（现在被标记为 fn），当我们这么做时，它覆盖在 foo() 内部作用域的闭包就可以通过 a 的访问观察到。

这样的函数传递也可以是间接的。

```
var fn;

function foo() {
    var a = 2;

    function baz() {
        console.log( a );
    }
```

```
    fn = baz; // 将 `baz`赋值给一个全局变量
}

function bar() {
    fn(); // 看妈妈，我看到闭包了！
}

foo();

bar(); // 2
```

无论我们使用什么方法将内部函数 *传送*到它的词法作用域之外，它都将维护一个指向它最开始被声明时的作用域的引用，而且无论我们什么时候执行它，这个闭包就会被行使。

### 现在我能看到了

前面的代码段有些学术化，而且是人工构建来说明 *闭包的使用*的。但我保证过给你的东西不止是一个新的酷玩具。我保证过闭包是在你的现存代码中无处不在的东西。现在让我们 *看看*真相。

```
function wait(message) {

    setTimeout( function timer(){
        console.log( message );
    }, 1000 );

}

wait( "Hello, closure!" );
```

我们拿来一个内部函数（名为 `timer`）将它传递给 `setTimeout(..)`。但是 `timer` 拥有覆盖 `wait(..)` 的作用域的闭包，实际上保持并使用着对变量 `message` 的引用。

在我们执行 `wait(..)` 一千毫秒之后，要不是内部函数 `timer` 依然拥有覆盖着 `wait()` 内部作用域的闭包，它早就会消失了。

在 *引擎*的内脏深处，内建的工具 `setTimeout(..)` 拥有一些参数的引用，可能称为 `fn` 或者 `func` 或者其他诸如此类的东西。*引擎*去调用这个函数，它调用我们的内部 `timer` 函数，而词法作用域依然完好无损。

**闭包**。

或者，如果你信仰 jQuery（或者就此而言，其他的任何 JS 框架）：

```
function setupBot(name,selector) {
    $( selector ).click( function activator(){
        console.log( "Activating: " + name );
    } );
}

setupBot( "Closure Bot 1", "#bot_1" );
setupBot( "Closure Bot 2", "#bot_2" );
```

我不确定你写的是什么代码，但我通常写一些代码来负责控制全球的闭包无人机军团，所以这完全是真实的！

把玩笑放在一边，实质上 *无论何时何地*只要你将函数作为头等的值看待并将它们传来传去的话，你就可能看到这些函数行使闭包。计时器、事件处理器、Ajax 请求、跨窗口消息、web worker、或者任何其他的异步（或同步！）任务，当你传入一个 *回调函数*，你就在它周围悬挂了一些闭包！

**注意**：第三章介绍了 IIFE 模式。虽然人们常说 IIFE（独自）是一个可以观察到闭包的例子，但是根据我们上面的定义，我有些不同意。

```
var a = 2;

(function IIFE(){
    console.log( a );
})();
```

这段代码“好用”，但严格来说它不是在观察闭包。为什么？因为这个函数（就是我们这里命名为“IIFE”的那个）没有在它的词法作用域之外执行。它仍然在它被声明的相同作用域中（那个同时持有 `a` 的外围/全局作用域）被调用。`a` 是通过普通的词法作用域查询找到的，不是通过真正的闭包。

虽说技术上闭包可能发生在声明时，但它 *不是*严格地可以观察到的，因此，就像人们说的，*它是一颗在森林中倒掉的树，但周围没人去听到它。*

虽然 IIFE *本身*不是一个闭包的例子，但是它绝对创建了作用域，而且它是我们用来创建可以被闭包的作用域的最常见工具之一。所以 IIFE 确实与闭包有强烈的关联，即便它们本身不行使闭包。

亲爱的读者，现在把这本书放下。我有一个任务给你。去打开一些你最近的 JavaScript 代码。寻找那些被你作为值的函数，并识别你已经在那里使用了闭包，而你以前甚至可能不知道它。

我会等你。

现在……你看到了！

### 循环 + 闭包

用来展示闭包最常见最权威的例子是老实巴交的 for 循环。

```
for (var i=1; i<=5; i++) {
    setTimeout( function timer(){
        console.log( i );
    }, i*1000 );
}
```

**注意**：当你将函数放在循环内部时 Linter 经常会抱怨，因为不理解闭包的错误 **在开发者中太常见了**。我们在这里讲解如何正确地利用闭包的全部力量。但是 Linter 通常不理解这样的微妙之处，所以它们不管怎样都将抱怨，认为你 实际上不知道你在做什么。

这段代码的精神是，我们一般将 期待它的行为是分别打印数字 “1”，“2”，……“5”，一次一个，一秒一个。

实际上，如果你运行这段代码，你会得到 “6” 被打印 5 次，一秒一个。

**啊?**

首先，让我们解释一下 “6” 是从哪儿来的。循环的终结条件是 `i` 不 `<=5`。第一次满足这个条件时 `i` 是 6。所以，输出的结果反映的是 `i` 在循环终结后的最终值。

如果多看两眼的话这其实很明显。超时的回调函数都将在循环的完成之后立即运行。实际上，就计时器而言，即便在每次迭代中它是 `setTimeout(.., 0)`，所有这些回调函数也都仍然是严格地在循环之后运行的，因此每次都打印 6。

但是这里有个更深刻的问题。要是想让它实际上如我们在语义上暗示的那样动作，我们的代码缺少了什么?

缺少的东西是，我们试图 暗示在迭代期间，循环的每次迭代都 “捕捉” 一份对 `i` 的拷贝。但是，虽然所有这 5 个函数在每次循环迭代中分离地定义，由于作用域的工作方式，它们 **都闭包在同一个共享的全局作用域上**，而它事实上只有一个 `i`。

这么说来，所有函数共享一个指向相同的 `i` 的引用是 理所当然的。循环结构的某些东西往往迷惑我们，使我们认为这里有其他更精巧的东西在工作。但是这里没有。这与根本没有循环，5 个超时回调仅仅一个接一个地被声明没有区别。

好了，那么，回到我们火烧眉毛的问题。缺少了什么? 我们需要更多 ~~铃声~~ 被闭包的作用域。明确地说，我们需要为循环的每次迭代都准备一个新的被闭包的作用域。

我们在第三章中学到，IIFE 通过声明并立即执行一个函数来创建作用域。

让我们试试：

```
for (var i=1; i<=5; i++) {
    (function(){
        setTimeout( function timer(){
            console.log( i );
```

```
        }, i*1000 );
    })();
}
```

这好用吗？试试。我还会等你。

我来为你终结悬念。**不好用**。但是为什么？很明显我们现在有了更多的词法作用域。每个超时回调函数确实闭包在每次迭代时分别被每个 IIFE 创建的作用域中。

拥有一个被闭包的 **空的作用域**是不够的。仔细观察。我们的 IIFE 只是一个空的什么也不做的作用域。它内部需要 *一些东西*才能变得对我们有用。

它需要它自己的变量，在每次迭代时持有值 `i` 的一个拷贝。

```
for (var i=1; i<=5; i++) {
    (function(){
        var j = i;
        setTimeout( function timer(){
            console.log( j );
        }, j*1000 );
    })();
}
```

**万岁！它好用了！**

有些人偏好一种稍稍变形的形式：

```
for (var i=1; i<=5; i++) {
    (function(j){
        setTimeout( function timer(){
            console.log( j );
        }, j*1000 );
    })( i );
}
```

当然，因为这些 IIFE 只是函数，我们可以传入 `i`，如果我们乐意的话可以称它为 `j`，或者我们甚至可以再次称它为 `i`。不管哪种方式，这段代码都能工作。

在每次迭代内部使用的 IIFE 为每次迭代创建了新的作用域，这给了我们的超时回调函数一个机会，在每次迭代时闭包一个新的作用域，这些作用域中的每一个都拥有一个持有正确的迭代值的变量给我们访问。

问题解决了！

### 重温块儿作用域

仔细观察我们前一个解决方案的分析。我们使用了一个 IIFE 来在每一次迭代中创建新的作用域。换句话说，我们实际上每次迭代都 *需要*一个 **块儿作用域**。我们在第三章展示了 `let` 声明，它劫持一个块儿并且就在这个块儿中声明一个变量。

**这实质上将块儿变成了一个我们可以闭包的作用域**。所以接下来的牛逼代码“就是好用”：

```
for (var i=1; i<=5; i++) {
    let j = i; // 呀，给闭包的块儿作用域!
    setTimeout( function timer(){
        console.log( j );
    }, j*1000 );
}
```

*但是，这还不是全部!*（用我最棒的 Bob Barker 嗓音）在用于 for 循环头部的 `let` 声明被定义了一种特殊行为。这种行为说，这个变量将不是只为循环声明一次，**而是为每次迭代声明一次**。并且，它将在每次后续的迭代中被上一次迭代末尾的值初始化。

```
for (let i=1; i<=5; i++) {
    setTimeout( function timer(){
        console.log( i );
    }, i*1000 );
}
```

这有多酷？块儿作用域和闭包携手工作，解决世界上所有的问题。我不知道你怎么样，但这使我成了一个快乐的 JavaScript 开发者。

### 模块

还有其他的代码模式利用了闭包的力量，但是它们都不像回调那样浮于表面。让我们来检视它们中最强大的一种：*模块*。

```
function foo() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
```

```
        console.log( another.join( " ! " ) );
    }
}
```

就现在这段代码来说，没有发生明显的闭包。我们只是拥有一些私有数据变量 something 和 another，以及几个内部函数 doSomething() 和 doAnother()，它们都拥有覆盖在 foo() 内部作用域上的词法作用域（因此是闭包!）。

但是现在考虑这段代码：

```
function CoolModule() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }

    return {
        doSomething: doSomething,
        doAnother: doAnother
    };
}

var foo = CoolModule();

foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3
```

在 JavaScript 中我们称这种模式为 *模块*。实现模块模式的最常见方法经常被称为 “揭示模块”，它是我们在这里展示的方式的变种。

让我们检视关于这段代码的一些事情。

首先，CoolModule() 只是一个函数，但它 *必须被调用*才能成为一个被创建的模块实例。没有外部函数的执行，内部作用域的创建和闭包都不会发生。

第二，CoolModule() 函数返回一个对象，通过对象字面量语法 { key: value, ... } 标记。这个我们返回的对象拥有指向我们内部函数的引用，但是 *没有*指向我们内部数据变量的引

用。我们可以将它们保持为隐藏和私有的。可以很恰当地认为这个返回值对象实质上是一个 **我们模块的公有 API**。

这个返回值对象最终被赋值给外部变量 `foo`，然后我们可以在这个 API 上访问那些属性，比如 `foo.doSomething()`。

**注意**：从我们的模块中返回一个实际的对象（字面量）不是必须的。我们可以仅仅直接返回一个内部函数。jQuery 就是一个很好地例子。`jQuery` 和 `$` 标识符是 jQuery "模块" 的公有 API，但是它们本身只是一个函数（这个函数本身可以有属性，因为所有的函数都是对象）。

`doSomething()` 和 `doAnother()` 函数拥有模块 "实例" 内部作用域的闭包（通过实际调用 `CoolModule()` 得到的）。当我们通过返回值对象的属性引用，将这些函数传送到词法作用域外部时，我们就建立好了可以观察和行使闭包的条件。

更简单地说，行使模块模式有两个 "必要条件"：

1. 必须有一个外部的外围函数，而且它必须至少被调用一次（每次创建一个新的模块实例）。
2. 外围的函数必须至少返回一个内部函数，这样这个内部函数才拥有私有作用域的闭包，并且可以访问和/或修改这个私有状态。

仅一个带有函数属性的对象本身不是 *真正*的模块。从可观察的角度来说，一个从函数调用中返回的对象，仅带有数据属性而没有闭包的函数，也不是 *真正*的模块。

上面的代码段展示了一个称为 `CoolModule()` 独立的模块创建器，它可以被调用任意多次，每次创建一个新的模块实例。这种模式的一个稍稍的变化是当你只想要一个实例的时候，某种 "单例"：

```
var foo = (function CoolModule() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }

    return {
        doSomething: doSomething,
        doAnother: doAnother
    };
```

```
})();

foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3
```

这里，我们将模块放进一个 IIFE（见第三章）中，而且我们 立即调用它，并把它的返回值直接赋值给我们单独的模块实例标识符 foo。

模块只是函数，所以它们可以接收参数：

```
function CoolModule(id) {
    function identify() {
        console.log( id );
    }

    return {
        identify: identify
    };
}

var foo1 = CoolModule( "foo 1" );
var foo2 = CoolModule( "foo 2" );

foo1.identify(); // "foo 1"
foo2.identify(); // "foo 2"
```

另一种在模块模式上微小但是强大的变化是，为你作为公有 API 返回的对象命名：

```
var foo = (function CoolModule(id) {
    function change() {
        // 修改公有 API
        publicAPI.identify = identify2;
    }

    function identify1() {
        console.log( id );
    }

    function identify2() {
        console.log( id.toUpperCase() );
```

```
    }

    var publicAPI = {
        change: change,
        identify: identify1
    };

    return publicAPI;
})( "foo module" );

foo.identify(); // foo module
foo.change();
foo.identify(); // FOO MODULE
```

通过在模块实例内部持有一个指向公有 API 对象的内部引用，你可以 **从内部**修改这个模块，包括添加和删除方法，属性，和改变它们的值。

### 现代的模块

各种模块依赖加载器/消息机制实质上都是将这种模块定义包装进一个友好的 API。与其检视任意一个特定的库，不如让我 **（仅）为了说明的目的**展示一个 非常简单的概念证明：

```
var MyModules = (function Manager() {
    var modules = {};

    function define(name, deps, impl) {
        for (var i=0; i<deps.length; i++) {
            deps[i] = modules[deps[i]];
        }
        modules[name] = impl.apply( impl, deps );
    }

    function get(name) {
        return modules[name];
    }

    return {
        define: define,
        get: get
```

```
    };
})();
```

这段代码的关键部分是 `modules[name] = impl.apply(impl, deps)`。这为一个模块调用了它的定义的包装函数（传入所有依赖），并将返回值，也就是模块的 API，存储到一个用名称追踪的内部模块列表中。

这里是我可能如何使用它来定义一个模块：

```
MyModules.define( "bar", [], function(){
    function hello(who) {
        return "Let me introduce: " + who;
    }

    return {
        hello: hello
    };
} );

MyModules.define( "foo", ["bar"], function(bar){
    var hungry = "hippo";

    function awesome() {
        console.log( bar.hello( hungry ).toUpperCase() );
    }

    return {
        awesome: awesome
    };
} );

var bar = MyModules.get( "bar" );
var foo = MyModules.get( "foo" );

console.log(
    bar.hello( "hippo" )
); // Let me introduce: hippo

foo.awesome(); // LET ME INTRODUCE: HIPPO
```

模块 “foo” 和 “bar” 都使用一个返回公有 API 的函数来定义。“foo” 甚至接收一个 “bar” 的实例作为依赖参数，并且可以因此使用它。

花些时间检视这些代码段，来完全理解将闭包的力量付诸实践给我们带来的好处。关键之处在于，对于模块管理器来说真的没有什么特殊的 “魔法”。它们只是满足了我在上面列出的模块模式的两个性质：调用一个函数定义包装器，并将它的返回值作为这个模块的 API 保存下来。

换句话说，模块就是模块，即便你在它们上面放了一个友好的包装工具。

### 未来的模块

ES6 为模块的概念增加了头等的语法支持。当通过模块系统加载时，ES6 将一个文件视为一个独立的模块。每个模块可以导入其他的模块或者特定的 API 成员，也可以导出它们自己的公有 API 成员。

**注意：**基于函数的模块不是一个可以被静态识别的模式（编译器可以知道的东西），所以它们的 API 语义直到运行时才会被考虑。也就是，你实际上可以在运行时期间修改模块的 API（参见早先 `publicAPI` 的讨论）。

相比之下，ES6 模块 API 是静态的（这些 API 不会在运行时改变）。因为编译器知道它，它可以（也确实在这么作！）在（文件加载和）编译期间检查一个指向被导入模块的成员的引用是否 *实际存在*。如果 API 引用不存在，编译器就会在编译时抛出一个 “早期” 错误，而不是等待传统的动态运行时解决方案（和错误，如果有的话）。

ES6 模块 **没有** “内联” 格式，它们必须被定义在一个分离的文件中（每个模块一个）。浏览器/引擎拥有一个默认的 “模块加载器”（它是可以被覆盖的，但是这超出我们在此讨论的范围），它在模块被导入时同步地加载模块文件。

考虑这段代码：

**bar.js**

```
function hello(who) {
    return "Let me introduce: " + who;
}

export hello;
```

**foo.js**

```
// 仅导入 “bar” 模块中的 `hello()`
import hello from "bar";

var hungry = "hippo";

function awesome() {
    console.log(
```

```
        hello( hungry ).toUpperCase()
    );
}

export awesome;

// 导入 `foo`和 `bar`整个模块
module foo from "foo";
module bar from "bar";

console.log(
    bar.hello( "rhino" )
); // Let me introduce: rhino

foo.awesome(); // LET ME INTRODUCE: HIPPO
```

**注意**：需要使用前两个代码片段中的内容分别创建两个分离的文件 **“foo.js”** 和 **“bar.js”**。然后，你的程序将加载/导入这些模块来使用它们，就像第三个片段那样。

`import` 在当前的作用域中导入一个模块的 API 的一个或多个成员，每个都绑定到一个变量（这个例子中是 `hello`）。`module` 将整个模块的 API 导入到一个被绑定的变量（这个例子中是 `foo`，`bar`）。`export` 为当前模块的公有 API 导出一个标识符（变量，函数）。在一个模块的定义中，这些操作符可以根据需要使用任意多次。

在 *模块文件*内部的内容被视为像是包围在一个作用域闭包中，就像早先看到的使用函数闭包的模块那样。

## 复习

对于那些还蒙在鼓里的人来说，闭包就像在 JavaScript 内部被隔离开的魔法世界，只有很少一些最勇敢的灵魂才能到达。但是它实际上只是一个标准的，而且几乎明显的事实 —— 我们如何在函数即是值，而且可以被随意传递的词法作用域环境中编写代码，

**闭包就是当一个函数即使是在它的词法作用域之外被调用时，也可以记住并访问它的词法作用域。**

如果我们不能小心地识别它们和它们的工作方式，闭包可能会绊住我们，例如在循环中。但它们也是一种极其强大的工具，以各种形式开启了像 *模块*这样的模式。

模块要求两个关键性质：1）一个被调用的外部包装函数，来创建外围作用域。2）这个包装函数的返回值必须包含至少一个内部函数的引用，这个函数才拥有包装函数内部作用域的闭包。

现在我们看到了闭包在我们的代码中无处不在，而且我们有能力识别它们，并为了我们自己的利益利用它们!

# 附录 A：动态作用域

在第二章中，作为与 JavaScript 中（事实上，其他大多数语言也是）作用域的工作方式模型 —— “词法作用域” 的对比，我们谈到了 “动态作用域”。

我们将简单地检视动态作用域，来彻底说明这种比较。但更重要的是，对于 JavaScript 中的另一种机制（`this`）来说动态作用域实际上是它的一个近亲表兄，我们将在本系列的 “*this 与对象原型*” 中详细讲解这种机制。

正如我们在第二章中看到的，词法作用域是一组关于 *引擎* 如何查询变量和它在何处能够找到变量的规则。词法作用域的关键性质是，它是在代码编写时被定义的（假定你不使用 `eval()` 或 `with` 作弊的话）。

动态作用域看起来在暗示，有充分的理由，存在这样一种模型，它的作用域是在运行时被确定的，而不是在编写时静态地确定的。让我们通过代码来说明这样的实际情况：

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

function bar() {
    var a = 3;
    foo();
}

var a = 2;

bar();
```

在 `foo()` 的词法作用域中指向 `a` 的 RHS 引用将被解析为全局变量 `a`，它将导致输出结果为值 `2`。

相比之下，动态作用域本身不关心函数和作用域是在哪里和如何被声明的，而是关心 **它们是从何处被调用的**。换句话说，它的作用域链条是基于调用栈的，而不是代码中作用域的嵌套。

所以，如果 JavaScript 拥有动态作用域，当 `foo()` 被执行时，**理论上**下面的代码将得出 `3` 作为输出结果。

```
function foo() {
    console.log( a ); // 3 (不是 2!)
}

function bar() {
    var a = 3;
```

```
    foo();
}

var a = 2;

bar();
```

这怎么可能？因为当 `foo()` 不能为 `a` 解析出一个变量引用时，它不会沿着嵌套的（词法）作用域链向上走一层，而是沿着调用栈向上走，以找到 `foo()` 是 *从何处被调用的*。因为 `foo()` 是从 `bar()` 中被调用的，它就会在 `bar()` 的作用域中检查变量，并且在这里找到持有值 `3` 的 `a`。

奇怪吗？此时此刻你可能会这样认为。

但这可能只是因为你仅在拥有词法作用域的代码中工作过。所以动态作用域看起来陌生。如果你仅使用动态作用域的语言编写过代码，它看起来就是很自然的，而词法作用域将是个怪东西。

要清楚，JavaScript **实际上没有动态作用域**。它拥有词法作用域。就这么简单。但是 `this` 机制有些像动态作用域。

关键的差异：**词法作用域是编写时的，而动态作用域（和 `this`）是运行时的**。词法作用域关心的是 *函数在何处被声明*，但是动态作用域关心的是函数 *从何处被调用*。

最后：`this` 关心的是 *函数是如何被调用的*，这揭示了 `this` 机制与动态作用域的想法有多么紧密的关联。要了解更多关于 `this` 的细节，请阅读 “*this* 与对象原型”。

## 附录 B：填补块儿作用域

在第三章中，我们探索了块儿作用域。我们看到最早在 ES3 中引入的 `with` 和 `catch` 子句都是存在于 JavaScript 中的块儿作用域的小例子。

但是 ES6 引入的 `let` 最终使我们的代码有了完整的，不受约束的块作用域能力。不论是在功能上还是在代码风格上，块作用域都使许多激动人心的事情成为可能。

但要是我们想在前 ES6 环境中使用块儿作用域呢？

考虑这段代码：

```
{
    let a = 2;
    console.log( a ); // 2
}

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

它在 ES6 环境下工作的非常好。但是我们能在前 ES6 中这么做吗？`catch` 就是答案。

```
try{throw 2}catch(a){
    console.log( a ); // 2
}

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

哇！这真是难看和奇怪的代码。我们看到一个 `try/catch` 似乎强制抛出一个错误，但是这个它抛出的“错误”只是一个值 2。然后接收它的变量声明是在 `catch(a)` 子句中。三观：毁尽。

没错，`catch` 子句拥有块儿作用域，这意味着它可以被用于在前 ES6 环境中填补块儿作用域。

“但是……”，你说。“……没人愿意写这么丑的代码！”你是对的。也没人编写由 CoffeeScript 编译器输出的（某些）代码。这不是重点。

重点是工具可以将 ES6 代码转译为能够在前 ES6 环境中工作的代码。你可以使用块儿作用域编写代码，并从这样的功能中获益，然后让一个编译工具来掌管生成将在部署之后实际工作的代码。

这实际上是所有（嗯哼，大多数）ES6 特性首选的迁移路径：在从前 ES6 到 ES6 的转变过程中，使用一个代码转译器将 ES6 代码转换为 ES5 兼容的代码。

### Traceur

Google 维护着一个称为“Traceur”[7]的项目，它的任务正是为了广泛使用 ES6 特性而将它转译为前 ES6（大多数是 ES5，但不是全部！）代码。TC39 协会依赖这个工具（和其他的工具）来测试他们所规定的特性的语义。

Traceur 将从我们的代码段中产生出什么？你猜对了！

```
{
    try {
        throw undefined;
    } catch (a) {
        a = 2;
        console.log( a );
    }
}

console.log( a );
```

所以，使用这种工具，我们可以开始利用块儿作用域，无论我们是否面向 ES6，因为 `try/catch` 从 ES3 那时就开始存在了（并且这样工作）。

[7] Google Traceur

### 隐含的与明确的块儿

在第三章中，在我们介绍块儿作用域时，我们认识了一些关于代码可维护性/可重构性的潜在陷阱。有什么其他的方法可以利用块儿作用域同时减少这些负面影响吗?

考虑一下 `let` 的这种形式，它被称为 “let 块儿” 或 “let 语句”（和以前的 “let 声明” 对比来说）。

```
let (a = 2) {
    console.log( a ); // 2
}

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

与隐含地劫持一个既存的块儿不同，let 语句为它的作用域绑定明确地创建了一个块儿。这个明确的块儿不仅更显眼，而且在代码重构方面健壮得多，从文法上讲，它通过强制所有的声明都位于块儿的顶部而产生了某种程度上更干净的代码。这使任何块儿都更易于观察，更易于知道什么属于这个作用域和什么不属于这个作用域。

作为一种模式，它是与许多人在函数作用域中采纳的方式相对称的 —— 它们手动地将所有 `var` 声明移动/提升到函数的顶部。let 语句有意地将它们放在块儿的顶部，而且如果你没有通篇到处使用 `let` 声明，那么你的块儿作用域声明就会在某种程度上更易于识别和维护。

但是，这里有一个问题。let 语句的形式没有包含在 ES6 中。就连官方的 Traceur 编译器也不接受这种形式的代码。

我们有两个选择。我们可以使用 ES6 合法的语法格式化，再加上一点儿代码规则:

```
/*let*/ { let a = 2;
    console.log( a );
}

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

但是，工具就意味着要解决我们的问题。所以另一个选项是编写明确的 let 语句块儿，并让工具将他转换为合理的，可以工作的代码。

所以，我建造了一个称为 “let-er”[ˆnote-let_er] 的工具来解决这个问题。*let-er* 是一个编译期代码转译器，它唯一的任务就是找到 let 语句形式并转译它们。它将你的代码其他部分原封不动地留下，包括任何 let 声明。你可以安全地将 *let-er* 用于 ES6 转译器的第一步，然后如果有需要，你可以将你的代码通过 Traceur 这样的东西。

另外，*let-er* 有一个配置标志 `--es6`，当它打开时（默认是关闭），会改变生成的代码的种类。与使用 `try/catch` 的 ES3 填补黑科技不同的是，*let-er* 将拿着我们的代码并产生完全兼容 ES6 的代码，没有黑科技:

```
{
    let a = 2;
    console.log( a );
}

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

所以，你可以立即开始使用 *let-er*，而且可以面向所有前 ES6 环境，当你仅关心 ES6 时，你可以加入配置标志并立即仅面向 ES6。

而且最重要的是，**你可以使用更好的和更明确的 let 语句形式**，即便它（还）不是任何 ES 官方版本的一部分。

### 性能

让我在 `try/catch` 的性能问题上加入最后一个快速的说明，并/或解决这个问题："为什么不使用一个 IIFE 来创建作用域？"

首先，`try/catch` 的性能 *是*慢一些，但是没有任何合理的假设表明它 *必须*是这样，或者它 *总是*这样。因为 TC39 认可的官方 ES6 转译器使用 `try/catch`，Traceur 团队已经让 Chrome 去改进 `try/catch` 的性能了，而且它们有很明显的动力这样做。

第二，IIFE 和 `try/catch` 不是一个合理的 "苹果对苹果" 的比较，因为一个包装着任意代码的函数改变了这段代码的含义，以及它的 `this`、`return`、`break`、和 `continue` 的含义。IIFE 不是一个合适一般替代品。它只能在特定的情况下手动使用。

真正的问题变成了：你是否想要使用块儿作用域。如果是，这些工具给你提供了这些选择。如果不，那就在你的代码中继续使用 `var`！

[^note-let_er]: let-er

## 附录 C：词法 this

这本书通篇没有讲解 `this` 机制的任何细节，有一个 ES6 的话题以一种重要的方式将 `this` 与词法作用域联系了起来，我们将快速检视它一下。

ES6 为函数声明增加了一种特殊的语法形式，称为 "箭头函数"。它看起来像这样：

```
var foo = a => {
    console.log( a );
};

foo( 2 ); // 2
```

这个所谓的 "大箭头" 经常被称为是 *乏味烦冗的*（讽刺）`function` 关键字的缩写。

但是在箭头函数上发生的一些事情要重要得多，而且这与在你的声明中少敲几下键盘无关。

简单地说，这段代码有一个问题：

```
var obj = {
    id: "awesome",
    cool: function coolFn() {
        console.log( this.id );
    }
};

var id = "not awesome";

obj.cool(); // awesome

setTimeout( obj.cool, 100 ); // not awesome
```

这个问题就是在 cool() 函数上丢失了 this 绑定。有各种方法可以解决这个问题，但一个经常被重复的解决方案是 var self = this;。

它可能看起来像：

```
var obj = {
    count: 0,
    cool: function coolFn() {
        var self = this;

        if (self.count < 1) {
            setTimeout( function timer(){
                self.count++;
                console.log( "awesome?" );
            }, 100 );
        }
    }
};

obj.cool(); // awesome?
```

用不过于深入细节的方式讲，var self = this 的“解决方案”免除了理解和正确使用 this 绑定的整个问题，而是退回到我们也许感到更舒服的东西上面：词法作用域。self 变成了一个可以通过词法作用域和闭包解析的标识符，而且一直不关心 this 绑定发生了什么。

人们不喜欢写繁冗的东西，特别是当他们一次又一次重复它的时候。于是，ES6 的一个动机是帮助缓和这些场景，将常见的惯用法问题 固定下来，就像这一个。

ES6 的解决方案，箭头函数，引入了一种称为 “词法 this” 的行为。

```
var obj = {
    count: 0,
    cool: function coolFn() {
        if (this.count < 1) {
            setTimeout( () => { // 箭头函数能好用?
                this.count++;
                console.log( "awesome?" );
            }, 100 );
        }
    }
};

obj.cool(); // awesome?
```

简单的解释是，当箭头函数遇到它们的 this 绑定时，它们的行为与一般的函数根本不同。它们摒弃了 this 绑定的所有一般规则，而是采用它们的直接外围词法作用域的 this 的值，无论它是什么。

于是，在这个代码段中，箭头函数不会以不可预知的方式丢掉 this 绑定，它只是 “继承” cool() 函数的 this 绑定（如果像我们展示的那样调用它就是正确的!）。

虽然这使代码更短，但在我看来，箭头函数只不过是将一个开发者们常犯的错误固化成了语言的语法，这混淆了 “this 绑定” 规则与 “词法作用域” 规则。

换一种说法：为什么要使用 this 风格的编码形式来招惹麻烦和繁冗? 只要通过将它与词法作用域混合把它剔除掉就好。对于给定的一段代码只采纳一种方式或另一种看起来才是自然的，而不是在同一段代码中将它们混在一起。

**注意**：源自箭头函数的另一个非议是，它们是匿名的，不是命名的。参见第三章来了解为什么匿名函数不如命名函数理想的原因。

在我看来，这个 “问题” 的更恰当的解决方式是，正确地使用并接受 this 机制。

```
var obj = {
    count: 0,
    cool: function coolFn() {
        if (this.count < 1) {
            setTimeout( function timer(){
                this.count++; // `this` 因为 `bind(..)` 所以安全
                console.log( "more awesome" );
```

```
            }.bind( this ), 100 ); // 看, `bind()`!
        }
    }
};

obj.cool(); // more awesome
```

不管你是偏好箭头函数的新的词法 this 行为，还是偏好经得起考验的 `bind()`，重要的是要注意箭头函数 **不**仅仅是关于可以少打一些 “function”。

它们拥有一种我们应当学习并理解的，*有意的行为上的不同*，而且如果我们这样选择，就可以利用它们。

现在我们完全理解了词法作用域（和闭包!），理解词法 this 应该是小菜一碟!

## 附录 D: 鸣谢

为了这本书和整个系列的诞生，我有很多人要感谢。

首先，我必须感谢我的妻子 Christen Simpson，和我的两个孩子 Ethan 和 Emily，忍受着老爹总是在电脑上敲打。即使在没有写书时，我对 JavaScript 的痴迷也将我的眼睛粘在屏幕上太久了。我从家庭那里借来的时间是这些书可以如此深入和完整地向你，读者，解释 JavaScript 的原因。我欠我的家庭一切。

我要感谢我在 O’Reilly 的编辑，他们是 Simon St.Laurent 和 Brian MacDonald，还有其他的编辑和市场员工。和他们一起工作很棒，而且在这种 “开源” 写作，编辑，和生产的实验期间提供了特别的通融。

感谢许多通过提供编辑意见和订正来参与使这部丛书变得更好的朋友们，他们是 Shelley Powers，Tim Ferro，Evan Borden，Forrest L. Norvell，Jennifer Davis，Jesse Harlin，Kris Kowal，Rick Waldron，Jordan Harband，Benjamin Gruenbaum，Vyacheslav Egorov，David Nolen，和许多其他人。一个巨大感谢送给为本书作序的 Shane Hudson。

感谢社区中无数的朋友们，包括 TC39 协会的成员，他们和我们分享了那么多的知识，特别是以耐心和细节容忍我无休止的问题和探究。John-David Dalton，Juriy “kangax” Zaytsev，Mathias Bynens，Axel Rauschmayer，Nicholas Zakas，Angus Croll，Reginald Braithwaite，Dave Herman，Brendan Eich，Allen Wirfs-Brock，Bradley Meck，Domenic Denicola，David Walsh，Tim Disney，Peter van der Zee，Andrea Giammarchi，Kit Cambridge，Eric Elliott，和其他许多我甚至不能接触到的人。

*你不懂 JS* 系列丛书诞生于 Kickstarter，所以我也要感谢我的所有（将近）500 位慷慨的支持者，没有他们这部丛书不可能诞生：

> Jan Szpila, nokiko, Murali Krishnamoorthy, Ryan Joy, Craig Patchett, pdqtrader, Dale Fukami, ray hatfield, R0drigo Perez [Mx], Dan Petitt, Jack Franklin, Andrew Berry, Brian Grinstead, Rob Sutherland, Sergi Meseguer, Phillip Gourley, Mark

Watson, Jeff Carouth, Alfredo Sumaran, Martin Sachse, Marcio Barrios, Dan, AimelyneM, Matt Sullivan, Delnatte Pierre-Antoine, Jake Smith, Eugen Tudorancea, Iris, David Trinh, simonstl, Ray Daly, Uros Gruber, Justin Myers, Shai Zonis, Mom & Dad, Devin Clark, Dennis Palmer, Brian Panahi Johnson, Josh Marshall, Marshall, Dennis Kerr, Matt Steele, Erik Slagter, Sacah, Justin Rainbow, Christian Nilsson, Delapouite, D.Pereira, Nicolas Hoizey, George V. Reilly, Dan Reeves, Bruno Laturner, Chad Jennings, Shane King, Jeremiah Lee Cohick, od3n, Stan Yamane, Marko Vucinic, Jim B, Stephen Collins, Ægir Þorsteinsson, Eric Pederson, Owain, Nathan Smith, Jeanetteurphy, Alexandre ELISÉ, Chris Peterson, Rik Watson, Luke Matthews, Justin Lowery, Morten Nielsen, Vernon Kesner, Chetan Shenoy, Paul Tregoing, Marc Grabanski, Dion Almaer, Andrew Sullivan, Keith Elsass, Tom Burke, Brian Ashenfelter, David Stuart, Karl Swedberg, Graeme, Brandon Hays, John Christopher, Gior, manoj reddy, Chad Smith, Jared Harbour, Minoru TODA, Chris Wigley, Daniel Mee, Mike, Handyface, Alex Jahraus, Carl Furrow, Rob Foulkrod, Max Shishkin, Leigh Penny Jr., Robert Ferguson, Mike van Hoenselaar, Hasse Schougaard, rajan venkataguru, Jeff Adams, Trae Robbins, Rolf Langenhuijzen, Jorge Antunes, Alex Koloskov, Hugh Greenish, Tim Jones, Jose Ochoa, Michael Brennan-White, Naga Harish Muvva, Barkóczi Dávid, Kitt Hodsden, Paul McGraw, Sascha Goldhofer, Andrew Metcalf, Markus Krogh, Michael Mathews, Matt Jared, Juanfran, Georgie Kirschner, Kenny Lee, Ted Zhang, Amit Pahwa, Inbal Sinai, Dan Raine, Schabse Laks, Michael Tervoort, Alexandre Abreu, Alan Joseph Williams, NicolasD, Cindy Wong, Reg Braithwaite, LocalPCGuy, Jon Friskics, Chris Merriman, John Pena, Jacob Katz, Sue Lockwood, Magnus Johansson, Jeremy Crapsey, Grzegorz Pawłowski, nico nuzzaci, Christine Wilks, Hans Bergren, charles montgomery, Ariel - Fogel, Ivan Kolev, Daniel Campos, Hugh Wood, Christian Bradford, Frédéric Harper, Ionuţ Dan Popa, Jeff Trimble, Rupert Wood, Trey Carrico, Pancho Lopez, Joël kuijten, Tom A Marra, Jeff Jewiss, Jacob Rios, Paolo Di Stefano, Soledad Penades, Chris Gerber, Andrey Dolganov, Wil Moore III, Thomas Martineau, Kareem, Ben Thouret, Udi Nir, Morgan Laupies, jory carson-burson, Nathan L Smith, Eric Damon Walters, Derry Lozano-Hoyland, Geoffrey Wiseman, mkeehner, KatieK, Scott MacFarlane, Brian LaShomb, Adrien Mas, christopher ross, Ian Littman, Dan Atkinson, Elliot Jobe, Nick Dozier, Peter Wooley, John Hoover, dan, Martin A. Jackson, Héctor Fernando Hurtado, andy ennamorato, Paul Seltmann, Melissa Gore, Dave Pollard, Jack Smith, Philip Da Silva, Guy Israeli, @megalithic, Damian Crawford, Felix Gliesche, April Carter Grant, Heidi, jim tierney, Andrea Giammarchi, Nico Vignola, Don Jones, Chris Hartjes, Alex Howes, john gibbon, David J. Groom, BBox, Yu 'Dilys' Sun, Nate Steiner,

Brandon Satrom, Brian Wyant, Wesley Hales, Ian Pouncey, Timothy Kevin Oxley, George Terezakis, sanjay raj, Jordan Harband, Marko McLion, Wolfgang Kaufmann, Pascal Peuckert, Dave Nugent, Markus Liebelt, Welling Guzman, Nick Cooley, Daniel Mesquita, Robert Syvarth, Chris Coyier, Rémy Bach, Adam Dougal, Alistair Duggin, David Loidolt, Ed Richer, Brian Chenault, GoldFire Studios, Carles Andrés, Carlos Cabo, Yuya Saito, roberto ricardo, Barnett Klane, Mike Moore, Kevin Marx, Justin Love, Joe Taylor, Paul Dijou, Michael Kohler, Rob Cassie, Mike Tierney, Cody Leroy Lindley, tofuji, Shimon Schwartz, Raymond, Luc De Brouwer, David Hayes, Rhys Brett-Bowen, Dmitry, Aziz Khoury, Dean, Scott Tolinski - Level Up, Clement Boirie, Djordje Lukic, Anton Kotenko, Rafael Corral, Philip Hurwitz, Jonathan Pidgeon, Jason Campbell, Joseph C., SwiftOne, Jan Hohner, Derick Bailey, getify, Daniel Cousineau, Chris Charlton, Eric Turner, David Turner, Joël Galeran, Dharma Vagabond, adam, Dirk van Bergen, dave furf, Vedran Zakanj, Ryan McAllen, Natalie Patrice Tucker, Eric J. Bivona, Adam Spooner, Aaron Cavano, Kelly Packer, Eric J, Martin Drenovac, Emilis, Michael Pelikan, Scott F. Walter, Josh Freeman, Brandon Hudgeons, vijay chennupati, Bill Glennon, Robin R., Troy Forster, otaku_coder, Brad, Scott, Frederick Ostrander, Adam Brill, Seb Flippence, Michael Anderson, Jacob, Adam Randlett, Standard, Joshua Clanton, Sebastian Kouba, Chris Deck, SwordFire, Hannes Papenberg, Richard Woeber, hnzz, Rob Crowther, Jedidiah Broadbent, Sergey Chernyshev, Jay-Ar Jamon, Ben Combee, luciano bonachela, Mark Tomlinson, Kit Cambridge, Michael Melgares, Jacob Adams, Adrian Bruinhout, Bev Wieber, Scott Puleo, Thomas Herzog, April Leone, Daniel Mizieliński, Kees van Ginkel, Jon Abrams, Erwin Heiser, Avi Laviad, David newell, Jean-Francois Turcot, Niko Roberts, Erik Dana, Charles Neill, Aaron Holmes, Grzegorz Ziółkowski, Nathan Youngman, Timothy, Jacob Mather, Michael Allan, Mohit Seth, Ryan Ewing, Benjamin Van Treese, Marcelo Santos, Denis Wolf, Phil Keys, Chris Yung, Timo Tijhof, Martin Lekvall, Agendine, Greg Whitworth, Helen Humphrey, Dougal Campbell, Johannes Harth, Bruno Girin, Brian Hough, Darren Newton, Craig McPheat, Olivier Tille, Dennis Roethig, Mathias Bynens, Brendan Stromberger, sundeep, John Meyer, Ron Male, John F Croston III, gigante, Carl Bergenhem, B.J. May, Rebekah Tyler, Ted Foxberry, Jordan Reese, Terry Suitor, afeliz, Tom Kiefer, Darragh Duffy, Kevin Vanderbeken, Andy Pearson, Simon Mac Donald, Abid Din, Chris Joel, Tomas Theunissen, David Dick, Paul Grock, Brandon Wood, John Weis, dgrebb, Nick Jenkins, Chuck Lane, Johnny Megahan, marzsman, Tatu Tamminen, Geoffrey Knauth, Alexander Tarmolov, Jeremy Tymes, Chad Auld, Sean Parmelee, Rob Staenke, Dan Bender, Yannick derwa, Joshua Jones, Geert Plaisier, Tom LeZotte, Christen Simpson, Stefan Bruvik, Justin Falcone, Car-

los Santana, Michael Weiss, Pablo Villoslada, Peter deHaan, Dimitris Iliopoulos, seyDoggy, Adam Jordens, Noah Kantrowitz, Amol M, Matthew Winnard, Dirk Ginader, Phinam Bui, David Rapson, Andrew Baxter, Florian Bougel, Michael George, Alban Escalier, Daniel Sellers, Sasha Rudan, John Green, Robert Kowalski, David I. Teixeira (@ditma), Charles Carpenter, Justin Yost, Sam S, Denis Ciccale, Kevin Sheurs, Yannick Croissant, Pau Fracés, Stephen McGowan, Shawn Searcy, Chris Ruppel, Kevin Lamping, Jessica Campbell, Christopher Schmitt, Sablons, Jonathan Reisdorf, Bunni Gek, Teddy Huff, Michael Mullany, Michael Fürstenberg, Carl Henderson, Rick Yoesting, Scott Nichols, Hernán Ciudad, Andrew Maier, Mike Stapp, Jesse Shawl, Sérgio Lopes, jsulak, Shawn Price, Joel Clermont, Chris Ridmann, Sean Timm, Jason Finch, Aiden Montgomery, Elijah Manor, Derek Gathright, Jesse Harlin, Dillon Curry, Courtney Myers, Diego Cadenas, Arne de Bree, João Paulo Dubas, James Taylor, Philipp Kraeutli, Mihai Păun, Sam Gharegozlou, joshjs, Matt Murchison, Eric Windham, Timo Behrmann, Andrew Hall, joshua price, Théophile Villard

这部丛书是以开源的风格书写的，包括编辑和生产。我们感激 GitHub 使这样的事情在社区中成为可能!

再次感谢所有无数的朋友，尽管我不能叫上名字但是我依然亏欠感谢。但愿这部丛书被我们所有人 “拥有”，并为增进对 JavaScript 语言的意识和理解做出贡献，成为当下和未来所有社区贡献者的助益。