

05 | for (let x of [1,2,3]) ...: for循环并不比使用函数递归节省开销

2019-11-20 周爱民

JavaScript核心原理解析

[进入课程 >](#)



讲述：周爱民

时长 19:38 大小 17.99M



你好，我是周爱民。欢迎回到我的专栏，我将为你揭示 JavaScript 最为核心的那些实现细节。

语句，是 JavaScript 中组织代码的基础语法组件，包括函数声明等等在内的六种声明，其实都被归为“语句”的范畴。因此，如果将一份 JavaScript 代码中的所有语句抽离掉，那么大概就只会剩下为数不多的、在全局范围内执行的表达式了。

所以，理解“语句”在 JavaScript 中的语义是重中之重。

尽管如此，你实际上要了解的也无非是**顺序**、**分支**、**循环**这三种执行逻辑而已，相比于它们，其它语句在语义上的复杂性通常不值一提。而这三种逻辑中尤其复杂的就是**循环**，今天的这一讲，我就来给你讲讲它。

块

在 ECMAScript 6 之后，JavaScript 实现了**块级作用域**。因此，现在绝大多数语句都基于这一作用域的概念来实现。近乎想当然的，几乎所有开发者都认为一个 JavaScript 语句就有一个自己的块级作用域。这看起来很好理解，因为这样处理是典型的、显而易见的**代码分块的结果**。

然而，事实上正好相反。

真正的状况是，**绝大多数 JavaScript 语句都并没有自己的块级作用域**。从语言设计的原则上来看，越少作用域的执行环境调度效率也就越高，执行时的性能也就越好。

基于这个原则，`switch`语句被设计为有且仅有一个作用域，无论它有多少个 `case` 语句，其实都是运行在一个块级作用域环境中的。例如：

 复制代码

```
1 var x = 100, c = 'a';
2 switch (c) {
3   case 'a':
4     console.log(x); // ReferenceError
5     break;
6   case 'b':
7     let x = 200;
8     break;
9 }
```

在这个例子中，`switch` 语句内是无法访问到外部变量 `x` 的，即便声明变量 `x` 的分支 `case 'b'` 永远都执行不到。这是因为所有分支都处在同一个块级作用域中，所以任意分支的声明都会给该作用域添加这个标识符，从而覆盖了全局的变量 `x`。

一些简单的、显而易见的块级作用域包括：

```
1 // 例 1
2 try {
3   // 作用域 1
4 }
5 catch (e) { // 表达式 e 位于作用域 2
6   // 作用域 2
7 }
8 finally {
9   // 作用域 3
10 }
11
12 // 例 2
13 // (注: 没有使用大括号)
14 with (x) /* 作用域 1 */; // <- 这里存在一个块级作用域
15
16 // 例 3, 块语句
17 {
18   // 作用域 1
19
20 除了这三个语句和“** 一个特例 **”之外, 所有其它的语句都是没有块级作用域的。例如`if`条件语句的
21
22 if (x) {
23   ...
24 }
25
26 // or
27 if (x) {
28   ...
29 }
30 else {
31   ...
32 }
```

这些语法中的“块级作用域”都是一对大括号表示的“块语句”自带的, 与上面的“例 3”是一样的, 而与`if`语句本身无关。

那么, 这所谓的“一个特例”是什么呢? 这个特例, 就是今天这一讲标题中的`for`循环。

循环语句中的块

并不是所有的循环语句都有自己的块级作用域, 例如 `while` 和 `do...while` 语句就没有。而且, 也不是所有 `for` 语句都有块级作用域。在 JavaScript 中, 有且仅有:

```
for (<let/const>...) ...
```

这个语法有自己的块级作用域。当然，这也包括相同设计的`for await`和`for .. of/in`...。例如：

```
for await (<let/const>x of ...) ...  
for (<let/const>x ... in ...) ...  
for (<let/const>x ... of ...) ...
```

等等。你应该已经注意到了，这里并没有按照惯例那样列出“`var`”关键字。关于这一点，后面写到的时候我也会再次提及到。就现在来说，你可能需要关心的是：**为什么这是个特例？**以及，**如果它是拥有自己的块级作用域的特例，那么它有多少个块级作用域呢？**

后面这个问题的答案，是：“说不准”。

看起来，我是被 JavaScript 的古怪设计击败了。我居然给出了这么一个显而易见是在糊弄大众的答案！但是要知道，所谓的“块级作用域”有两种形式，一种是静态的词法作用域，这对于上面的 `for` 语句来说，它们都只有两个块级作用域。但是对于另一种动态的、“块级作用域”的实例来说，这答案就真的是“说不准”了。

不过，先放下这个，我接下来先给你解释一下“**为什么这里需要一个特例**”。

特例

除了语句的关键字和语法结构本身之外，语句中可以包括什么呢？

如果你归纳一下语句中可以包含的全部内容，你应该可以看到一个简单的结果：所有在语句内可以存在的东西只有四种：表达式、其它语句、标识符声明（取决于声明语句或其它的隐式声明的方式），以及一种特殊的语法元素，称为“标签（例如标签化语句，或 `break` 语句指向的目标位置）”。

所谓“块级作用域”，本质上只包括一组标识符。因此，只有当存在潜在标识符冲突的时候，才有必要新添加一个作用域来管理它们。例如函数，由于函数存在“重新进入”的问题，所以它必须有一个作用域来管理“重新进入之前”的那些标识符。这个东西想必你是听说过的，它被称为“**闭包**”。

NOTE: 在语言设计时，有三种需求会促使语句构建自己的作用域，标识符管理只是其中之一。其它两种情况，要么是因为在语法上支持多语句（例如 try...catch...finally 语句），要么是语句所表达的语义要求有一个块，例如“块语句{ }”在语义上就要求它自己是一个块级作用域。

所以**标签**、**表达式**和**其它语句**这三种东西都不需要使用一个“独立作用域”去管理起来。所谓“其它语句”当然存在这种冲突，不过显然这种情况下它们也应该自己管理这个作用域。所以，对于当前语句来说，就只需要考虑剩下的唯一——一种情况，就是在“**语句中包含了标识符声明**”的情况下，需要创建块级作用域来管理这些声明出来的标识符。

在所有六种声明语句之外，只剩下for (<let/const>...)...这一个语句能在它的语法中去做这样的标识符声明。所以，它就成了块级作用域的这个唯一特例。

那么这个语法中为什么单单没有了“var 声明”呢？

特例中的特例

“var 声明”是特例中的特例。

这一特性来自于 **JavaScript 远古时代的作用域设计**。在早期的 JavaScript 中，并没有所谓的块级作用域，那个时候的作用域设计只有“函数内”和“函数外”两种，如果一个标识符不在任何（可以多层嵌套的）函数内的话，那么它就一定是在“全局作用域”里。

“函数内→全局”之间的作用域，就只有概念上无限层级的“函数内”。

而在这个时代，变量也就只有“var 声明”的变量。由于作用域只有上面两个，所以任何一个“var 声明”的标识符，要么是在函数内的，要么就是在全局的，没有例外。按照这个早期设计，如下语句中的变量x：

```
1 for (var x = ...)
2     ...
```

 复制代码

是不应该出现在“**for 语句所在的**”块级作用域中的。它应该出现其外层的某个函数作用域，或者全局作用域中。这种越过当前语法范围，而在更外围的作用域中登记名字行为就称为“**提升 (Hoisting/Hoistable)**”。

ECMAScript 6 开始的 JavaScript 在添加块级作用域特性时充分考虑了对旧有语法的兼容，因此当上述语法中出现“var 声明”时，它所声明的标识符是与该语句的块级作用域无关的。在 ECMAScript 中，这是两套标识符体系，也是使用两套作用域来管理的。确切地说：

所有“var 声明”和函数声明的标识符都登记为 varNames，使用“**变量作用域**”管理；

其它情况下的标识符 / 变量声明，都作为 lexicaNames 登记，使用“**词法作用域**”管理。

NOTE: 考虑到对传统 JavaScript 的兼容，函数内部的顶层函数名是提升到变量作用域中来管理的。 > > NOTE: 我通常会将“在变量声明语句前使用该变量”也称为一种提升效果 (Hoisting effect)，但这种说法不见于 ECMAScript 规范。ES 规范将这种“提前使用”称为“访问一个未初始化的绑定 (uninitialized mutable/immutable binding)” 。而所谓“var 声明能被提前使用”的效果，事实上是“var 变量总是被引擎预先初始化为 undefined”的一种后果。

所以，语句 `for (<const/let> x ...) ...` 语法中的标识符 `x` 是一个**词法名字**，应该由 `for` 语句为它创建一个（块级的）词法作用域来管理之。

然而进一步问题是：一个词法作用域是足够的吗？

第二个作用域

首先，必须要拥有至少一个块级作用域。如之前讲到的，这是出于管理标识符的必要性。下面的示例简单说明这个块级作用域的影响：

 复制代码

```
1 var x = 100;
2 for (let x = 102; x < 105; x++)
3   console.log('value:', x); // 显示“value: 102~104”
4 console.log('outer:', x); // 显示“outer: 100”
```


因为for语句的这个块级作用域的存在，导致循环体内访问了一个局部的x值（循环变量），而外部的（outer）变量x是不受影响的。

那么在循环体内是否需要一个新的块级作用域呢？这取决于在语言设计上是否支持如下代码：

```
1 for (let x = 102; x < 105; x++)  
2   let x = 200;
```

 复制代码

也就是说，如果循环体（单个语句）允许支持新的变量声明，那么为了避免它影响到循环变量，就必须为它再提供另一个块级作用域。很有趣的是，**在这里，JavaScript 是不允许声明新的变量的**。上述的示例会抛出一个异常，提示“单语句不支持词法声明”：

SyntaxError: Lexical declaration cannot appear in a single-statement context


这个语法错误并不常见，因为很少有人会尝试构建这样的特殊代码。然而事实上，它是一个普遍存在的语法禁例，例如以下语句语法：

```
1 // if 语句中的禁例  
2 if (false) let x = 100;  
3  
4 // while 语句中的禁例  
5 while (false) let x = 200;  
6  
7 // with 语句中的禁例  
8 with (0) let x = 300
```

 复制代码

所以，现在可以确定：循环语句（对于支持“let/const”的for语句来说）“通常情况下”只支持一个块级作用域。更进一步地说，在上面的代码中，我们并没有机会覆盖for语句中的“let/const”声明。

但是如果在 for 语句支持了 let/const 的情况下，仅仅只有一个块级作用域是不方便的。例如：

 复制代码

```
1 for (let i=0; i<2; i++) /* 用户代码 */;
```

在这个例子中，“只有一个块级作用域”的设计，将会导致“用户代码”直接运行在与“let 声明”相同的词法作用域中。对于这个例子来说，这一切还好，因为“let i = 0”这个代码只执行了一次，因为它是 for 语句的初始化表达式。

但是对于下面这个例子来说，“只有一个块级作用域”就不够了：

 复制代码

```
1 for (let i in x) ...;
```

在这个例子中，“let i ...”在语义上就需要被执行多次——因为在静态结构中它的多次迭代都作用于同一个语法元素。而你是知道的，let 语句的变量不能重复声明的。所以，这里就存在了一个冲突：“let/const”语句的单次声明（不可覆盖）的设计，与迭代多次执行的现实逻辑矛盾了。

这个矛盾的起点，就是“只有一个块级作用域”。所以，在 JavaScript 引擎实现“支持 let/const 的 for 语句”时，就在这个地方做了特殊处理：为循环体增加一个作用域。

这样一来，“let i”就可以只执行一次，然后将“i in x”放在每个迭代中来执行，这样避免了与“let/const”的设计冲突。

上面讲的，其实是 JavaScript 在语法设计上的处理，也就是在语法设计上，需要为使用 let/const 声明循环变量的 for 语句多添加一个作用域。然而，这个问题到了具体的运行环境中，变量又有些不同了。


for 循环的代价

在 JavaScript 的具体执行过程中，作用域是被作为环境的上下文来创建的。如果将 for 语句的块级作用域称为 **forEnv**，并将上述为循环体增加的作用域称为 **loopEnv**，那么

loopEnv 它的外部环境就指向 **forEnv**。

于是在 **loopEnv** 看来，变量 **i** 其实是登记在父级作用域 **forEnv** 中，并且 **loopEnv** 只能使用它作为名字 “**i**” 的一个引用。更准确地说，在 **loopEnv** 中访问变量 **i**，在本质上就是通过环境链回溯来查找标识符（Resolve identifier, or Get Identifier Reference）。

上面的矛盾“貌似”被解决了，但是想想程序员可以在每次迭代中做的事情，这个解决方案的结果就显得并不那么乐观了。例如：

 复制代码

```
1 for (let i in x)
2   setTimeout(()=>console.log(i), 1000);
```

这个例子创建了一些定时器。当定时器被触发时，函数会通过它的闭包（这些闭包处于 **loopEnv** 的子级环境中）来回溯，并试图再次找到那个标识符 **i**。然而，当定时器触发时，整个 **for** 迭代有可能都已经结束了。这种情况下，要么上面的 **forEnv** 已经没有了、被销毁了，要么它即使存在，那个 **i** 的值也已经变成了最后一次迭代的终值。

所以，要想使上面的代码符合预期，这个 **loopEnv** 就必须是“随每次迭代变化的”。也就是说，需要为每次迭代都创建一个新的作用域副本，这称为**迭代环境**（**iterationEnv**）。因此，每次迭代在实际上都并不是运行在 **loopEnv** 中，而是运行在该次迭代自有的 **iterationEnv** 中。

也就是说，在语法上这里只需要两个“块级作用域”，而实际运行时却需要为其中的第二个块级作用域创建无数个副本。

这就是 **for** 语句中使用 “**let/const**” 这种块级作用域声明所需要付出的代价。

知识回顾

今天我讲述了 **for** 循环为了支持局部的标识符声明而付出的代价。

在传统的 JavaScript 中是不存在这个问题的，因为 “**var** 声明” 是直接提升到函数的作用域中登记的，不存在上面的矛盾。这里讲的 **for** 语句的特例，是在 ECMAScript 6 支持了

块级作用域之后，才出现的特殊语法现象。当然，它也带来了便利，也就是可以在每个 for 迭代中使用独立的循环变量了。

当在这样的 for 循环中添加块语句时（这是很常见的），块语句是作为 iterationEnv 的子级作用域的，因此块语句在每个迭代中都会都会创建一次它自己的块级作用域副本。这个循环体越大，支持的层次越多，那么这个环境的创建也就越频繁，代价越高昂。再加上可以使用函数闭包将环境传递出去，或交给别的上下文引用，这里的负担就更是雪上加霜了。

注意，无论用户代码是否直接引用 loopEnv 中的循环变量，这个过程都是会发生的。这是因为 JavaScript 允许动态的 eval()，所以引擎并不能依据代码文本静态地分析出循环体 (ForBody) 中是否引用哪些循环变量。

你应该知道一种理论上的观点，也就是所谓“**循环与函数递归在语义上等价**”。所以在事实上，上述这种 **for 循环并不比使用函数递归节省开销**。在函数调用中，这里的循环变量通常都是通过函数参数传递来处理的。因而，那些支持“let/const”的 for 语句，本质上也就与“在函数参数界面中传递循环控制变量的递归过程”完全等价，并且在开销上也是完全一样的。

因为每一次函数调用其实都会创建一个**新的闭包**——也就是函数的作用域的一个副本。

思考题

为什么单语句（single-statement）中不能出现词法声明（lexical declaration）？

如果你喜欢我的分享，也欢迎你把文章分享给你的朋友。

小程序学习打卡邀请

打卡 46 天，彻底搞定 JavaScript



扫一扫参与小程序打卡



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 04 | `export default function() {}`：你无法导出一个匿名函数表达式

下一篇 06 | `x: break x`；搞懂如何在循环外使用`break`，方知语句执行真解

精选留言 (12)

写留言



Y

2019-11-20

老师，在es6中，其实for只要写大括号就代表着块级作用域。所以只要写大括号，不管用`let`还是`var`，一定是会创建相应循环数量的块级作用域的。

如果不用大括号，在for中使用了`let`，也会创建相应循环数量的块级作用域。

也就是说，可以提高性能的唯一情况只有（符合业务逻辑的情况下），循环体是单行语句就不使用大括号且for中使用`var`。

展开 ▾

作者回复：是的。

赞，好几个赞。^^.



9



Wiggle Wiggle

2019-11-22

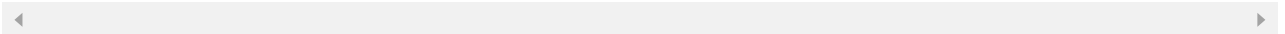
词法、词法作用域、语法元素.....等等，这些概念特别模糊，老师有什么推荐的书吗？

作者回复: 《JavaScript语言精髓与编程实践》第三版。^^.

已经交稿，大概快要出了。

如果急用，可以看ECMAScript~ 别的书很少用语言层面来讲的。不过，另外，你可以看《程序原本》，对很多概念都是讲到的。在这里可以直接下载：

<https://github.com/aimingoo/my-ebooks>



1



Fans

2019-11-21

看到标题感觉的终于有一个可能会看的懂了

展开 ∨



1



westfall

2019-11-20

因为单语句没有块级作用域，而词法声明是不可覆盖的，单语句后面的词法声明会存在潜在的冲突。

作者回复: :)

+1



1



Y

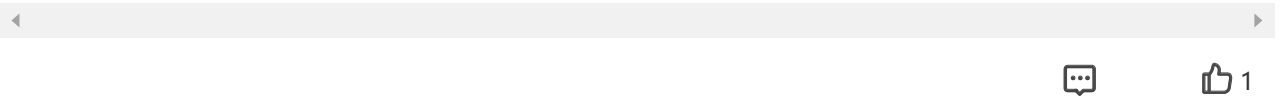
2019-11-20

既然是单语句就说明只有一句话，如果就一句话，还是词法声明，那就会创建一个块级作用域，但是因为是单语句，那一定就没有地方会用到这个声明了。那这个声明就是没有意义的。所以js为了避免这种没有意义的声明，就会直接报错。是这样嘛

展开 ∨

作者回复: 不是，单语句也可以实现很复杂的逻辑的。如果单语句使用let/const声明，也一样可以包括逻辑。例如（这个当然不能执行）：

if (false) let x = 100, y = x++; // < 这里的x就被使用了

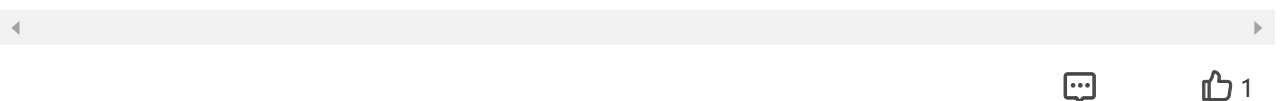


许童童

2019-11-20

为什么单语句（single-statement）中不能出现词法声明（lexical declaration）？
我觉得应该是语法规定 单语句后面需要一个表达式，而一个声明语句是不行的。

作者回复: 你可以在后面用var声明试试😊



qqq

2019-11-20

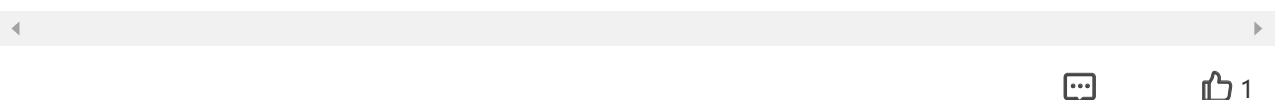
单语句对应的是变量作用域，不能出现词法声明吗

展开 ∨

作者回复: 不是。

而是这种情况下并没有所谓的“块级作用域（变量作用域）”。

这就是需要仔细地“数”清楚一个语法有几个作用域的原因。

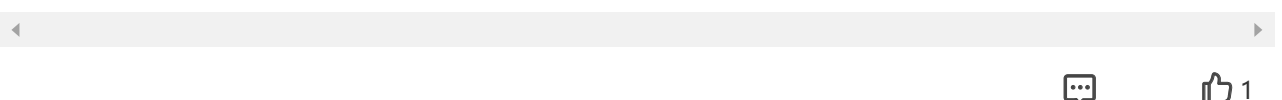


Chao

2019-11-20

临时死区使得 通过let / const 定义的变量。 在定义之前调用报错。

作者回复: 临时死区 = ？



zcdll

2019-11-20

看不懂。。。第一个 switch 那个例子都看不懂。。

展开 ▾

作者回复: case 'b' 永远执行不到, 但它里面的x却已经声明了, 并且导致case 'a'中的代码无法访问到外部的`x = 100`。

这说明case 'a'和case 'b'使用了同一个闭包。

◀ ▶

💬 2

👍 1



海绵薇薇

2019-11-23

hello, 老师好, 一如既往有许多问题等待解答:)

for(let/const ...) ... 这个语句有两个词法作用域, 分别是 forEnv 和 loopEnv。还有一个概念是iterationEnv, 这个是迭代时生成的loopEnv的副本。

...

展开 ▾

作者回复: 1. 这个问题出在我对 “for(let/const...)” 这个语法没有展开讲, 它跟 “for(var...)”, 以及后面的 “for(let/const ... in/of)” 其实都有区别。所以你套用它们的处理方法, 结果都有点差异, 对你结论会带来干扰。

你读一下ECMA这个部分:

<https://tc39.es/ecma262/#sec-for-statement-runtime-semantics-labelledevaluation>

注意其中的第三节的具体说明:

> IterationStatement:

for(LexicalDeclarationExpression;Expression)Statement

在后续调用中, 简单地说, 就是这种情况下for语句会为每次循环创建 CreatePerIterationEnvironment()来产生一个新的IterationEnv。并且thisIterationEnv 与lastIterationEnv 之间会有关联。

2. with({}) let b = 1 这个语法报错, 不是因为with()没有作用域, 而是它的作用域称为 “对象作用域”, 而不是 “词法作用域”。对象作用域只有在用作global的时候可以接受var和泄露的变量声明, 其它情况下, 它不能接受 “向作用域添加名字” 这样的行为——它的名字列表来自于属性名, 例如obj.x对吧。

3. eval有一个自己的作用域。

◀ ▶

💬

👍



晓小东

2019-11-21

老师您看下这段代码，我在Chrome 打印有点不符合直觉，Second 最终打印的应该是2，为什么还是1，2，3；

```
for (let i = 0; i < 3; i ++, setTimeout(() => console.log("Second" + i), 20))...
```

展开 ∨

作者回复: 在node里很合理呀。

在node里的second值是：Second1，Second2，Second3

如果你把setTimeout()超时值都改成0，就看得计算过程了。

0

1

2

Last:0

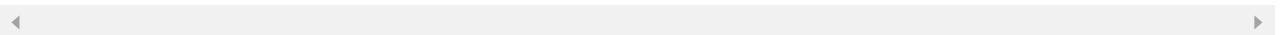
Second1

Last:1

Second2

Last:2

Second3



Summer

2019-11-21

假设允许的话，没有块语句创建的iterationEnv的子作用域，let声明就直接在iterationEnv作用域中，会每次循环重复声明。

展开 ∨

作者回复: 是的。^^.

