你并不了解 JavaScript：作用域与闭包 - 第二版

> 版本：v1.1.4 构建时间：2023/12/4 14:47:58 Asia/Shanghai
> 最新版地址：https://github.com/liunnn1994/You-Dont-Know-JS-zh-CN/releases/latest



从 Leanpub 上购买电子书/PDF

目录

- 前言（由 Sarah Drasner 撰写）
- 序
- 第一章：什么是作用域？
- 第二章：图解作用域词法
- 第三章：作用域链
- 第四章：全局作用域
- 第五章：（并不）神秘的变量生命周期
- 第六章：限制作用域的过度暴露
- 第七章：闭包的使用
- 第八章：模块化模式
- 附录 A：欲穷千里目，更上一层楼

- 附录 B：练习

目录

- 前言
- 序
- 第一章：什么是作用域？
  - 关于本书
  - 编译型与解释型
  - 编译代码
  - 编译器说
  - 取巧：修改运行时的作用域
  - 词法作用域
- 第二章：图解作用域词法
  - 弹珠、水桶和气泡……哦，我的老伙计！
  - 朋友之间的对话
  - 作用域嵌套
  - 未完待续
- 第三章：作用域链
  - 「查找」（大部分）是概念性的
  - 遮蔽（Shadowing）
  - 函数名的作用域
  - 箭头函数
  - 思维转换
- 第四章：全局作用域
  - 为什么是全局作用域？
  - 全局作用域具体范围是什么？
  - Global This
  - 掌握全局作用域
- 第五章：（并不）神秘的变量生命周期
  - 何时使用变量？
  - 变量提升：另一种隐喻
  - 重复声明？
  - 未初始化变量警告（又称 TDZ 暂时性死区）

- 终于初始化
- 第六章：限制作用域的过度暴露
  - 最少暴露
  - 隐藏在普通（函数）作用域内
  - 块的作用域
  - 块中的声明函数
  - 刀过竹解
- 第七章：闭包的使用
  - 来看看闭包
  - 闭包生命周期和垃圾回收（GC）
  - 为什么是闭包？
  - 一种替代观点
  - 完美谢幕
- 第八章：模块化模式
  - 封装和最小暴露（POLE）
  - 什么是模块？
  - Node CommonJS 模块
  - 现代 ES 模块（ESM）
  - 周天圆满
- 附录 A：欲穷千里目，更上一层楼
  - 隐式作用域
  - 匿名函数与具名函数
  - 提升：函数和变量
  - 使用 `var` 的理由
  - TDZ 是怎么回事？
  - 同步回调还是闭包吗？
  - 类模块的变体
- 附录 B：练习
  - 弹珠桶
  - 闭包（第一部分）
  - 闭包（第二部分）
  - 闭包（第三部分）
  - 模块
  - 参考答案

第一章：什么是作用域？

当你编写了最初的几个程序后，你可能会对创建变量并在其中存储值感到有些得心应手了。使用变量是我们编程中最基础的工作之一！

但你可能没有仔细考虑过引擎用来组织和管理这些变量的底层机制。我指的不是内存在计算机上是如何分配的，而是 JS 是如何知道任何给定语句可以访问哪些变量的，以及如何处理两个同名变量？

类似这些问题的答案都以明确的规则定义形式出现，这些规则被称为作用域。本书将深入探讨作用域的方方面面：它是如何工作的、有什么用处、要避免哪些麻烦，然后列出程序结构的常见作用域模式。

我们的第一步是揭示 JS 引擎如何在程序运行**之前**处理我们的程序的。

关于本书

欢迎阅读《你并不了解 JavaScript》系列第二部！如果你已经读完了*入门*（第一册），那你就来对地方了！如果没有，在继续阅读之前，我建议您*从入门开始*，以便打下最好的基础。

我们将重点关注 JS 语言三大支柱中的第一支柱：作用域系统及其函数闭包，以及模块设计模式的强大功能。

JS 通常被归类为解释型脚本语言，因此大多数人都认为 JS 程序是自上而下一次性处理的。但事实上，JS 是在**开始执行**前的一个单独阶段进行解析/编译的。在初始解析/编译阶段，代码编写者会根据作用域规则分析变量、函数和代码块之间的位置关系。由此产生的作用域结构一般不受运行时条件的影响。

JS 函数本身就是头等公民；它们可以像数字或字符串一样被赋值和传递。并且，由于这些函数持有和访问变量，因此无论函数最终在程序中的哪个位置执行，它们都能保持原有的作用域。这就是所谓的闭包。

模块是一种代码组织模式，其特点是公共方法具有访问模块内部隐藏变量和函数的权限（通过闭包）。

编译型与解释型

您可能听说过*代码编译*，但它就像一个神秘的黑盒子，源代码从一端滑入，可执行程序从另一端弹出。

不过，这并非神秘莫测。代码编译是处理代码文本并将其转化为计算机可以理解的指令列表的一系列步骤。通常情况下，整个源代码会被一次性转换，而这些转换后的指令会被保存为输出（通常是文件），然后就可以被执行。

您可能还听说过代码可以*解释*，那么这与*编译*有什么区别呢？

解释执行的任务与编译类似，都是将程序转化为机器可理解的指令。但处理模式不同。与一次性编译程序不同，解释程序是逐行转换源代码；每一行或每一条语句都要先执行，然后才立即处理下一行源代码。

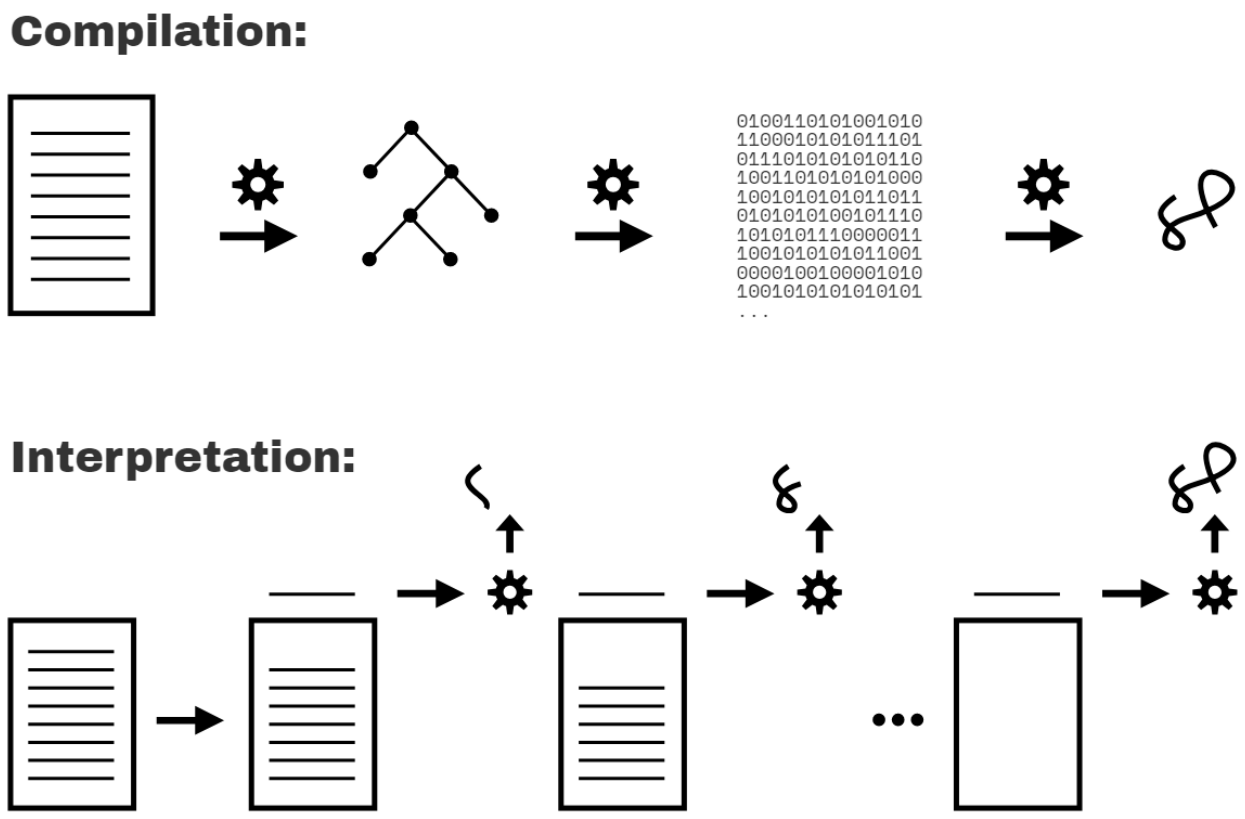

图 1：编译代码与解释代码

图 1 说明了程序的编译与解释。

这两种处理模式相互排斥吗？一般来说，是的。不过，这个问题更加微妙的是解释代码实际上可以采取其他形式，而不仅仅是逐行操作源代码文本。现代 JS 引擎在处理 JS 程序时，实际上采用了编译和解释的多种变体。

记得我们在《入门》一书的第 1 章中讨论过这个话题。我们的结论是，JS 最准确的描述是一种**编译语言**。为了读者的利益，以下章节将重新讨论并扩展这一论断。

编译代码

但首先，JS 是否编译又有什么关系呢？

作用域主要是在编译过程中确定的，因此了解编译和执行之间的关系是掌握作用域的关键。

根据经典的编译器理论，编译器对程序的处理分为三个基本阶段：

1. **标记化/词法：** 将字符串分解成有意义的（对语言而言）块，称为标记。例如，请看下面的程序 `var a = 2;`。这个程序可能会被分解成以下标记：`var`、`a`、`=`、`2`和`;`。空白可以作为一个标记，也可以不作为一个标记，这取决于它是否有意义。

   (标记化和词法之间的区别是微妙的、学术性的，但其核心在于这些标记是以*无状态*还是*有状态*的方式识别的。简单地说，如果标记化器调用有状态的解析规则来确定 `a` 应该被视为一个独立的标记，还是只是另一个标记的一部分，那就是**词法**）。

2. **解析：** 获取标记流（数组）并将其转化为嵌套元素树，这些嵌套元素共同代表了程序的语法结构。这就是所谓的抽象语法树（AST)。

   例如 `var a = 2;` 的语法树可能以一个名为 `VariableDeclaration` 的顶层节点开始，它有一个名为 `Identifier` 的子节点（其值为 `a`）和另一个名为 `AssignmentExpression` 的子节点（其本身有一个名为 `NumericLiteral` 的子节点（其值为 `2`）。

3. **代码生成：** 提取 AST 并将其转化为可执行代码。这部分因语言、目标平台和其他因素的不同而有很大差异。

   JS 引擎将刚才描述的 AST 中的 `var a = 2;` 转化为一组机器指令，以实际*创建*一个名为 `a` 的变量（包括预留内存等），然后将一个值存储到 `a` 中。

> **注意：**
>
> JS 引擎的实现细节（利用系统内存资源等）远比我们在此讨论的要深入得多。我们将继续关注程序的可观察行为，让 JS 引擎来管理这些更深层次的系统级抽象。

JS 引擎*远比*这三个阶段复杂得多。在解析和代码生成的过程中，还有一些优化执行性能的步骤（即折叠冗余元素）。事实上，代码甚至可以在执行过程中重新编译和优化。

所以，我在这里只是粗略地描述一下。但你很快就会明白，为什么我们涉及的*这些细节*，即使是高层次的，也是相关的。

JS 引擎没有充裕的时间来进行工作和优化，因为 JS 编译不像其他语言那样在构建步骤中提前进行。编译通常必须在代码执行前的几微秒（或更短时间）内完成。为了确保在这些限制条件下实现最快的性能，JS 引擎使用了各种技巧（如 JIT、懒编译甚至热重新编译）；这些远远超出了我们在此讨论的「作用域」。

必经的两个阶段

尽可能简单地说，我们对 JS 程序处理的最重要看法是，它（至少）分为两个阶段：首先是解析/编译，然后是执行。

解析/编译阶段与后续执行阶段的分离是可以观察到的事实，而不是理论或观点。虽然 JS 规范并没有明确要求「编译」，但它所要求的行为基本上只有编译-执行方法才切实可行。

您可以通过观察以下三个程序特征来证明这一点：语法错误、早期错误和变量提升。

从一开始就出现语法错误

思考一下这个程序：

```js
var greeting = "Hello";

console.log(greeting);

greeting = ."Hi";
// SyntaxError: unexpected token .
```

该程序不会有任何输出（`"Hello"` 不会被打印），而是会抛出一个 `SyntaxError`，原因是在 `"Hi"` 字符串之前出现了一个意外的 `.` 标记。你可能认为语法错误发生在语法没问题的 `console.log(..)` 语句之后，如果 JS 是自上而下逐行执行的，那么在抛出语法错误之前就会打印出 `"Hello"` 消息。但事实并非如此。

事实上，在执行第一行和第二行之前，JS 引擎要想知道第三行的语法错误，唯一的办法就是在执行任何程序之前先解析整个程序。

早期错误

下一个思考：

```js
console.log("Howdy");

saySomething("Hello", "Hi");
// Uncaught SyntaxError: Duplicate parameter name not
// allowed in this context

function saySomething(greeting, greeting) {
    "use strict";
    console.log(greeting);
}
```

尽管 `"Howdy"` 信息是一个格式良好的语句，但却没有打印出来。

相反，就像上一节中的代码段一样，这里的 `SyntaxError` 会在程序执行前被抛出。在这种情况下，这是因为严格模式（此处只特指 `saySomething(..)` 函数）禁止函数有重复的参数名，而在非严格模式下，这一直是允许的。

抛出的错误并不是语法错误，也不是畸形的标记串（如之前的 `. "Hi"`），但在严格模式下，规范要求在开始执行前作为「早期错误」抛出。

但是，JS 引擎如何知道 `greeting` 参数被重复使用了呢？在处理参数列表时，它如何知道 `saySomething(..)` 函数处于严格模式的呢（ `"use strict"` 只出现在后面的函数里）？

再次说明，唯一合理的解释是，在执行任何代码之前，必须先对代码进行*完整*的解析。

变量提升

最终思考：

```js
function saySomething() {
    var greeting = "Hello";
    {
        greeting = "Howdy"; // 错误来自这里
        let greeting = "Hi";
        console.log(greeting);
    }
}

saySomething();
// ReferenceError: Cannot access 'greeting' before
// initialization
```

注意到的 `ReferenceError` 出现在语句 `greeting = "Howdy"` 的那一行。发生这种情况的原因是，该语句的 `greeting` 变量属于下一行的声明 `let greeting = "Hi"`，而不是前一条语句 `var greeting = "Hello"`。

JS 引擎要想知道*下一条语句*将声明一个同名的块作用域变量（`greeting`）会出错，唯一的办法就是 JS 引擎已经在之前处理过这段代码，并设置了所有作用域及其变量关联。只有在执行前解析程序，才能准确地完成作用域和声明的处理。

这里的 `ReferenceError` 从技术上讲是由于 `greeting = "Howdy"` **过早**访问了 `greeting` 变量，这种冲突被称为暂时性死区（TDZ）。第 5 章将对此进行详细介绍。

> **警告：**
>
> 人们经常断言，`let` 和 `const` 声明不会导致变量提升，这是对 TDZ 行为的一种解释。但这种说法并不准确。我们将在第 5 章中解释 `let` / `const` 的变量提升和 TDZ。

希望你现在已经确信，JS 程序在开始执行之前是经过解析的。但这能证明它们是经过编译的吗？

这是一个值得思考的有趣问题。JS 能否解析一个程序，然后通过**解释** AST 中的操作来执行该程序，而**不需要**首先编译该程序？是的，这是*可行的*。但可能性极小，主要是因为这样做的性能效率极低。

很难想象一个高质量的 JS 引擎会费尽心思将程序解析成 AST，但却不将 AST 转换（又称「编译」）成最有效的（二进制）表示形式供引擎执行。

许多人都在努力对这一术语进行区分，因为其中有许多细微差别和「实际上……」的插话。但无论在精神上还是在实践中，引擎在处理 JS 程序时所做的工作**更像是编译**但非编译。

将 JS 归类为编译语言并不涉及其二进制（或字节码）可执行表示的分发模式，而是要在我们的头脑中保持对 JS 代码处理和分析阶段的明确区分；这一阶段在代码开始执行*之前*就已发生，这是可以观察到的，也是无可争议的。

如果我们想有效地理解 JS 和作用域，就需要对 JS 引擎如何处理我们的代码建立正确的心智模型。

编译器说

了解了 JS 程序的两阶段处理过程（编译，然后执行）后，我们再来看看 JS 引擎在编译程序时是如何识别变量和确定程序作用域的。

首先，让我们研究一个简单的 JS 程序，用于接下来几章的分析：

```js
var students = [
    { id: 14, name: "Kyle" },
    { id: 73, name: "Suzy" },
    { id: 112, name: "Frank" },
    { id: 6, name: "Sarah" },
];

function getStudentName(studentID) {
    for (let student of students) {
        if (student.id == studentID) {
            return student.name;
```

```
        }
    }
}

var nextStudent = getStudentName(73);

console.log(nextStudent);
// Suzy
```

除声明外，程序中出现的所有变量/标识符都有两种「角色」：要么是赋值的*目标*，要么是值的*来源*。

(我在攻读计算机科学学位时第一次学习编译器理论，当时我们学习的术语 "LHS"（又称*目标*）和 "RHS"（又称*源*）分别表示这些角色。正如你可能从 "L" 和 "R" 中猜到的，这两个缩写词的意思是左手边（"Left-Hand Side"）和右手边（"Right-Hand Side"），就像 `=` 赋值运算符的左侧和右侧。不过，赋值目标和赋值源并不总是出现在 `=` 的左侧或右侧，所以用*目标/源*而不是*左/右*来表示可能更清楚）。

如何知道一个变量是否是*目标*？检查是否有一个值正在分配给它；如果有，它就是*目标*。如果没有，那么该变量就是*源*。

为了让 JS 引擎正确处理程序中的变量，它必须首先将每个变量标记为*目标*或*源*。我们现在就来深入探讨如何确定每种角色。

目标

是什么让变量成为*目标*？思考一下：

```js
students = [ // ..
```

这条语句显然是一个赋值操作；请记住，`var students` 部分在编译时完全是作为声明处理的，因此在执行时无关紧要；为了清晰和突出重点，我们把它省略了。`nextStudent = getStudentName(73)` 语句也是如此。

不过，代码中还有三个*目标*赋值操作可能不太明显。其中之一是：

```js
for (let student of students) {
```

该语句会在循环的每次迭代中为 `student` 赋值。另一个*目标*引用：

```js
getStudentName(73);
```

但这怎么会是对*目标*的赋值呢？仔细看：参数 `73` 被赋值给了参数 `studentID`.

在我们的程序中，还有最后一个（微妙的）*目标*参考。你能发现吗？

..

..

..

你认出这个了吗？

```js
function getStudentName(studentID) {
```

一个 `function` 的声明是*目标*引用的一种特殊情况。你可以把它想象成 `var getStudentName = function(studentID)`，但这并不完全准确。一个标识符 `getStudentName` 是在编译时声明的，但 `= function(studentID)` 部分也是在编译时处理的；`getStudentName` 和函数之间的关联是在作用域开始时自动建立的，而不是等待 `=` 赋值语句被执行。

> **注意：**
>
> 这种函数与变量的自动关联被称为「函数提升」，将在第 5 章中详细介绍。

源

这样，我们就确定了程序中所有五个*目标*引用。其他变量引用必须是*源*引用（因为这是唯一的选择！）。

在 `for (let student of students)` 中，我们说 `student` 是*目标*，而 `students` 是*源*引用。在语句 `if (student.id == studentID)` 中，`student` 和 `studentID` 都是*源*引用。在 `return student.name` 中，`student` 也是*源*引用。

在 `getStudentName(73)` 中，`getStudentName` 是*源*引用（我们希望它能解析为函数引用值）。在 `console.log(nextStudent)` 中，`console` 和 `nextStudent` 都是*源*引用。

> **注意：**
>
> 如果你想知道的话 `id`、`name` 和 `log` 都是属性，而不是变量引用。

了解*目标与源*有什么实际意义呢？在第 2 章中，我们将再次讨论这一主题，并介绍变量的角色如何影响其查找（特别是在查找失败的情况下）。

取巧：修改运行时的作用域

现在应该清楚了，作用域是在程序编译时确定的，一般不会受运行时条件的影响。不过，在非严格模式下，技术上仍有两种方法可以欺骗这一规则，即在运行时修改程序的作用域。

这两种技术都*不应该*使用 — 它们既危险又容易混淆，无论如何你都应该使用严格模式（不允许使用它们）。不过，如果你在某些程序中遇到了它们，还是有必要了解一下。

`eval(..)` 函数接收一串代码，并在程序运行时编译和执行。如果该代码串中有 `var` 或 `function` 声明，这些声明将修改 `eval(..)` 当前执行的作用域：

```js
function badIdea() {
    eval("var oops = 'Ugh!';");
    console.log(oops);
}
badIdea(); // Ugh!
```

如果没有 `eval(..)`，`console.log(oops)` 中的 `oops` 变量将不存在，并将抛出一个 `ReferenceError`。但是，`eval(..)` 在运行时修改了 `badIdea()` 函数的作用域。这是不好的，原因有很多，包括每次运行 `badIdea()` 时修改已编译和优化的作用域对性能的影响。

第二个欺骗手段是 `with` 关键字，它基本上是动态地将一个对象变成一个本地作用域 — 它的属性在这个新作用域的块中被视为标识符：

```js
var badIdea = { oops: "Ugh!" };

with (badIdea) {
    console.log(oops); // Ugh!
}
```

全局作用域在这里没有被修改，但 `badIdea` 在运行时而不是在编译时变成了一个作用域，它的属性 `oops` 变成了该作用域中的一个变量。出于性能和可读性的考虑，这同样是一个糟糕的想法。

无论如何，都要避免使用 `eval(..)`（至少要避免使用 `eval(..)` 创建声明）和 `with`。同样，严格模式下也不能使用这两种欺骗手段，所以如果你只使用严格模式（你应该这样做！），那么诱惑就会消失！

词法作用域

我们已经证明 JS 的作用域是在编译时确定的；这种作用域的术语是「词法作用域」。正如本章前面所讨论的，「词法」与编译的「词法」阶段有关。

本章的主要观点是，「词法作用域」完全由函数、代码块和变量声明之间的位置关系来控制。

如果将变量声明放在函数中，编译器会在解析函数时处理该声明，并将该声明与函数的作用域相关联。如果变量是以块作用域声明的（`let` / `const`），那么它会与最近的外层 `{ .. }` 块相关联，而不是与其外层函数相关联（与 `var` 相同）。

此外，一个变量的引用（*目标*或*源*角色）必须被解析为来自于对它来说*词法可用*的作用域之一；否则，该变量将被称为「未声明」（通常会导致错误！）。如果变量不是在当前作用域中声明的，那么将查询下一个外层/封闭作用域。这种一步步向外递增的过程会一直持续下去，直到找到匹配的变量声明，或者达到全局作用域而无处可去为止。

需要注意的是，在为作用域和变量保留内存方面，编译实际上并没有*做任何事情*。没有程序在执行。

相反，编译会创建一个所有词法作用域的映射，列出程序在执行时需要的内容。你可以将该计划视为运行时使用的插入代码，它定义了所有作用域（又称「词法环境」），并为每个作用域注册了所有标识符（变量）。

换句话说，虽然作用域是在编译过程中确定的，但直到运行时，每次需要运行作用域时，才会真正创建作用域。在下一章中，我们将勾画出词法作用域的概念基础。

第二章：图解作用域词法

在第 1 章中，我们探讨了在代码编译过程中如何确定作用域，这种模式被称为「词法作用域」。术语「词法」指的是编译的第一阶段（词法/解析）。

要对我们的程序进行正确的*推理*，就必须对作用域的工作原理有一个坚实的概念基础。如果我们依靠猜测和直觉，我们可能会在某些时候意外地得到正确的答案，但在其他更多时候，我们的答案却大相径庭。这不是成功的秘诀。

就像在小学数学课上一样，如果我们不展示正确的步骤，仅得到正确的答案是不够的！我们需要建立准确而有用的心智模型，作为前进的基础。

本章将用几个比喻来说明*作用域*。目的是*思考* JS 引擎如何处理您的程序，使其更接近 JS 引擎的实际工作方式。

弹珠、水桶和气泡.....哦，我的老伙计

在理解作用域方面，我发现一个有效的比喻，就是把彩色弹珠分类装进颜色相匹配的桶里。

想象一下，你拥有一堆弹珠，发现所有弹珠的颜色都是红色、蓝色或绿色。让我们把所有弹珠分类，红色的扔进红色桶，绿色的扔进绿色桶，蓝色的扔进蓝色桶。分类后，当你需要绿色弹珠时，你已经知道从绿色桶里取到它。

在这个比喻中，弹珠就是我们程序中的变量。桶则是作用域（函数和块），我们只是在概念上为它们指定了各自的颜色，以方便讨论。因此，每个弹珠的颜色取决于我们发现弹珠最初是在哪个*颜色*作用域中创建的。

让我们在注释中用颜色标签来定义第一章中的运行程序示例的作用域：

```js
// 外部/全局作用域：红色

var students = [
    { id: 14, name: "Kyle" },
    { id: 73, name: "Suzy" },
    { id: 112, name: "Frank" },
    { id: 6, name: "Sarah" },
];

function getStudentName(studentID) {
    // 函数作用域：蓝色
```

```
    for (let student of students) {
        // 循环作用域：绿色

        if (student.id == studentID) {
            return student.name;
        }
    }
}

var nextStudent = getStudentName(73);
console.log(nextStudent); // Suzy
```

我们在代码注释中指定了三种作用域颜色：红色（最外层的全局作用域）、蓝色（函数 `getStudentName(..)` 的作用域）和绿色（`for` 循环的/内部作用域）。但在查看代码时，可能仍然很难识别这些作用域桶的边界具体是哪里。

图 2 通过在每个作用域周围绘制彩色气泡（又称水桶），帮助直观显示作用域的边界：

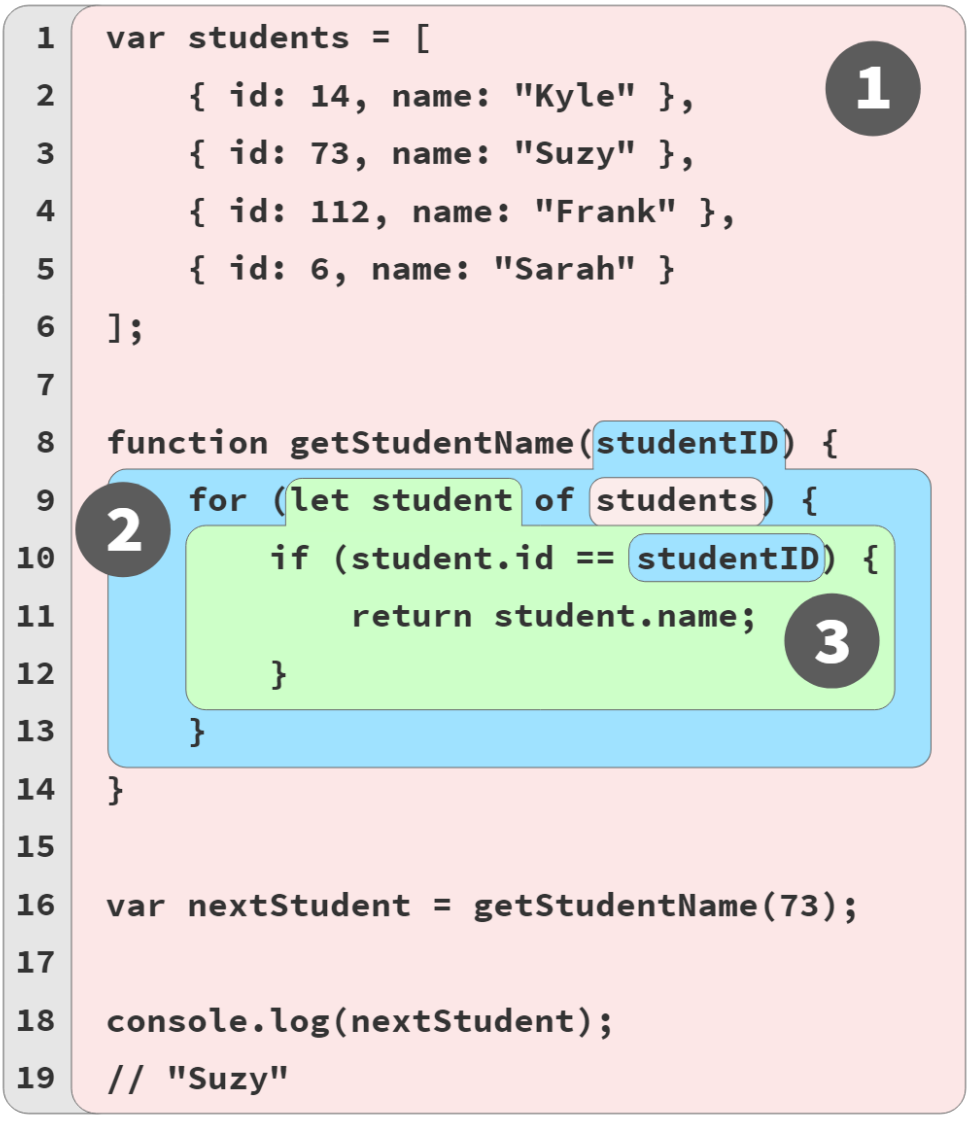

图 2：彩色作用域气泡

1. **气泡 1**（红色）包含全局作用域，其中有三个标识符/变量：`students`（第 1 行）、`getStudentName`（第 8 行）和 `nextStudent`（第 16 行）。

2. **气泡 2**（蓝色）包括函数 `getStudentName(..)` （第 8 行）的作用域，该函数仅有一个标识符/变量：参数 `studentID` （第 8 行）。
3. **气泡 3**（绿色）包含 `for` 循环（第 9 行）的作用域，该循环只包含一个标识符/变量：`student` （第 9 行）。

> 注意：
>
> 从技术上讲，参数 `studentID` 并不完全在蓝色（2）的作用域中。我们将在附录 A 的「隐式作用域」中解开这个困惑。现在，把 `studentID` 标注为蓝色（2）的弹珠已经足够了。

作用域气泡是在编译过程中根据函数/作用域块的编写位置、内部嵌套等因素确定的。每个作用域气泡都完全包含在其父作用域气泡中即作用域永远不会部分地包含在两个不同的外层作用域中。

每个弹珠（变量/标识符）的颜色取决于它在哪个气泡（桶）中声明，而不是它可能被访问的作用域的颜色（例如，第 9 行的 `students` 和第 10 行的 `studentID`）。

> 注意：
>
> 请记住，我们在第 1 章中说过 `id`、`name` 和 `log` 都是属性，而不是变量；换句话说，它们不是桶中的弹珠，因此它们不会根据我们在本书中讨论的任何规则着色。要了解如何处理此类属性访问，请参阅本系列的第三本书*对象与类*。

当 JS 引擎处理程序（在编译过程中）并找到一个变量的声明时，它基本上会问：「我当前处于哪个*颜色*的作用域（气泡或水桶）中？」变量被指定为相同的*颜色*，这意味着它属于该桶/气泡。

绿色（3）桶完全嵌套在蓝色（2）桶内，同样，蓝色（2）桶也完全嵌套在红色（1）桶内。如图所示，作用域可以相互嵌套，嵌套深度可以根据程序需要而定。

如果在当前作用域或当前作用域之上/之外的任何作用域中有匹配的声明，则允许引用（非声明）变量/标识符，但不允许引用来自下层/嵌套作用域的声明。

红色（1）桶中的表达式只能使用红色（1）弹珠，**不能**使用蓝色（2）或绿色（3）。蓝色（2）桶中的表达式可以引用蓝色（2）或红色（1）弹珠，**不能**使用绿色(3)。而绿色（3）桶中的表达式可以访问红色（1）、蓝色（2）和绿色（3）弹珠。

我们可以把在运行时确定这些未声明弹珠颜色的过程概念化为查找。由于第 9 行的 `for` 循环语句中的 `students` 变量引用不是一个声明，所以它没有颜色。因此我们询问当前蓝色（2）作用域桶是否有与该名称匹配的弹珠。既然没有，那么就继续查找下一个外层/包含的作用域：红色（1）。红色（1）桶有一个名称为 `students` 的弹珠，因此循环语句的 `students` 变量引用被确定为一个红色（1）弹珠。

第 10 行的 `if (student.id == studentID)` 语句同样被确定为引用名为 `student` 的绿色（3）弹珠和 `studentID` 的蓝色（2）弹珠。

> 注意：
>
> JS 引擎通常不会在运行时确定这些弹珠的颜色；这里的「查找」只是一种修辞手法，目的是帮助您理解这些概念。在编译过程中，大多数或所有变量引用都会匹配已知的作用域桶，因此它们的颜色已经确定，并与每个弹珠引用一起存储，以避免程序运行时不必要的查找。有关这一细微差别的更多信息，请参见第 3 章。

从弹珠和水桶（还有气泡！）中获得的主要启示：

- 变量是在特定作用域中声明的，可以将其视为来自匹配颜色桶的彩色弹珠。
- 任何变量引用，如果出现在它被声明的作用域中，或出现在任何更深的嵌套作用域中，都会被标记为相同颜色的弹珠，除非中间的作用域「遮蔽」了变量声明；请参阅第 3 章中的「遮蔽」。
- 在编译过程中，会确定颜色桶及其包含的弹珠。在代码执行过程中，这些信息将用于变量（弹珠颜色）的「查找」。

朋友之间的对话

在分析变量及其作用域的过程中，另一个有用的比喻是想象代码在处理和执行过程中在引擎内部发生的各种对话。我们可以「监听」这些对话，从而更好地从概念上理解作用域是如何工作的。

现在让我们来认识一下 JS 引擎的成员，他们在处理我们的程序时会进行对话：

- *引擎*：负责从头到尾编译和执行 JavaScript 程序。
- *编译器*：*引擎*的朋友之一；处理所有解析和代码生成的脏活累活（见上一节）。
- *作用域管理器*：*引擎*的另一个朋友；收集并维护所有已声明变量/标识符的查找列表，并执行一系列规则，规定当前执行代码如何访问这些变量/标识符。

要想*完全理解* JavaScript 的工作原理，你需要开始像*引擎*（和朋友们）那样思考，提出他们提出的问题，并同样回答他们的问题。

为了探讨这些对话，请再次回顾我们正在运行的程序示例：

```js
var students = [
    { id: 14, name: "Kyle" },
    { id: 73, name: "Suzy" },
    { id: 112, name: "Frank" },
    { id: 6, name: "Sarah" },
```

```
];

function getStudentName(studentID) {
    for (let student of students) {
        if (student.id == studentID) {
            return student.name;
        }
    }
}

var nextStudent = getStudentName(73);

console.log(nextStudent);
// Suzy
```

让我们来看看 JS 将如何处理该程序，特别是从第一条语句开始。数组及其内容只是基本的 JS 值字面形式（因此不受任何作用域问题的影响），所以我们在这里的重点是 `var students = [ .. ]` 声明和初始化赋值部分。

我们通常将其视为一条语句，但我们的朋友*引擎*并不这么看。事实上，JS 将这些操作视为两个不同的操作，一个由*编译器*在编译时处理，另一个由*引擎*在执行时处理。

*编译器*要做的第一件事就是进行词法处理，将程序分解为词块，然后将其解析为一棵树 (AST)。

一旦*编译器*进入代码生成阶段，需要考虑的细节可能比看到的要多。一个合理的假设是，*编译器*将为第一条语句生成代码，例如「为一个变量分配内存，给它贴上 `students` 的标签，然后将数组的引用插入该变量」。但这并不是故事的全部。

下面是*编译器*处理该语句的步骤：

1. 遇到 `var students` 时，*编译器*会询问*作用域管理器*看看该特定作用域桶是否已经存在名为 `students` 的变量。如果是，*编译器*将忽略此声明并继续。否则，*编译器*将生成代码，（在执行时）要求*作用域管理器*在该作用域桶中创建一个名为 `students` 的新变量。
2. 然后，*编译器*会生成供*引擎*稍后执行的代码，以处理 `students = []` 赋值。*引擎*运行的代码将首先询问*作用域管理器*在当前作用域桶中是否存在可访问的名为 `students` 的变量。如果没有，*引擎*就会继续查找其他地方（请参阅下面的「嵌套作用域」）。一旦*引擎*找到一个变量，它就会将数组 `[ .. ]` 的引用赋给它。

以对话的形式来看，程序编译的第一阶段可以在*编译器*和*作用域管理器*之间这样进行：

> ***编译器***：嘿，*作用域管理器*（全局作用域），我发现了一个名为 `students` 的标识符的正式声明，听说过吗？

> ***（全局）作用域管理器***：没听说过，所以我就为你创建了它。

> ***编译器***：嘿，*作用域管理器*，我找到了一个名为 `getStudentName` 的标识符的正式声明，听说过吗？

> ***（全局）作用域管理器***：没有，但我刚刚为你创建了它。

> ***编译器***：嘿，*作用域管理器*，`getStudentName` 指向一个函数，所以我们需要一个新的作用域桶。

> ***（函数）作用域管理器***：知道了，这是作用域桶。

> ***编译器***：嘿，*作用域管理器*（函数的管理器），我找到了 `studentID` 的正式参数声明，听说过吗？

> ***（函数）作用域管理器***：没有，但现在它是在这个作用域内创建的。

> ***编译器***：嘿，*作用域管理器*（函数的管理器），我发现一个 `for` 循环需要它自己的作用域桶。

> ...

对话是一种问答式的交流，**编译器**会询问当前的*作用域管理器*是否已经遇到过所遇到的标识符声明。如果回答「否」，*作用域管理器*就会在该作用域中创建该变量。如果答案是「是」，那么实际上就跳过了，因为不需要该*作用域管理器*做什么了。

*编译器*在运行到函数或块作用域时也会发出信号，以便实例化新的作用域桶和*作用域管理器*。

之后，在执行程序时，对话将转移到*引擎*和*作用域管理器*上，可能会这样进行：

> ***引擎***：嘿，*作用域管理器*（全局作用域），在我们开始之前，你能否查找标识符 `getStudentName` 以便我将此函数赋给它？

> ***（全局）作用域管理器***：好的，这个变量给你。

> ***引擎***：嘿，*作用域管理器*，我找到了 `students` 的*目标参考*，听说过吗？

> ***（全局）作用域管理器***：是的，它是在这一作用域定义的，给你。

> ***引擎***：谢谢，我将 `students` 初始化为 `undefined`，这样就可以使用了。

> 嘿，*作用域管理器*（全局作用域），我找到了`nextStudent` 的*目标*引用，听说过吗？

> ***（全局）作用域管理器***：是的，它是在这一作用域定义的，给你。

> ***引擎***：谢谢，我将 `nextStudent` 初始化为 `undefined`，这样就可以使用了。

> 嘿，*作用域管理器*（全局作用域），我找到了`getStudentName` 的*源*引用，听说过吗？

> ***（全局）作用域管理器***：是的，它是在这一作用域定义的，给你。

> *引擎*：很好，`getStudentName` 中的值是一个函数，所以我要执行它。

> *引擎*：嘿，*作用域管理器*，现在我们需要实例化函数的作用域。

> ...

这次对话又是一次问答式交流，*引擎*首先要求当前的*作用域管理器*查找被提升的 `getStudentName` 标识符，以便将函数与之关联。然后，*引擎*继续询问*作用域管理器*有关 `students` 的*目标*引用，以此类推。

回顾并总结一下类似 `var students = [ .. ]` 这样的语句是如何处理的，分为两个不同的步骤：

1. *编译器*会设置作用域变量的声明（因为它之前没有在当前作用域中声明）。
2. 当*引擎*执行时，为了处理语句中的赋值部分，*引擎*会要求*作用域管理器*查找变量，将其初始化为 `undefined` 以便随时使用，然后将数组值赋值给它。

作用域嵌套

当执行 `getStudentName()` 函数时，*引擎*会为该函数的作用域请求一个*作用域管理器*实例，然后它将继续查找参数（`studentID`），并将 `73` 参数值赋给该参数，依此类推。

`getStudentName(..)` 的函数作用域嵌套在全局作用域中。`for` 循环的块作用域同样嵌套在该函数作用域内。作用域可以按照程序定义的任意深度进行词法嵌套。

每次执行作用域（一次或多次）时，每个作用域都会获得自己的*作用域管理器*实例。每个作用域在开始执行时都会自动注册其所有标识符（这称为「变量提升」；请参阅第 5 章）。

在作用域开始时，如果任何标识符来自一个 `function` 声明，该变量会被自动初始化为其关联的函数引用。如果任何标识符来自 `var` 声明（而不是 `let` / `const` 声明），该变量会被自动初始化为 `undefined` 以便可以使用；否则，该变量将保持未初始化状态（又称 "TDZ"，参见第 5 章），在执行完整的声明和初始化之前不能使用。

在 `for (let student of students) {` 语句中，`students` 是一个必须查找的*源*引用。但由于函数的作用域找不到这样的标识符，如何处理这种查找呢？

为了解释清楚，让我们想象一下这样的对话：

> *引擎*：嘿，*作用域管理器*（用于函数），我有一个关于 `students` 的*源*引用，听说过吗？

> ***（函数）作用域管理器***：没听说过。试试下一个外部作用域。

> *引擎*：嘿，*作用域管理器*（用于全局作用域），我有一个 `students` 的*源*引用，听说过吗？

> ***（全局）作用域管理器***：是的，它是在这一作用域定义的，给你。

> ...

词法作用域的一个重要方面是，如果在当前作用域中找不到标识符引用，就会查询嵌套中的下一个外层作用域；这个过程会一直重复，直到找到答案或没有更多的作用域可查询为止。

查找失败

当*引擎*用尽所有*词法可用*作用域（向外移动）仍无法解决标识符的查找问题时，就会出现错误条件。不过，根据程序的模式（严格模式或非严格模式）和变量的作用（即*目标*与*源*；参见第 1 章），这种错误条件会有不同的处理方式。

混乱的 Undefined

如果变量是*源*，则未解决的标识符查找会被视为未声明（未知、缺失）变量，总是会导致抛出 `ReferenceError` 。此外，如果变量是*目标*，且代码当时正在严格模式下运行，则该变量会被视为未声明变量，同样会引发 `ReferenceError`。

在大多数 JS 环境中，未声明变量条件的错误信息类似于 "Reference Error: XYZ is not defined"。就英语而言，「未定义（not defined)」与 "undefined" 几乎完全相同。但在 JS 中，这两个词却截然不同，不幸的是，这条错误信息造成了长期的混淆。

「未定义（not defined)」的真正意思是「标识符被定义但缺少说明（not declared)」，或者更确切地说，是「未声明（undeclared)」，就像一个变量在任何*词法可用*作用域中都没有匹配的正式声明一样。相比之下，"undefined" 的真正含义是找到了一个变量（已声明），但该变量目前没有其他值，所以它的值默认为 `undefined`。

为了进一步加深混淆，JS 的 `typeof` 运算符会为两种状态下的变量引用返回字符串 `"undefined"`：

```js
var studentName;
typeof studentName; // "undefined"

typeof doesntExist; // "undefined"
```

这两个变量引用的条件截然不同，但 JS 确实把水搅浑了。术语混乱令人困惑，也非常不幸。不幸的是，JS 开发人员必须密切关注，以免混淆他们正在处理的*哪种*「未定义」！

全局... 什么!?

如果该变量是一个*目标*变量，并且严格模式没有生效，那么就会出现一种令人困惑和惊讶的遗留行为。麻烦的结果是，全局作用域的*作用域管理器*会创建一个**意外的全局变量**来完成目标赋值！

思考一下：

```js
function getStudentName() {
    // 赋值给未声明的变量 :(
    nextStudent = "Suzy";
}

getStudentName();

console.log(nextStudent);
// "Suzy" -- 哎呀，一个意外的全局变量!
```

下面是*对话*的过程：

> ***引擎***：嘿，*作用域管理器*（用于函数），我有一个关于 `nextStudent` 的*目标*引用，听说过吗？

> ***（函数）作用域管理器***：没听说过。试试下一个外部作用域。

> ***引擎***：嘿，*作用域管理器*（全局作用域），我有一个关于 `nextStudent` 的*目标*引用，听说过吗？

> ***（全局）作用域管理器***：没有，不过既然我们现在是非严格模式，我就帮你创建了一个全局变量，给你！

呸。

这种意外（几乎肯定最终会导致错误）是严格模式提供有益保护的一个很好的例子，也是为什么*不使用*严格模式是个坏主意的原因。在严格模式下，**_全局作用域管理器_**会作出如下响应：

> ***（全局）作用域管理器***：没听说过。对不起，我必须抛出一个 `ReferenceError`。

赋值给一个从未声明过的变量*是*一个错误，因此我们在这里收到一个 `ReferenceError` 是正确的。

切勿依赖意外的全局变量。始终使用严格模式，并正式声明变量。如果你错误地将变量赋值给一个未声明的变量，你会得到一个有用的 `ReferenceError`。

建立在隐喻之上

为了使嵌套的作用域解析可视化，我更喜欢另一个比喻：办公楼，如图 3 所示：

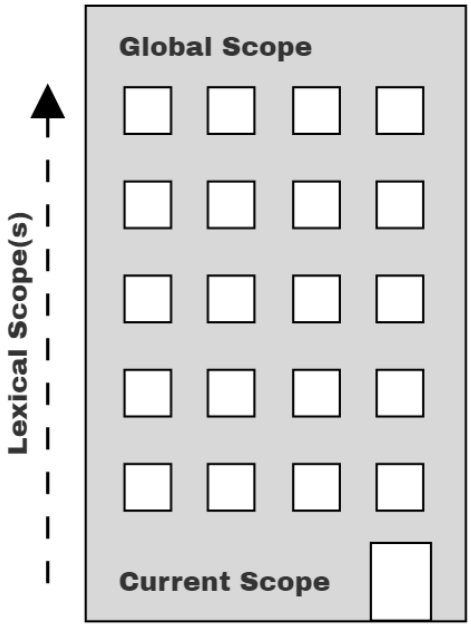

图 3：作用域「建筑」

大楼代表我们程序的嵌套作用域集合。大楼的第一层代表当前执行的作用域。大楼的顶层是全局作用域。

要解析一个*目标*或*源*变量引用，首先要在当前楼层查找，如果找不到，就乘电梯到下一层（即外层作用域），在那里查找，然后再到下一层，依此类推。一旦到达顶层（全局作用域），要么找到要找的东西，要么找不到。但不管怎样，你都必须停下来。

未完待续

到目前为止，您应该可以建立更丰富的心智模型，了解什么是作用域，以及 JS 引擎如何从您的代码中确定和使用作用域。

在*继续*之前，请在你的某个项目中找到一些代码，然后进行这些对话。真的，要大声说出来。找一个朋友，和他们一起练习每个角色。如果你们中的任何一个人发现自己感到困惑或被绊倒了，请花更多时间复习这些资料。

当我们移动（向上）到下（外）一章时，我们将探索程序的词法作用域是如何以链条形式连接起来的。

第三章：作用域链

第 1 章和第 2 章为*词法作用域*（及其组成部分）下了一个具体的定义，并为其概念基础提供了有用的隐喻。在继续学习本章之前，请找其他人用你自己的话（书面或大声）解释什么是词法作用域，以及为什么理解它是有用的。

这一步似乎可以省略，但我发现，花时间把这些想法重新表述为对他人的解释，确实很有帮助。这有助于我们的大脑消化所学知识！

现在是深入探讨具体问题的时候了，所以从现在开始，事情会变得更加详细。不过请坚持下去，因为这些讨论确实会让我们明白，我们对作用域还有多少*不了解*。请务必慢慢阅读文本和提供的所有代码片段。

为了重温我们正在运行的示例的上下文，让我们回顾一下第 2 章图 2 中嵌套作用域气泡的彩色编码图示：

```
1  var students = [
2      { id: 14, name: "Kyle" },
3      { id: 73, name: "Suzy" },
4      { id: 112, name: "Frank" },
5      { id: 6, name: "Sarah" }
6  ];
7
8  function getStudentName(studentID) {
9      for (let student of students) {
10         if (student.id == studentID) {
11             return student.name;
12         }
13     }
14 }
15
16 var nextStudent = getStudentName(73);
17
18 console.log(nextStudent);
19 // "Suzy"
```

图 2（第 2 章）：彩色作用域气泡

嵌套在其他作用域中的作用域之间的连接称为作用域链，它决定了变量的访问路径。作用域链是定向的，这意味着查找只能向上/向外移动。

「查找」（大部分）是概念性的

在图 2 中，请注意 `for` 循环中引用的 `students` 变量的颜色。我们究竟是如何确定它是一个 红色（1）弹珠的呢？

在第 2 章中，我们将变量的运行时访问描述为「查找」，在查找过程中，*引擎*必须先询问当前作用域的*作用域管理器*是否知道某个标识符/变量，然后通过嵌套作用域链（朝向全局作用域）向上/向外查找，直到找到为止（如果有的话）。一旦在作用域桶中找到第一个匹配的命名声明，查找就会停止。

因此，查找过程确定了 `students` 是一个 红色（1）全局作用域，因为我们在遍历作用域链时还没有找到匹配的变量名，直到我们到达最终的红色（1）全局作用域。

同样，`if` 语句中的 `studentID` 也被确定为蓝色（2）弹珠。

这种运行时查找过程的暗示在概念上很好理解，但实际上在实际操作中并非如此。

弹珠桶的颜色（也就是变量来自哪个作用域的元信息）通常是在初始编译处理过程中确定的。由于词法作用域在此时已基本确定，因此弹珠的颜色不会因运行时可能发生的任何情况而改变。

由于弹珠的颜色在编译时就已知晓，而且不可更改，因此该信息很可能与 AST 中每个变量的条目一起存储（或至少可从条目中访问）；然后构成程序运行时的可执行指令会明确使用该信息。

换句话说，*引擎*（来自第 2 章）不需要在一堆作用域中查找变量来自哪个作用域桶。这些信息都是已知的！避免运行时查找是词法作用域的一个关键优化优势。无需在所有这些查找上花费时间，运行时就能更高效地运行。

但我刚才说「……通常是确定的……」，说的是在编译过程中确定弹珠的颜色。那么，在什么情况下，在编译过程中会*不*知道呢？

当前文件中的任何词法作用域中都没有声明变量的引用 — 参见 *入门* 第 1 章，该章中提到，从 JS 编译的角度来看，每个文件都是独立的程序。如果找不到任何声明，这并不一定是错误。运行时中的另一个文件（程序）可能确实在共享全局作用域中声明了该变量。

因此，最终确定变量是否曾在某个可访问的桶中进行过适当的声明，可能需要推迟到运行时进行。

在编译文件的过程中，任何对最初*未声明*的变量的引用都是未着色的弹珠；在编译完其他相关文件并开始应用程序运行之前，无法确定其颜色。这种延迟查找最终会将颜色解析为变量所在的作用域（可能是全局作用域）。

不过，每个变量最多只需要进行一次查询，因为在运行过程中，没有任何其他因素可以改变弹珠的颜色。

第 2 章中的「查找失败」一节介绍了在运行时执行某个弹珠的引用时，如果该弹珠最终仍未着色会发生什么情况。

遮蔽 (Shadowing)[1]

「遮蔽」听起来可能很神秘，而且有点不靠谱。但别担心，它是完全合法的！

我们这几章的运行示例在不同的作用域中使用了不同的变量名。由于它们都有唯一的名称，因此从某种程度上说，如果所有变量都存储在一个桶中（如红色（1））也没有关系。

当您有两个或更多变量时，拥有不同的词法作用域桶开始变得更重要，每个变量都在不同的作用域中，但具有相同的词法名称。一个作用域中不能有两个或更多同名变量；这种多重引用将被视为一个变量。

因此，如果需要维护两个或多个同名变量，就必须使用不同的（通常是嵌套的）作用域。在这种情况下，如何布局不同的作用域桶就非常重要了。

思考一下：

```js
var studentName = "Suzy";

function printStudent(studentName) {
    studentName = studentName.toUpperCase();
    console.log(studentName);
}

printStudent("Frank");
// FRANK

printStudent(studentName);
// SUZY

console.log(studentName);
// Suzy
```

> 贴士：
>
> 在继续之前，请花一些时间使用我们在书中介绍的各种技术/隐喻来分析这段代码。特别是，请务必识别这段代码中的弹珠/气泡颜色。这是很好的练习！

第 1 行的 `studentName` 变量（语句 `var studentName = ..`）创建了一个红色（1）弹珠。在第 3 行，即 `printStudent(..)` 函数定义的参数中，同样命名的变量被声明为蓝色（2）弹珠。

在 `studentName = studentName.toUpperCase()` 赋值语句和 `console.log(studentName)` 语句中，`studentName` 将是什么颜色的弹珠？所有三个 `studentName` 引用都将是蓝色（2）。

根据「查找」的概念，我们断言它会从当前作用域开始向外/向上查找，一旦找到匹配的变量就停止。蓝色（2）`studentName` 立即被找到。红色（1）`studentName` 甚至从未被考虑过。

这是词法作用域行为的一个关键方面，称为*遮蔽*。蓝色（2）`studentName` 变量（参数）会对红色（1）`studentName` 产生遮蔽。因此，参数对全局变量产生了遮蔽。请多重复几遍这句话，以确保您对术语的理解是正确的！

这就是为什么重新分配 `studentName` 只影响内部（参数）变量：蓝色（2）`studentName`，而不是全局的红色（1）`studentName`。

当您选择从外层作用域对变量进行遮蔽处理时，一个直接的影响是，从该作用域向内/向下（通过任何嵌套的作用域），现在任何弹珠都不可能被着色为被遮蔽处理的变量（本例中为红色（1））。换句话说，任何 `studentName` 标识符引用都将对应于该参数变量，而绝不会对应于全局的 `studentName` 变量。从词法上讲，在 `printStudent(..)` 函数内部的任何地方（或任何嵌套作用域）都不可能引用全局 `studentName` 变量。

全局无遮蔽把戏

请注意：使用我将要描述的技术并不是很好的做法，因为它的实用性有限，会让你的代码读者感到困惑，而且很可能会给你的程序带来错误。我之所以介绍它，只是因为你可能会在现有程序中遇到这种行为，而了解发生了什么是避免被绊倒的关键。

在全局变量已被遮蔽化的作用域中访问该变量*是*可能的，但不是通过典型的词法标识符引用。

在全局作用域（红色（1））中，`var` 声明和 `function` 声明也会作为*全局（global）对象*（本质上是全局作用域的对象表示）上的属性（与标识符同名）暴露自己。如果您为浏览器环境编写过 JS，您可能会将全局对象识别为 `window`。这并不*完全*准确，但对于我们的讨论来说已经足够了。在下一章中，我们将进一步探讨全局作用域/对象的话题。

请看这个程序，它是在浏览器环境中作为独立的 .js 文件执行的：

```js
var studentName = "Suzy";

function printStudent(studentName) {
    console.log(studentName);
```

```
    console.log(window.studentName);
}

printStudent("Frank");
// "Frank"
// "Suzy"
```

注意到 `window.studentName` 引用了吗？这个表达式正在访问作为 `window` 属性的全局变量 `studentName`（我们现在假定它与全局对象同义）。这是从存在遮蔽变量的作用域内部访问遮蔽变量的唯一方法。

`window.studentName` 是全局 `studentName` 变量的镜像，而不是单独的快照副本。对其中一个变量的更改仍然可以从另一个变量中看到，无论方向如何。您可以将 `window.studentName` 视为访问实际 `studentName` 变量的 getter/setter。事实上，您甚至可以通过在全局对象上创建/设置一个属性，在全局作用域中*添加*一个变量。

> **警告:**
>
> 记住：*可以*并不意味着*应该*。不要对需要访问的全局变量进行遮蔽处理，反之，也要避免使用此技巧访问已被遮蔽处理的全局变量。绝对不要将全局变量创建为 `window` 属性而不是正式声明，从而混淆代码的读者！

这个小「把戏」只适用于访问全局作用域变量（而不是来自嵌套作用域的遮蔽变量），而且只适用于使用 `var` 或 `function` 声明的变量。

其他形式的全局作用域声明不会创建镜像全局对象属性：

```js
var one = 1;
let notOne = 2;
const notTwo = 3;
class notThree {}

console.log(window.one); // 1
console.log(window.notOne); // undefined
console.log(window.notTwo); // undefined
console.log(window.notThree); // undefined
```

存在于全局作用域以外的任何其他作用域中的变量（无论它们是如何声明的！），在它们已被遮蔽化的作用域中是完全不可访问的：

```js
var special = 42;

function lookingFor(special) {
```

```
    // The identifier `special` (parameter) in this
    // scope is shadowed inside keepLooking(), and
    // is thus inaccessible from that scope.

    function keepLooking() {
        var special = 3.141592;
        console.log(special);
        console.log(window.special);
    }

    keepLooking();
}

lookingFor(112358132134);
// 3.141592
// 42
```

全局的红色（1）`special` 被蓝色（2）`special`（参数）遮蔽，而蓝色（2）`special`本身又被 `keepLooking()`内的绿色（3）`special`遮蔽。我们仍然可以使用间接引用 `window.special` 访问红色（1）`special`。但 `keepLooking()` 无法访问保存数字 `112358132134` 的蓝色（2）`special` 。

复制不是访问

下面这个「但是……呢？」的问题我已经被问过几十次了。思考一下：

```js
var special = 42;

function lookingFor(special) {
    var another = {
        special: special,
    };

    function keepLooking() {
        var special = 3.141592;
        console.log(special);
        console.log(another.special); // Ooo, tricky!
        console.log(window.special);
    }

    keepLooking();
}

lookingFor(112358132134);
```

```
// 3.141592
// 112358132134
// 42
```

哦！那么，这种「另一种」对象技术是否推翻了我的说法，即在 `keepLooking()` 中「完全无法访问」`special` 参数？不，这个说法仍然正确。

`special: special` 是将 `special` 参数变量的值复制到另一个容器（同名属性）中。当然，如果把一个值放到另一个容器中，遮蔽就不再适用了（除非 `another` 也被遮蔽化了！）。但这并不意味着我们正在访问参数 `special`；而是意味着我们正在通过*另一个*容器（对象属性）访问它当时所具有的值的副本。我们不能在 `keepLooking()` 中将蓝色（2）`special` 参数重新赋值。

你可能会提出另一个「但是……？」的问题：如果我用对象或数组代替数字（`112358132134` 等）作为值呢？我们有了对象的引用而不是原始值的副本，就能「解决」无法访问的问题吗？

通过引用副本更改对象值的内容与通过词法访问变量本身并不是一回事。我们仍然无法重新分配蓝色（2）`special` 参数。

非法遮蔽

并非所有的声明遮蔽方式都是允许的。`let` 可以对 `var` 产生遮蔽，但 `var` 不能对 `let` 产生遮蔽：

```js
function something() {
    var special = "JavaScript";

    {
        let special = 42; // 完全 ok 的遮蔽

        // ..
    }
}

function another() {
    // ..

    {
        let special = "JavaScript";

        {
            var special = "JavaScript";
            // ^^^ Syntax Error
```

```
            // ..
        }
    }
}
```

注意在 `another()` 函数中，内部的 `var special` 声明试图声明一个函数作用域内的 `special`，这本身是没有问题的（如 `something()` 函数所示）。

在这种情况下，语法错误描述表明 `special` 已经被定义，但该错误信息有点误导。同样，在 `something()` 中不会出现这种错误，因为通常允许遮蔽。

它被当作 `SyntaxError` 引发的真正原因是 `var` 基本上是在试图「越界」（或跳过）同名的 `let` 声明，而这是不允许的。

禁止跨越作用域的规定实际上止于每个函数作用域，因此这个变体也不例外：

```js
function another() {
    // ..

    {
        let special = "JavaScript";

        ajax("https://some.url", function callback() {
            // 完全 ok 的遮蔽
            var special = "JavaScript";

            // ..
        });
    }
}
```

摘要: `let`（在内作用域中）总是可以遮蔽外作用域的 `var`。内作用域中的 `var` 只有在两者之间有函数作用域的情况下才会对外作用域中的 `let` 产生遮蔽。

函数名的作用域

正如你现在已经看到的，`function` 声明看起来像这样：

```js
function askQuestion() {
    // ..
```

```
}
```

正如第 1 章和第 2 章所讨论的，这样的 `function` 声明将在外层作用域（这里是全局作用域）中创建一个名为 `askQuestion` 的标识符。

那以下代码呢？

```js
var askQuestion = function () {
    // ..
};
```

正在创建的变量 `askQuestion` 也是如此。但由于它是一个 `function` 表达式吗，一个用作值的函数定义，而不是一个独立的声明。函数本身不会「提升」（参见第 5 章）。

函数声明与函数表达式的一个主要区别是函数名称标识符的处理。请考虑命名的 `function` 表达式：

```js
var askQuestion = function ofTheTeacher() {
    // ..
};
```

我们知道 `askQuestion` 最后会进入外层作用域。但是 `ofTheTeacher` 的标识符呢？对于正式的 `function` 声明，名称标识符会在外层/封闭的作用域中结束，因此可以合理地认为这里的情况也是如此。但 `ofTeacher` 被声明为**函数本身内部的标识符**：

```js
var askQuestion = function ofTheTeacher() {
    console.log(ofTheTeacher);
};

askQuestion();
// function ofTheTeacher()...

console.log(ofTheTeacher);
// ReferenceError: ofTheTeacher is not defined
```

> **注意：**
>
> 实际上，`ofTheTeacher` 并不完全*在函数的作用域内*。附录 A 中「隐式作用域」会有进一步解释。

`ofTheTeacher` 不仅是在函数内部而不是外部声明的，而且还被定义为只读：

```js
var askQuestion = function ofTheTeacher() {
    "use strict";
    ofTheTeacher = 42; // TypeError

    //..
};

askQuestion();
// TypeError
```

由于我们使用了严格模式，赋值失败会被报告为「类型错误（TypeError）」；而在非严格模式下，这种赋值失败是无声的，不会抛出异常。

函数表达式没有名称标识符时怎么办？

```js
var askQuestion = function () {
    // ..
};
```

有名称标识符的函数表达式称为「命名函数表达式」，而没有名称标识符的则称为「匿名函数表达式」。匿名函数表达式显然没有名称标识符来影响任何作用域。

> **注意：**
>
> 我们将在附录 A 中更详细地讨论命名表达式和匿名函数表达式，包括影响使用它们的因素。

箭头函数

ES6 在语言中添加了一种额外的函数表达式，称为「箭头函数」：

```js
var askQuestion = () => {
    // ..
};
```

箭头函数 `=>` 不需要使用 `function` 来定义。此外，在某些简单的情况下，参数列表两侧的 `( .. )` 是可选的。同样，函数体两侧的 `{ .. }` 在某些情况下也是可选的。如果省略了 `{ .. }` 将直接返回值，而无需使用 `return` 关键字。

> 注意：
>
> 箭头函数 `=>` 常常被说成是「语法糖」，并声称这等同于客观上更可读的代码。这种说法是可疑的，我认为完全是一种误导。我们将在附录 A 中探讨各种函数形式的「可读性」。

箭头函数在词法上是匿名的，这意味着它们没有直接相关的标识符来引用函数。对 `askQuestion` 的赋值创建了一个推断名称 "askQuestion"，但**这和匿名可不一样**：

```js
var askQuestion = () => {
    // ..
};

askQuestion.name; // askQuestion
```

箭头函数的语法简洁，是以不得不在头脑中为不同的形式/条件变来变去为代价的。举几个例子

```js
() => 42;

(id) => id.toUpperCase();

(id, name) => ({ id, name });

(...args) => {
    return args[args.length - 1];
};
```

我谈论箭头函数的真正原因是，箭头函数在词法作用域方面的行为与标准的 `function` 函数有所不同，这种说法很常见，但并不正确。

这是不正确的。

除了匿名（没有声明形式）之外，`=>` 箭头函数与 `function` 函数具有相同的词法作用域规则。一个箭头函数，带或不带 `{ .. }` ，仍然会创建一个单独的、内部嵌套的作用域桶。在这个嵌套的作用域桶中的变量声明与在 `function` 作用域中的行为相同。

思维转换

当定义一个函数（声明或表达式）时，就会创建一个新的作用域。作用域相互嵌套，在整个程序中形成了一个自然的作用域层次结构，称为作用域链。作用域链控制变量的访问，向上和向外

的。

每一个新的作用域都是一片净土，都有自己的变量空间。当一个变量名在作用域链的不同层级重复出现时，就会产生遮蔽，阻止从该层级向内访问外层变量。

当我们从这些更微小的细节中走出来时，下一章的重点将转移到所有 JS 程序都包含的主要作用域：全局作用域。

---

1. *在程序设计中，变量遮蔽（英语：variable shadowing，或称变量隐藏）指的是：当新变量与旧有变量同名，此名称暂时不再可用于访问旧有变量*，https://zh.wikipedia.org/zh-cn/變數遮蔽 ，2022年12月19日。 ↩

第四章：全局作用域

第 3 章多次提到了「全局作用域」，但您可能仍想知道为什么程序的最外层作用域在现代 JS 中如此重要。现在，绝大多数工作都是在函数和模块内部完成的，而不是在全局作用域内。

仅仅断言「避免使用全局作用域」就够了吗？

JS 程序的全局作用域是一个内容丰富的话题，其实用性和细微差别远远超出你的想象。本章首先探讨了全局作用域在今天编写 JS 程序中（仍然）是如何有用和相关的，然后探讨了在不同的 JS 环境中访问全局作用域的位置和*方式*的差异。

充分理解全局作用域对于掌握使用词法作用域来构建程序至关重要。

为什么是全局作用域？

大多数应用程序都是由多个（有时是很多！）独立的 JS 文件组成的，读者对此可能并不陌生。那么，JS 引擎究竟是如何将所有这些独立文件拼接到一个运行时上下文中的呢？

关于浏览器执行应用程序，主要有三种方式。

首先，如果您直接使用 ES 模块（而不是将它们转换成其他模块包的格式），JS 环境会单独加载这些文件。然后，每个模块会 `import` 它需要访问的其他模块的引用。独立的模块文件只通过这些共享导入相互合作，而不需要任何共享的外层作用域。

其次，如果您在构建过程中使用了打包工具，那么所有文件通常都会在交付给浏览器和 JS 引擎之前串联在一起，然后只处理一个大文件。即使应用程序的所有部分都位于一个文件中，也需要建立某种机制，让每个部分都能注册一个*名称*，以便被其他部分引用，还需要建立某种设施，以便进行访问。

在某些编译设置中，文件的全部内容都被封装在一个单一的外层作用域中，如封装函数、通用模块（UMD — 见附录 A）等。每个片段都可以通过共享作用域中的局部变量注册自己，以便其他片段访问。例如

```js
(function wrappingOuterScope() {
    var moduleOne = (function one() {
        // ..
    })();

    var moduleTwo = (function two() {
        // ..
```

```
        function callModuleOne() {
            moduleOne.someMethod();
        }

        // ..
    })();
})();
```

如上所示，`wrappingOuterScope()` 函数作用域内的 `moduleOne` 和 `moduleTwo` 局部变量的声明是为了让这些模块可以互相访问，以实现合作。

虽然 `wrappingOuterScope()` 的作用域是一个函数，而不是整个环境的全局作用域，但它确实充当了一种「全应用程序作用域」，一个可以存储所有顶级标识符的容器，尽管它不在真正的全局作用域中。在这方面，它有点像全局作用域的替身。

最后是第三种方式：无论应用程序使用的是打包工具，还是（非 ES 模块）文件只是在浏览器中单独加载（通过 `<script>` 标签或其他动态 JS 资源加载），如果没有包含所有这些部分的单一作用域，那么**全局作用域**就是它们相互合作的唯一方式：

这类文件通常看起来像这样：

```js
var moduleOne = (function one() {
    // ..
})();
var moduleTwo = (function two() {
    // ..

    function callModuleOne() {
        moduleOne.someMethod();
    }

    // ..
})();
```

在这里，由于周围没有函数作用域，这些 `moduleOne` 和 `moduleTwo` 声明被简单地放到了全局作用域中。这实际上等同于没有串联文件，而是单独加载文件：

module1.js:

```js
var moduleOne = (function one() {
    // ..
})();
```

module2.js:

```js
var moduleTwo = (function two() {
    // ..

    function callModuleOne() {
        moduleOne.someMethod();
    }

    // ..
})();
```

如果在浏览器环境中将这些文件作为普通的独立 .js 文件单独加载，每个顶层变量声明最终都将成为全局变量，因为全局作用域是这两个独立文件之间唯一的共享资源。从 JS 引擎的角度来看，它们是独立的程序。

除了（有可能）说明运行时应用程序的代码所在位置，以及每个部分如何访问其他部分以进行合作外，全局作用域也是一个重要因素：

- JS 提供的内置组件:
  - 原始值： `undefined`, `null`, `Infinity`, `NaN`
  - 内置： `Date()`, `Object()`, `String()`, 等等。
  - 全局函数： `eval()`, `parseInt()`, 等等。
  - 命名空间： `Math`, `Atomics`, `JSON`
  - JS 的朋友们： `Intl`, `WebAssembly`
- 托管 JS 引擎的环境提供的内置功能：
  - `console` (以及它的函数)
  - DOM（`window`, `document`, 等等)
  - timers（`setTimeout(..)`, 等等)
  - web 平台 API: `navigator`, `history`, geolocation, WebRTC, 等等。

这些只是您的程序将与之交互的众多 *globals* 中的一部分。

> **注意：**
>
> Node 还在「全局」作用域内公开了几个元素，但从技术上讲，它们并不在 `global` 作用域内：`require()`, `__dirname`, `module`, `URL` 等。

大多数开发人员都认为，全局作用域不应该只是应用程序中每个变量的垃圾场。这样做会导致错误丛生。但同样不可否认的是，全局作用域几乎是每个 JS 应用程序的重要「粘合剂」。

全局作用域具体范围是什么?

全局作用域位于文件的最外层，即不在任何函数或其他代码块内，这一点似乎显而易见。但事实并非如此简单。

不同的 JS 环境会以不同的方式处理程序的作用域，尤其是全局作用域。JS 开发人员经常会在不知不觉中产生误解。

浏览器 "Window"

关于全作用域的处理，JS 最*纯粹*的运行环境是在浏览器的网页环境中运行独立的 .js 文件。我说的「纯粹」并不是指没有自动添加任何东西，而是指对代码的侵入或对其预期全局作用域行为的干扰最小。

假设有一个这样的 .js 文件：

```js
var studentName = "Kyle";

function hello() {
    console.log(`Hello, ${studentName}!`);
}

hello();
// Hello, Kyle!
```

这段代码可以在网页环境中使用内联 `<script>` 标签加载或者 `<script src=..>` script 标签，甚至是动态创建的 `<script>` DOM 元素。在这三种情况下，`studentName` 和 `hello` 标识符都是在全局作用域声明的。

这意味着，如果访问全局对象（通常是浏览器中的 `window`），就会在那里找到这些相同名称的属性：

```js
var studentName = "Kyle";

function hello() {
    console.log(`Hello, ${window.studentName}!`);
}
```

```
window.hello();
// Hello, Kyle!
```

这正是阅读 JS 规范时所期望的默认行为：外层作用域*是*全局作用域，而 `studentName` 被合法地创建为全局变量。

这就是我所说的*纯粹*。但遗憾的是，这并不总是你所遇到的所有 JS 环境的真实情况，而这往往会让 JS 开发人员感到惊讶。

全局遮蔽全局

回想一下第 3 章中关于遮蔽（和解除全局遮蔽）的讨论，其中一个变量声明可以覆盖并阻止从外层作用域访问同名的声明。

全局变量和同名的全局属性之间的一个不同寻常的结果是，在全局作用域内，全局对象属性可以被全局变量遮蔽：

```js
window.something = 42;

let something = "Kyle";

console.log(something);
// Kyle

console.log(window.something);
// 42
```

`let` 声明添加了一个 `something` 全局变量，但没有添加全局对象属性（参见第 3 章）。这样做的结果是 `something` 词法标识符会影响 `something` 全局对象属性。

在全局对象和全局作用域之间产生分歧肯定是个坏主意。你的代码读者很大概率会被绊倒。

避免全局声明出现这种问题的简单方法是：全局声明始终使用 `var`。将 `let` 和 `const` 保留给块作用域（参见第 6 章「块作用域」）。

全局 DOM

我曾断言，浏览器托管的 JS 环境拥有我们将看到的最*纯粹*的全局作用域行为。然而，它并不完全*纯粹*。

在基于浏览器的 JS 应用程序中，您可能会在全局作用域内遇到一个令人惊讶的行为：带有 `id` 属性的 DOM 元素会自动创建一个引用它的全局变量。

请看以下代码：

```text
<ul id="my-todo-list">
   <li id="first">Write a book</li>
   ..
</ul>
```

但是这个页面的 JS 可能出现：

```js
first;
// <li id="first">..</li>

window["my-todo-list"];
// <ul id="my-todo-list">..</ul>
```

如果 `id` 值是一个有效的词法名称（如 `first`），则会创建词法变量。否则，访问全局变量的唯一途径是通过全局对象（`window[..]`）。

将所有带有 `id` 的 DOM 元素自动注册为全局变量是一种古老的传统浏览器行为，但由于许多老网站仍然依赖于这种行为，因此这种行为必须保留。我的建议是永远不要使用这些全局变量，尽管它们总是会被静默创建。

(Window) 名称有什么含义?

基于浏览器的 JS 中的另一个全局作用域怪现象：

```js
var name = 42;

console.log(name, typeof name);
// "42" string
```

在浏览器上下文中，`window.name` 是一个预定义的「全局」；它是全局对象上的一个属性，因此看起来就像一个普通的全局变量（但它并不「普通」）。

我们在声明中使用了 `var`，它**不会**对预定义的 `name` 全局属性产生遮蔽。这实际上意味着，`var` 声明会被忽略，因为已经有了一个全局作用域对象属性的名称。正如我们之前讨论过的，如果我们使用 `let name`，我们就会使用一个单独的全局变量 `name` 来遮蔽 `window.name`。

但真正令人惊讶的行为是，尽管我们将数字 `42` 赋值给了 `name`（因此也赋值给了 `window.name`），但当我们检索其值时，它却是一个字符串 `"42"`！在这种情况下，出现这种奇怪的情况是因为 `name` 实际上是 `window` 对象上的一个预定义 getter/setter，它坚持其值必须是一个字符串值。哎呀！

除了 DOM 元素 ID 和 `window.name` 等极少数情况外，在浏览器页面中作为独立文件运行的 JS 具有我们会遇到的最*纯粹*的全局作用域行为。

Web Workers

Web Workers 是 browser-JS 行为之上的网络平台扩展，它允许 JS 文件在与运行主 JS 程序的线程完全独立的线程中运行（从操作系统角度看）。

由于这些 Web Worker 程序运行在单独的线程上，因此它们与主程序线程的通信受到限制，以避免/限制竞争条件和其他复杂情况。例如，Web Worker 代码不能访问 DOM。不过，Web Worker 可以使用某些 Web API，如 `navigator`。

由于 Web Worker 被视为一个完全独立的程序，因此它不会与主 JS 程序共享全局作用域。不过，浏览器的 JS 引擎仍在运行代码，因此我们可以期待其全局作用域行为具有类似的*纯净度*。由于不存在 DOM 访问，因此全局作用域的 `window` 别名并不存在。

在 Web Worker 中，全局对象引用通常使用 `self`：

```js
var studentName = "Kyle";
let studentID = 42;

function hello() {
    console.log(`Hello, ${self.studentName}!`);
}

self.hello();
// Hello, Kyle!

self.studentID;
// undefined
```

与主 JS 程序一样，`var` 和 `function` 声明会在全局对象（又称 `self`）上创建镜像属性，而其他声明（`let` 等）则不会。

因此，我们在这里看到的全局作用域行为对于运行的 JS 程序来说是*纯粹*的；由于没有 DOM 的干扰，也许会更加*纯粹*！

开发人员工具控制台/REPL

回顾入门中的第 1 章，开发者工具并不能创建一个完全贴合 JS 的环境。它们会处理 JS 代码，但也会偏向于对开发人员最友好的用户体验交互（又称开发人员体验或 DX）。

在某些情况下，在键入简短的 JS 代码段时偏向于使用 DX，而不是处理完整 JS 程序所需的常规严格步骤，会在程序和工具之间产生可观察到的代码行为差异。例如，在将代码输入开发工具时，适用于 JS 程序的某些错误条件可能会被放宽而不显示。

就我们在这里讨论的作用域言，这种可观察到的行为差异可能包括：

- 全局作用域的行为
- 提升（见第 5 章）
- 在最外层作用域使用 Block-scoping 声明符时(`let` / `const`，见第 6 章)

虽然在使用控制台/REPL 时，在最外层作用域输入的语句似乎是在真正的全局作用域中处理的，但这并不十分准确。这类工具通常会在一定程度上模拟全局作用域的位置；这只是模拟，而不是严格遵守。这些工具环境优先考虑的是开发人员的便利性，这意味着有时（比如我们当前关于作用域的讨论），观察到的行为可能会偏离 JS 规范。

我们的结论是，开发者工具虽然经过优化，对各种开发者活动非常方便和有用，但**不**适合用来确定或验证实际 JS 程序上下文的明确和细微行为。

ES 模块 (ESM)

ES6 引入了对模块模式的一流支持（将在第 8 章中介绍）。使用 ESM 最明显的影响之一是它如何改变文件中可观察到的顶级作用域的行为。

回顾一下前面的代码片段（我们将使用 `export` 关键字将其调整为 ESM 格式）：

```js
var studentName = "Kyle";

function hello() {
    console.log(`Hello, ${ studentName }!`);
}

hello();
// Hello, Kyle!

export hello;
```

如果该代码位于作为 ES 模块加载的文件中，其运行方式仍然完全相同。不过，从整个应用程序的角度来看，可观察到的效果将有所不同。

尽管 `studentName` 和 `hello` 被声明在（模块）文件的顶层，显而易见时在最外层的作用域内，但它们并不是全局变量。相反，它们是模块作用域内的变量，如果你愿意，也可以称为「模块全局变量」。

然而，在模块中，这些顶层声明并没有隐式的「模块作用域象」可以作为属性添加到模块中，而当声明出现在非模块 JS 文件的顶层时，就会出现这种情况。这并不是说全局变量不能存在或不能在此类程序中被访问。这只是说，全局变量不会通过在模块的顶层作用域中声明变量来*创建*。

模块的顶级作用域是从全局作用域降级而来的，就好像模块的全部内容都被封装在一个函数中一样。因此，存在于全局作用域中的所有变量（无论它们是否在全局对象上！）都可以在模块的作用域中作为词法标识符使用。

ESM 鼓励尽量减少对全局作用域的依赖，您可以导入当前模块运行所需的任何模块。因此，您较少看到使用全局作用域或其全局对象。

不过，如前所述，无论您意识到与否，仍有大量的 JS 和 Web 全局将继续从全局作用域访问！

Node

Node 的一个方面常常让 JS 开发人员措手不及，那就是 Node 将加载的每个 .js 文件（包括启动 Node 进程时使用的主文件）都视为*模块*（ES 模块或 CommonJS 模块，参见第 8 章）。这样做的实际效果是，Node 程序的顶层**实际上并不是全局作用域**，就像在浏览器中加载非模块文件时那样。

截至本文撰写之时，Node 最近新增了对 ES 模块的支持。此外，Node 从一开始就支持一种称为 "CommonJS" 的模块格式，它看起来像这样：

```js
var studentName = "Kyle";

function hello() {
    console.log(`Hello, ${studentName}!`);
}

hello();
// Hello, Kyle!

module.exports.hello = hello;
```

在处理之前，Node 会有效地将此类代码封装在一个函数中，这样 `var` 和 `function` 声明就包含在封装函数的作用域中，而**不会**被视为全局变量。

将前面的代码想象成 Node 看到的样子（示例，非实际）：

```js
function Module(module,require,__dirname,...) {
    var studentName = "Kyle";
```

```
    function hello() {
        console.log(`Hello, ${ studentName }!`);
    }

    hello();
    // Hello, Kyle!

    module.exports.hello = hello;
}
```

然后，Node 会调用 `Module(..)` 函数来运行模块。在这里，您可以清楚地看到为什么 `studentName` 和 `hello` 标识符不是全局的，而是在模块作用域中声明的。

如前所述，Node 定义了许多类似 `require()` 的「全局」，但它们实际上并不是全局作用域中的标识符（也不是全局对象的属性）。它们被注入到每个模块的作用域中，本质上有点像 `Module(..)` 函数声明中列出的参数。

那么，如何在 Node 中定义实际的全局变量呢？唯一的方法就是在 Node 自动提供的另一个「全局」中添加属性，具有讽刺意味的是，这个「全局」被称为`global`。`global` 是对真正全局作用域对象的引用，有点像在浏览器 JS 环境中使用 `window` 一样。

思考一下：

```js
global.studentName = "Kyle";

function hello() {
    console.log(`Hello, ${studentName}!`);
}

hello();
// Hello, Kyle!

module.exports.hello = hello;
```

在这里，我们将 `studentName` 添加为 `global` 对象的一个属性，然后在 `console.log(..)` 语句中，我们就可以将 `studentName` 作为一个普通的全局变量来访问。

请记住，标识符 `global` 不是由 JS 定义的，而是由 Node 专门定义的。

Global This

回顾我们迄今为止所看到的 JS 环境，程序可能有，也可能没有：

- 在顶层作用域中使用 `var` 或 `function` 声明或 `let`，`const` 和 `class` 声明全局变量。
- 如果使用了 `var` 或 `function` 声明，还可将全局变量声明添加为全局作用域对象的属性。
- 使用 `window`、`self` 或 `global` 引用全局作用域对象（用于添加或检索全局变量或者作为属性）。

我认为可以这样说，全局作用域的访问和行为比大多数开发人员想象的要复杂得多，前面的章节已经说明了这一点。但是，这种复杂性在试图确定对全局作用域对象的通用引用时表现得最为明显。

获取全局作用域对象引用的另一个「把戏」是：

```js
const theGlobalScopeObject = new Function("return this")();
```

> **注意：**
>
> 可以使用 `Function()` 构造函数从存储在字符串值中的代码动态构造函数，类似于 `eval(..)`（请参阅第 1 章中的「取巧：修改运行时的作用域」）。如图所示，当使用普通的 `()` 函数调用时，这样的函数将自动以非严格模式运行（出于遗留原因）；其 `this` 指向全局对象。有关确定 `this` 绑定的更多信息，请参阅本系列丛书第三册*对象与类*。

因此，我们有了 `window`、`self`、`global` 和这个丑陋的 `new Function(..)` 把戏。有很多不同的方法来获取这个全局对象。各有利弊。

要不再介绍一个！？！？

从 ES2020 开始，JS 终于定义了对全局作用域对象的标准化引用，称为 `globalThis`。因此，根据您代码运行的 JS 引擎的最新情况，您可以使用 `globalThis` 代替任何其他方法。

我们甚至可以尝试定义一种跨环境的 polyfill，让之前不支持 `globalThis` 的 JS 环境中更安全，例如

```js
const theGlobalScopeObject =
    typeof globalThis != "undefined"
        ? globalThis
        : typeof global != "undefined"
          ? global
          : typeof window != "undefined"
            ? window
            : typeof self != "undefined"
```

```
            ? self
            : new Function("return this")();
```

呼！这当然不是最理想的，但如果你需要一个可靠的全局作用域参考，它还是有用的。

（在将该功能添加到 JS 时，`globalThis` 这个拟议名称曾引起相当大的争议。具体来说，我和其他许多人都认为名称中的 "this" 引用具有误导性，因为引用此对象的原因是访问全局作用域，而不是访问某种全局/默认的 `this` 绑定。我们曾考虑过许多其他名称，但由于种种原因都被排除了。不幸的是，最终选择的名称是不得已而为之。如果您打算在您的程序中与全局作用域对象交互，为了减少混淆，我强烈建议您选择一个更好的名称，比如这里使用的（长得可笑但准确的！）`theGlobalScopeObject`（全局作用域对象）。）

掌握全局作用域

尽管将代码组织成模块的现代模式下不再强调在命名空间中存储标识符的重要性，但全局作用域在每一个 JS 程序中都存在并与之相关。

尽管如此，随着我们的代码越来越多地超出浏览器的限制，掌握全局作用域（和全局作用域对象！）在不同 JS 环境中的行为差异就显得尤为重要。

有了全局作用域的全貌，下一章将再次深入词法作用域的细节，研究如何以及何时使用变量。

第五章：（并不）神秘的变量生命周期

现在，您应该对从全局作用域到程序的作用域链的嵌套有了一定的了解了。

但仅仅知道变量来自哪个作用域还只是管中窥豹。如果变量的声明出现在作用域的第一条语句之后，那么在声明之前对该标识符的任何引用将如何表现？如果在作用域中两次声明同一个变量会发生什么？

JS 的特殊词法作用域在变量如何以及何时出现并提供给程序使用方面有着丰富的细微差别。

何时可以使用变量?

变量在什么时候可以在其作用域内使用？答案似乎显而易见： 在变量被声明/创建*之后*。对吗？不尽然。

思考一下：

```js
greeting();
// Hello!

function greeting() {
    console.log("Hello!");
}
```

这段代码运行良好。您以前可能见过或写过类似的代码。但你是否想过它是如何或为什么能运行的？具体地说，为什么您可以从第 1 行开始访问标识符 `greeting`（以检索和执行函数引用），尽管 `greeting()` 函数声明直到第 4 行才出现？

回顾第一章，所有标识符都在编译时注册到各自的作用域中。此外，每个标识符都是在其所属作用域的起始位置*创建的*，**每次进入该作用域时**。

最常用的术语是**变量提升(hoisting)** 变量从其外层作用域的起始位置开始可见，即使它的声明可能出现在作用域的更下面。

但仅仅是变量提升并不能完全回答这个问题。我们可以从作用域的开头看到一个名为 `greeting` 的标识符，但为什么我们可以在声明 `greeting()` 函数之前**调用**它呢？

换句话说，从作用域开始运行的那一刻起，变量 `greeting` 如何被赋值（函数引用）？答案是这是 `function` 声明的一个特殊特性，称为*函数提升 (function hoisting)*。当一个

`function` 声明的名称标识符被注册在其作用域的顶部时，它就会被自动初始化为该函数的引用。这就是为什么该函数可以在整个作用域中被调用！

一个关键的细节是*函数提升*和 `var` 声明的*变量提升*都将它们的名称标识符附加到最近的外层 **function scope** 中（如果没有，则附加到全局作用域），而不是块作用域。

> **注意：**
>
> 带有 `let` 和 `const` 的声明仍然可以提升（参见本章后面关于 TDZ 的讨论）。但这两种声明形式附加于它们的外层块，而不是像 `var` 和 `function` 声明那样只附加于外层函数。更多信息请参阅第 6 章中的「块的作用域」。

提升： 声明 (Declaration) vs. 表达 (Expression)

*函数提升*仅适用于正式的函数声明（特别是那些出现在代码块之外的声明，参见第 6 章中的 "FiB"），不适用于函数表达式赋值。思考一下：

```js
greeting();
// TypeError

var greeting = function greeting() {
    console.log("Hello!");
};
```

第 1 行（`greeting();`）抛出了一个错误。但要注意的是抛出的是*哪种*错误。类型错误（`TypeError`）意味着我们试图对一个不允许的值进行操作。根据你的 JS 环境，错误信息会类似于 "'undefined' is not a function" 或者更有用的说法是 "'greeting' is not a function"。

请注意，该错误并**不是** `ReferenceError`（引用错误）。JS 并没有告诉我们它无法在作用域中找到作为标识符的 `greeting`。它是在告诉我们，虽然找到了 `greeting`，但当时并没有函数引用。只有函数才能被调用，因此试图调用某个非函数值会导致错误。

但是，如果不是函数引用，`greeting` 又是什么呢？

除了被提升之外，使用 `var` 声明的变量在其作用域开始时也会被自动初始化为 `undefined`，同样，在最近的外层函数或全局中。一旦初始化，它们就可以在整个作用域中被使用（赋值、检索等）。

因此，在第一行，`greeting` 已经存在，但它只保留了默认的 `undefined` 值。直到第 4 行，`greeting` 才被分配了函数引用。

请注意这里的区别。函数声明被提升**并初始化其为函数值**（*函数提升*）。 `var` 声明的变量也被提升，然后自动初始化为 `undefined`。随后对该变量的任何 `function` 表达式赋值都要等到运行时执行过程中处理该赋值时才会发生。

在这两种情况下，标识符的名称都会被提升。但除非标识符是在正式的 `function` 声明中创建的，否则在初始化时（作用域的开始）函数引用关联不会被处理。

变量提升

让我们再来看一个*变量提升*的例子：

```js
greeting = "Hello!";
console.log(greeting);
// Hello!

var greeting = "Howdy!";
```

虽然 `greeting` 在第 5 行才被声明，但它早在第 1 行就可以被赋值了。为什么？

这其中有两个必要的说明：

- 标识符被提升，
- **并**从作用域的顶部自动初始化为`undefined`值。

> **注意：**
>
> 使用这种*变量提升*可能会让人感觉不自然，许多读者可能会理所当然地希望在自己的程序中避免使用这种情况。但是，是否应该避免所有的提升（包括*函数提升*）呢？我们将在附录 A 中更详细地探讨关于提升的这些不同观点。

提升：又一个隐喻

第 2 章中有很多隐喻（用来说明作用域），但这里我们又要面对另一个隐喻：提升本身。与其说「提升」是 JS 引擎执行的一个具体执行步骤，不如将「提升」视为 JS 在**执行前**设置程序时所采取的各种行动的可视化。

提升的传统含义是什么？*举起 (lifting)*：就像把重物向上提升一样，将任何标识符一直提升到作用域的顶部。通常的解释是，JS 引擎实际上会在执行前*重写*该程序，使其看起来更像这样：

```js
var greeting; // 提升声明
greeting = "Hello!"; // 原来的第一行
```

```
console.log(greeting); // Hello!
greeting = "Howdy!"; // `var` 消失了！
```

提升（隐喻）建议 JS 在执行前对原始程序进行预处理并稍作重新排列，以便将所有声明移至各自作用域的顶部。此外，提升隐喻还断言，「函数」声明会被全部提升到每个作用域的顶部。思考一下：

```js
studentName = "Suzy";
greeting();
// Hello Suzy!

function greeting() {
    console.log(`Hello ${studentName}!`);
}
var studentName;
```

提升隐喻的「规则」是提升函数声明，然后在所有函数之后紧接着提升变量。因此，「提升」的故事表明，JS 引擎会将程序*重新编排*，使其看起来像这样：

```js
function greeting() {
    console.log(`Hello ${studentName}!`);
}
var studentName;

studentName = "Suzy";
greeting();
// Hello Suzy!
```

这种「提升」的隐喻很方便。它的好处在于，我们可以省去那些神奇的前瞻性预处理，将所有这些深埋在作用域中的声明以某种方式移动（提升）到顶部；我们只需将程序看作是由 JS 引擎自上而下地**一次性执行**的。

与第 1 章中关于两阶段处理的说法相比，单阶段处理无疑显得更加简单明了。

提升作为一种重新排序代码的机制可能是一种有吸引力的简化，但它并不准确。JS 引擎实际上并不会重新排列代码。它无法神奇地提前找到声明；准确找到声明以及程序中所有作用域边界的唯一方法是完全解析代码。

猜猜什么是解析？二相处理的第一阶段！没有什么神奇的脑力体操能绕过这一事实。

那么，如果「提升」这个词的隐喻（充其量）不准确，我们该如何处理这个词呢？我认为它仍然有用。事实上，甚至 TC39 的成员也经常使用它！但我认为我们不应该声称它是对源代码的实际重新编排。

> **警告：**
>
> 不正确或不完整的心智模型往往看起来够用，因为它们偶尔会意外地带来正确的答案。但从长远来看，如果你的思维与 JS 引擎的工作方式不完全一致，就很难准确地分析和预测结果。

我断言，「提升」*应该*用来指**编译时的操作**，即每次进入变量的作用域时，生成运行时指令，以便在其作用域的起始位置自动注册该变量。

这是一个微妙而重要的转变，从作为运行时行为的提升转变为编译时任务中的适当位置。

重复声明？

如果在同一作用域中多次声明变量，你认为会发生什么情况？思考一下：

```js
var studentName = "Frank";
console.log(studentName);
// Frank

var studentName;
console.log(studentName); // ???
```

您希望打印出的第二条信息是什么？许多人认为第二个 `var studentName` 重新声明了变量（从而「重置」了变量），因此他们认为会打印出 `undefined`。

但在同一作用域中是否存在「重新声明」变量的情况呢？没有。

如果从提升隐喻的角度来考虑这个程序，为了执行的目的，代码可以这样重新编排：

```js
var studentName;
var studentName; // 显然是在做无用功！

studentName = "Frank";
console.log(studentName);
// Frank

console.log(studentName);
// Frank
```

由于提升实际上是在作用域的开头注册变量，因此在作用域的中间没有什么可做的，在那里原程序实际上有第二条 `var studentName` 语句。这只是一个 no-op(eration)，一个毫无意义的语句。

贴士：

按照第 2 章的对话叙述方式，*编译器*会找到第二个 `var` 声明语句，并询问 *作用域管理员*是否已经看到了 `studentName` 标识符；既然已经看到了，就没有其他事情可做了。

还需要指出的是，`var studentName;` 并不是大多数人认为的 `var studentName = undefined;`。让我们通过考虑这个程序的变体来证明它们是不同的：

```js
var studentName = "Frank";
console.log(studentName); // Frank

var studentName;
console.log(studentName); // Frank <--- 依旧！

// 让我们显式地添加初始化
var studentName = undefined;
console.log(studentName); // undefined <--- 看！？
```

看到显式的 `= undefined` 初始化与省略隐式初始化所产生的不同结果了吗？在下一节中，我们将重新讨论变量声明中的初始化问题。

在作用域中重复声明相同标识符名称的 `var` 实际上是一种无用的操作。下面是另一个例子，这次是在一个同名函数中：

```js
var greeting;

function greeting() {
    console.log("Hello!");
}

// 基本上是无意义的
var greeting;

typeof greeting; // "function"

var greeting = "Hello!";

typeof greeting; // "string"
```

第一个 `greeting` 声明将标识符注册到作用域中，因为它是一个 `var` ，所以自动初始化将是 `undefined`。 `function` 的声明不需要重新注册标识符，但由于*函数提升*，它会覆盖自动

初始化以使用函数引用。第二个 `var greeting` 本身没有任何作用，因为 `greeting` 已经是一个标识符，而且*函数提升*已经优先于自动初始化。

实际上，将 `"Hello!"` 赋值给 `greeting` 会将其值从初始函数 `greeting()` 变为字符串；而 `var` 本身没有任何影响。

那么使用 `let` 或 `const` 在作用域内重复声明会如何？

```js
let studentName = "Frank";

console.log(studentName);

let studentName = "Suzy";
```

该程序不会执行，而是会立即抛出 `SyntaxError`（语法错误）。根据您的 JS 环境，错误信息将显示如下内容 "studentName has already been declared."。换句话说，这是明确不允许尝试「重复声明」的情况！

不仅仅是涉及 `let` 的两个声明会产生这个错误。如果其中一个声明使用了 `let`，另一个声明可以是 `let` 或 `var`，错误仍然会发生，如下面两个变体所示：

```js
var studentName = "Frank";

let studentName = "Suzy";
```

以及：

```js
let studentName = "Frank";

var studentName = "Suzy";
```

在这两种情况下，都会在二次声明中抛出一个 `SyntaxError`。换句话说，「重复声明」变量的唯一方法是在所有（两个或两个以上）声明中使用 `var`。

但为什么不允许呢？错误的原因并不在于技术本身，因为 `var` 的「重复声明」一直是被允许的；显然，`let` 也可以有同样的允许。

这其实更像是一个「社会工程」问题。包括 TC39 成员在内的一些人认为，「重复声明」变量是一种可能导致程序错误的坏习惯。因此，当 ES6 引入 `let` 时，他们决定用一个错误来防止「重新声明」。

注意：

当然，这只是一种风格上的观点，并非真正的技术论据。许多开发者都同意这一立场，这可能也是 TC39 包含这一错误（以及 `let` 符合 `const`）的部分原因。但有一个合理的理由可以证明，与 `var` 的先例保持一致是更谨慎的做法，而且这种意见执行最好留给像 linters 这样的选择性工具。在附录 A 中，我们将探讨 `var`（及其相关行为，如「重复声明」）在现代 JS 中是否仍然有用。

当*编译器*向*作用域管理员*询问一个声明时，如果该标识符已经被声明过，而且其中一个/两个声明都是用 `let` 进行的，那么就会产生一个错误。向开发人员发出的信号是：「不要再依赖马虎的重复声明了！」

常量?

`const` 关键字比 `let` 受到更多限制。和 `let` 一样，`const` 不能在同一作用域中重复使用相同的标识符。但实际上，不允许这种「重复声明」是有技术原因的，不像 `let` 主要是出于风格上的原因而不允许「重复声明」。

`const` 关键字要求对变量进行初始化，因此在声明中省略赋值会导致 `SyntaxError`（语法错误）：

```js
const empty;   // SyntaxError
```

`const` 声明创建的变量不能重新赋值：

```js
const studentName = "Frank";
console.log(studentName);
// Frank

studentName = "Suzy"; // TypeError
```

`studentName` 变量不能被重新赋值，因为它是用 `const` 声明的。

警告：

重新分配 `studentName` 时抛出的错误是 `TypeError`，而不是 `SyntaxError`。这里的细微差别实际上非常重要，但不幸的是很容易被忽略。语法错误代表程序中的故障，甚至会阻止程序开始执行。类型错误代表程序执行过程中出现的故障。在前面的代码段中，`"Frank"` 在我们处理 `studentName` 的重新赋值之前被打印出来，然后产生错误。

因此，如果 `const` 声明不能重新赋值，而 `const` 声明总是需要赋值，那么我们就有了一个明确的技术理由来解释为什么 `const` 必须禁止任何"重复声明"：任何 `const` 的"重复声

明"也必然是 `const` 的重新赋值，而这是不允许的！

```js
const studentName = "Frank";

// 显然这是一个错误
const studentName = "Suzy";
```

由于 `const` 「重复声明」必须被禁止（基于这些技术原因），TC39 基本上认为 `let` 「重复声明」也应被禁止，以保持一致性。这是否是最好的选择还有待商榷，但至少我们知道了决定背后的理由。

循环

因此，从我们之前的讨论中可以清楚地看出，JS 并不希望我们在同一作用域内「重复声明」变量。这似乎是一个简单明了的告诫，直到你考虑到它对重复执行循环中的声明语句意味着什么。思考一下：

```js
var keepGoing = true;
while (keepGoing) {
    let value = Math.random();
    if (value > 0.5) {
        keepGoing = false;
    }
}
```

在这个程序中，`value`是否被反复「重复声明」？会不会报错？不会。

作用域的所有规则（包括对 `let` 创建的变量的「重复声明」）都是按*每个作用域实例*应用的。换句话说，每次在执行过程中进入作用域，一切都会重置。

每个循环迭代都是它自己的新作用域实例，而在每个作用域实例中，`value` 只被声明一次。因此不存在试图「重复声明」的情况，也就不会出错。在我们考虑其他循环形式之前，如果把前面代码段中的 `value` 声明改为 `var` 会怎样？

```js
var keepGoing = true;
while (keepGoing) {
    var value = Math.random();
    if (value > 0.5) {
        keepGoing = false;
    }
}
```

我们知道 `var` 允许这样做，那么 `value` 在这里是否被「重复声明」了呢？不是的。因为 `var` 不被视为块作用域声明（参见第 6 章），它将自己附加到全局作用域中。因此只有一个 `value` 变量，与 `keepGoing` 处于同一作用域（本例中为全局作用域）。这里也没有「重复声明」！

有一种方法可以让我们记住，当代码开始执行时，`var`、`let` 和 `const` 关键字实际上已从代码中删除。它们完全由编译器处理。

如果你在头脑中抹去声明器关键字，然后尝试处理代码，应该可以帮助你判断是否以及何时可能出现（重复）声明。

其他循环形式（如 `for` 循环）的「重复声明」又是怎么回事？

```js
for (let i = 0; i < 3; i++) {
    let value = i * 10;
    console.log(`${i}: ${value}`);
}
// 0: 0
// 1: 10
// 2: 20
```

很明显，每个作用域实例只能声明一个 `value`。那么 `i` 呢？它被「重复声明」了吗？

要回答这个问题，请考虑一下 `i` 所处的作用域。它看起来像是在外层（在本例中是全局）作用域，但其实不是。它在 `for` 循环体的作用域中，就像 `value` 一样。事实上，你可以用这种更简洁的等价形式来考虑这个循环：

```js
{
    // 用于说明的虚构变量
    let $$i = 0;

    for (; /* nothing */ $$i < 3; $$i++) {
        // 这是我们实际循环的 `i`！
        let i = $$i;

        let value = i * 10;
        console.log(`${i}: ${value}`);
    }
    // 0: 0
    // 1: 10
    // 2: 20
}
```

现在应该清楚了： `i` 和 `value` 变量都是在**每个作用域实例**中声明一次。这里没有「重复声明」。

其他 `for` 循环形式呢？

```js
for (let index in students) {
    // 这么做没问题
}

for (let student of students) {
    // 俺也一样
}
```

`for..in` 和 `for..of` 循环也是如此：声明的变量被视为循环体的*内部*变量，因此每次迭代（又称每个作用域实例）都会被处理。没有「重复声明」。

好吧，我知道你在想我现在的语气就像个坏掉的唱片。但让我们来探讨一下 `const` 如何影响这些循环结构。思考一下：

```js
var keepGoing = true;
while (keepGoing) {
    // 哦，一个闪亮的常数！
    const value = Math.random();
    if (value > 0.5) {
        keepGoing = false;
    }
}
```

就像我们之前看到的这个程序的 `let` 变体一样，`const` 在每个循环迭代中都会被运行一次，所以它不会有「重复声明」的问题。但当我们谈到 `for` 循环时，情况就变得复杂了。

`for..in` 和 `for..of` 可以与 `const` 一起使用：

```js
for (const index in students) {
    // 这么做没问题
}

for (const student of students) {
    // 同样没问题
}
```

但是传统的 `for` 循环不行：

```js
for (const i = 0; i < 3; i++) {
    // 哎呀，这样做会失败的
    // 第一次迭代后出现类型错误
}
```

这里有什么问题吗？我们可以在这个结构中很好地使用 `let` ，而且我们断言它为每个循环迭代作用域创建了一个新的 `i` ，所以它看起来甚至不是一个「重复声明」。

让我们像刚才那样在头脑中「扩展」这个循环：

```js
{
    // 用于说明的虚构变量
    const $$i = 0;

    for (; $$i < 3; $$i++) {
        //  这是我们实际循环的 `i`！
        const i = $$i;
        // ..
    }
}
```

您发现问题了吗？我们的 `i` 的确只是在循环中创建了一次。这不是问题所在。问题在于概念中的 `$$i` 每次都必须用 `$$i++` 表达式递增。这是**重新赋值**（而不是「重复声明」），常量不允许这样做。

请记住，这种「扩展」形式只是一种概念模型，可以帮助您直观地了解问题的根源。您可能会想，JS 是否可以有效地将 `const $$i = 0` 变成 `let $ii = 0` ，从而使 `const` 与我们经典的 `for` 循环一起工作？这是可能的，但这样可能会给 `for` 循环语义带来令人惊讶的异常。

例如，如果允许 `for` 循环头中的 `i++` 避开 `const` 赋值的严格性，但不允许在循环迭代中对 `i` 进行其他重新赋值，这将是一个相当武断（而且很可能引起混淆）的细微例外。

直接的答案是：`const` 不能与传统的 `for` 循环形式一起使用，因为需要重新赋值。

有趣的是，如果你不重新分配，那么它就是有效的：

```js
var keepGoing = true;

for (const i = 0; keepGoing /* 这什么也没有 */; ) {
    keepGoing = Math.random() > 0.5;
    // ..
}
```

这样做是可行的，但毫无意义。没有理由在该位置用 `const` 声明 `i`，因为在该位置声明这样一个变量的目的就是**用来计算迭代次数**。只需使用不同的循环形式，如 `while` 循环，或使用 `let` 循环！

未初始化变量警告（又称 TDZ 暂时性死区）

通过 `var` 声明，变量会被「提升」到其作用域的顶部。但它也会被自动初始化为 `undefined` 值，这样变量就可以在整个作用域中使用。

然而，`let` 和 `const` 声明在这方面并不完全相同。

思考一下：

```js
console.log(studentName);
// ReferenceError 引用错误

let studentName = "Suzy";
```

该程序的结果是在第一行抛出了一个 `ReferenceError`（引用错误）。根据您的 JS 环境，错误信息可能会如下所示 "Cannot access studentName before initialization"。

> **注意：**
>
> 这里看到的错误信息过去更含糊不清或具有误导性。值得庆幸的是，我们社区中的一些人成功地游说 JS 引擎改进了这条错误信息，使它能更准确地告诉你出了什么问题！

这条错误信息很好地说明了问题所在：`studentName` 存在于第 1 行，但尚未初始化，因此还不能使用。让我们试试这个：

```js
studentName = "Suzy"; // 让我们试着初始化它！
// ReferenceError

console.log(studentName);

let studentName;
```

哎呀。我们仍然收到了 `ReferenceError`，但现在我们正试图在第一行赋值给（也就是初始化！）这个所谓的「未初始化」变量 `studentName`。这是怎么回事？

真正的问题是，我们如何初始化一个未初始化的变量？对于 `let` / `const` 来说，**唯一的方法**是在声明语句中附加赋值。赋值本身是不够的！思考一下：

```js
let studentName = "Suzy";
console.log(studentName); // Suzy
```

在这里，我们通过带有赋值的 `let` 声明语句形式来初始化 `studentName` （在本例中，初始化为 `"Suzy"` 而不是 `undefined` ）。

或者：

```js
// ..

let studentName;
// 或：
// let studentName = undefined;

// ..

studentName = "Suzy";

console.log(studentName);
// Suzy
```

> **注意：**
>
> 这很有趣！回想一下，我们在前面说过 `var studentName;` 与 `var studentName = undefined;` 并不相同，但在 `let` 中，它们的行为是一样的。区别在于 `var studentName` 会在作用域的顶层自动初始化，而 `let studentName` 不会。

请记住，到目前为止我们已经多次断言*编译器*最终会删除任何 `var` / `let` / `const` 声明符，取而代之的是在每个作用域的顶部注册相应标识符的指令。

因此，如果我们分析一下这里发生的事情，就会发现一个额外的细微差别，那就是*编译器*在程序中间，也就是在声明变量 `studentName` 的地方添加了一条指令，以处理声明的自动初始化。在初始化之前，我们不能使用该变量。对于 `const` 和 `let` 也是一样。

TC39 创造了一个术语来指从进入作用域到变量发生自动初始化的*这段时间*： 暂时性死区（TDZ）。

TDZ 是变量存在但仍未初始化的时间窗口，因此无法以任何方式访问。只有执行*编译器*在最初声明时留下的指令才能完成初始化。之后，TDZ 就完成了，变量就可以在其余的作用域中自由

使用了。

从技术上讲，`var` 也有一个 TDZ，但它的长度为零，因此我们的程序无法观察到！只有 `let` 和 `const` 才有可观测的 TDZ。

顺便说一句，TDZ（Temporal Dead Zone）中的 "temporal" 指的确实是*时间*，而不是*代码中的位置*。思考一下：

```js
askQuestion();
// ReferenceError 引用错误

let studentName = "Suzy";

function askQuestion() {
    console.log(`${studentName}, do you know?`);
}
```

尽管从位置上看，引用 `studentName` 的 `console.log(..)` 在 `let studentName` 声明之后，但从时间上看，`askQuestion()` 函数是在遇到 `let` 语句之前调用的，而 `studentName` 仍在其 TDZ 中！因此出现错误。

有一种常见的误解，认为 TDZ 意味着 `let` 和 `const` 没有提升。这种说法是不准确的，至少有一点误导。它们肯定会提升。

实际区别在于 `let`/`const` 声明不会像 `var` 那样在作用域的开头自动初始化。那么*争论*的问题是，自动初始化是否是*提升的*一部分？我认为在作用域的顶端自动注册变量（即我所说的「提升」）和在作用域的顶端自动初始化（到 `undefined`）是不同的操作，不应该混为一谈，统称为「提升」。

我们已经看到 `let` 和 `const` 不会在作用域的顶层自动初始化。但让我们来证明 `let` 和 `const` *可以*提升（在作用域顶端自动注册），这要归功于我们的朋友 shadowing（参见第 3 章中的「遮蔽（Shadowing）」）：

```js
var studentName = "Kyle";

{
    console.log(studentName);
    // ???

    // ..

    let studentName = "Suzy";
```

```
    console.log(studentName);
    // Suzy
}
```

第一条 `console.log(..)` 语句会发生什么？如果 `let studentName` 没有提升到作用域的顶部，那么第一条 `console.log(..)` 应该打印 `"Kyle"` ，对吗？此时，似乎只有外层的 `studentName` 存在，所以 `console.log(..)` 应该访问并打印这个变量。

但是，第一个 `console.log(..)` 却抛出了 TDZ 错误，因为事实上，内部作用域的 `studentName` **已经**提升了（自动注册到了作用域的顶部）。但内部的 `studentName` 还没有自动初始化；此时它仍未初始化，因此违反了 TDZ！

因此，概括地说，TDZ 错误的发生是因为 `let` / `const` 声明把它们的声明提升到了作用域的顶端，但与 `var` 不同的是，它们把变量的自动初始化推迟到了代码排序中原始声明出现的时刻。这个时间窗口（提示：时间性），无论其长度如何，就是 TDZ。

如何避免 TDZ 错误？

我的建议是：始终将 `let` 和 `const` 声明放在任何作用域的顶部。将 TDZ 窗口缩小到零（或接近零）长度，这样就没可以避免了。

但为什么还要有 TDZ 呢？为什么 TC39 没有规定 `let` / `const` 像 `var` 那样自动初始化？请耐心等待，我们会在附录 A 中再来探讨 TDZ 的*为什么*。

终于初始化

变量的使用远比乍看起来要细微得多。*提升*、*（重复）声明* 和 *TDZ* 是开发人员常见的困惑来源，尤其是那些在学习 JS 之前使用过其他语言的开发人员。在继续学习之前，请确保您的心智模型已完全建立在 JS 作用域和变量的这些方面之上。

提升通常被认为是 JS 引擎的一种显式机制，但实际上它更像是一种隐喻，用来描述 JS 在编译过程中处理变量声明的各种方式。但即使作为一种隐喻，提升也为思考变量的生命周期（何时创建、何时可用、何时消失）提供了有用的架构。

如果将变量的声明和重复声明视为运行时操作，往往会造成混乱。但如果将这些操作转换为编译时思维，这些怪异现象和*遮蔽*就会减少。

TDZ（temporal dead zone，暂时性死区）错误既奇怪又令人沮丧。幸运的是，如果您总是小心地将 `let` / `const` 声明放在任何作用域的顶部，TDZ 是可以相对直接地避免的。

当您成功地驾驭这些变量作用域时，下一章将介绍指导我们决定将声明置于不同作用域（尤其是嵌套块）的因素。

第六章：限制作用域的过度暴露

到目前为止，我们的重点是解释作用域和变量的工作原理。有了坚实的基础，我们的注意力将转移到更高层次的思考上：我们应用于整个程序的决策和模式。

首先，我们要看看如何以及为什么要使用不同级别的作用域（函数和块）来组织程序的变量，特别是要减少作用域的过度暴露。

最小暴露

函数定义自己的作用域是合情合理的。但为什么我们还需要块来创建作用域呢？

软件工程阐明了一门基本学科，通常应用于软件安全，称为「最小权限原则」(POLP)。[1]而适用于我们当前讨论的这一原则的变体通常被称为「最小暴露」(POLE)。

POLP 表达了对软件架构的一种防御姿态：系统组件应设计为以最小权限、最小访问、最小暴露的方式运行。如果每个组件都以最低限度的必要功能连接起来，那么从安全的角度来看，整个系统就会更强大，因为一个组件的受损或失效对系统其他组件的影响就会降到最低。

如果说 POLP 侧重于系统级的组件设计，那么 POLE *暴露*变体则侧重于更低层次；我们将把它应用于作用域之间的交互。

在遵循 POLE 的过程中，我们要尽量减少哪些变量的暴露？很简单：在每个作用域中注册的变量。

你可以这样想：为什么不把程序中的所有变量都放在全局作用域中呢？这可能会让人立刻觉得这是个坏主意，但值得考虑一下为什么会这样。当程序的一部分使用的变量通过作用域暴露给程序的另一部分时，通常会产生三大危害：

- **命名碰撞**：如果在程序的两个不同部分中使用了一个共同的、有用的变量/函数名，但标识符来自一个共享作用域（如全局作用域），那么就会发生命名碰撞，很可能会出现错误，因为一部分以另一部分意想不到的方式使用了该变量/函数。

  例如，试想一下，如果您的所有循环都使用了一个全局 `i` 索引变量，然后一个函数中的一个循环在另一个函数的循环迭代期间运行，现在共享的 `i` 变量得到了一个意想不到的值。

- **意想不到的行为**：如果你将变量/函数的用途公开，而这些变量/函数的用途在程序中是*私有*的，那么其他开发人员就可以用你没有想到的方式使用它们，这可能会违反预期行为并导致错误。

  例如，如果你的程序假定数组包含所有数字，但其他人的代码访问并修改了数组，使其包含布尔和字符串，那么你的代码可能会出现意想不到的错误行为。

更糟糕的是，暴露*隐私*细节会招致那些居心不良的人试图绕过你施加的限制，用你的那部分软件做一些不该做的事情。

- **意外的依赖**：如果你不必要地暴露变量/函数，就会招致其他开发人员使用和依赖这些原本*私人*的部分。虽然这不会破坏你现在的程序，但会给将来的重构带来隐患，因为现在你不可能轻易地重构该变量或函数，而不会破坏软件中你无法控制的其他部分。[2]

  例如，如果您的代码依赖于数字数组，而您后来决定最好使用其他数据结构而不是数组，那么您现在就必须承担调整软件其他受影响部分的责任。

POLE 应用于变量/函数的作用域时，主要是说，默认情况下只暴露必要的最低限度，其他一切尽可能保持私有。在尽可能小的深度嵌套作用域中声明变量，而不是把所有东西都放在全局（甚至外层函数）作用域中。

如果在设计软件时考虑到这一点，就有更大的机会避免（或至少减少）这三种危害。

思考一下：

```js
function diff(x, y) {
    if (x > y) {
        let tmp = x;
        x = y;
        y = tmp;
    }

    return y - x;
}

diff(3, 7); // 4
diff(7, 5); // 2
```

在这个 `diff(..)` 函数中，我们要确保 `y` 大于或等于 `x`，这样当我们减去（`y-x`），结果就是 `0` 或更大。如果 `x` 最初较大（结果将为负数！），我们将使用 `tmp` 变量交换 `x` 和 `y`，以保持结果为正数。

在这个简单的例子中，`tmp` 是否在 `if` 代码块中或是否属于函数级似乎并不重要。它当然不应该是一个全局变量！然而，根据 POLE 原则，`tmp` 应尽可能隐藏在作用域中。因此，我们将 `tmp` 的作用域屏蔽（使用 `let` ）到 `if` 代码块中。

隐藏在普通（函数）作用域内

现在我们应该清楚为什么要尽可能将变量和函数声明隐藏在最低（嵌套最深）的作用域中了。但我们该如何做呢？

我们已经看到了 `let` 和 `const` 关键字，它们是块作用域声明符；稍后我们将更详细地讨论它们。但首先，如何在作用域中隐藏 `var` 或 `function` 声明呢？这可以通过在声明周围包装一个 `function` 作用域来轻松实现。

让我们举一个例子来说明 `function` 作用域的作用。

数学运算「阶乘」[3]（写为 "6!"）是一个给定整数与所有依次低至 `1` 的整数相乘—实际上，你可以停在 `2` ，因为与 `1` 相乘没有任何作用。换句话说，"6!" 等同于 "6 *5!*" ，*同样等同于* "*6* 5 * 4!"，以此类推。由于所涉及数学的性质，一旦计算出任何给定整数的阶乘（如 "4!"），我们就不需要再做这项工作了，因为答案总是一样的。

因此，如果你天真地计算了 `6` 的阶乘，然后又想计算 `7` 的阶乘，你可能会不必要地重新计算从 2 到 6 的所有整数的阶乘。如果你愿意用内存来换取速度，你可以在计算时缓存每个整数的阶乘来解决这种浪费：

```js
var cache = {};

function factorial(x) {
    if (x < 2) return 1;
    if (!(x in cache)) {
        cache[x] = x * factorial(x - 1);
    }
    return cache[x];
}

factorial(6);
// 720

cache;
// {
//     "2": 2,
//     "3": 6,
//     "4": 24,
//     "5": 120,
//     "6": 720
// }

factorial(7);
// 5040
```

我们将所有计算出的阶乘存储在 `cache` 中，这样在多次调用 `factorial(..)` 时，之前的计算结果就会保留下来。但 `cache` 变量显然是 `factorial(..)` 工作原理中的一个*私有*细节，不应该暴露在外部作用域中，尤其是全局作用域。

> **注意：**
>
> 这里的 `factorial(..)` 是递归的—从内部调用自身，但这只是为了代码的简洁；非递归的实现会产生与 `cache` 相同的作用域。

然而，要解决这个过度暴露的问题，并不像看起来那样简单，只需将 `cache` 变量隐藏在 `factorial(..)` 中即可。因为我们需要 `cache` 在多次调用后仍能存活，所以它必须位于该函数之外的作用域中。那么我们该怎么办呢？

为 `cache` 定义另一个中间作用域（介于外部/全局作用域和 `factorial(..)` 内部之间）：

```js
// 外层/全局作用域

function hideTheCache() {
    // "中间作用域"，在这里我们隐藏了 `cache`
    var cache = {};

    return factorial;

    // **********************

    function factorial(x) {
        // 内部作用域
        if (x < 2) return 1;
        if (!(x in cache)) {
            cache[x] = x * factorial(x - 1);
        }
        return cache[x];
    }
}

var factorial = hideTheCache();

factorial(6);
// 720

factorial(7);
// 5040
```

`hideTheCache()` 函数除了为 `cache` 创建一个作用域，以便在多次调用 `factorial(..)` 时持续存在之外，没有其他作用。但为了让 `factorial(..)` 能够访问 `cache`，我们必须在同一个作用域中定义 `factorial(..)`。然后，我们从 `hideTheCache()` 返回函数引用作为值，并将其存储在外作用域变量中，该变量也命名为 `factorial`。现在，当我们调用 `factorial(..)`（多次！）时，其持久的 `cache` 将保持隐藏，但只有 `factorial(..)` 可以访问！

好吧，但是……每次需要隐藏变量/函数时，都要定义（并命名！）`hideTheCache(..)` 函数作用域，这将会非常繁琐，尤其是我们可能希望通过为每次出现的函数命名一个唯一的名称来避免名称冲突。唉。

> **注意：**
>
> 图解技术：缓存函数的计算输出[4]中提到，重复调用相同输入时优化性能在函数式编程（FP）领域非常常见，通常称为「记忆函数（memoization）」；这种缓存依赖于闭包（参见第 7 章）。此外，还有内存使用方面的问题（在附录 B 的「内存问题」中讨论）。FP 库通常会为函数的内存化提供一个经过优化和审查的实用程序，在这里它将取代 `hideTheCache(..)`。记忆函数超出了我们的讨论*范围*（双关语！*scope* 译作范围的同时也可理解为作用域），更多信息请参阅我（原作者）的 *Functional-Light JavaScript*[5] 一书。

与每次都定义一个新的、唯一命名的函数相比，也许更好的解决方案是使用函数表达式：

```js
var factorial = (function hideTheCache() {
    var cache = {};

    function factorial(x) {
        if (x < 2) return 1;
        if (!(x in cache)) {
            cache[x] = x * factorial(x - 1);
        }
        return cache[x];
    }

    return factorial;
})();

factorial(6);
// 720

factorial(7);
// 5040
```

等等！这仍然是在使用一个函数来创建隐藏 `cache` 的作用域，而且在这种情况下，该函数仍然被命名为 `hideTheCache`，这又能解决什么问题呢？

回顾一下「函数名的作用域」（第 3 章），函数表达式的名称标识符会发生什么变化。由于 `hideTheCache(..)` 被定义为一个 `function` 表达式而不是一个 `function` 声明，所以它的名称在它自己的作用域中，基本上与 `cache` 相同而不是在外层/全局作用域中。

这意味着，我们可以将此类函数表达式的每一次出现都命名为完全相同的名称，而绝不会出现任何冲突。更恰当地说，我们可以根据我们试图隐藏的内容，对每个出现的表达式进行语义命名，而不必担心我们选择的任何名称会与程序中的任何其他函数表达式作用域相冲突。

事实上，我们完全*可以*不使用名称，而是定义一个「匿名函数表达式」。但附录 A 将讨论名称的重要性，即使是对于这种 scope-only 的函数。

立即调用函数表达式 (IIFE)

在前面的阶乘递归程序中，还有一个很容易被忽略的重要部分：函数表达式末尾包含 `})();` 的那一行。

请注意，我们用一组 `( .. )` 包围了整个函数表达式，然后在最后添加了第二组 `()` 括号；这实际上是在调用我们刚刚定义的函数表达式。此外，在这种情况下，函数表达式周围的第一组 `( .. )` 严格来说并不是必需的（稍后会详细说明），但为了便于阅读，我们还是使用了它们。

换句话说，我们正在定义一个函数表达式，然后立即调用它。这种常见的模式有一个非常有创意的名字：立即调用函数表达式 (IIFE)。

当我们想创建一个隐藏变量/函数的作用域时，IIFE 就会派上用场。因为它是一个表达式，所以可以用在 JS 程序中任何允许使用表达式的地方。IIFE 可以命名，如 `hideTheCache()`，也可以（更常见！）不命名/匿名。它可以是独立的，也可以是另一个语句的一部分：`hideTheCache()` 返回 `factorial()` 函数引用，然后 `=` 赋值给变量 `factorial`。

下面是一个独立 IIFE 的示例，以资比较：

```js
// 外层作用域

(function () {
    // 内部隐藏作用域
})();

// 更多外层作用域
```

与之前的 `hideTheCache()` 不同，外围的 `(..)` 被认为是一种可选的样式选择，而对于独立的 IIFE，它们是 **必须的**；它们将 `function` 区分为一个表达式，而不是一个语句。不过，为了保持一致，在 IIFE `function` 的周围一定要加上 `( .. )` 。

> **注意：**
>
> 从技术上讲，周围的 `( .. )` 并不是确保 JS 解析器将 IIFE 中的 `function` 视为函数表达式的唯一语法方式。我们将在附录 A 中查看其他一些选择。

函数边界

请注意，使用 IIFE 定义作用域可能会产生一些意想不到的后果，这取决于它周围的代码。因为 IIFE 是一个完整的函数，函数边界会改变某些语句/结构体的行为。

例如，如果代码中的 `return` 语句被 IIFE 包住，它的含义就会改变，因为现在 `return` 指的是 IIFE 的函数。非箭头函数 IIFE 也会改变 `this` 关键字的绑定，更多内容请参阅*对象与类*一书。像 `break` 和 `continue` 这样的语句不会跨越 IIFE 函数边界来控制外循环或代码块。

因此，如果您需要封装作用域的代码中包含 `return`、`this`、`break` 或 `continue` 字符，那么 IIFE 可能不是最好的方法。在这种情况下，您可以使用代码块而不是函数来创建作用域。

块的作用域

至此，您应该对创建作用域以限制标识符暴露的优点感到相当满意了。

到目前为止，我们已经了解了如何通过函数（即 IIFE）作用域来实现这一点。现在我们来考虑使用嵌套代码块中的 `let` 声明。一般来说，任何 `{ .. }` 作为语句的大括号对将作为一个块，但**不一定**作为作用域。

块只有在必要时才成为作用域，以包含其块作用域声明（即 `let` 或 `const`）。思考一下：

```js
{
    // 不一定（尚未）是作用域

    // ..

    // 现在我们知道该区块需要是一个作用域
    let thisIsNowAScope = true;
```

```
    for (let i = 0; i < 5; i++) {
        // 每次循环这也是一个作用域
        // 迭代
        if (i % 2 == 0) {
            // 这只是一个块，不是作用域
            console.log(i);
        }
    }
}
// 0 2 4
```

并非所有的 `{ .. }` 大括号都会创建块（因此有资格成为作用域）：

- 对象字面使用 `{ .. }` 大括号对来分隔键值对，但这种对象值**不是**作用域。
- `class` 在其主体定义周围使用了 `{ .. }` 大括号，但这不是一个块或作用域。
- `function` 在其主体周围使用 `{ .. }` 围绕着它的主体，但从技术上讲，这并不是一个块：它是函数主体的单个语句。但它*是*一个（函数）作用域。
- `switch` 语句上的一对 `{ .. }` 大括号（围绕 `case` 子句集）并不定义块/作用域。

除了这些非代码块示例之外，`{ .. }` 大括号对可以定义一个附在语句（如 `if` 或 `for`）之后的块，也可以单独定义，见上一段中最外层的 `{ .. }` 括号对。对于这种显式块如果它没有声明，实际上就不是一个作用域，没有任何操作上的用途，尽管它作为语义信号仍然有用。

明确的独立 `{ .. }` 块一直都是有效的 JS 语法，但由于在 ES6 的 `let`/`const` 之前它们不能作为作用域，所以它们非常罕见。不过，在 ES6 之后，它们开始逐渐流行起来。

在大多数支持块作用域的语言中，显式块作用域是一种极为常见的模式，用于为一个或几个变量创建狭小的作用域。因此，根据 POLE 原则，我们也应该在 JS 中更广泛地采用这种模式；使用（显式）块作用域将标识符的暴露范围缩小到最小。

即使在另一个块内部，显式块作用域也是有用的（无论外部块是否是一个作用域）。

例如：

```js
if (somethingHappened) {
    // 这是一个块但不是作用域

    {
        // 这既是一个块
        // 也是一个显式作用域
        let msg = somethingHappened.message();
        notifyOthers(msg);
    }
```

```
    // ..

    recoverFromSomething();
}
```

在这里，`{ .. }` 大括号对在 `if` 语句的**内部**，是 `msg` 的一个更小的内部显式代码块作用域，因为整个 `if` 代码块都不需要该变量。大多数开发人员都会直接将 `msg` 代码块作用到 `if` 代码块中，然后继续开发。公平地说，当只需要考虑几行代码时，这是一个折腾人的判断。但随着代码的增加，这些过度暴露的问题就会变得更加明显。

那么，添加额外的 `{ .. }` 和缩进级别是否足够重要？我认为您应该遵循 POLE，始终（在合理作用域内）为每个变量定义最小的块。因此，我建议使用如图所示的额外显式块作用域。

回顾「未初始化变量警告（TDZ）」（第 5 章）中对 TDZ 误差的讨论。我在那里提出的建议是：为了尽量减少使用 `let` / `const` 声明时出现 TDZ 错误的风险，应始终将这些声明放在其作用域的顶端。

如果你发现自己在一个作用域中间放置了一个 `let` 声明，首先要想的是「哦，不！TDZ 警报！」如果该代码块的前半部分不需要这个 `let` 声明，您就应该使用内部显式代码块作用域来进一步缩小它的暴露范围！

另一个明确块作用域的例子：

```js
function getNextMonthStart(dateStr) {
    var nextMonth, year;

    {
        let curMonth;
        [, year, curMonth] = dateStr.match(/(\d{4})-(\d{2})-\d{2}/) || [];
        nextMonth = (Number(curMonth) % 12) + 1;
    }

    if (nextMonth == 1) {
        year++;
    }

    return `${year}-${String(nextMonth).padStart(2, "0")}-01`;
}
getNextMonthStart("2019-12-25"); // 2020-01-01
```

首先让我们来确定作用域及其标识符：

1. 外部/全局作用域有一个标识符，即函数 `getNextMonthStart(..)`。

2. `getNextMonthStart(..)` 的函数作用域有三个：`dateStr`（参数）、`nextMonth` 和 `year`。
3. `{ .. }` 大括号对定义了一个包含一个变量的内部块作用域：`curMonth`。

那么，为什么要将 `curMonth` 放在显式的块作用域中，而不是在顶级函数作用域中与 `nextMonth` 和 `year` 放在一起呢？因为 `curMonth` 只需要用于前两个语句；在函数作用域级别，它已经过度暴露了。

这个示例很小，因此过度暴露 `curMonth` 的危害非常有限。但是，当您把默认情况下最小化作用域暴露作为一种习惯时，POLE 原则的好处就能发挥得淋漓尽致。如果您能始终如一地遵循该原则，即使是在很小的情况下，随着程序的发展，它也会为您提供更多的作用。

现在我们来看一个更有说服力的例子：

```js
function sortNamesByLength(names) {
    var buckets = [];

    for (let firstName of names) {
        if (buckets[firstName.length] == null) {
            buckets[firstName.length] = [];
        }
        buckets[firstName.length].push(firstName);
    }

    // 一个块，以缩小作用域
    {
        let sortedNames = [];

        for (let bucket of buckets) {
            if (bucket) {
                // 按字母数字对每个桶进行排序
                bucket.sort();

                // 将排序后的名称添加到运行列表中
                sortedNames = [...sortedNames, ...bucket];
            }
        }

        return sortedNames;
    }
}

sortNamesByLength(["Sally", "Suzy", "Frank", "John", "Jennifer", "Scott"]);
```

```
// [ "John", "Suzy", "Frank", "Sally",
//   "Scott", "Jennifer" ]
```

在五个不同的作用域中声明了六个标识符。这些变量是否都可以存在于一个外部/全局作用域中？从技术上讲，是可以的，因为它们的名称都是唯一的，因此不会发生名称冲突。但这样的代码组织实在是太糟糕了，很可能会导致混乱和将来的错误。

我们根据需要将它们分割到每个内部嵌套作用域中。每个变量都定义在程序运行所需的最内层作用域。

`sortedNames` 本来可以在顶层函数作用域中定义，但这个函数的后半部分才需要它。为了避免在更高级的作用域中过度暴露该变量，我们再次遵循 POLE 的做法，在内部显式块作用域中对其进行块作用域定义。

`var` 和 `let`

接下来，让我们来谈谈 `var buckets` 声明。该变量用于整个函数（除了最后的 `return` 语句）。任何需要在函数的全部（甚至大部分）部分使用的变量，都应该声明得一目了然。

> **注意：**
>
> 参数 `names` 并未在整个函数中使用，但没有办法限制参数的作用域，因此无论如何，它都是函数作用域内的声明。

那么，为什么我们使用 `var` 而不是 `let` 来声明 `buckets` 变量呢？选择 `var` 既有语义上的原因，也有技术上的原因。

在风格上，`var` 从最早的 JS 开始就表示「属于整个函数的变量」。正如我们在「词法作用域」（第 1 章）中所断言的，`var` 不管出现在哪里，都会依附于最近的外层函数作用域。即使 `var` 出现在一个块中也是如此：

```js
function diff(x, y) {
    if (x > y) {
        var tmp = x; // `tmp` 是函数作用域
        x = y;
        y = tmp;
    }

    return y - x;
}
```

即使 `var` 位于代码块内部，它的声明也是函数作用域的（对 `diff(..)`），而不是代码块作用域的。

虽然您可以在代码块中声明 `var`（并且它仍然是函数作用域），但我建议您不要采用这种方法，除非在少数特殊情况下（在附录 A 中讨论）。否则，`var` 应保留给函数的顶级作用域使用。

为什么不在同一位置使用 `let` 呢？因为 `var` 与 `let` 在视觉上截然不同，因此可以清楚地表明「此变量属于函数作用域」。在顶层作用域中使用 `let`，尤其是如果不是在函数的前几行中使用，并且在代码块中的所有其他声明都使用 `let` 时，并不会在视觉上引起人们注意与函数作用域声明的区别。

换句话说，我觉得 `var` 比 `let` 更好地传达了函数作用域，而 `let` 既传达了（也实现了！）块作用域，而 `var` 则不够。只要你的程序既需要函数作用域变量又需要块作用域变量，那么最明智、最易读的方法就是同时使用 `var` *和* `let` 这两种变量，它们各自都有自己的最佳用途。

在不同的情况下，选择 `var` 或 `let` 还有其他语义和操作上的原因。我们将在附录 A 中更详细地探讨 `var` *和* `let` 的情况。

> **警告：**
>
> 我建议同时使用 `var` *和* `let` 显然是有争议的，而且与大多数人的观点相悖。更常见的说法是：「var 是坏的，let 能修复它」和「永远不要使用 var，let 是替代品」。这些观点都有道理，但它们只是观点，就像我的观点一样。事实上，`var` 并没有坏掉或被弃用；它从早期的 JS 开始就一直有效，只要 JS 还在，它就会一直有效。

在哪里使用 `let`？

我建议将 `var` 保留给（大部分）仅用于顶级函数作用域，这意味着其他大多数声明都应使用 `let`。但您可能仍想知道如何决定程序中每个声明的归属？

POLE 已经为您的决策提供了指导，但我们还是要明确指出这一点。决定的方式不是基于你想使用哪个关键字。决定的方法是问：「对这个变量来说，最起码的作用域暴露是什么？」

一旦回答了这个问题，你就会知道变量属于块作用域还是函数作用域。如果一开始决定某个变量应属于块作用域，但后来意识到需要将其提升为函数作用域，那么不仅需要改变变量的声明位置，还需要改变所使用的声明器关键字。决策过程确实应该这样进行。

如果声明属于块作用域，则使用 `let`。如果声明属于函数作用域，则使用 `var`（再次重申，这只是我的观点）。

但另一种可视化决策的方法是考虑程序的 ES6 前版本。例如，让我们回顾一下之前的 `diff(..)` ：

```js
function diff(x, y) {
    var tmp;

    if (x > y) {
        tmp = x;
        x = y;
        y = tmp;
    }

    return y - x;
}
```

在这个版本的 `diff(..)` 中，`tmp` 被清楚地声明在函数作用域中。这对 `tmp` 来说合适吗？我认为不合适。`tmp` 只需要在这几个语句中使用。在 `return` 语句中不需要它。因此，它应该是块作用域。

在 ES6 之前，我们没有`let` ，因此无法对其进行实际的块作用域化。不过，我们可以做一个次好的事情来表明我们的意图：

```js
function diff(x, y) {
    if (x > y) {
        // `tmp` 仍然在函数作用域
        // 但放置在这里
        // 在语义上表示块作用域
        var tmp = x;
        x = y;
        y = tmp;
    }

    return y - x;
}
```

将 `tmp` 的 `var` 声明放在 `if` 语句中，向代码读者发出了 `tmp` 属于该代码块的信号。尽管 JS 并不强制执行该作用域定义，但语义信号仍会给代码读者带来好处。

从这个角度看，你可以找到这种代码块中的任何 `var` 并将其切换为 `let` 以执行已经发送的语义信号。在我看来，这才是 `let` 的正确用法。

另一个历史上使用 `var` 但现在几乎总是使用 `let` 的例子是 `for` 循环：

```js
for (var i = 0; i < 5; i++) {
    // 做点什么
}
```

无论在哪里定义了这样的循环，`i` 基本上都只能在循环内部使用，在这种情况下，POLE 规定应该用 `let` 而不是 `var` 来声明它：

```js
for (let i = 0; i < 5; i++) {
    // 做点什么
}
```

将 `var` 替换为 `let` 几乎是唯一会「破坏」代码的情况，那就是在循环之外/之后访问循环的迭代器（`i`），例如：

```js
for (var i = 0; i < 5; i++) {
    if (checkValue(i)) {
        break;
    }
}

if (i < 5) {
    console.log("The loop stopped early!");
}
```

这种使用模式并不罕见，但大多数人都觉得它的代码结构很糟糕。更可取的方法是使用另一个外层作用域变量：

```js
var lastI;

for (let i = 0; i < 5; i++) {
    lastI = i;
    if (checkValue(i)) {
        break;
    }
}

if (lastI < 5) {
    console.log("The loop stopped early!");
}
```

整个作用域都需要 `lastI`，所以用 `var` 声明。只有在（每次）循环迭代中才需要 `i`，所以用 `let` 声明。

Catch 是什么?

到目前为止，我们已经断言 `var` 和参数是函数作用域，而 `let`/`const` 则是块作用域声明。有一个小例外需要指出：`catch` 子句。

自从在 ES3（1999 年）中引入 `try..catch` 以来，`catch` 子句一直使用附加的（鲜为人知的）块作用域声明功能：

```js
try {
    doesntExist();
} catch (err) {
    console.log(err);
    // ReferenceError: 'doesntExist' is not defined
    // ^^^^ 从捕获的异常中打印的消息

    let onlyHere = true;
    var outerVariable = true;
}

console.log(outerVariable); // true

console.log(err);
// ReferenceError: 'err' is not defined
// ^^^^ 这是另一个抛出（未捕获）的异常
```

由 `catch` 子句声明的 `err` 变量是该代码块的块作用域。这个 `catch` 子句块可以通过 `let` 保存其他块作用域声明。但该代码块中的 `var` 声明仍会附加到外层函数/全局作用域。

ES2019（最近，在撰写本文时）更改了 `catch` 子句，使其声明成为可选项；如果省略声明，则 `catch` 代码块不再是（默认情况下）作用域；但它仍然是一个代码块！

因此，如果您需要对异常发生的条件做出反应（以便优雅地恢复），但不关心错误值本身，则可以省略 `catch` 声明：

```js
try {
    doOptionOne();
} catch {
    // 省略 catch 的声明
```

```
    doOptionTwoInstead();
}
```

对于一个相当常见的用例来说，这是对语法的一个微小但却令人高兴的简化，而且在删除不必要的作用域方面也可能略胜一筹！

块中的声明函数 (FiB)

我们已经看到，使用 `let` 或 `const` 的声明是块作用域，而 `var` 声明是函数作用域。那么，直接出现在代码块中的 `function` 声明又是怎么回事呢？作为一种特性，这被称为 "FiB"。

我们通常认为 `function` 声明等同于 `var` 声明。那么，它们的函数作用域是否与 `var` 相同？

是又不是。我知道……这很令人困惑。让我们深入了解一下：

```js
if (false) {
    function ask() {
        console.log("Does this run?");
    }
}
ask();
```

您期望该程序做些什么？三个合理的结果：

1. 调用 `ask()` 时可能会出现 `ReferenceError` 异常，因为 `ask` 标识符是块作用域中的 `if` 块作用域，因此在外部/全局作用域中不可用。
2. `ask()` 调用可能会因 `TypeError` 异常而失败，因为 `ask` 标识符虽然存在，但它是 `undefined` 的（因为 `if` 语句没有运行），因此不是一个可调用函数。
3. `ask()` 调用可能会正确运行，并打印出 "Does this run?"

令人困惑的是：根据您在哪个 JS 环境中尝试该代码片段，可能会得到不同的结果！这是现有遗留行为暴露出可预测结果的少数几个疯狂领域之一。

根据 JS 规范，块内的 `function` 声明具有块作用域，因此答案应该是（1）。然而，大多数基于浏览器的 JS 引擎（包括 v8，它来自 Chrome 浏览器，但也用于 Node）会表现为（2），这意味着标识符的作用域在 `if` 块之外，但函数值不会自动初始化，因此它仍然是 `undefined`。

为什么允许浏览器 JS 引擎的行为违背规范？因为在 ES6 引入块作用域之前，这些引擎就已经有了与 FiB 相关的某些行为，而且有人担心，如果修改为遵守规范，可能会破坏某些现有的网站 JS 代码。因此，在 JS 规范的附录 B 中规定了例外情况，允许浏览器 JS 引擎（仅限！）做出某些偏离。

> **注意：**
>
> 你通常不会把 Node 归类为浏览器 JS 环境，因为它通常在服务器上运行。但 Node 的 v8 引擎与 Chrome（和 Edge）浏览器共享。由于 v8 首先是一个浏览器 JS 引擎，因此它采用了附录 B 中的例外，这就意味着浏览器例外也扩展到了 Node。

在代码块中放置 `function` 声明的最常见用例之一，是根据某些环境状态，以某种方式有条件地定义函数（如使用 `if..else` 语句）。例如：

```js
if (typeof Array.isArray != "undefined") {
    function isArray(a) {
        return Array.isArray(a);
    }
} else {
    function isArray(a) {
        return Object.prototype.toString.call(a) == "[object Array]";
    }
}
```

出于性能考虑，这样构建代码很有诱惑力，因为 `typeof Array.isArray` 检查只执行一次，而不是只定义一个 `isArray(..)` 并将 `if` 语句放在其中，这样每次调用时都会执行不必要的检查。

> **警告：**
>
> 除了 FiB 偏差的风险之外，函数的条件定义的另一个问题是调试这样的程序更加困难。如果最终在 `isArray(..)` 函数中发现了一个错误，那么首先必须弄清楚 `isArray(..)` 的实际运行情况！有时，错误是由于条件检查不正确而应用了错误的实现！如果定义了一个函数的多个版本，程序的推理和维护就会变得更加困难。

除了前面的片段，还潜伏着其他一些 FiB 情况；在各种浏览器和非浏览器 JS 环境（非基于浏览器的 JS 引擎）中，此类行为可能会有所不同。例如：

```js
if (true) {
    function ask() {
        console.log("Am I called?");
    }
```

```
}

if (true) {
    function ask() {
        console.log("Or what about me?");
    }
}

for (let i = 0; i < 5; i++) {
    function ask() {
        console.log("Or is it one of these?");
    }
}

ask();

function ask() {
    console.log("Wait, maybe, it's this one?");
}
```

回想一下「何时使用变量？」（第 5 章）中所述，本代码段中最后一个以 "Wait, maybe..." 为信息的 `ask()` 函数应该提升在 `ask()` 的调用之上。因为它是该名称的最后一个函数声明，所以它应该「赢」，对吗？很遗憾，不是。

我无意记录所有这些奇怪的角落情况，也不想解释它们为什么会以某种方式出现。在我看来，这些信息都是神秘的遗产[6]。

对于 FiB，我真正关心的是，我能给出什么建议来确保你的代码在任何情况下都能可预测地运行？

在我看来，要避免 FiB 的变幻莫测，唯一切实可行的办法就是完全避免 FiB。换句话说，永远不要将 `function` 声明直接放在任何代码块中。始终将 `function` 声明放在函数顶层作用域（或全局作用域）的任何地方。

因此，对于前面的 `if..else` 示例，我的建议是尽可能避免有条件地定义函数。是的，这可能会稍微降低性能，但这是顾全大局的方法：

```js
function isArray(a) {
    if (typeof Array.isArray != "undefined") {
        return Array.isArray(a);
    } else {
        return Object.prototype.toString.call(a) == "[object Array]";
    }
}
```

如果性能下降成为应用程序的关键路径问题，我建议您考虑这种方法：

```js
var isArray = function isArray(a) {
    return Array.isArray(a);
};

// 如果必要的话，覆盖该定义
if (typeof Array.isArray == "undefined") {
    isArray = function isArray(a) {
        return Object.prototype.toString.call(a) == "[object Array]";
    };
}
```

需要注意的是，这里我在 `if` 语句中放置了一个 `function` **表达式**，而不是一个声明。在代码块中出现 `function` 表达式是完全正确的。我们关于 FiB 的讨论是关于避免在代码块中使用 `function` **声明**。

即使您测试了您的程序并能正确运行，但在代码中使用 FiB 样式所带来的微小好处，远不能抵消将来被其他开发人员混淆的潜在风险，或您的代码在其他 JS 环境中运行的差异。

FiB 不值得一试，应该避免它。

刀过竹解

在编程语言中使用词法作用域规则的意义在于，我们可以适当地组织程序中的变量，以达到操作和语义代码交流的目的。

最重要的组织技巧之一就是确保没有变量过度暴露在不必要的作用域（POLE）中。希望你现在对块作用域的理解比以前更深刻了。

希望到此为止，你对词法作用域的理解已经有了更坚实的基础。在此基础上，下一章将进入闭包这一重要话题。

---

1. *最小权限原则*， https://zh.wikipedia.org/wiki/最小权限原则， 2021年12月26日。 ↩
2. *苹果底层开源代码被发现包含兼容微信的代码*，https://www.oschina.net/news/127177/apple-libmalloc-compliance-with-wechat，2021年1月20日。 ↩
3. *阶乘*， https://zh.wikipedia.org/zh-cn/階乘， 2023年8月23日。 ↩

4. *Incremental Computation via Function Caching*，https://dl.acm.org/doi/pdf/10.1145/75277.75305， 1989年。 ↩

5. *getify/Functional-Light-JS: Pragmatic, balanced FP in JavaScript*，https://github.com/getify/Functional-Light-JS， 2016年8月17日。 ↩

6. *代码异味*， https://zh.wikipedia.org/zh-cn/代码异味， 2021年12月16日。 ↩

第七章：闭包的使用

目前，我们主要介绍了词法作用域的来龙去脉，以及它如何影响程序中变量的组织和使用。

我们的注意力再次从抽象的角度转移到历史上令人生畏的「闭包」话题上。别担心！你并不需要一个高级计算机科学学位来理解它。我们在本书中的总体目标不仅仅是理解作用域，而是在我们的程序结构中更有效地使用它；闭包是这一目标的核心。

回顾第 6 章的主要结论：*最小暴露* (POLE) 鼓励我们使用块（和函数）作用域来限制变量的作用域暴露。这有助于保持代码的可理解性和可维护性，并有助于避免许更多的作用域陷阱（如命名冲突等）。

闭包基于这种方法：对于我们需要长期使用的变量，我们可以将它们封装起来（更小的作用域），而不是将它们放在更大的外部作用域中，但仍保留函数内部的访问权限，以便更广泛地使用。函数通过闭包*记住*这些被引用的作用域变量。

我们在上一章中已经看到过这种闭包的示例（第 6 章中的 `factorial(..)`），几乎可以肯定，你已经在自己的程序中使用过它了。如果你写过一个回调函数，它可以访问自身作用域之外的变量……你猜怎么着？这就是闭包。

闭包是编程领域有史以来最重要的语言特点之一，它是包括函数式编程（FP）、模块甚至一些面向类的设计在内的主要编程范式的基础。要掌握 JS 并在整个代码中有效利用许多重要的设计模式，就必须熟悉闭包。

在本章中，需要讨论和编写大量代码，才能对闭包有全面的了解。请务必花点时间，确保在进入下一章节之前对每一部分都能得心应手。

来看看闭包

闭包原本是一个数学概念，来自 lambda 微积分。但我不会列出数学公式或使用一堆符号和术语来定义它。

相反，我将从实用的角度出发。首先，我们将从定义闭包开始，根据我们在程序的不同行为中可以观测到何如来定义闭包，而不是如果闭包在 JS 中不存在。不过，在本章的后面部分，我们将从*另一个角度*来看待闭包。

闭包是函数的一种行为，也只能是函数的一种行为。如果你处理的不是函数，闭包就不适用。对象不具有闭包性，类也不具有闭包性（尽管其函数/方法可能具有闭包性）。只有函数才有闭包。

要观测到闭包，函数必须被调用，具体地说，它必须在与最初定义它的作用域链不同的分支中被调用。在定义的同一作用域中执行的函数，无论是否可能闭包，都不会表现出任何可观测到的不同行为；根据观测的角度和定义，这不是闭包。

让我们来看一些代码，并用相关的作用域气泡颜色加以注释（见第 2 章）：

```js
// 外部/全局作用域：红色(1)

function lookupStudent(studentID) {
    // 函数作用域：蓝色(2)

    var students = [
        { id: 14, name: "Kyle" },
        { id: 73, name: "Suzy" },
        { id: 112, name: "Frank" },
        { id: 6, name: "Sarah" },
    ];

    return function greetStudent(greeting) {
        // 函数作用域：绿色(3)

        var student = students.find((student) => student.id == studentID);

        return `${greeting}, ${student.name}!`;
    };
}

var chosenStudents = [lookupStudent(6), lookupStudent(112)];

// 访问函数名：
chosenStudents[0].name;
// greetStudent

chosenStudents[0]("Hello");
// Hello, Sarah!

chosenStudents[1]("Howdy");
// Howdy, Frank!
```

这段代码首先要注意的是，`lookupStudent(..)` 外层函数创建并返回一个名为 `greetStudent(..)` 的内层函数。`lookupStudent(..)` 被调用了两次，产生了两个独立的内部函数 `greetStudent(..)` 实例，这两个实例都被保存到 `chosenStudents` 数组中。

我们通过检查保存在 `chosenStudents[0]` 中的返回函数的 `.name` 属性来验证，它确实是内部 `greetStudent(..)` 的实例。

每次调用 `lookupStudent(..)` 结束后，它的所有内部变量似乎都会被丢弃和 GC（垃圾回收）。内部函数似乎是唯一被返回和保留的东西。但我们可以从这里开始观测行为的不同之处。

虽然 `greetStudent(..)` 收到一个名为 `greeting` 的参数，但它也同时引用了 `students` 和 `studentID` 这两个来自 `lookupStudent(..)` 的外层作用域的标识符。从内部函数到外层作用域变量的每一次引用都被称为*闭包 (closure)*。用学术术语来说，`greetStudent(..)` 的每个实例都*封闭了*外层变量 `students` 和 `studentID`。

那么，在具体的、可观测到的意义上，这些闭包在这里有什么作用呢？

闭包允许 `greetStudent(..)` 继续访问这些外层变量，即使在外层作用域结束后（当每次调用 `lookupStudent(..)` 完成时）。`students` 和 `studentID` 的实例没有被 GC，而是留在了内存中。以后再调用 `greetStudent(..)` 函数实例时，这些变量仍然存在，并保持其当前值。

如果 JS 函数没有闭包，那么每次完成`lookupStudent(..)` 调用后都会立即缩小其作用域并 GC `students` 和 `studentID` 变量。之后，当我们调用其中一个 `greetStudent(..)` 函数时，会发生什么呢？

如果 `greetStudent(..)` 试图访问它认为是蓝色(2) 的弹珠，但该弹珠实际上并不存在（不再存在），合理的假设是我们应该得到一个 `ReferenceError` 不是吗？

但我们并没有得到错误信息。事实上，`chosenStudents[0]("Hello")` 的执行是有效的，并返回了信息 "Hello, Sarah!"，这意味着它仍然能够访问 `students` 和 `studentID` 变量。这就是对闭包的直接观测！

明显的闭包

事实上，我们在之前的讨论中忽略了一个小细节，我猜很多读者都没有注意到！

由于 `=>` 箭头函数的语法非常简洁，我们很容易忘记它们仍然创建了一个作用域（正如第 3 章「箭头函数」中所断言的）。`student => student.id == studentID` 箭头函数在 `greetStudent(..)` 函数作用域内创建了另一个作用域气泡。

根据第 2 章中彩色水桶和气泡的比喻，如果我们要为这段代码创建一个彩色图，在最内层的嵌套层中有第四个作用域，因此我们需要第四种颜色；也许我们会为该作用域选择橙色(4)：

```js
var student = students.find(
    (student) =>
        // 函数作用域：橙色(4)
```

```
        student.id == studentID,
);
```

蓝色(2) `studentID` 引用实际上是在橙色(4) 作用域内，而不是在 `greetStudent(..)` 的绿色(3) 作用域内；而且，箭头函数的 `student` 参数是橙色(4)，对绿色(3) `student` 有遮蔽。

这样做的后果是，作为数组的 `find(..)` 方法的回调函数传递的箭头函数必须保持 `studentID` 的闭包，而不是 `greetStudent(..)` 保持该闭包。这并不是什么大问题，因为一切仍按预期运行。重要的是，即使是很小的箭头函数也可以参与闭包。

增加闭包

让我们来看看一个经常被引用的经典案例：

```js
function adder(num1) {
    return function addTo(num2) {
        return num1 + num2;
    };
}

var add10To = adder(10);
var add42To = adder(42);

add10To(15); // 25
add42To(9); // 51
```

内部 `addTo(..)` 函数的每个实例都记住了自己的 `num1` 变量（值分别为 `10` 和 `42`），因此这些 `num1` 并不会因为 `adder(..)` 的结束而消失。当我们以后调用这些内部的 `addTo(..)` 实例之一时，例如调用 `add10To(15)` 时，其中的 `num1` 变量仍然存在，并且仍然保持原来的 `10` 值。因此，该操作能够执行 `10 + 15` 并返回答案 `25`。

在上一段中，有一个重要的细节可能很容易被忽略，所以让我们来强化一下：闭包与函数的实例相关联，而不是与函数的单个词法定义相关联。在前面的代码段中，`adder(..)` 内部只定义了一个内部`addTo(..)` 函数，因此这似乎意味着只有一个闭包。

但实际上，外层的 `adder(..)` 函数每运行一次，就会创建一个*新*的内部 `addTo(..)` 函数实例，而每个新实例都有一个新的闭包。因此，每个内部函数实例（在我们的程序中标记为 `add10To(..)` 和 `add42To(..)`）在执行 `adder(..)` 时都有自己的作用域环境实例闭包。

尽管闭包是基于编译时处理的词法作用域，但闭包却是函数实例的运行时特征。

实时链接，而非快照

在前面的两个例子中，我们都是从闭包中的**变量读取值**。这让人感觉闭包可能是某个特定时刻值的快照。事实上，这是一个常见的误解。

闭包实际上是一个实时链接，保留了对完整变量本身的访问。我们不仅限于读取一个值，还可以更新（重新赋值）被封闭的变量！通过封闭函数中的变量，只要程序中存在函数引用，我们就可以继续使用该变量（读取和写入），也可以在任何地方调用该函数。这就是为什么闭包是一种强大的技术，被广泛应用于编程的各个领域！

图 4 描述了函数实例和作用域链接：

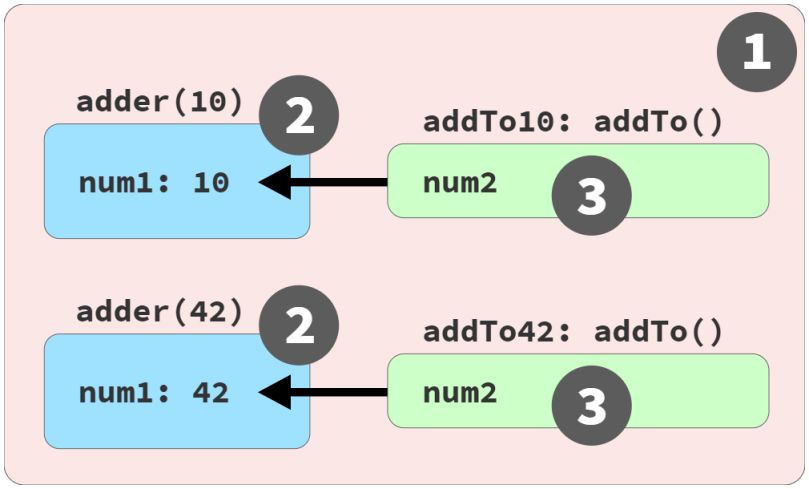

图 4：可视化闭包

如图 4 所示，对 `adder(..)` 的每次调用都会创建一个新的包含 `num1` 变量的蓝色（2）作用域，以及一个新的作为绿色（3）作用域的 `addTo(..)` 函数实例。请注意，函数实例（`addTo10(..)` 和 `addTo42(..)`）存在于红色（1）作用域中并被调用。

现在，我们来看看一个更新封闭变量的例子：

```js
function makeCounter() {
    var count = 0;

    return function getCurrent() {
        count = count + 1;
        return count;
    };
}

var hits = makeCounter();

// 然后

hits(); // 1
```

```
// 然后

hits(); // 2
hits(); // 3
```

`count` 变量被内部的 `getCurrent()` 函数封闭，该函数会保留该变量，而不是将其用于 GC。`hits()` 函数会调用访问*并*更新该变量，每次都会返回一个递增的计数。

虽然闭包的外层作用域通常来自一个函数，但实际上这并不是必需的；外层作用域中只需有一个内部函数即可：

```js
var hits;
{
    // 外层作用域（但不是函数）
    let count = 0;
    hits = function getCurrent() {
        count = count + 1;
        return count;
    };
}
hits(); // 1
hits(); // 2
hits(); // 3
```

注意：

我故意将 `getCurrent()` 定义为一个函数表达式，而不是一个函数声明。这与闭包无关，而是与 FiB 的危险怪癖有关（第 6 章）。

由于将闭包误认为是面向值而不是面向变量的做法非常普遍，开发人员有时会在试图使用闭包来快照保存某个时刻的值时被绊倒。思考一下：

```js
var studentName = "Frank";

var greeting = function hello() {
    // 我们正在封闭 `studentName`，
    // 而不是 "Frank"
    console.log(`Hello, ${studentName}!`);
};

// later
```

```
studentName = "Suzy";

// 然后

greeting();
// Hello, Suzy!
```

当 `studentName` 的值为 `"Frank"`（在重新赋值给 `"Suzy"` 之前）时，通过定义 `greeting()`（又名 `hello()`），错误的假设往往是闭包将捕获 `"Frank"`。但是，`greeting()` 是对变量 `studentName` 的闭包，而不是对其值的闭包。无论何时调用 `greeting()`，都会反映变量（本例中为 `"Suzy"`）的当前值。

这种错误的典型例子就是在循环中定义函数：

```js
var keeps = [];

for (var i = 0; i < 3; i++) {
    keeps[i] = function keepI() {
        // 封闭 `i`
        return i;
    };
}

keeps[0](); // 3 -- 为什么！？
keeps[1](); // 3
keeps[2](); // 3
```

> **注意：**
>
> 这种闭包示例通常在循环内使用 `setTimeout(..)` 或其他回调（如事件处理程序）。我简化了示例，将函数引用存储在数组中，这样我们在分析时就不需要考虑异步时序。无论如何，闭包原理是相同的。

你可能以为 `keeps[0]()` 的调用会返回 `0`，因为该函数是在循环的第一次迭代中创建的，当时 `i` 为 `0`。但同样，这种假设源于将闭包视为面向值而不是面向变量。

`for` 循环的结构会欺骗我们，让我们以为每一次迭代都会得到一个新的 `i` 变量；事实上，这个程序只有一个 `i`，因为它是用 `var` 声明的。

每个保存的函数都会返回 `3`，因为在循环结束时，程序中唯一的 `i` 变量已经被赋值为 `3`。在 `keeps` 数组中的三个函数都有各自的闭包，但它们都是在同一个共享的 `i` 变量上闭包的。

当然，一个变量在任何时候都只能保存一个值。因此，如果要保存多个值，就需要为每个值设置不同的变量。

如何在循环中做到这一点呢？让我们为每次迭代创建一个新变量：

```js
var keeps = [];

for (var i = 0; i < 3; i++) {
    // 每次迭代都会创建新的 `j`，
    // 获取此刻 `i` 值的副本
    let j = i;

    // 这里的 `i` 没有被封闭，
    // 所以在每个循环迭代中
    // 立即使用它的当前值就可以了
    keeps[i] = function keepEachJ() {
        // 封闭了 `j`，而不是 `i`！
        return j;
    };
}
keeps[0](); // 0
keeps[1](); // 1
keeps[2](); // 2
```

现在，每个函数在每次迭代中都封闭了一个单独的（新）变量，尽管所有变量都被命名为 `j`。每个 `j` 都会获得循环迭代中 `i` 值的副本，而 `j` 永远不会被重新赋值。因此，这三个函数现在都会返回它们的预期值： `0`、 `1` 和 `2`！

请再次记住，即使我们在此程序中使用异步，例如将每个内部 `keepEachJ()` 函数传递到 `setTimeout(..)` 或某些事件处理程序订阅中，仍然可以观测到相同的闭包行为。

回想一下第 5 章中的「循环」一节，其中说明了在一个 `for` 循环中的 `let` 声明实际上不仅为循环创建了一个变量，而且实际上为循环的*每次迭代*创建了一个新变量。这个技巧正是我们的循环闭包所需要的：

```js
var keeps = [];

for (let i = 0; i < 3; i++) {
    // `let i` 会自动为每次迭代
    // 提供一个新的 "i"！
    keeps[i] = function keepEachI() {
        return i;
    };
}
```

```
keeps[0](); // 0
keeps[1](); // 1
keeps[2](); // 2
```

由于我们使用了 `let`，所以创建了三个 `i`，每个循环一个，因此三个闭包都能按预期运行。

常见闭包： Ajax 和事件

闭包常见于回调：

```js
function lookupStudentRecord(studentID) {
    ajax(`https://some.api/student/${studentID}`, function onRecord(record) {
        console.log(`${record.name} (${studentID})`);
    });
}

lookupStudentRecord(114);
// Frank (114)
```

`onRecord(..)` 回调将在 Ajax 调用返回响应后的某个时间点被调用。这种调用将从 `ajax(..)` 工具的内部发生，无论它来自何处。此外，当发生这种情况时，`lookupStudentRecord(..)` 调用早已完成。

那么，为什么 `studentID` 仍然存在并可被回调访问呢？闭包。

事件处理程序是闭包的另一种常见用法：

```js
function listenForClicks(btn, label) {
    btn.addEventListener("click", function onClick() {
        console.log(`The ${label} button was clicked!`);
    });
}

var submitBtn = document.getElementById("submit-btn");

listenForClicks(submitBtn, "Checkout");
```

`onClick(..)` 事件处理回调封闭了 `label` 参数。点击按钮后，`label` 仍然存在，可以使用。这就是闭包。

如果我看不到怎么办？

你可能听说过这句谚语：

> 假如一棵树在森林里倒下而没有人在附近听见，它有没有发出声音？[1]

这是一种愚蠢的哲学体操。当然，从科学的角度来看，声波是被创造出来的。但真正的问题是声音是否发生*重要吗*？

请记住，我们对闭包定义的重点是可观测性。如果闭包存在（在技术、实现或学术意义上），但在我们的程序中却无法观测到，*这有什么关系吗*？

为了强化这一点，让我们来看一些*无法*观测到的基于闭包的例子。

例如，调用一个使用词法作用域查找的函数：

```js
function say(myName) {
    var greeting = "Hello";
    output();

    function output() {
        console.log(`${greeting}, ${myName}!`);
    }
}

say("Kyle");
// Hello, Kyle!
```

内层函数 `output()` 从其外层作用域访问变量 `greeting` 和 `myName`。但是，`output()` 的调用发生在同一个作用域中，当然 `greeting` 和 `myName` 仍然可用；这只是词法作用域，而不是闭包。

任何函数不支持闭包的词法作用域语言都会有同样的行为。

事实上，全局作用域变量本质上是无法（可观测）闭包的，因为它们总是可以从任何地方访问。在作用域链的任何部分，只要不是全局作用域的后代，就不能调用任何函数。

想想看：

```js
var students = [
    { id: 14, name: "Kyle" },
    { id: 73, name: "Suzy" },
    { id: 112, name: "Frank" },
    { id: 6, name: "Sarah" },
];

function getFirstStudent() {
```

```
    return function firstStudent() {
        return students[0].name;
    };
}

var student = getFirstStudent();

student();
// Kyle
```

内部的 `firstStudent()` 函数确实引用了 `students`，这是一个在其自身作用域之外的变量。但由于 `students` 恰好来自全局作用域，因此无论在程序中的哪个位置调用该函数，它访问 `students` 的能力都不会比正常的词法作用域更特殊。

无论语言是否支持闭包，所有函数调用都可以访问全局变量。全局变量不需要闭包。

仅存在但从未被访问的变量不会导致闭包：

```js
function lookupStudent(studentID) {
    return function nobody() {
        var msg = "Nobody's here yet.";
        console.log(msg);
    };
}

var student = lookupStudent(112);

student();
// Nobody's here yet.
```

内部函数 `nobody()` 不会封闭任何外部变量，它只使用自己的变量 `msg`。即使 `studentID` 存在于外层作用域中，`nobody()` 也不会引用 `studentID`。在 `lookupStudent(..)` 运行结束后，JS 引擎不需要继续使用 `studentID`，因此 GC 希望清理该内存！

无论 JS 函数是否支持闭包，这个程序的行为都是一样的。因此，这里没有观测到闭包。

如果没有函数调用，就无法观测闭包：

```js
function greetStudent(studentName) {
    return function greeting() {
        console.log(`Hello, ${studentName}!`);
    };
}
```

```
greetStudent("Kyle");

// 别无其他
```

这个问题很棘手，因为外层函数肯定会被调用。但内部函数是*可能*具有闭包性的函数，但它从未被调用；返回的函数在这里只是被丢弃。因此，即使从技术上讲，JS 引擎在短时间内创建了闭包，在这个程序中也没有以任何有意义的方式观测到它。

一棵树可能倒下了……但我们没听到，所以我们也不关心。

可观测定义

现在，我们准备定义闭包：

> 当函数使用外作用域的变量时，即使是在这些变量无法访问的作用域中运行，也会出现闭包。

这一定义的关键部分是：

- 必须涉及一个函数
- 必须引用至少一个外部作用域变量
- 必须在不同于变量的作用域链分支中调用

这种以观测为导向的定义意味着，我们不应将闭包视为一些间接的学术琐事。相反，我们应该关注并计划闭包对程序行为产生的直接、具体的影响。

闭包生命周期和垃圾回收 (GC)

由于闭包本质上是与函数实例绑定的，因此只要仍存在对该函数的引用，它对变量的闭包就会持续。

如果十个函数都对同一个变量进行了闭包，随着时间的推移，其中九个函数的引用被丢弃，剩下的唯一一个函数引用仍然保留着该变量。一旦最后一个函数引用被丢弃，该变量的最后一次闭包也就消失了，变量本身也会被 GC。

这对构建高效、高性能的程序有着重要影响。闭包可能会意外地阻止一个变量的 GC，会导致内存使用时间过长。这就是为什么当不再需要函数引用（以及它们的闭包）时，必须将其丢弃的原因。

想想看：

```js
function manageBtnClickEvents(btn) {
    var clickHandlers = [];

    return function listener(cb) {
        if (cb) {
            let clickHandler = function onClick(evt) {
                console.log("clicked!");
                cb(evt);
            };
            clickHandlers.push(clickHandler);
            btn.addEventListener("click", clickHandler);
        } else {
            // 不传递回调将取消订阅
            // 所有点击处理程序
            for (let handler of clickHandlers) {
                btn.removeEventListener("click", handler);
            }

            clickHandlers = [];
        }
    };
}

// var mySubmitBtn = ..
var onSubmit = manageBtnClickEvents(mySubmitBtn);

onSubmit(function checkout(evt) {
    // 办理结账
});

onSubmit(function trackAction(evt) {
    // 记录日志和分析
});

// 之后，取消订阅所有处理程序：
onSubmit();
```

在此程序中，内部的 `onClick(..)` 函数对传递的 `cb`（提供的事件回调）进行闭包。这意味着只要这些事件处理程序被订阅，`checkout()` 和 `trackAction()` 函数表达式引用就会通过闭包被保留（并且不能被 GC）。

当我们在最后一行没有输入的情况下调用 `onSubmit()` 时，所有事件处理程序都会被取消订阅，而 `clickHandlers` 数组也会被清空。一旦所有点击处理程序函数的引用都被丢弃，`cb` 的闭包引用 `checkout()` 和 `trackAction()` 将被丢弃。

在考虑程序的整体健康和效率时，在不再需要时取消订阅事件处理程序可能比最初的订阅更加重要！

按变量还是按作用域？

我们需要解决的另一个问题是：我们应该认为闭包只适用于引用的外层变量，还是闭包会保留整个作用域链及其所有变量？

换句话说，在前面的事件订阅代码段中，内部的 `onClick(..)` 函数是只对 `cb` 产生闭包，还是也对 `clickHandler`、 `clickHandlers` 和 `btn` 产生闭包？

从概念上讲，闭包是**按变量**，而不是*按作用域*。Ajax 回调、事件处理程序和所有其他形式的函数闭包通常被认为只闭包它们明确引用的内容。

但现实情况比这更复杂。

另一个值得思考的程序：

```js
function manageStudentGrades(studentRecords) {
    var grades = studentRecords.map(getGrade);

    return addGrade;

    // ************************

    function getGrade(record) {
        return record.grade;
    }

    function sortAndTrimGradesList() {
        // 按成绩降序排序
        grades.sort(function desc(g1, g2) {
            return g2 - g1;
        });

        // 只保留前 10 名
        grades = grades.slice(0, 10);
    }

    function addGrade(newGrade) {
        grades.push(newGrade);
        sortAndTrimGradesList();
        return grades;
    }
}
```

```
var addNextGrade = manageStudentGrades([
    { id: 14, name: "Kyle", grade: 86 },
    { id: 73, name: "Suzy", grade: 87 },
    { id: 112, name: "Frank", grade: 75 },
    // ..many more records..
    { id: 6, name: "Sarah", grade: 91 },
]);

// 然后

addNextGrade(81);
addNextGrade(68);
// [ .., .., ... ]
```

外部函数 `manageStudentGrades(..)` 接收一个学生记录列表，并返回一个 `addGrade(..)` 函数引用，我们将其外部标记为 `addNextGrade(..)`。每次我们使用新成绩调用 `addNextGrade(..)` 时，都会返回当前前 10 个成绩的列表，并按数字降序排序（参见 `sortAndTrimGradesList()`）。

从最初的 `manageStudentGrades(..)` 调用结束到多个 `addNextGrade(..)` 调用之间，`grades` 变量通过闭包被保存在 `addGrade(..)` 中；这就是最高分的运行列表的保存方式。请记住，这是对变量 `grades` 本身的闭包，而不是它所包含的数组。

然而，这并不是唯一的闭包。您还能发现其他被封闭的变量吗？

你发现 `addGrade(..)` 引用了 `sortAndTrimGradesList` 吗？这意味着它也封闭了该标识符，而该标识符恰好持有对 `sortAndTrimGradesList()` 函数的引用。第二个内部函数必须继续存在，这样 `addGrade(..)` 才能继续调用它，这也意味着 *它* 封闭的任何变量都必须继续存在，尽管在本例中，没有任何额外的变量被封闭。

还有什么被封闭了？

考虑一下 `getGrade` 变量（及其函数）；它是否被封闭？在 `manageStudentGrades(..)` 的外层作用域中的 `.map(getGrade)` 调用中引用了它。但在 `addGrade(..)` 或 `sortAndTrimGradesList()` 中没有引用。

我们作为 `studentRecords` 传入的（可能是）大量学生记录列表怎么办？该变量是否被封闭？如果是，那么学生记录数组就永远不会被 GC，这导致程序占用的内存比我们想象的要大。但如果我们再仔细观察一下，没有一个内部函数引用了 `studentRecords`。

根据闭包的*按变量*的定义，由于 `getGrade` 和 `studentRecords` 没有被内部函数引用，所以它们没有被封闭。在 `manageStudentGrades(..)` 调用完成后，它们应该可以自由地用于

GC。

事实上，请尝试在最新的 JS 引擎（如 Chrome 浏览器中的 v8）中调试此代码，在 `addGrade(..)` 函数中设置断点。您可能会注意到，检查器**并没有**列出 `studentRecords` 变量。这就证明，无论如何，从调试的角度来看，引擎并没有通过闭包来维护 `studentRecords` 变量。呼！

但是，这种观察作为证据的可靠性如何呢？请看这个程序（相当别扭！）：

```js
function storeStudentInfo(id, name, grade) {
    return function getInfo(whichValue) {
        // 警告：
        //   使用 `eval(..)` 是个坏主意！
        var val = eval(whichValue);
        return val;
    };
}

var info = storeStudentInfo(73, "Suzy", 87);

info("name");
// Suzy

info("grade");
// 87
```

请注意，内部函数 `getInfo(..)` 并没有显式地封闭任何 `id`、`name` 或 `grade` 变量。然而，对 `info(..)` 的调用似乎仍然能够访问这些变量，尽管是通过使用 `eval(..)` 词法作用域欺骗（参见第 1 章）。

因此，尽管内部函数没有明确引用，但所有变量都通过闭包得到了保留。那么，这是否推翻了*按变量*的断言，转而支持*按作用域*的断言呢？这要看情况。

许多现代 JS 引擎都会应用一种*优化*方法，从闭包作用域中移除任何未明确引用的变量。然而，正如我们在使用 `eval(..)` 时所看到的，在某些情况下无法应用这种优化，闭包作用域仍然包含其所有原始变量。换句话说，从实现上来说，闭包必须是按作用域闭包的，然后可选的优化会将作用域缩减到只有被闭包的变量（与按变量闭包的结果类似）。

即使在几年前，许多 JS 引擎也没有应用这种优化；您的网站有可能仍在此类浏览器中运行，尤其是在较旧或较低端的设备上。这意味着事件处理程序等长寿命闭包占用内存的时间可能比我们想象的要长很多。

而事实上，它首先是一个可选的优化，而不是规范的要求，这意味着我们不应该随意地过度假设它的适用性。

在变量持有较大值（如对象或数组）且该变量存在于闭包作用域中的情况下，如果不再需要该值，也不希望内存被占用，那么手动丢弃该值比依赖闭包优化/GC 更安全（内存占用）。

让我们对先前的 `manageStudentGrades(..)` 示例进行*修复*，以确保在 `studentRecords` 中保存的潜在大数组不会不必要地陷入闭包作用域：

```js
function manageStudentGrades(studentRecords) {
    var grades = studentRecords.map(getGrade);

    // 取消设置 `studentRecords` 以防止
    // 在闭包中保留不必要的内存
    studentRecords = null;

    return addGrade;
    // ..
}
```

我们并没有将 `studentRecords` 从闭包作用域中移除；这是我们无法控制的。我们要确保即使 `studentRecords` 仍在闭包作用域中，该变量也不再引用可能很大的数据数组；该数组可以被 GC。

同样，在许多情况下，JS 可能会自动优化程序，以达到同样的效果。但我们还是要养成小心谨慎的好习惯，明确确保不会在不必要的情况下占用大量设备内存。

事实上，在 `.map(getGrade)` 调用完成后，我们在技术上也不再需要函数 `getGrade()`。如果对我们的应用程序进行剖析后发现这是内存使用过多的关键区域，我们可以通过释放该引用来释放更多内存，这样它的值也不会被占用。在这个玩具示例中，这可能是不必要的，但如果要优化应用程序的内存占用，这是一个需要牢记的通用技术。

启示：了解闭包在程序中出现的位置和包含的变量非常重要。我们应该仔细管理这些闭包，以便只保留最不需要的部分，避免浪费内存。

一种替代观点

回顾我们对闭包的工作定义，我们可以断言函数是「头等值」[2]，可以像其他值一样在程序中传递。闭包是将函数与函数本身之外的作用域/变量连接起来的链接关联，无论函数走到哪里。

让我们回顾一下本章前面的一个代码示例，同样标注了相关的作用域气泡颜色：

```js
// 外部/全局作用域：红色 (1)
```

```
function adder(num1) {
    // 函数作用域：蓝色 (2)

    return function addTo(num2) {
        // 函数作用域：绿色 (3)

        return num1 + num2;
    };
}

var add10To = adder(10);
var add42To = adder(42);

add10To(15); // 25
add42To(9); // 51
```

我们目前的观点认为，无论函数在哪里被传递和调用，闭包都会保留一个回到原始作用域的隐藏链接，以方便访问被闭包的变量。为方便起见，我们在此重复图 4，以说明这一概念：

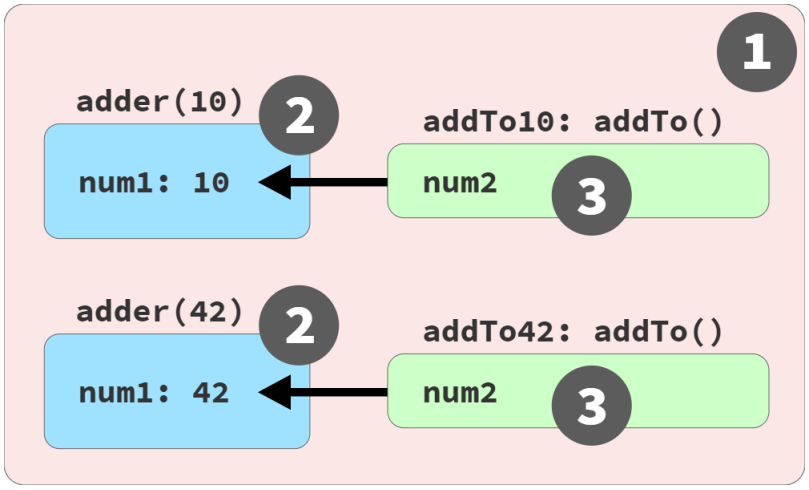

图 4（重复）：可视化闭包

不过，还有另一种思考闭包的方法，更确切地说，是思考被*传递*的函数的性质，这可能有助于深化心智模型。

这种替代模型不再强调「函数是头等值」，而是接受函数（像所有非原始值一样）在 JS 中通过引用持有，并通过引用拷贝分配/传递的方式。更多信息请参见*入门*一书的附录 A。

我们不用考虑内部函数实例 `addTo(..)` 通过 `return` 和赋值移动到外部红色（1）作用域，我们可以设想函数实例实际上只是停留在它们自己的作用域环境中，当然，它们的作用域链也是完好无损的。

*发送*到红色（1）作用域的只是对本地函数实例的**引用**，而不是函数实例本身。图 5 描述了分别由红色（1）`addTo10` 和 `addTo42` 引用指向的保持原位的内部函数实例：

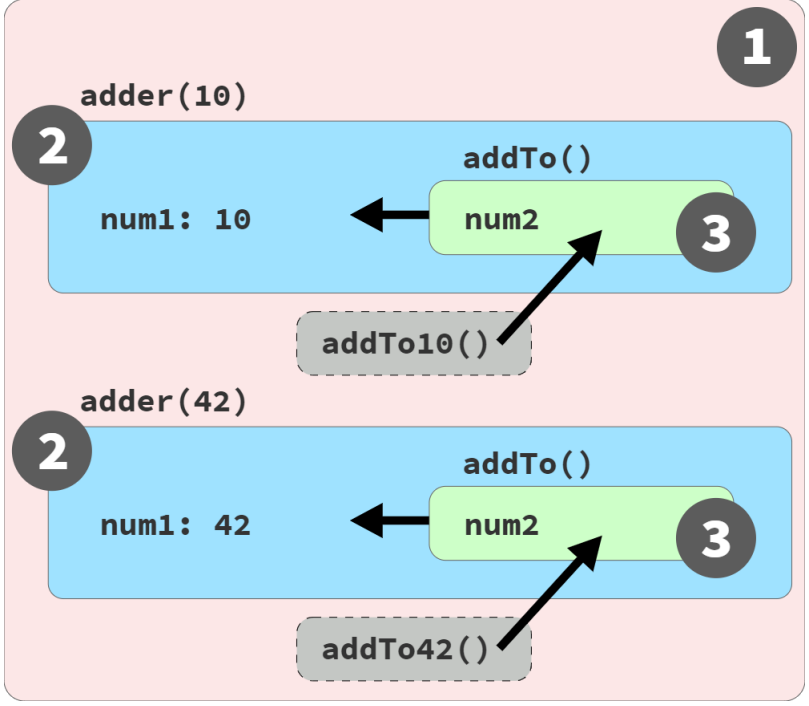

图 5：闭包可视化（备选方案）

如图 5 所示，每次调用 `adder(..)` 都会创建一个包含 `num1` 变量的新蓝色（2）作用域，以及一个绿色（3）`addTo(..)` 作用域实例。但与图 4 不同的是，现在这些绿色（3）实例保持不变，自然地嵌套在它们的蓝色（2）作用域实例中。`addTo10` 和 `addTo42` 引用被移到了红色（1）外层作用域，而不是函数实例本身。

当调用 `adTo10(15)` 时，会调用函数实例 `addTo(..)`（仍在其原来的蓝色（2）作用域环境中）。由于函数实例本身从未移动过，它当然仍然可以自然地访问其作用域链。同样，调用 `adTo42(9)` 也是如此。除了词法作用域之外，没有什么特别之处。

那么，如果闭包不是一种让函数在其他作用域中移动时仍能保持与其原始作用域链链接的魔法，那它又*是*什么呢？在这一替代模型中，函数保持原位，并能像以前一样访问其原始作用域链。

闭包描述的是一种**保持函数实例**及其整个作用域环境和链的活力的魔法，只要程序的任何其他部分中还存在至少一个对该函数实例的引用。

与传统的学术观点相比，闭包的这一定义不那么注重观察，听起来也不那么耳熟能详。但它仍然有用，因为它的好处是，我们将闭包的解释简化为引用和本地函数实例的直接组合。

前面的模型（图 4）在描述 JS 中的封闭性方面并没有错。它只是在概念上更有启发，是从学术角度来看待闭包的。相比之下，另一个模型（图 5）可以说是更注重实现，即 JS 实际是如何工作的。

这两种观点/模式都有助于理解闭包，但读者可能会觉得其中一种比另一种更容易接受。无论你选择哪一种，我们计划中的可观察结果都是一样的。

> **注意：**
>
> 这种闭包的替代模型确实会影响我们是否将同步回调归类为闭包的例子。关于这一细微差别，详见附录 A。

为什么是闭包？

既然我们已经全面了解了什么是闭包及其工作原理，那么就让我们来探讨一下闭包改善代码结构和程序组织的一些方法。

想象一下，页面上有一个按钮，当点击它时，应该通过 Ajax 请求获取并发送一些数据。在不使用闭包的情况下：

```js
var APIendpoints = {
    studentIDs: "https://some.api/register-students",
    // ..
};

var data = {
    studentIDs: [14, 73, 112, 6],
    // ..
};

function makeRequest(evt) {
    var btn = evt.target;
    var recordKind = btn.dataset.kind;
    ajax(APIendpoints[recordKind], data[recordKind]);
}

// <button data-kind="studentIDs">
//    Register Students
// </button>
btn.addEventListener("click", makeRequest);
```

`makeRequest(..)` 方法只会从点击事件中接收一个 `evt` 对象。然后，它必须从目标按钮元素中获取 `data-kind` 属性，并使用该值查找 API 端点的 URL 以及 Ajax 请求中应包含的数据。

这样做效果还可以，但令人遗憾的是（效率不高，更容易引起混淆），每次触发事件时，事件处理程序都必须读取 DOM 属性。为什么事件处理程序不能*记住*这个值呢？让我们尝试使用闭包来改进代码：

```js
var APIendpoints = {
    studentIDs: "https://some.api/register-students",
    // ..
};

var data = {
    studentIDs: [14, 73, 112, 6],
    // ..
};

function setupButtonHandler(btn) {
    var recordKind = btn.dataset.kind;

    btn.addEventListener("click", function makeRequest(evt) {
        ajax(APIendpoints[recordKind], data[recordKind]);
    });
}

// <button data-kind="studentIDs">
//    Register Students
// </button>

setupButtonHandler(btn);
```

使用 `setupButtonHandler(..)` 方法时，会在初始设置时检索一次 `data-kind` 属性并将其分配给 `recordKind` 变量。然后，内部的 `makeRequest(..)` 点击处理程序会封闭 `recordKind` 变量，并在每次事件触发时使用其值来查找应发送的 URL 和数据。

> **注意：**
>
> `evt` 仍会传递给 `makeRequest(..)`，但在本例中我们不再使用它。为了与前面的代码段保持一致，我们仍然列出了它。

通过将 `recordKind` 放在 `setupButtonHandler(..)` 中，我们将该变量的作用域限制在了程序的一个更合适的子集中；如果将它存储在全局中，代码的组织性和可读性都会变差。闭包可以让内部的 `makeRequest()` 函数实例记住这个变量，并在需要时访问它。

根据这种模式，我们可以在设置时一次性查询 URL 和数据：

```js
function setupButtonHandler(btn) {
    var recordKind = btn.dataset.kind;
    var requestURL = APIendpoints[recordKind];
    var requestData = data[recordKind];
```

```
    btn.addEventListener("click", function makeRequest(evt) {
        ajax(requestURL, requestData);
    });
}
```

现在，`makeRequest(..)` 封闭了 `requestURL` 和 `requestData`，这样理解起来更简洁，性能也略有提高。

函数式编程（FP）范式中有两种依赖于闭包的类似技术，它们是偏函数[3]（partial application）和柯里化[4]（currying）。简而言之，利用这些技术，我们可以改变需要多个输入的函数的*形态*，这样就可以先提供一些输入，然后再提供其他输入；通过闭包记住初始输入。一旦提供了所有输入，就会执行底层操作。

通过创建一个封装了某些信息的函数实例（通过闭包），以后就可以直接使用存储了信息的函数，而无需重新提供输入。这使得代码的这一部分更加简洁，同时也为使用语义更好的名称标记部分应用的函数提供了机会。

通过调整应用的部分内容，我们可以进一步改进前面的代码：

```js
function defineHandler(requestURL, requestData) {
    return function makeRequest(evt) {
        ajax(requestURL, requestData);
    };
}

function setupButtonHandler(btn) {
    var recordKind = btn.dataset.kind;
    var handler = defineHandler(APIendpoints[recordKind], data[recordKind]);
    btn.addEventListener("click", handler);
}
```

`requestURL` 和 `requestData` 输入会提前提供，从而产生 `makeRequest(..)` 部分应用函数，我们在本地将其标记为 `handler`。当事件最终触发时，最后的输入（`evt`，尽管它被忽略了）会传递给 `handler()`，完成其输入并触发底层的 Ajax 请求。

从行为上看，这个程序与前一个程序非常相似，使用了相同类型的闭包。但通过将 `makeRequest(..)` 的创建分离到一个单独的实用程序（`defineHandler(..)`）中，我们使得该定义在整个程序中的重用性更高。我们还明确地将闭包的范围限制为只需要两个变量。

完美谢幕

在我们结束密集的一章时，请做一些深呼吸，让一切都沉淀下来。说真的，对于任何人来说，这都是大量的信息！

我们探讨了从思想上解决闭包问题的两种模式：

- 观测：闭包是一个函数实例，即使该函数被传递到其他作用域并在其他作用域中**被调用**，它也会记住其外部变量。
- 实现：闭包是一个函数实例，它的作用域环境就地保留，而对它的任何引用都被传递并**从**其他作用域**调用**。

总结一下我们程序的优势：

- 闭包可以让函数实例记住先前确定的信息，而不必每次都进行计算，从而提高效率。
- 闭包可以提高代码的可读性，通过将变量封装在函数实例中来限制作用域的暴露，同时还能确保这些变量中的信息可供将来使用。由此产生的更小、更专业的函数实例，由于无需在每次调用时都传递所保留的信息，因此交互起来更简洁。

在继续学习之前，请花一些时间*用自己的话*重述一下这个总结，解释一下什么是闭包，为什么它对你的程序有帮助。本书正文的最后一章将在闭包的基础上使用模块模式。

---

1. *假如一棵树在森林里倒下*， https://zh.wikipedia.org/wiki/假如一棵樹在森林裡倒下，2022年10月29日。 ↩
2. *头等函数*，https://zh.wikipedia.org/wiki/头等函数，2023年8月21日。 ↩
3. *Partial application*，https://en.wikipedia.org/wiki/Partial_application， 2023年6月28日。 ↩
4. *柯里化*，https://zh.wikipedia.org/zh-hans/柯里化， 2021年9月1日。 ↩

第八章：模块化模式

在本章中，我们将通过探索所有编程中最重要的代码组织模式之一：模块，来结束本书的正文。正如我们将看到的那样，模块本质上是由我们已经讲过的内容构建而成：这是你学习词法作用域和闭包所付出努力的回报。

我们研究了词法作用域的各个角度，从全局作用域的广度到嵌套的块作用域，再到错综复杂的变量生命周期。然后，我们利用词法作用域来了解闭包的全部功能。

花点时间回顾一下，你在这段旅程中已经走了多远；在更深入地了解 JS 方面，你已经迈出了一大步！

本书的核心主题是，理解并掌握作用域和闭包是正确构建和组织代码的关键，尤其是决定在变量中存储信息的位置。

在最后一章中，我们的目标是了解模块如何体现这些主题的重要性，如何将它们从抽象概念提升为构建程序中具体而实用的改进。

封装和最小暴露 (POLE)

封装经常被作为面向对象（OO）编程的一项原则，但它的基础性和广泛适用性远不止于此。封装的目的是将信息（数据）和行为（功能）捆绑在一起或置于同一位置，共同达到一个目的。

与任何语法或代码机制无关，封装的精神可以通过一些简单的事情来实现，比如使用单独的文件来保存整个程序中具有共同目的的部分。如果我们将为搜索结果列表提供功能的所有内容都打包到一个名为 "search-list.js" 的文件中，我们就封装了程序的这一部分。

现代前端编程最近的趋势是围绕组件架构来组织应用程序，这进一步推动了封装。对于许多人来说，将构成搜索结果列表的所有内容（甚至包括代码以外的内容，包括呈现标记和样式）整合到一个单一的程序逻辑单元中，即我们可以与之交互的有形内容中，感觉很自然。然后，我们给这个集合贴上 "SearchList" 组件的标签。

另一个关键目标是控制封装数据和功能某些方面的可见性。请回顾第 6 章中的*最小暴露原则 (POLE)*，该原则旨在防御各种作用域过度暴露的*危险*；这些危险同时影响变量和函数。在 JS 中，我们通常通过词法作用域的机制来实现可见性控制。

这样做的目的是将程序中相似的片段组合在一起，有选择性地限制程序对我们认为*私有*细节部分的访问。不被视为*私有*的部分则被标记为*公共*，整个程序都可以访问。

这种努力的自然会得到更好的代码组织。当我们知道事物的位置、清晰明确的边界和连接点时，构建和维护软件就会变得更加容易。如果我们能避免过度暴露数据和功能的隐患，也就更容易保证质量。

这些就是将 JS 程序组织成模块的一些主要好处。

什么是模块?

模块是相关数据和函数（在此通常称为方法）的集合，其特点是分为隐藏的*私有*细节和*公共*可访问的细节，通常称为「公共 API」。

模块也是有状态的：它随着时间的推移维护一些信息，以及访问和更新这些信息的功能。

> **注意：**
>
> 模块模式的一个更广泛的关注点是，通过解耦和其他程序架构技术，全面拥抱系统级模块化。这是一个复杂的话题，远远超出了我们的讨论范围，但值得在本书之外进一步研究。

为了更好地理解什么是模块，让我们将模块的一些特征与非模块的有用代码模式进行比较。

命名空间（无状态分组）

如果你将一组相关的函数组合在一起，但不包含数据，那么你就没有真正意义上的模块封装。命名空间（namespace）是这种*无状态*函数分组的更贴切的术语：

```js
// 命名空间，而不是模块
var Utils = {
    cancelEvt(evt) {
        evt.preventDefault();
        evt.stopPropagation();
        evt.stopImmediatePropagation();
    },
    wait(ms) {
        return new Promise(function c(res) {
            setTimeout(res, ms);
        });
    },
    isValidEmail(email) {
        return /[^@]+@[^@.]+\.[^@.]+/.test(email);
    },
};
```

这里的 `Utils` 是实用工具的集合，但它们都是与状态无关的函数。将功能集合在一起通常是一种好的做法，但这并不能使其成为一个模块。相反，我们定义了一个 `Utils` 命名空间，并在其下组织了这些函数。

数据结构（有状态分组）

即使你把数据和有状态函数捆绑在一起，如果你没有限制其中任何一个函数的可见性，那么你就没有达到封装的 POLE 层面；给它贴上模块的标签也没什么用。

想想看：

```js
// 数据结构，而不是模块
var Student = {
    records: [
        { id: 14, name: "Kyle", grade: 86 },
        { id: 73, name: "Suzy", grade: 87 },
        { id: 112, name: "Frank", grade: 75 },
        { id: 6, name: "Sarah", grade: 91 },
    ],
    getName(studentID) {
        var student = this.records.find((student) => student.id == studentID);
        return student.name;
    },
};

Student.getName(73);
// Suzy
```

由于 `records` 是可公开访问的数据，而不是隐藏在公共 API 后面的数据，因此这里的 `Student` 并不是一个真正的模块。

`Student` 确实具有封装的数据和功能方面，但不具有可见性控制方面。最好给它贴上数据结构实例的标签。

模块（状态访问控制）

要充分体现模块模化的精神，我们不仅需要分组和状态，还需要通过可见性（私有与公用）进行访问控制。

让我们把上一节中的 `Student` 变成一个模块。我们将从一种我称之为「传统模块」的形式开始，这种模块在 2000 年代初首次出现时被称为「揭示模块」。思考一下：

```js
var Student = (function defineStudent() {
    var records = [
        { id: 14, name: "Kyle", grade: 86 },
        { id: 73, name: "Suzy", grade: 87 },
        { id: 112, name: "Frank", grade: 75 },
        { id: 6, name: "Sarah", grade: 91 },
    ];

    var publicAPI = {
        getName,
    };

    return publicAPI;

    // ************************

    function getName(studentID) {
        var student = records.find((student) => student.id == studentID);
        return student.name;
    }
})();

Student.getName(73); // Suzy
```

`Student` 现在是一个模块实例。它有一个公共 API，只有一个方法：`getName(..)`。该方法可以访问隐藏的私有 `records` 数据。

> **警告：**
>
> 我需要指出的是，将明确的学生数据硬编码到模块定义中只是为了说明问题。程序中的典型模块将从外部来源接收数据，通常是从数据库、JSON 数据文件、Ajax 调用等加载的数据。然后，这些数据通常会通过模块公共 API 上的方法注入模块实例。

传统模块模式是如何运作的？

请注意，模块实例是通过执行 `defineStudent()` IIFE 创建的。该 IIFE 返回一个对象（名为 `publicAPI`），该对象上有一个引用内部 `getName(..)` 函数的属性。

将对象命名为 `publicAPI` 是我的风格偏好。对象可以随心所欲地命名（JS 并不关心），也可以直接返回对象，而不将其赋值给任何内部命名变量。附录 A 将详细介绍这种选择。

从外部看，`Student.getName(..)` 会调用这个公开的内部函数，该函数通过闭包保持对内部 `records` 变量的访问。

你并不一定要返回一个以函数作为属性之一的对象。你可以直接返回一个函数来代替对象。这样仍然可以满足传统模块的所有核心要求。

根据词法作用域的工作原理，在外层模块定义函数中定义变量和函数，默认情况下会将所有内容都设为私有。只有添加到函数返回的公共 API 对象中的属性才会被导出供外部公共使用。

使用 IIFE 意味着我们的程序只需要模块的一个中心实例，也就是通常所说的「单例」。事实上，这个例子非常简单，没有明显的理由让我们只需要一个 `Student` 模块实例。

模块工厂（多实例）

但如果我们确实想在程序中定义一个支持多个实例的模块，我们可以对代码稍作调整：

```js
// 工厂函数，而不是单例 IIFE
function defineStudent() {
    var records = [
        { id: 14, name: "Kyle", grade: 86 },
        { id: 73, name: "Suzy", grade: 87 },
        { id: 112, name: "Frank", grade: 75 },
        { id: 6, name: "Sarah", grade: 91 },
    ];

    var publicAPI = {
        getName,
    };

    return publicAPI;

    // ************************

    function getName(studentID) {
        var student = records.find((student) => student.id == studentID);
        return student.name;
    }
}

var fullTime = defineStudent();
fullTime.getName(73); // Suzy
```

我们并没有将 `defineStudent()` 指定为 IIFE，而是将其定义为一个普通的独立函数，在这里通常称为「模块工厂」函数。

然后，我们调用模块工厂，生成一个标为 `fullTime` 的模块实例。这个模块实例意味着一个新的内部作用域实例，因此，`getName(..)` 对 `records` 拥有一个新的闭包。现在，`fullTime.getName(..)` 将调用该特定实例上的方法。

传统模块的定义

因此，为了澄清什么是传统模块：

- 必须有一个外部作用域，通常是至少运行一次的模块工厂函数。
- 模块的内部作用域必须至少有一条代表模块状态的隐藏信息。
- 模块必须在其公共 API 中返回至少一个函数的引用，该函数对隐藏的模块状态具有闭包（以便实际保留该状态）。

我们将在附录 A 中详细介绍这种传统模块方法的其他变体。

Node CommonJS 模块

在第 4 章中，我们介绍了 Node 使用的 CommonJS 模块格式。与前面介绍的传统模块格式不同，CommonJS 模块是基于文件的；每个文件一个模块。

让我们调整一下模块示例，使其符合这种格式：

```js
module.exports.getName = getName;

// ************************

var records = [
    { id: 14, name: "Kyle", grade: 86 },
    { id: 73, name: "Suzy", grade: 87 },
    { id: 112, name: "Frank", grade: 75 },
    { id: 6, name: "Sarah", grade: 91 },
];

function getName(studentID) {
    var student = records.find((student) => student.id == studentID);
    return student.name;
}
```

`records` 和 `getName` 标识符位于本模块的顶层作用域，但这不是全局作用域（如第 4 章所述）。因此，这里的所有内容*默认*都是模块的私有内容。

要在 CommonJS 模块的公共 API 上公开某些内容，需要向作为 `module.exports` 提供的空对象添加一个属性。在一些较旧的遗留代码中，您可能会遇到对仅有的 `exports` 的引用，但为了代码的清晰度，您应始终使用 `module.` 前缀对该引用进行完全限定。

出于风格考虑，我喜欢将 "exports" 放在顶部，而将模块实现放在底部。但这些导出可以放在任何地方。我强烈建议将它们集中在一起，放在文件的顶部或底部。

有些开发人员习惯于替换默认导出为对象，就像这样：

```js
// 为 API 定义新对象
module.exports = {
    // ..exports..
};
```

这种方法有一些怪异之处，包括多个此类模块相互循环依赖时出现意外行为。因此，我建议不要替换对象。如果你想使用对象字面风格定义一次分配多个导出，你可以这样做：

```js
Object.assign(module.exports, {
    // .. exports ..
});
```

这里做的事情是在定义 `{ .. }` 然后，`Object.assign(..)` 将所有这些属性浅拷贝到现有的 `module.exports` 对象上，而不是替换它。

要在模块/程序中包含另一个模块实例，请使用 Node 的 `require(..)` 方法。假设该模块位于 "/path/to/student.js"，我们可以这样访问它：

```js
var Student = require("/path/to/student.js");

Student.getName(73);
// Suzy
```

现在，`Student` 引用了我们示例模块的公共 API。

CommonJS 模块表现为单例，类似于之前介绍的 IIFE 模块定义风格。无论你多少次 `require(..)` 同一个模块，你都只会得到对单个共享模块实例的额外引用。

`require(..)` 是一种全有或全无的机制；它包括对模块整个公开 API 的引用。如果只想有效地访问 API 的一部分，典型的方法如下：

```js
var getName = require("/path/to/student.js").getName;

// 或者使用：

var { getName } = require("/path/to/student.js");
```

与传统模块格式类似，CommonJS 模块 API 的公开导出方法对模块内部细节进行闭包。这就是模块单例状态在整个程序生命周期中的维护方式。

> 注意：
>
> 在 Node `require("student")` 语句中，非绝对路径（`"student"`）假定文件扩展名为 ".js"，并搜索 "node_modules"。

现代 ES 模块 (ESM)

ESM 格式与 CommonJS 格式有几处相似之处。ESM 基于文件，模块实例是单例，*默认情况下*所有内容都是私有的。一个显著的不同点是，ESM 文件被假定为严格模式，而无需在顶部添加 `"use strict"` 编译指示。没有办法将 ESM 定义为非严格模式。

与 CommonJS 中的 `module.exports` 不同，ESM 使用 `export` 关键字来公开模块的公共 API。`import` 关键字取代了 `require(..)` 语句。让我们调整 "students.js" 以使用 ESM 格式：

```js
export { getName };

// ************************

var records = [
    { id: 14, name: "Kyle", grade: 86 },
    { id: 73, name: "Suzy", grade: 87 },
    { id: 112, name: "Frank", grade: 75 },
    { id: 6, name: "Sarah", grade: 91 },
];

function getName(studentID) {
    var student = records.find((student) => student.id == studentID);
    return student.name;
}
```

这里唯一的变化是 `export { getName }` 语句。和以前一样，`export` 语句可以出现在整个文件的任何地方，但 `export` 必须位于顶层作用域；它不能位于任何其他代码块或函数内部。

对于如何指定 `export` 语句，ESM 提供了相当多的写法。例如：

```js
export function getName(studentID) {
    // ..
```

```
}
```

尽管 `export` 出现在这里的 `function` 关键字之前，但这个表单仍然是一个 `function` 声明，而且恰好也是导出的。也就是说，`getName` 标识符是*函数提升*（参见第 5 章），因此它在整个模块作用域内都可用。

另一种写法：

```js
export default function getName(studentID) {
    // ..
}
```

这就是所谓的「默认导出」，其语义与其他导出不同。从本质上讲，「默认导出」是模块消费者在「导入」时的一种速记，当他们只需要这个单一的默认 API 成员时，会给他们提供一种更简洁的语法。

非默认（`default`）导出称为「命名导出」。

`import` 关键字与 `export` 关键字一样，只能在任何模块或函数之外的 ESM 顶层使用，其语法也有多种变化。第一种称为「命名导入」：

```js
import { getName } from "/path/to/students.js";

getName(73); // Suzy
```

如您所见，这种形式只导入模块中已明确命名的公共 API 成员（跳过任何未明确命名的成员），并将这些标识符添加到当前模块的顶层作用域中。这种导入方式对于那些习惯于使用 Java 等语言中的包导入的人来说并不陌生。

可在 `{ .. }` 集合中列出多个 API 成员，中间用逗号隔开。还可以使用 `as` 关键字对已命名的导入进行*重命名*：

```js
import { getName as getStudentName } from "/path/to/students.js";

getStudentName(73);
// Suzy
```

如果 `getName` 是模块的「默认导出」，我们可以这样导入它：

```js
import getName from "/path/to/students.js";
```

```
getName(73); // Suzy
```

这里唯一的区别是去掉了导入周围的 `{ }`。如果要将默认导入与其他已命名的导入混合使用：

```js
import { default as getName /* .. others .. */ } from "/path/to/students.js";

getName(73); // Suzy
```

相比之下，`import` 的另一个主要变体称为「命名空间导入」：

```js
import * as Student from "/path/to/students.js";

Student.getName(73); // Suzy
```

显而易见，`*` 会导入所有输出到 API 的内容，包括默认内容和已命名内容，并将其存储在指定的单一命名空间标识符下。这种方法最接近 JS 历史上大多数传统模块的形式。

> **注意：**
>
> 截至本文撰写之时，现代浏览器支持 ESM 已有数年时间，而 Node 对 ESM 的稳定支持则是最近才开始的，并且已经发展了相当长的一段时间。这种演进可能还要持续一年或更长时间；ES6 将 ESM 引入 JS 时，为 Node 与 CommonJS 模块的互操作带来了许多具有挑战性的兼容性问题。有关所有最新细节，请查阅 Node 的 ESM 文档：https://nodejs.org/api/esm.html。

周天圆满

无论是使用经典的模块格式（浏览器或 Node）、CommonJS 格式（在 Node 中）还是 ESM 格式（浏览器或 Node），模块都是结构化和组织程序功能和数据的最有效方法之一。

模块模式是我们在本书中学习如何使用词法作用域规则将变量和函数放置在适当位置的旅程的终点。POLE 是我们一贯采取的*默认情况下的私有*防御姿态，确保我们避免过度暴露，只与最小的公共 API 表面区域进行必要的交互。

在模块之下，利用词法作用域系统的闭包是维护所有模块状态的*法宝*。

正文到此结束。恭喜你完成了这一段旅程！正如我在整个过程中多次说过的，暂停、思考和实践我们刚刚讨论的内容是一个非常好的主意。

当您感到舒适并准备就绪时，请查看附录，这些附录深入探讨了这些主题的一些角落，还通过一些练习题来巩固您所学的知识。

附录 A：欲穷千里目，更上一层楼

现在，我们将围绕本书正文中涉及的许多主题，探讨一些细微差别和边缘问题。本附录为可选的辅修资料。

有些人认为，过于深入地研究细枝末节的案例和不同的意见只会制造噪音和分散注意力。他们认为，开发人员最好坚持走常人走过的路。有人批评我的方法不切实际、适得其反。我理解并欣赏这种观点，尽管我并不一定赞同。

我认为，了解事物是如何运作的，要比凭空臆想和缺乏好奇心来忽略细节要好得多。归根结底，你会遇到这样的情况：在你未曾探索过的地方，会冒出一些新的东西。换句话说，你不可能一直走在平坦的「幸福之路」上。难道你不愿意为越野过程中不可避免的颠簸做好准备吗？

与正文相比，这些讨论也会更多地受到我的观点的影响，因此在阅读和思考所介绍的内容时，请记住这一点。本附录有点像小型博客文章集，详细阐述了书中的各种主题。它篇幅很长，内容很深，所以请慢慢阅读，不要急于看完这里的所有内容。

隐式作用域

作用域有时会在不显眼的地方创建。实际上，这些隐式作用域并不总会影响程序的运行，但知道它们的存在还是很有用的。请留意以下令人惊讶的作用域：

- 参数作用域
- 函数名的作用域

参数作用域

第 2 章中的对话隐喻意味着函数参数基本上等同于函数作用域中本地声明的变量。但事实并非总是如此。

想想看：

```js
// 外部/全局作用域：红色 (1)

function getStudentName(studentID) {
    // 函数作用域：蓝色 (2)
    // ..
}
```

在这里，`studentID` 被视为一个「简单」的参数，因此它确实是蓝色（2）函数作用域的成员。但如果我们把它改为非简单参数，技术上就不再是这样了。被视为非简单参数的参数形式包括带默认值的参数、其余参数（使用 `...`）和非结构化参数。

想想看：

```js
// 外部/全局作用域：红色 (1)

function getStudentName(/*蓝色 (2)*/ studentID = 0) {
    // 函数作用域：绿色 (3)
    // ..
}
```

在这里，参数列表实质上成为了它自己的作用域，而函数的作用域则嵌套在*那个*作用域内。

为什么？有什么区别？非简单的参数形式会带来各种各样的情况，因此参数列表变成了自己的作用域，以便更有效地处理这些情况。

想想看：

```js
function getStudentName(studentID = maxID, maxID) {
    // ..
}
```

假设从左到右操作，`studentID` 参数的默认 `= maxID` 要求 `maxID` 已经存在（并且已经初始化）。这段代码会产生 TDZ 错误（第 5 章）。原因是 `maxID` 已在参数作用域中声明，但由于参数的顺序，它尚未被初始化。如果调换参数顺序，就不会发生 TDZ 错误：

```js
function getStudentName(maxID, studentID = maxID) {
    // ..
}
```

如果我们在默认参数位置引入一个函数表达式，它就可以在这个隐含参数作用域中的参数上创建自己的闭包（第 7 章），那么情况就会变得更加复杂：

```js
function whatsTheDealHere(id, defaultID = () => id) {
    id = 5;
    console.log(defaultID());
}
```

```
whatsTheDealHere(3);
// 5
```

该代码段可能是合理的，因为 `defaultID()` 箭头函数封闭了 `id` 参数/变量，然后我们将其重新赋值给 `5`。但现在让我们在函数作用域中引入 `id` 的遮蔽定义：

```js
function whatsTheDealHere(id, defaultID = () => id) {
    var id = 5;
    console.log(defaultID());
}

whatsTheDealHere(3);
// 3
```

啊哦！`var id = 5` 是对参数 `id` 的遮蔽，但函数 `defaultID()` 的闭包是在参数上，而不是在函数体中的遮蔽变量上。这证明在参数列表周围存在一个作用域气泡。

但还有比这更疯狂的！

```js
function whatsTheDealHere(id, defaultID = () => id) {
    var id;

    console.log(`local variable 'id': ${id}`);
    console.log(`parameter 'id' (closure): ${defaultID()}`);

    console.log("reassigning 'id' to 5");
    id = 5;

    console.log(`local variable 'id': ${id}`);
    console.log(`parameter 'id' (closure): ${defaultID()}`);
}

whatsTheDealHere(3);
// local variable 'id': 3   <--- Huh!? Weird!
// parameter 'id' (closure): 3
// reassigning 'id' to 5
// local variable 'id': 5
// parameter 'id' (closure): 3
```

这里最奇怪的是第一条控制台消息。此时，遮蔽的 `id` 局部变量刚刚被声明为 `var id`，而第 5 章断言该变量通常会在其作用域的顶端自动初始化为 `undefined`。为什么不打印 `undefined`？

在这种特殊情况下（出于传统兼容的原因），JS 不会将 `id` 自动初始化为 `undefined`，而是初始化为参数 `id` 的值（`3`）！

虽然此时两个 `id` 看起来像是一个变量，但实际上它们仍然是分开的（并且在不同的作用域中）。`id = 5` 赋值使分歧可观察到，其中 `id` 参数保持 `3`，而局部变量变成了 `5`。

我的建议是，避免被这些奇怪的细微差别所困扰：

- 切勿使用本地变量作为参数的遮蔽
- 避免使用封闭任何参数的默认参数函数

至少你现在已经意识到，如果有任何参数是非简单参数，那么参数列表就是它自己的作用域。

函数名的作用域

在第 3 章「函数名的作用域」一节中，我断言函数表达式的名称会被添加到函数自身的作用域中。请回想一下：

```js
var askQuestion = function ofTheTeacher() {
    // ..
};
```

的确，`ofTheTeacher` 并没有被添加到外层作用域（`askQuestion` 被声明的地方），但它也不是像您假设的那样*只是*被添加到函数的作用域。这是另一个奇怪的隐式作用域细节。

函数表达式的名称标识符在它自己的隐式作用域中，嵌套在外层作用域和主内部函数作用域之间。

如果 `ofTheTeacher` 在函数的作用域中，我们就会在这里发现错误：

```js
var askQuestion = function ofTheTeacher() {
    // 为什么这不是重复声明错误？
    let ofTheTeacher = "Confused, yet?";
};
```

`let` 声明形式不允许重新声明（参见第 5 章）。但这是完全合法的映射，而不是重新声明，因为两个 `ofTheTeacher` 标识符处于不同的作用域中。

你很少会遇到函数名称标识符的作用域的情况。不过，了解这些机制的实际工作原理还是有好处的。为了避免踩坑，永远不要对函数名标识符进行遮蔽处理。

匿名函数与具名函数

正如第 3 章所述，函数可以用命名或匿名形式表达。使用匿名形式更为常见，但这是个好主意吗？

在考虑为您的函数命名时，请先脑补一下：

- 命名推断不完整
- 词法命名允许自我引用
- 命名是有用的描述
- 箭头函数没有词法命名
- IIFE 也需要命名

显示命名还是推断命名？

程序中的每个函数都有其用途。如果没有目的，就把它删除，因为你只是在浪费空间。如果它*有*用途，那么它*就有*用途的名称。

到目前为止，许多读者可能都同意我的观点。但这是否意味着我们应该总是把这个名字写入代码呢？在这里，我会让很多人瞠目结舌。我明确地说，是的！

首先，堆栈跟踪中显示的「匿名（anonymous）」对调试没有什么帮助：

```js
btn.addEventListener("click", function () {
    setTimeout(function () {
        ["a", 42].map(function (v) {
            console.log(v.toUpperCase());
        });
    }, 100);
});
// Uncaught TypeError: v.toUpperCase is not a function
//     at myProgram.js:4
//     at Array.map (<anonymous>)
//     at myProgram.js:3
```

哎呀。与我给出函数名称后的报错相比：

```js
btn.addEventListener("click", function onClick() {
    setTimeout(function waitAMoment() {
        ["a", 42].map(function allUpper(v) {
            console.log(v.toUpperCase());
        });
```

```
    }, 100);
});
// Uncaught TypeError: v.toUpperCase is not a function
//     at allUpper (myProgram.js:4)
//     at Array.map (<anonymous>)
//     at waitAMoment (myProgram.js:3)
```

`waitAMoment` and `allUpper` 名称是如何出现并为堆栈跟踪提供更多有用的调试信息/上下文的？如果我们为所有函数使用合理的命名，程序就更容易调试。

> **注意：**
>
> 不幸的是，"<anonymous>" 仍然出现，这是因为 `Array.map(..)` 的实现并不存在于我们的程序中，而是内置于 JS 引擎中。这并不是因为我们的程序引入了可读性快捷方式而造成的混淆。

顺便说一下，让我们对什么是具名函数有一个统一的认识：

```js
function thisIsNamed() {
    // ..
}

ajax("some.url", function thisIsAlsoNamed() {
    // ..
});

var notNamed = function () {
    // ..
};

makeRequest({
    data: 42,
    cb /* 也不是命名 */: function () {
        // ..
    },
});

var stillNotNamed = function butThisIs() {
    // ..
};
```

「但等等！」你说。其中一些*是*具名的，对吧！？

```js
var notNamed = function () {
    // ..
```

```
};

var config = {
    cb: function () {
        // ..
    },
};

notNamed.name;
// notNamed

config.cb.name;
// cb
```

这些被称为*推断*名命名。推断命名很好，但它们并不能真正解决我所讨论的全部问题。

缺少命名？

是的，这些推断命名可能会显示在堆栈跟踪中，这肯定比显示「匿名」要好。但是...

```js
function ajax(url, cb) {
    console.log(cb.name);
}

ajax("some.url", function () {
    // ..
});
// ""
```

哎呀。作为回调传递的匿名 `function` 表达式无法接收推断命名，因此 `cb.name` 只保留空字符串 `""`。绝大多数的 `function` 表达式，尤其是匿名表达式，都被用作回调参数；这些表达式都没有命名。因此，依靠命名推断充其量只是不完整的推断。

推断的不足之处不仅仅在于回调：

```js
var config = {};

config.cb = function () {
    // ..
};

config.cb.name;
// ""
```

```
var [noName] = [function () {}];
noName.name;
// ""
```

函数表达式的任何赋值如果不是*简单赋值*，也会导致命名推断失败。因此，换句话说，除非你小心谨慎，否则程序中几乎所有的匿名函数表达式实际上都没有命名。

推断命名只是...不太够用。

即使一个函数表达式*确实*得到了一个推断命名，它仍然不能算作一个完整的具名函数。

我是谁?

如果没有词法名称标识符，函数就没有内部方式来引用自身。自我引用对于递归和事件处理等工作非常重要：

```js
// broken
runOperation(function (num) {
    if (num <= 1) return 1;
    return num * oopsNoNameToCall(num - 1);
});

// 同样 broken
btn.addEventListener("click", function () {
    console.log("should only respond to one click!");
    btn.removeEventListener("click", oopsNoNameHere);
});
```

如果不在回调中使用词法名称，就很难可靠地自我引用函数。您*可以*在引用函数的外层作用域中声明一个变量，但这个变量是由外层作用域*控制*的，它可能被重新赋值等。因此它不如函数有自己的内部自引用那么可靠。

命名即描述

最后，我认为最重要的一点是，省略函数名称会让读者更难一眼看出函数的用途。他们必须阅读更多的代码，包括函数内部的代码和函数外部的代码，才能弄明白。

想想看：

```js
[1, 2, 3, 4, 5].filter(function (v) {
    return v % 2 == 1;
});
// [ 1, 3, 5 ]
```

```
[1, 2, 3, 4, 5].filter(function keepOnlyOdds(v) {
    return v % 2 == 1;
});
// [ 1, 3, 5 ]
```

没有任何合理的理由可以证明，**省略**第一个回调的名称 `keepOnlyOdds` 可以更有效地向读者传达这个回调的目的。你节省了 13 个字符，却失去了重要的可读性信息。`keepOnlyOdds` 这个名称可以非常清楚地告诉读者，第一眼看到它的时候，发生了什么。

JS 引擎并不关心命名。但阅读代码的人类读者绝对会关心。

读者能否通过观察 `v % 2 == 1` 来知道它在做什么？当然可以，但他们必须通过在头脑中执行代码来推断其目的（和名称）。即使是短暂的停顿，也会减慢代码的阅读速度。一个好的描述性命名可以让这一过程几乎毫不费力，而且立竿见影。

试想一下：代码的作者在将函数名称添加到代码中之前，需要弄清楚函数的用途多少次？大约一次。如果需要调整命名，可能需要两三次。但这段代码的读者需要多少次弄清命名/目的呢？每一次阅读这行代码。成百上千次？还是更多？

无论函数的长度或复杂程度如何，我的主张是，作者都应该想出一个好的描述性命名，并将其添加到代码中。即使是 `map(..)` 和 `then(..)` 语句中的单行函数也应该命名：

```js
lookupTheRecords(someData)
    .then(function extractSalesRecords(resp) {
        return resp.allSales;
    })
    .then(storeRecords);
```

`extractSalesRecords` （提取销售记录）这个名称告诉读者这个 `then(..)` 处理程序的目的，比仅仅从精神上执行 `return resp.allSales` 来推断要好。

不为函数命名的唯一借口要么是懒惰（不想多输入几个字符），要么是没有创意（想不出好名字）。如果你想不出一个好名字，很可能是你还不了解这个函数及其用途。这个功能可能设计得不好，或者做了太多事情，应该重新设计。一旦你有了一个设计良好、用途单一的函数，它的正确名称就应该显而易见了。

我有一个小窍门：在第一次编写一个函数时，如果我不完全理解它的目的，也想不出一个好名字，我就用 `TODO` 作为它的名字。这样，在以后审查我的代码时，我就有可能发现这些名称占位符，我就更愿意（也更有准备！）回头去想一个更好的名称，而不是仅仅把它作为 `TODO`。

所有函数都需要名称。每一个都是。没有例外。省略任何命名都会使程序更难阅读、更难调试、更难扩展和维护。

箭头函数

箭头函数**始终是**匿名的，即使（很少）在使用时赋予了它一个推断命名。我刚刚花了好几页纸解释为什么匿名函数是个坏主意，所以你大概能猜到我对箭头函数的看法。

不要用它们来替代常规函数。它们的确更简洁，但这种简洁是以省略关键的视觉分隔符为代价的，而视觉分隔符可以帮助我们的大脑快速分辨出我们正在阅读的内容。而且，就本次讨论的重点而言，它们是匿名的，从这个角度来看，可读性也更差。

箭头函数有其目的，但不是为了节省击键次数。箭头函数具有*lexical this*[1]行为，这在某种程度上超出了我们在本书中讨论的范围。

简而言之：箭头函数根本没有定义 `this` 标识符关键字。如果在箭头函数中使用 `this`，它的行为与其他变量引用完全相同，即通过查询作用域链来找到它*被*定义的函数作用域（非箭头函数），并使用该函数作用域。

换句话说，箭头函数对待 `this` 就像对待其他词法变量一样。

如果你习惯于使用 `var self = this` 这样的 hack 手段，或者如果你喜欢在内部函数表达式上调用 `.bind(this)`，只是为了迫使它们像词法变量一样从外部函数继承一个 `this`，那么 `=>` 箭头函数绝对是更好的选择。它们就是专门为解决这个问题而设计的。

因此，在极少数情况下，如果您需要*lexical this*，请使用箭头函数。它是最合适的工具。但需要注意的是，这样做的同时，你也接受了匿名函数的缺点。你应该花更多的精力来减轻可读性的*代价*，比如使用更多描述性的变量名和代码注释。

IIFE 变体

所有函数都应该有名称。我说过好几次了，对吧？包括 IIFE。

```js
(function () {
    // 不要这么做！
})();

(function doThisInstead() {
    // ..
})();
```

如何为 IIFE 取名？确定 IIFE 的用途。为什么需要在该处设置作用域？您是否隐藏了一个「学生记录（studentRecords）」的缓存变量？

```js
var getStudents = (function StoreStudentRecords() {
    var studentRecords = [];
```

```
    return function getStudents() {
        // ..
    };
})();
```

我将 IIFE 命名为 `StoreStudentRecords`（存储学生记录），因为这就是它的工作：存储学生记录。每个 IIFE 都应该有一个名字。没有例外。

IIFE 通常是通过在 `function` 表达式周围放置 `( .. )` 来定义的，如前面的代码段所示。但这并不是定义 IIFE 的唯一方法。从技术上讲，我们使用第一组周围的 `( .. )` 的唯一原因是，这样 `function` 关键字在 JS 解析器中就不会被视为 `function` 声明。但还有其他语法方式可以避免被解析为声明：

```js
!(function thisIsAnIIFE() {
    // ..
})();

+(function soIsThisOne() {
    // ..
})();

~(function andThisOneToo() {
    // ..
})();
```

`!`，`+`，`~` 和其他一些一元运算符（只有一个操作数的运算符）都可以放在 `function` 前面，将它变成一个表达式。然后，最后的 `()` 调用是有效的，这使它成为一个 IIFE。

实际上，我挺喜欢在定义独立的 IIFE 时使用 `void` 一元操作符：

```js
void (function yepItsAnIIFE() {
    // ..
})();
```

`void` 的好处是，它在函数的开头清楚地表明，这个 IIFE 不会返回任何值。

无论如何定义 IIFE，都要给它们起个名字，以示关爱。

提升：函数和变量

第 5 章阐述了*函数提升*和*变量提升*。由于提升经常被认为是 JS 设计中的失误，我想简要探讨一下为什么这两种形式的提升都*是*有益的，而且仍应加以考虑。

通过思考以下方面的优点，对提升进行更深层次的理解：

- 可执行代码在前，函数声明在后
- 变量声明的语义位置

函数提升

回顾一下，这个程序的工作原理是*函数提升*：

```js
getStudents();

// ..

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

函数声明在编译过程中被提升，这意味着 `getStudents` 是为整个作用域声明的标识符。此外，`getStudents` 标识符会通过函数引用自动初始化，同样是在作用域的开头。

这有什么用？我更喜欢使用*函数提升*的原因是，它将任何作用域中的*可执行的*代码放在最上面，而任何进一步的声明（函数）则放在下面。这就意味着可以更容易地找到将在任何给定区域中运行的代码，而不必不停地滚动，希望在某个地方找到标记作用域/函数结束的尾部 `}`。

我在各级作用域内都利用了这种反向定位的优势：

```js
getStudents();

// *************

function getStudents() {
    var whatever = doSomething();

    // 其他内容

    return whatever;

    // *************

    function doSomething() {
        // ..
```

```
    }
}
```

当我第一次打开这样的文件时，第一行就是启动其行为的可执行代码。这很容易发现！然后，如果我需要找到并检查 `getStudents()`，我喜欢它的第一行也是可执行代码。只有当我需要查看 `doSomething()` 的细节时，我才会去下面查找它的定义。

换句话说，我认为*函数提升*通过从上到下的流畅、渐进的阅读顺序，使代码更具可读性。

变量提升

那么*变量提升*呢？

即使是 `let` 和 `const` 变量，也不能在它们的 TDZ 中使用（参见第 5 章）。因此，下面的讨论只适用于 `var` 声明。在继续之前，我得承认：几乎在所有情况下，我都完全同意*变量提升*是个坏主意：

```js
pleaseDontDoThis = "bad idea";

// 很久以后
var pleaseDontDoThis;
```

虽然这种颠倒的排序对*函数提升*很有帮助，但在这里，我认为它通常会使代码更难推理。

不过，在我自己的代码编写过程中，我发现了一个例外情况，而且是很少见的例外情况。这与我在 CommonJS 模块定义中放置 `var` 声明的位置有关。

以下是我在 Node 中通常使用的模块定义结构：

```js
// 依赖
var aModuleINeed = require("very-helpful");
var anotherModule = require("kinda-helpful");

// 公共 API
var publicAPI = Object.assign(module.exports, {
    getStudents,
    addStudents,
    // ..
});

// ********************************
// 私有实现
```

```
var cache = {};
var otherData = [];

function getStudents() {
    // ..
}

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

注意到 `cache` 和 `otherData` 变量位于模块布局的「私有」部分了吗？这是因为我不打算公开它们。因此，我在组织模块时，将它们与模块的其他隐藏实现细节放在了一起。

但在一些罕见的情况下，我需要在声明模块的导出公共 API 之前，在上面对这些值进行赋值。例如：

```js
// 公共 API
var publicAPI = Object.assign(module.exports, {
    getStudents,
    addStudents,
    refreshData: refreshData.bind(null, cache),
});
```

我需要为 `cache` 变量赋值，因为该值将用于公共 API（`.bind(..)` 部分应用程序）的初始化。

我是否应该将 `var cache = { .. }` 移到顶部，在公共 API 初始化的上方？也许吧。但现在就不那么明显了，`var cache` 是一个*私有*实现细节。以下是我曾经（很少）使用过的折中方案：

```js
cache = {}; // 此处使用，但在下面声明

// 公共 API
var publicAPI = Object.assign(module.exports, {
    getStudents,
    addStudents,
    refreshData: refreshData.bind(null, cache),
});

// ********************************
// 私有实现
```

```
var cache /* = {}*/;
```

看到*变量提升*了吗？从逻辑上讲，我已经在下面声明了 `cache`，但在这种罕见的情况下，我在上面需要初始化的地方提前使用了它。我甚至还在代码注释中提示了分配给 `cache` 的值。

这是我发现的唯一一种利用*变量提升*来在作用域中比变量声明更早地赋值的情况。但我认为这是一个合理的例外，应谨慎使用。

使用 `var` 的理由

说到*变量提升*，让我们来谈谈 `var`。开发人员最喜欢将 JS 开发中的许多问题归咎于它的「连环套」。在第 5 章中，我们探讨了 `let` / `const` 的问题，并承诺将重新审视 `var` 在整个组合中的位置。

当我陈述案例时，千万不要错过：

- `var` 从未被弃用
- `let` 你的朋友
- `const` 效用有限
- 两全其美：`var` *和* `let`

不要扔掉 `var`

`var` 很好，运行的也很好。它已经存在了 25 年，在未来的 25 年甚至更长的时间里，它都将是有用的、功能强大的。声称 `var` 被破坏、被弃用、过时、危险或设计不合理，都是虚假的哗众取宠。

这是否意味着 `var` 是程序中每个声明的正确声明符？当然不是。但它在你的程序中仍有一席之地。因为团队中有人选择了激进的 linter[2] 观点而拒绝使用 `var` 就等于是掩耳盗铃。

好吧，既然我已经把你激怒了，那就让我来解释一下我的立场吧。

声明一下，我是块作用域声明的 `let` 的粉丝。我非常不喜欢 TDZ，我认为那是个错误。但 `let` 本身很棒。我经常使用它。事实上，我使用它的次数可能和使用 `var` 的次数一样多，甚至更多。

感到迷茫的 `const`

另一方面，我并不经常使用 `const`。我不想深究其中的原因，但归根结底，还是因为 `const` 无法承担其自身的重量。也就是说，虽然 `const` 在某些情况下有一点点好处，但在 JS 中出现之前，它就已经在各种语言中造成了长期的混乱。

`const` 假装创建了不能被修改的值。在许多语言的开发者社区中这是一个非常普遍的误解。而它真正的作用是防止重新赋值。

```js
const studentIDs = [14, 73, 112];

// 然后

studentIDs.push(6); // 哇偶,等等……什么！？
```

在可变值（如数组或对象）中使用 `const`，会让未来的开发人员（或您代码的读者）掉入您设置的陷阱，即他们要么不知道，要么有点忘了，*值不变性*与*赋值不变性*完全不是一回事。

我只是觉得我们不应该设置这些陷阱。我唯一一次使用 `const` 是在给一个已经不可更改的值（比如 `42` 或 `"Hello, friends!"`）赋值的时候，而且从语义上来说，它显然是一个「常量」，是一个字面值的命名占位符。这就是 `const` 的最佳用途。不过这在我的代码中很少见。

如果变量重赋值是个大问题，那么 `const` 就会更有用。但就导致错误而言，变量重赋值并不是什么大问题。导致程序错误的原因有一长串，但「意外的重新赋值」在那一长串中排得很靠后。

再加上 `const`（和 `let`）应该在代码块中使用，而代码块应该是很短的，所以在代码中 `const` 声明适用的区域非常小。十行代码块中第一行的 `const` 声明只能告诉你后面九行的内容。而它告诉你的东西，只要往下看这九行就已经很明显了：变量永远不会在 `=` 的左侧；它不会被重新赋值。

仅此而已，这就是 `const` 的真正作用。除此之外，它就没什么用了。在值与赋值不变性的重大混淆中，`const` 失去了很多光彩。

从不重新赋值的 `let`（或 `var`！）在行为上已经是一个「常量」，尽管它没有编译器的保证。这在大多数情况下已经足够了。

`var` *和* `let`

在我看来，`const` 很少有用，所以这只是 `let` 和 `var` 之间的两匹赛马竞争。但这也不是一场真正的比赛，因为不一定只有一个赢家。它们可以同时赢得……不同的比赛。

事实上，您应该在程序中同时使用 `var` 和 `let`。它们不能互换：在需要使用 `let` 的地方不应使用 `var`，但在最适合使用 `var` 的地方也不应使用 `let`。

那么，在什么情况下我们仍然应该使用 `var`？在什么情况下它比 `let` 更合适？

首先，我总是在任何函数的顶层作用域中使用 `var`，不管它是在函数的开头、中间还是结尾。我也会在全局作用域中使用 `var`，不过我会尽量减少全局作用域的使用。

为什么在函数作用域内使用 `var`？因为这正是 `var` 的作用。25 年来，没有任何工具能比一个声明器更好地完成对声明进行函数作用域界定的工作了。

您*可以*在这个顶层作用域中使用 `let`，但它并不是最好的工具。我还发现，如果在所有地方都使用 `let`，那么哪些声明是为了本地化而设计的，哪些声明是为了在整个函数中使用的，就不那么明显了。

相比之下，我很少在代码块中使用 `var`。这就是 `let` 的作用。使用最合适的工具。如果你看到 `let`，它告诉你正在处理一个本地化声明。如果你看到 `var`，它告诉你你正在处理的是一个函数作用域的声明。就这么简单。

```js
function getStudents(data) {
    var studentRecords = [];

    for (let record of data.records) {
        let id = `student-${ record.id }`;
        studentRecords.push({
            id,
            record.name
        });
    }

    return studentRecords;
}
```

`studentRecords` 变量用于整个函数。`var` 是告诉读者这一点的最佳声明符。相比之下，`record` 和 `id` 仅用于循环迭代的较小作用域，因此 `let` 是最好的工具。

除了这个*最佳工具*语义论证外，`var` 还有一些其他特性，在某些有限的情况下，这些特性使它变得更加强大。

其中一个例子是，当一个循环专门使用一个变量时，其条件子句却无法看到迭代内部的块作用域声明：

```js
function commitAction() {
    do {
        let result = commit();
        var done = result && result.code == 1;
```

```
    } while (!done);
}
```

在这里，`result` 显然只能在代码块内部使用，所以我们使用 `let`。但 `done` 有点不同。它只对循环有用，但 `while` 子句无法看到出现在循环内部的 `let` 声明。所以我们折中使用 `var`，这样 `done` 就会被提升到可以看到它的外层作用域。

另一种方法：在循环外声明 `done`，将它与首次使用它的地方分开，要么必须选择一个默认值来赋值，要么更糟，让它不被赋值，从而让读者看起来模棱两可。我认为在循环内使用 `var` 更合适。

`var` 的另一个有用特性体现在非预期块内的声明上。非预期块是由于语法需要块而创建的块，但开发者的意图并不是创建一个本地化的作用域。非预期作用域的最佳示例是 `try..catch` 语句：

```js
function getStudents() {
    try {
        // 不是块作用域
        var records = fromCache("students");
    } catch (err) {
        // 哎呀，退回到默认值
        var records = [];
    }
    // ..
}
```

是的，还有其他结构化代码的方法。但考虑到各种权衡，我认为这是*最好*的方法。

我不想在 `try` 代码块之外声明 `records`（使用 `var` 或 `let`），然后在一个或两个代码块中对其赋值。我更希望初始声明总是尽可能靠近变量的第一次使用（理想情况下是同一行）。在这个简单的例子中，这只是几行的距离，但在实际代码中，这可能会增加到更多行。距离越大，就越难弄清楚你要赋值给哪个作用域中的哪个变量。在实际赋值时使用 `var` 就不会那么模糊。

另外，请注意我在 `try` 和 `catch` 块中都使用了 `var`。这是因为我想向读者表明，无论采用哪种路径，`records` 都会被声明。从技术上讲，这样做是可行的，因为 `var` 被提升到函数作用域一次。但这仍然是一个很好的语义信号，提醒读者 `var` 确保了什么。如果 `var` 只在其中一个代码块中使用，而你只阅读另一个代码块，你就不会那么容易发现 `records` 来自何处。

在我看来，这是 `var` 的一个小小的超能力。它不仅可以躲避无意的 `try..catch` 块，还可以在函数的作用域中多次出现。而 `let` 就做不到这一点。这并不坏，它实际上是一个小小的

有用功能。把 `var` 想象成一个声明式注释，每次使用时都会提醒你变量的来源。「啊哈，对了，它属于整个函数」。

这种重复标注的超能力在其他情况下也很有用：

```js
function getStudents() {
    var data = [];

    // 用数据做点什么
    // .. 50 多行代码 ..

    // 纯粹是一个注释来提醒我们
    var data;

    // 再次使用数据
    // ..
}
```

第二个 `var data` 并不是重新声明 `data`，只是为了读者的利益，注释 `data` 是一个函数作用域的声明。这样，读者就不需要滚动 50 多行代码来查找初始声明了。

我完全同意在整个函数作用域中为多种目的重复使用变量。我也完全不介意一个变量的两次使用之间相隔几行代码。在这两种情况下，使用 `var` 安全地「重新声明」（注释）的能力有助于确保无论我在函数的哪个位置，都能知道我的 `data` 来自何处。

遗憾的是，`let` 不能做到这一点。

还有其他一些细微差别和情况，`var` 可以提供一些帮助，但我不想再赘述了。我想说的是，在我们的程序中，`var` 可以与 `let` （以及偶尔使用的 `const`）一起发挥作用。你是否愿意创造性地使用 JS 语言提供的工具，向读者讲述一个更丰富的故事？

不要因为有人羞辱你，让你觉得 `var` 不再酷，就丢弃像 `var` 这样有用的工具。不要因为多年前的一次困惑而回避 `var`。学习这些工具，各尽其用。

TDZ 是怎么回事？

第 5 章解释了 TDZ（暂时性死区）。我们说明了它是如何发生的，但对于*为什么*首先要引入它的原因却一笔带过。让我们简单了解一下 TDZ 的动机。

TDZ 起源故事中的一些线索：

- `const` 不应改变

- 一切都与时间有关
- `let` 的行为应该更像 `const` 还是 `var`？

一切从何开始

实际上，TDZ 源自 `const`。

在早期的 ES6 开发工作中，TC39 不得不决定 `const`（和 `let`）是否要提升到代码块的顶部。他们决定将这些声明提升到与 `var` 类似的位置。如果不是这样的话，我想有些人的担心是对作用域中间遮蔽的混淆，例如：

```js
let greeting = "Hi!";

{
    // 这里应该打印什么?
    console.log(greeting);

    // .. 几行代码 ..

    // 现在对 `greeting` 变量进行遮蔽处理
    let greeting = "Hello, friends!";

    // ..
}
```

我们应该如何处理 `console.log(..)` 语句？让它打印 "Hi!" 对 JS 开发人员来说有意义吗？这似乎是个问题，因为遮蔽只在代码块的后半部分起作用，而在前半部分则不起作用。这并不是非常直观、类似 JS 的行为。因此，`let` 和 `const` 必须提升到代码块的顶部，在整个代码块中可见。

但是，如果 `let` 和 `const` 提升到代码块的顶部（就像 `var` 提升到函数的顶部一样），为什么 `let` 和 `const` 不像 `var` 那样自动初始化（到 `undefined`）呢？这就是主要的问题所在：

```js
{
    // 这里应该打印什么?
    console.log(studentName);

    // 然后

    const studentName = "Frank";
```

```
    // ..
}
```

让我们设想一下，`studentName` 不仅被提升到了代码块的顶部，而且还被自动初始化为 `undefined`。在程序块的前半部分，可以观察到 `studentName` 具有 `undefined` 值，例如我们的 `console.log(..)` 语句。一旦进入 `const studentName = ...` 语句，`studentName` 就会被赋值为 `"Frank"`。自此以后，`studentName` 就不能再被重新赋值了。

但是，一个常量有两个不同的观测值，先是 `undefined`，然后是 `"Frank"`，这奇怪对吗？这似乎有悖于我们对 「常量」的理解：「常量」只能有一个值。

所以……现在我们遇到了一个问题。我们不能将 `studentName` 自动初始化为 `undefined`（或任何其他值）。但变量必须在整个作用域中存在。我们该如何处理从变量首次存在（作用域开始）到变量赋值的这段时间呢？

我们称这段时间为「死区」，如「暂时性死区」(TDZ)。为了防止混淆，我们规定，在 TDZ 期间对变量的任何访问都是非法的，都会导致 TDZ 错误。

好吧，我必须承认，这种推理确实有些道理。

是谁「允许 (`let`)」TDZ 的?

但这只是 `const`。那么`let`呢？

TC39 做出了这样的决定：既然 `const` 需要一个 TDZ，那么 `let` 也需要一个 TDZ。事实上，如果我们让`let` 也有一个 TDZ，那么我们就能阻止人们做的那些丑陋的变量提升行为。

我的反驳是：如果你倾向于一致性，那就用 `var` 而不是 `const`；`let` 绝对比 `const` 更像 `var`。这一点尤其正确，因为他们已经选择了与 `var` 保持一致，因为他们已经把 `var` 提升到了作用域的顶端。让 `const` 成为它自己与 TDZ 的独特处理方式，让 TDZ 的答案纯粹是：通过始终在作用域的顶层声明常量来避免 TDZ。我认为这样会更合理。

但可惜，事情并非如此。`let` 有一个 TDZ，因为 `const` 需要一个 TDZ，因为 `let` 和 `const` 模仿 `var` 提升到（块）作用域的顶部。就是这样。太绕了？再读几遍。

同步回调还是闭包吗?

第 7 章介绍了两种不同的闭包模式：

- 闭包是一个函数实例，它会记住其外部变量，即使该函数在其他作用域中被传递和**调用**。

- 闭包是一个函数实例，它的作用域环境被就地保留，而对它的任何引用都会被传递并从其他作用域中**调用**。

这些模式并没有天壤之别，但它们确实从不同的角度切入。而这种不同的视角改变了我们对闭包的认定。

不要在闭包和回调中迷失方向：

- 回调到什么（或哪里）？
- 也许「同步回调」不是最好的标签
- 如果 *IIF* 函数不能移动，为什么需要闭包？
- 随着时间的推移而推迟是闭包的关键

什么是回调？

在重新讨论闭包之前，请允许我花一点时间来谈谈「回调（callback）」这个词。人们普遍认为，「回调（callback）」是*异步回调 (asynchronous callbacks)* 和*同步回调 (synchronous callbacks)* 的同义词。我不认为这是个好主意，所以我想解释一下原因，并建议我们不要使用这个词，而改用另一个词。

让我们先来理解一下*异步回调*，一个将在未来*稍后*调用的函数引用。在这种情况下，「回调」是什么意思？

这意味着当前代码已经完成或暂停，自身也已暂停，当稍后调用相关函数时，执行将重新进入暂停的程序，恢复执行。具体来说，重新进入的点就是被函数引用包裹的代码：

```js
setTimeout(function waitForASecond() {
    // 当计时器计时结束时
    // JS 应调用
}, 1000);

// 这是当前程序结束
// 或暂停的位置
```

在这种情况下，「回调」就很有意义了。JS 引擎通过在特定位置*回调*来恢复我们暂停的程序。好吧，回调是异步的。

同步回调？

那么*同步回调*呢？想想看：

```js
function getLabels(studentIDs) {
    return studentIDs.map(function formatIDLabel(id) {
```

```
        return `Student ID: ${String(id).padStart(6)}`;
    });
}

getLabels([14, 73, 112, 6]);
// [
//    "Student ID: 000014",
//    "Student ID: 000073",
//    "Student ID: 000112",
//    "Student ID: 000006"
// ]
```

我们应该把 `formatIDLabel(..)` 称作回调吗？通过调用我们提供的函数，`map(..)` 工具函数是否真的回调到我们的程序中？

因为程序没有暂停或退出，所以本身没有什么可调用的。我们只是将一个函数（引用）从程序的一部分传递到另一部分，然后立即调用它。

还有其他既定术语可能与我们正在做的事情相匹配：传递一个函数（引用），以便程序的另一部分可以代表我们调用它。你可能会认为这是*依赖注入*（DI[3]）或*控制反转*（IoC[4]）。

DI 可以概括为将必要的功能部分传递给程序的另一部分，以便其调用这些功能完成工作。对于上面的 `map(..)` 调用来说，这样的描述很恰当，不是吗？`map(..)` 知道要遍历列表的值，但不知道如何处理这些值。这就是我们将 `formatIDLabel(..)` 函数传递给它的原因。我们传递依赖关系。

IoC 是一个非常相似的相关概念。控制反转的意思是，不要让程序的当前区域控制正在发生的事情，而是将控制权交给程序的另一部分。我们将计算字符串标签的逻辑封装在函数 `formatIDLabel(..)` 中，然后将调用控制权交给 `map(..)`。

值得注意的是，Martin Fowler 认为 IoC 是框架与库的区别：对于库，你调用它的函数；而对于框架，它调用你的函数。[5]

在我们的讨论中，DI 或 IoC 都可以作为*同步回调*的替代标签。

但我有一个不同的建议。让我们把（以前称为）*同步回调*的函数称为*内部调用函数*（IIF）。是的，没错，我就是在模仿 IIFE。这类函数是*inter-invoked*（内部调用）的，意思是：另一个实体调用它们，而不是 IIFE，后者会立即调用自己。

*异步回调*与 IIF 之间有什么关系？*异步回调 (asynchronous callback)* 是异步而非同步调用的 IIF。

同步作用域？

既然我们已经把*同步回调*重新标记为 IIF，我们就可以回到我们的主要问题：IIF 是一个闭包吗？显然，IIF *必须*引用外层作用域中的变量才有可能成为闭包。前面的 `formatIDLabel(..)` IIF 并没有引用它自身作用域之外的任何变量，所以它肯定不是闭包。

如果 IIF 中确实有外部引用，是否是闭包？

```js
function printLabels(labels) {
    var list = document.getElementById("labelsList");

    labels.forEach(function renderLabel(label) {
        var li = document.createElement("li");
        li.innerText = label;
        list.appendChild(li);
    });
}
```

内部的 `renderLabel(..)` IIF 引用了外层作用域中的 `list`，因此它是一个*可以*有闭包的 IIF。但这里我们为闭包选择的定义/模型很重要：

- 如果`renderLabel(..)` 是一个**函数，它被传递到其他地方**，然后调用该函数，那么是的，`renderLabel(..)` 确实是闭包，因为闭包保留了对其原始作用域链的访问。
- 但是，如果像第 7 章中的另一种概念模型那样，`renderLabel(..)` 保留在原处，只把对它的引用传递给`forEach(..)`，那么当`renderLabel(..)` 在自己的作用域内同步执行时，还需要闭包来保留`renderLabel(..)` 的作用域链吗？

不，这只是正常的词法作用域。

要了解原因，请看 `printLabels(..)` 的另一种形式：

```js
function printLabels(labels) {
    var list = document.getElementById("labelsList");

    for (let label of labels) {
        // 只是在它自己的作用域中的一个普通函数调用，对吗？
        // 这并不是真正的闭包！
        renderLabel(label);
    }

    // **************

    function renderLabel(label) {
        var li = document.createElement("li");
        li.innerText = label;
        list.appendChild(li);
```

```
    }
}
```

这两个版本的 `printLabels(..)` 基本上是一样的。

后一个绝对不是闭包的例子，至少在任何有用或可观察的意义上都不是。这只是词法作用域。前一个版本中，`forEach(..)` 调用我们的函数引用，本质上是一样的。它也不是闭包，而只是一个普通的词法作用域函数调用。

推迟闭包

顺便提一下，第 7 章简要提到了偏函数和柯里化（它们*依赖*于闭包！）。这是一个可以使用手动柯里化的有趣场景：

```js
function printLabels(labels) {
    var list = document.getElementById("labelsList");
    var renderLabel = renderTo(list);

    // 这次绝对是闭包!
    labels.forEach(renderLabel);

    // **************

    function renderTo(list) {
        return function createLabel(label) {
            var li = document.createElement("li");
            li.innerText = label;
            list.appendChild(li);
        };
    }
}
```

我们赋值给 `renderLabel` 的内部函数 `createLabel(..)` 是在 `list` 上闭包的，因此闭包肯定被使用了。

闭包允许我们稍后再记住 `list`，而将实际标签创建逻辑的执行从 `renderTo(..)` 调用推迟到随后的 `forEach(..)` 调用 `createLabel(..)` IIF。在这里，这可能只是一个短暂的瞬间，但随着闭包从一个调用桥接到另一个调用，任何时间都可能过去。

类模块的变体

第 8 章介绍了经典的模块模式，它看起来像这样：

```js
var StudentList = (function defineModule(Student) {
    var elems = [];

    var publicAPI = {
        renderList() {
            // ..
        },
    };

    return publicAPI;
})(Student);
```

请注意，我们将 `Student`（另一个模块实例）作为依赖关系传入。但是，你可能会遇到这种模块形式的许多有用的变化。以下是一些识别这些变化的提示：

- 模块知道自己的 API 吗？
- 即使我们使用花哨的模块加载器，它也只是一个传统的模块
- 有些模块需要通用

我的 API 在哪里？

首先，大多数传统模块不会像我在代码中展示的那样定义和使用 `publicAPI` 。相反，它们通常看起来像：

```js
var StudentList = (function defineModule(Student) {
    var elems = [];

    return {
        renderList() {
            // ..
        },
    };
})(Student);
```

这里唯一的区别是直接返回作为模块公共 API 的对象，而不是先将其保存到内部的 `publicAPI` 变量中。到目前为止，大多数传统模块都是这样定义的。

但我非常喜欢并一直使用前一种 `publicAPI` 形式。理由有两个：

- `publicAPI` 是一个语义描述符，通过使对象的目的更加明显来提高可读性。

- 如果需要在模块生命周期内访问或修改 API，存储一个内部的 `publicAPI` 变量，引用返回的相同外部公共 API 对象，会非常有用。

  例如，您可能想从模块内部调用一个公开暴露的函数。或者，您可能想根据某些条件添加或删除方法，或者更新公开属性的值。

  无论如何，在我看来，*不*维护一个引用来访问我们自己的应用程序接口是非常愚蠢的。对吗？

异步模块定义 (AMD)

传统模块形式的另一种变体是 AMD 风格模块（几年前很流行），例如 RequireJS 工具支持的模块：

```js
define(["./Student"], function StudentList(Student) {
    var elems = [];

    return {
        renderList() {
            // ..
        },
    };
});
```

如果仔细观察 `StudentList(..)`，就会发现这是一个典型的模块工厂函数。在 `define(..)`（由 RequireJS 提供）的机制内，`StudentList(..)` 函数被执行，并传递给它作为依赖关系声明的任何其他模块实例。返回值是一个代表模块公共 API 的对象。

其原理（包括闭包的工作原理！）与我们在传统模块中探索的完全相同。

通用模块 (UMD)

我们要了解的最后一种变体是 UMD，它并不是一种特定的精确格式，而更像是一种非常相似的格式集合。它的设计目的是为可能在浏览器、AMD 风格加载器或 Node 中加载的模块创建更好的互操作性（无需任何构建工具转换）。我个人仍在使用 UMD 发布我的许多实用程序库。

以下是 UMD 的典型结构：

```js
(function UMD(name, context, definition) {
    // 由 AMD 风格的加载器 (loader) 加载?
    if (typeof define === "function" && define.amd) {
        define(definition);
    }
    // 在 Node 里？
```

```
    else if (typeof module !== "undefined" && module.exports) {
        module.exports = definition(name, context);
    }
    // 假设浏览器的独立脚本
    else {
        context[name] = definition(name, context);
    }
})("StudentList", this, function DEF(name, context) {
    var elems = [];

    return {
        renderList() {
            // ..
        },
    };
});
```

虽然 UMD 看起来有点不寻常，但它实际上只是一个 IIFE。

不同的是，IIFE 的主要函数表达式部分（位于顶部）包含一系列 `if..else if` 语句，用于检测模块加载时所处的三种支持环境。

通常调用 IIFE 的最后一个 `()` 被传递了三个参数： `"StudentsList"` 、 `this` 和另一个 `function` 表达式。如果将这些参数与其参数相匹配，就会发现它们分别是名称(`name`)、上下文（ `context` ）和定义（ `definition`)。 `"StudentList"` （ `name` ）是模块的名称标签，主要用于模块被定义为全局变量的情况。 `this` （ `context` ）通常是 `window` （又称全局对象；参见第 4 章），用于通过模块名称定义模块。

调用 `definition(..)` 实际是为了获取模块的定义，你会注意到，当然，这只是一个传统的模块形式！

毫无疑问，ESM（ES 模块）正在迅速流行和普及。但是，在过去 20 年里，有数百万个模块被编写出来，所有这些模块都使用了经典模块的 ESM 前变体，因此，当你遇到这些模块时，能够阅读和理解它们仍然非常重要。

---

1. *箭头函数*，https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Functions/Arrow_functions，2023年9月14日。 ↩
2. *lint*，https://zh.wikipedia.org/zh-cn/Lint，2021年10月07日。 ↩
3. *依赖注入*，https://zh.wikipedia.org/wiki/依赖注入，2023年1月7日。 ↩
4. *控制反转*，https://zh.wikipedia.org/zh-hans/控制反转，2022年9月1日。 ↩

5. *Inversion of Control*, Martin Fowler,https://martinfowler.com/bliki/InversionOfControl.html, 2005年6月26日。 ↩

附录 B：练习

本附录旨在为您提供一些具有挑战性的有趣练习，以测试和巩固您对本书主要内容的理解。最好能用实际的代码编辑器亲自尝试一下这些练习，而不是直接跳到最后的解答。不要作弊！

这些练习并没有特定的正确答案，你必须准确的理解它。您的方法可能与所提供的解决方案有一些（或很大！）不同，这没关系。

我们不会因为你如何编写代码而对你进行评判。我希望你能从本书中获得自信，相信自己可以在扎实的知识基础上完成这些编码任务。这就是本书的唯一目的。如果你对自己的代码感到满意，我也会！

弹珠桶

还记得第 2 章中的图 2 吗？

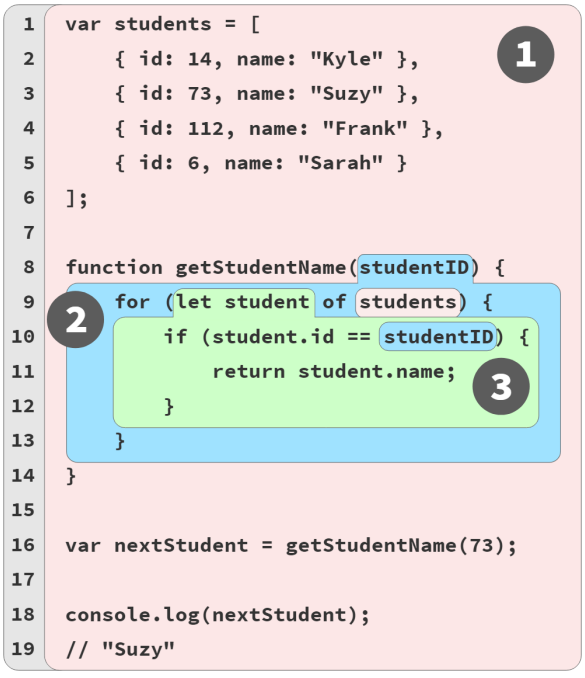

图 2（第 2 章）： 彩色作用域气泡

本练习要求你编写一个包含嵌套函数和块作用域的程序（任何程序！），并满足这些限制条件：

- 如果将所有作用域（包括全局作用域！）都染成不同的颜色，则至少需要六种颜色。请务必添加代码注释，为每个作用域标注其颜色。

  附加题：识别代码可能具有的任何隐式作用域。

- 每个作用域至少有一个标识符。
- 包含至少两个函数作用域和至少两个块作用域。
- 至少有一个外部作用域变量必须被嵌套作用域变量所遮蔽（见第 3 章）。
- 至少有一个变量引用必须解析到作用域链中至少高两级的变量声明。

> 贴士：
>
> 你*可以*写一些无意义的 foo/bar/baz 之类的代码来做这个练习，但我建议你试着写一些非繁琐的真实代码，至少能做一些合理的事情。

请自己尝试一下，然后查看本附录末尾的参考答案。

闭包（第 1 部分）

首先，让我们通过一些常见的计算机数学运算来练习闭包：判断一个数值是否是质数（除了 1 和它本身之外没有其他除数），以及生成给定数字的质因数（除数）列表。

例如：

```js
isPrime(11); // true
isPrime(12); // false

factorize(11); // [ 11 ]
factorize(12); // [ 3, 2, 2 ] --> 3*2*2=12
```

下面是 `是否是质数 isPrime(..)` 的实现，改编自 Math.js 库：[1]

```js
function isPrime(v) {
    if (v <= 3) {
        return v > 1;
    }
    if (v % 2 == 0 || v % 3 == 0) {
        return false;
    }
    var vSqrt = Math.sqrt(v);
    for (let i = 5; i <= vSqrt; i += 6) {
        if (v % i == 0 || v % (i + 2) == 0) {
            return false;
        }
```

```
    }
    return true;
}
```

下面是 因式分解 `factorize(..)` 不要与第 6 章中的 `factorial(..)` 混淆）的基本实现：

```js
function factorize(v) {
    if (!isPrime(v)) {
        let i = Math.floor(Math.sqrt(v));
        while (v % i != 0) {
            i--;
        }
        return [...factorize(i), ...factorize(v / i)];
    }
    return [v];
}
```

> 注意：
>
> 我之所以称其为「基本」，是因为它没有进行性能优化。它是二进制递归的（没有尾调优化），而且会创建大量的中间数组副本。它也没有以任何方式对发现的因子进行排序。有很多其他算法可以完成这项任务，但我想用一些简短且易于理解的算法来完成我们的练习。

如果你在一个程序中多次调用 `isPrime(4327)` ，你可以看到它每次都要经历几十个比较/计算步骤。如果考虑到 `factorize(..)` ，它在计算因数列表时会多次调用 `isPrime(..)` 。而这些调用很有可能大部分都是重复的。这就浪费了很多性能！

本练习的第一部分是使用闭包实现缓存，以记住 `isPrime(..)` 的结果，这样给定数字的基元性（`true` 或 `false` ）只需计算一次。提示：在第 6 章中，我们已经用 `factorial(..)` 演示了这种缓存。

如果你看一下 `factorize(..)` ，它是通过递归实现的，这意味着它会重复调用自己。这同样意味着我们可能会看到大量浪费的调用，来计算同一个数字的质因数。因此，练习的第二部分是对 `factorize(..)` 使用相同的闭包缓存技术优化。

使用单独的闭包缓存 `isPrime(..)` 和 `factorize(..)` ，而不是将它们放在一个作用域中。

请自己尝试一下，然后查看本附录末尾的参考答案。

关于内存

我想和大家分享一下关于这种闭包缓存技术的小提示，以及它对应用程序性能的影响。

我们可以看到，通过节省重复调用，我们提高了计算速度（在某些情况下，速度会大幅提高）。但是，这种闭包的使用是一种明确的权衡，你应该非常清楚。

代价就是内存。我们的缓存（内存）基本上是无限制增长的。如果相关函数被调用了数百万次，且大部分输入都是唯一的，那么我们就会占用大量内存。这样做绝对是值得的，但前提是我们认为我们很可能会看到重复的共同输入，这样我们才能利用缓存。

如果每次调用都有一个唯一的输入，而缓存基本上从未被*使用*过，那么采用这种技术就不合适了。

此外，采用更复杂的缓存方法（如 LRU（最近最少使用）缓存）来限制缓存的大小也是一个好主意；当缓存运行到极限时，LRU 会驱逐……嗯，最近最少使用的值！[2]

缺点是 LRU 本身并不简单。你需要使用高度优化的 LRU 实现，并清楚地意识到所有的权衡取舍。

闭包（第 2 部分）

在本练习中，我们将再次通过定义一个 `toggle(..)` 工具来练习闭包，该工具将为我们提供一个值切换器。

您将向 `toggle(..)` 传递一个或多个值（作为参数），并返回一个函数。在反复调用时，返回的函数将按顺序在所有传入的值之间交替/轮换，每次一个。

```js
function toggle(/* .. */) {
    // ..
}

var hello = toggle("hello");
var onOff = toggle("on", "off");
var speed = toggle("slow", "medium", "fast");

hello(); // "hello"
hello(); // "hello"

onOff(); // "on"
onOff(); // "off"
onOff(); // "on"

speed(); // "slow"
speed(); // "medium"
```

```
speed(); // "fast"
speed(); // "slow"
```

没有向 `toggle(..)` 传递任何值的边缘情况并不重要；这样的 toggler 实例可以总是返回 `undefined`。

请自己尝试一下，然后查看本附录末尾的参考答案。

闭包（第 3 部分）

在第三个也是最后一个关于闭包的练习中，我们将实现一个基本的计算器。`calculator(..)`（计算器）函数将生成一个计算器实例，它以函数的形式（`calc(..)`，下同）保持自己的状态：

```js
function calculator() {
    // ..
}

var calc = calculator();
```

每次调用 `calc(..)`，你都要输入一个字符，它代表按下计算器按钮的一次按键操作。为了保持简单明了，我们将限制计算器只支持输入数字（0-9）、算术运算（+，-，*，/）和 "=" 来计算运算。运算严格按照输入的顺序进行，没有 "( )" 分组或运算符优先。

我们不支持输入小数，但除法运算可以产生小数。我们不支持输入负数，但 "-" 操作可以产生负数。因此，只要先输入一个运算来计算负数或小数，就可以得出任何负数或小数。然后，您就可以继续使用该值进行计算。

`calc(..)` 调用的返回值应模拟真实计算器上的显示，如反映刚按下的内容，或在按下 "=" 时计算总数。

例如：

```js
calc("4"); // 4
calc("+"); // +
calc("7"); // 7
calc("3"); // 3
calc("-"); // -
calc("2"); // 2
calc("="); // 75
calc("*"); // *
```

```
calc("4"); // 4
calc("="); // 300
calc("5"); // 5
calc("-"); // -
calc("5"); // 5
calc("="); // 0
```

由于这种用法有点笨拙，这里有一个 `useCalc(..)` 助手，它可以使用字符串中的字符一个一个地运行计算器，并计算每次的显示结果：

```js
function useCalc(calc, keys) {
    return [...keys].reduce(function showDisplay(display, key) {
        var ret = String(calc(key));
        return display + (ret != "" && key == "=" ? "=" : "") + ret;
    }, "");
}

useCalc(calc, "4+3="); // 4+3=7
useCalc(calc, "+9="); // +9=16
useCalc(calc, "*8="); // *5=128
useCalc(calc, "7*2*3="); // 7*2*3=42
useCalc(calc, "1/0="); // 1/0=ERR
useCalc(calc, "+3="); // +3=ERR
useCalc(calc, "51="); // 51
```

使用 `useCalc(..)` 助手的最合理方法是始终将 "=" 作为最后输入的字符。

计算器显示的总计的某些格式需要特殊处理。我提供了这个 `formatTotal(..)` 函数，只要计算器要返回当前计算的总数（在输入 `"="` 后），就应该使用这个函数：

```js
function formatTotal(display) {
    if (Number.isFinite(display)) {
        // 将显示限制为最大11个字符
        let maxDigits = 11;
        // 为 "e+" 符号保留空间?
        if (Math.abs(display) > 99999999999) {
            maxDigits -= 6;
        }
        // 为 "-" 预留位置?
        if (display < 0) {
            maxDigits--;
        }

        // 整数？
```

```
            if (Number.isInteger(display)) {
                display = display.toPrecision(maxDigits).replace(/\.0+$/, "");
            }
            // 小数
            else {
                // 为 "." 预留空间
                maxDigits--;
                // 为前导 "0" 预留空间？
                if (Math.abs(display) >= 0 && Math.abs(display) < 1) {
                    maxDigits--;
                }
                display = display.toPrecision(maxDigits).replace(/0+$/, "");
            }
        } else {
            display = "ERR";
        }
        return display;
    }
```

不要太在意 `formatTotal(..)` 的工作原理。它的大部分逻辑是限制计算器显示最多 11 个字符的一系列处理，即使需要使用负数、重复小数甚至科学记数法 "e+"。

再次强调，不要陷入计算器特定行为的泥潭。把重点放在闭包的*内存*上。

请亲自尝试练习，然后查看本附录末尾的参考答案。

模块

本练习是将闭包（第 3 部分）中的计算器转换成模块。

我们不会为计算器添加任何额外功能，只是更改其接口。我们将不再调用单个函数 `calc(..)`，而是为计算器的每次「按键」调用公共 API 中的特定方法。输出保持不变。

该模块应表示为一个名为 `calculator()` 的传统模块工厂函数，而不是一个单例 IIFE，以便根据需要创建多个计算器。

公共 API 应包括以下方法：

- `number(..)`(输入：「按下」的字符/数字）
- `plus()`
- `minus()`
- `mult()`

- `div()`
- `eq()`

使用方法如下：

```js
var calc = calculator();

calc.number("4"); // 4
calc.plus(); // +
calc.number("7"); // 7
calc.number("3"); // 3
calc.minus(); // -
calc.number("2"); // 2
calc.eq(); // 75
```

`formatTotal(..)` 与之前的练习保持一致。但需要调整 `useCalc(..)` 助手，以便与模块 API 配合使用：

```js
function useCalc(calc, keys) {
    var keyMappings = {
        "+": "plus",
        "-": "minus",
        "*": "mult",
        "/": "div",
        "=": "eq",
    };

    return [...keys].reduce(function showDisplay(display, key) {
        var fn = keyMappings[key] || "number";
        var ret = String(calc[fn](key));
        return display + (ret != "" && key == "=" ? "=" : "") + ret;
    }, "");
}

useCalc(calc, "4+3="); // 4+3=7
useCalc(calc, "+9="); // +9=16
useCalc(calc, "*8="); // *5=128
useCalc(calc, "7*2*3="); // 7*2*3=42
useCalc(calc, "1/0="); // 1/0=ERR
useCalc(calc, "+3="); // +3=ERR
useCalc(calc, "51="); // 51
```

请自己尝试一下，然后查看本附录末尾的参考答案。

在做这个练习时，还需要花一些时间考虑将计算器表示为模块而不是前面练习中的闭包函数方法的优缺点。

附加题：写一些注释解释你的想法。

附加题 #2：尝试将模块转换为其他模块格式，包括：UMD、CommonJS 和 ESM（ES 模块）。

参考答案

希望您在阅读到这里之前，已经尝试过这些练习。不要作弊！

请记住，每个参考答案只是处理问题的多种不同方法之一。它们并不是「正确答案」，但确实说明了处理每个练习的合理方法。

阅读这些参考答案的最大益处在于将它们与您的代码进行比较，并分析我们各自做出相似或不同选择的原因。不要过多地纠缠于「帕金森琐碎定律[^帕金森琐碎定律]」；尽量将注意力集中在主题上，而不是小细节上。

建议：弹珠桶

*弹珠桶练习*可以这样解决[3]：

```js
// 红色(1)
const howMany = 100;

// 埃拉托斯特尼筛法
function findPrimes(howMany) {
    // 蓝色(2)
    var sieve = Array(howMany).fill(true);
    var max = Math.sqrt(howMany);

    for (let i = 2; i < max; i++) {
        // 绿色(3)
        if (sieve[i]) {
            // 橙色(4)
            let j = Math.pow(i, 2);
            for (let k = j; k < howMany; k += i) {
                // 紫色(5)
                sieve[k] = false;
            }
        }
    }
```

```
    return sieve
        .map(function getPrime(flag, prime) {
            // 粉色(6)
            if (flag) return prime;
            return flag;
        })
        .filter(function onlyPrimes(v) {
            // 黄色(7)
            return !!v;
        })
        .slice(1);
}

findPrimes(howMany);
// [
//    2, 3, 5, 7, 11, 13, 17,
//    19, 23, 29, 31, 37, 41,
//    43, 47, 53, 59, 61, 67,
//    71, 73, 79, 83, 89, 97
// ]
```

建议：闭包（第 1 部分）

闭包练习（第 3 部分） 的 `isPrime(..)` 和 `factorize(..)` 可以这样解决：

```js
var isPrime = (function isPrime(v) {
    var primes = {};

    return function isPrime(v) {
        if (v in primes) {
            return primes[v];
        }
        if (v <= 3) {
            return (primes[v] = v > 1);
        }
        if (v % 2 == 0 || v % 3 == 0) {
            return (primes[v] = false);
        }
        let vSqrt = Math.sqrt(v);
        for (let i = 5; i <= vSqrt; i += 6) {
            if (v % i == 0 || v % (i + 2) == 0) {
                return (primes[v] = false);
            }
        }
```

```
        return (primes[v] = true);
    };
})();

var factorize = (function factorize(v) {
    var factors = {};

    return function findFactors(v) {
        if (v in factors) {
            return factors[v];
        }
        if (!isPrime(v)) {
            let i = Math.floor(Math.sqrt(v));
            while (v % i != 0) {
                i--;
            }
            return (factors[v] = [...findFactors(i), ...findFactors(v / i)]);
        }
        return (factors[v] = [v]);
    };
})();
```

我使用每种实用工具的一般步骤：

1. 包裹一个 IIFE，以定义缓存变量所在的作用域。
2. 在底层调用中，首先检查缓存，如果已经知道结果，则返回。
3. 在每一个原本 `return` 的地方，将赋值操作赋给缓存，然后返回赋值操作的结果，这是一个节省空间的技巧，主要是为了在书中简洁明了。

我还将内部函数从 `factorize(..)` 重命名为 `findFactors(..)`。这在技术上并无必要，但它有助于更清楚地说明递归调用调用的是哪个函数。

建议：闭包（第 2 部分）

闭包练习（第 2 部分） 的 `toggle(..)` 可以这样解：

```js
function toggle(...vals) {
    var unset = {};
    var cur = unset;

    return function next() {
        // 将前一个值保存回
        // 列表末尾
        if (cur != unset) {
            vals.push(cur);
```

```
        }
        cur = vals.shift();
        return cur;
    };
}

var hello = toggle("hello");
var onOff = toggle("on", "off");
var speed = toggle("slow", "medium", "fast");

hello(); // "hello"
hello(); // "hello"

onOff(); // "on"
onOff(); // "off"
onOff(); // "on"

speed(); // "slow"
speed(); // "medium"
speed(); // "fast"
speed(); // "slow"
```

建议：闭包（第 3 部分）

闭包练习（第 3 部分） 的 `calculator(..)` 可以这样解：

```js
// 前置代码：
//
// function useCalc(..) { .. }
// function formatTotal(..) { .. }

function calculator() {
    var currentTotal = 0;
    var currentVal = "";
    var currentOper = "=";

    return pressKey;

    // ********************

    function pressKey(key) {
        //  数字键？
        if (/\d/.test(key)) {
            currentVal += key;
            return key;
```

```
        }
        // 操作键？
        else if (/[+*/-]/.test(key)) {
            // 在一个系列中进行多个操作？
            if (currentOper != "=" && currentVal != "") {
                // 隐含 '=' 键
                pressKey("=");
            } else if (currentVal != "") {
                currentTotal = Number(currentVal);
            }
            currentOper = key;
            currentVal = "";
            return key;
        }
        // = 键？
        else if (key == "=" && currentOper != "=") {
            currentTotal = op(currentTotal, currentOper, Number(currentVal));
            currentOper = "=";
            currentVal = "";
            return formatTotal(currentTotal);
        }
        return "";
    }

    function op(val1, oper, val2) {
        var ops = {
            // 注意：书中使用箭头功能
            // 只是为了简洁明了
            "+": (v1, v2) => v1 + v2,
            "-": (v1, v2) => v1 - v2,
            "*": (v1, v2) => v1 * v2,
            "/": (v1, v2) => v1 / v2,
        };
        return ops[oper](val1, val2);
    }
}

var calc = calculator();

useCalc(calc, "4+3="); // 4+3=7
useCalc(calc, "+9="); // +9=16
useCalc(calc, "*8="); // *5=128
useCalc(calc, "7*2*3="); // 7*2*3=42
useCalc(calc, "1/0="); // 1/0=ERR
useCalc(calc, "+3="); // +3=ERR
useCalc(calc, "51="); // 51
```

注意：

记住：本练习是关于闭包的。不要过于关注计算器的实际机制，而要关注在调用函数时是否正确地*记住*了计算器的状态。

建议：模块

*模块练习*中的 `calculator()` 可以这样解决：

```js
// 前置代码：
//
// function useCalc(..) { .. }
// function formatTotal(..) { .. }

function calculator() {
    var currentTotal = 0;
    var currentVal = "";
    var currentOper = "=";

    var publicAPI = {
        number,
        eq,
        plus() {
            return operator("+");
        },
        minus() {
            return operator("-");
        },
        mult() {
            return operator("*");
        },
        div() {
            return operator("/");
        },
    };

    return publicAPI;

    // ********************

    function number(key) {
        // number key?
        if (/\d/.test(key)) {
            currentVal += key;
            return key;
```

```
        }
    }

    function eq() {
        // = 键？
        if (currentOper != "=") {
            currentTotal = op(currentTotal, currentOper, Number(currentVal));
            currentOper = "=";
            currentVal = "";
            return formatTotal(currentTotal);
        }
        return "";
    }

    function operator(key) {
        // 在一个系列中进行多个操作？
        if (currentOper != "=" && currentVal != "") {
            // 隐含 '=' 键
            eq();
        } else if (currentVal != "") {
            currentTotal = Number(currentVal);
        }
        currentOper = key;
        currentVal = "";
        return key;
    }

    function op(val1, oper, val2) {
        var ops = {
            // 注意：书中使用箭头功能
            // 只是为了简洁明了
            "+": (v1, v2) => v1 + v2,
            "-": (v1, v2) => v1 - v2,
            "*": (v1, v2) => v1 * v2,
            "/": (v1, v2) => v1 / v2,
        };
        return ops[oper](val1, val2);
    }
}

var calc = calculator();

useCalc(calc, "4+3="); // 4+3=7
useCalc(calc, "+9="); // +9=16
useCalc(calc, "*8="); // *5=128
useCalc(calc, "7*2*3="); // 7*2*3=42
useCalc(calc, "1/0="); // 1/0=ERR
```

```
useCalc(calc, "+3="); // +3=ERR
useCalc(calc, "51="); // 51
```

本书到此结束，祝贺你取得的成绩！准备好后，请继续阅读第 3 册*对象与类*。

---

1. *Math.js: isPrime(..)*, https://github.com/josdejong/mathjs/blob/develop/src/function/utils/isPrime.js, 2020年3月3日。 ↩
2. Cache replacement policies, https://en.wikipedia.org/wiki/Cache_replacement_policies, 2023年11月2日。 ↩
3. 埃拉托斯特尼筛法，https://zh.wikipedia.org/wiki/，2023年9月27日。 ↩