

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》

课程介绍 >



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：

 复制代码

```
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

```

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条件语句的几种常见书写形式：

 复制代码

```

1  if (x) {
2      ...
3  }
4
5  // or
6  if (x) {
7      ...
8  }
9  else {
10     ...
11 }
12 }

```

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 lexicalNames 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。

然而进一步后，新的问题产生了：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：

```
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”
```

 复制代码

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，提示你“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：

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

复制代码

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：

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

 复制代码



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。

分享给需要的人，Ta购买本课程，你将得 20 元

 生成海报并分享

 赞 6  提建议

© 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](#) 加餐 | 捡豆吃豆的学问（上）：这门课讲的是什么？



JVM + NIO + Spring

各大厂面试题及知识点详解

限时免费



精选留言 (54)

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。

作者回复: 是的。

赞，好几个赞。^^.

共 13 条评论 >

34



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





小毛

2020-03-05

老师，最后的思考题感觉有点懵，按你文章里说的，`for(let/const...)`这中语法，不管怎样在执行阶段，都会为每次循环创建一个`iterationEnv`块级作用域，那又为什么在单语句语法中不能有`let`词法声明呢，像`if`不能有是可以理解的，但是对于`for(let/const...)`就不能理解了。另外如果要是提高`for`的性能，是不是不`for(let/const...)`这样写，把`let x`放在`for`语句体外，在其之前声明，是不是就可以在运行阶段只有一个`loopEnv`，而不创建`iterationEnv`，从而提高性能。

作者回复: 这个问题点问得非常细，解释起来也不容易。

> 1. `for (let/const x...)`

在这个结构中，`for`语句总是会生成一个块级作用域，用来放`x`等等变量。这一点是没有疑问的。并且，由于`var`声明，或者没有任何声明的`for`语句在这里不需要放变量名，所以在那些语法格式，也就不产生块级作用域。这个理由和逻辑也很清晰。这个地方创建的作用域（环境）称为`forEnv`。

基于此，我们继续讨论。

> 2. `for (let/const x...) ...`

在第二个`...`位置，亦即是`forBody`的位置如果没有块语句，那么这里就会被识别为“单语句上下文（single-statement context）”，也就是说这种情况下`for`被理解为单语句。`if`语句在这里的情况也一样，也是没有块语句，就理解为单语句。

对于`forBody`来说，它每一次循环都需要创建一个`iterationEnv`，这个`iterationEnv`抄写自`loopEnv`。——注意这里是抄写，而不是简单地将`parent`指向`loopEnv`，所以它确实比较消耗资源。（再次说明，`loopEnv`的`parent`指向`forEnv`，但`iterationEnv`是抄写`loopEnv`而不是指向它）。

但为什么要“抄写”呢？这个部分在正文里面有仔细讲，使用了一个基于`setTimeout()`的例子，请再回顾一下。

但是上面（在这个评论的）第1部分中说到的单语句的部分为什么仍然要`iterationEnv`呢？——这个才是你的问题本身不是？

其实这就与单语句或块语句无关了。`for`语句是不包括后面的大括号的。它的语法就是``for (...) ...``，后面是大括号还是单语句上下文，无关。所以``for (let/const ...) ...``语句就约定了每次循环都创建`iterationEnv`并抄写自`loopEnv`，以确保在`forBody`部分可以创建新的作用域，而至于在`forBody`中是`setTimeout`打开中的函数闭包，还是一个块语句，它们的处理逻辑（以及对块级作用域的需求）其实都是一样的。



最后，你的问题提到是不是可以将let x放for语句外。是的，这会提高效率，并且也不需要创建loopEnv和iterationEnv。你也可以考虑用var，以及用一个函数来包起来，避免变量泄露到全局。简单地说，使用函数内套一个for循环，并在函数内管理变量名，比将这些变量名放到for (let/const ...)循环语句中，要效率高一些。

如上。



👍 12



wDaLian

2020-01-12

```
const array = Array.from({length:100000},{item,i}=>i)
```

```
// 案例一
```

```
console.time('a')
const cc = []
for(let i in array){
  cc.push(i)
}
console.log(cc)
console.timeEnd('a')
```

```
// 案例二
```

```
console.time('b')
const ccc = []
for(var i in array){
  ccc.push(i)
}
console.log(ccc)
console.timeEnd('b')
```

```
// 案例三
```

```
console.time('c')
const cccv = []
for(let i in array)
  cccv.push(i);
console.log(cccv)
console.timeEnd('c')
```



1.老师你上次的评论我没看懂，第一我案例一和案例三是为了做区分所以案例一有大括号的

- 2.编译引擎的debug版本然后track内核，或者你可以尝试一个prepack-core这个项目，这两个东西是啥 我百度也没查到
- 3.老师你讲的都是概念的，我就想看到一个肉眼的案例然后根据概念消化，要不现在根本就是
这个for循环到底应该咋写我都懵了

作者回复: 很晚才回复你的这个问题，原因是确实不好回复，不知道哪种方法才能有效地解决你的疑惑。

首先，不要相信你写的代码，它并不是最终执行的，引擎会做一些优化，这些优化不是语言本身的，所以也不适用于我们在这个课程中所讨论的。

其次，如果你需要用你所列举的类似代码来（粗略地）检查性能，那么建议把数量提高100~1000倍以上，我运行了你的代码，单个测试case大概才20ms，这种情况下，随便的一个后台进程的波动就影响了结果，有效性成问题。再一次强调，不要用这种方法来检测性能，不要相信你的代码在“字面上的表现出来的”效率。

第三，关于debug版本并track内核，我建议你参考一下下面这两篇，一篇是讲编译的，一篇是讲优化的：

...

<https://zhuanlan.zhihu.com/p/25120909>

<https://segmentfault.com/a/1190000008285728>

...

我原来的意思是说，你可能会在原生语言（例如C）这个层面调试和分析内核有困难，所以就向你推荐了一下prepack-core。这个也是一个js引擎，但是是用javascript写的，你分析起来会好一些。——但坦率地说，也并不容易，这个项目还是很难的。在这里：

...

<https://github.com/aimingoo/prepack-core>

...

第四，我认为我还是应该给你一个简单的分析路径，来解释你的问题。从你的代码来看，你只是想尝试for let/var两种语法到底性能上有什么样的差异。我的建议是这样：

...

```
var array = Array.from({length:10},(item,i)=>i);
```

```
// 例1
```

```
var a = [], checker1 = ()=>console.log(a[1] == a[5]); // anything
```

```
for (var i in array) setTimeout(()=>a.push(i), 0);
```

```
setTimeout(checker1, 0); // true
```

```
// 例2
```

```
var b = [], checker2 = ()=>console.log(b[1] == b[5]); // anything
```



```
for (let i in array) setTimeout(()=>b.push(i), 0);  
setTimeout(checker2, 0); // false  
...
```

进一步测试如下：

```
...  
  
> a  
[ '9', '9', '9', '9', '9', '9', '9', '9', '9', '9' ]  
  
> b  
[ '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' ]  
...
```

我们分析一个问题：

* 一、在checker1()中，a[]的元素保存了相同的i值，是不是意味着所有的setTimeout()中a.push()操作其实是工作在一个环境中的？而相对的，由于在checker2()中，b[]保存了不同的i值，那么b.push()就得工作在不同的环境中（从而才能访问不同的i值）。是不是？

* 二、所以，如果在checker2()中每一次迭代都在不同的环境中，是不是说每一次迭代都要消耗一个“创建一个环境”所需要的时间和空间？如果是这样，是不是就说明了`let i`其实效率远低于`var i`？

OK. 最后说明一下，百度查不到东西是正常的，查到才不正常。^^.



10



Elmer

2019-12-08

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。

单语句如果支持变量声明，相当于需要支持为iteration env新增子作用域，降低了效率？

如果需要完全可以自己写{} 来强制生成一个子作用域

不知道这样说对不对

作者回复: 正是如此👍



10



Geek_8d73e3

2020-06-08

老师我发现运行以下代码会报错


```
for(let x = 0;x<1;x++){  
    var x = 100;  
}
```

//Uncaught SyntaxError: Identifier 'x' has already been declared

在我理解中，let声明的x是在forEnv中，而我使用var声明的x因为JavaScript早期设计，会在全局中声明一个x。这两个作用域是不会冲突的呀，为什么报错了？

作者回复: 我个问题真的把我考到了，很花了一些时间来分析它。

首先，简单地说，这个问题可以视为对如下两个语法的比较：

...

// 例1：如下成立

```
var x = 100;  
for (let x = 0; x < 1; x++)  
    console.log(x); // 0
```

// 例2：如下不成立

```
for (let x = 0; x<1; x++) {  
    var x = 100;  
}
```

...

就是说，为什么上述“例2”是不成立的呢？从我们之前的分析来说，`var x`声明的变量`x`是位于外部（例如全局）的，因此与当前块中的`let x`应该是没有关系的。——这类似于“例1”。

先说答案：这是语法限制。

下面.....重点来了：JavaScript在语法上不允许在同一个块中出现“声明与词法名字相同的标识符”。

也就是说，在语法上：

...

// 例3：你既不能写

```
let x = 100;  
const x = 100;
```

// 例4：（所以，）也不能写

```
let x = 100;  
var x = 100;
```

...

只要是在同一个词法作用域中，与`let/const`相同的“标识符声明”就是不被许可的。

再次强调：这是语法声明上的限制，而与执行过程是无关的。



不过在ECMAScript的规范上，对这一点也是语焉不详的。——唯一与此相关的，就是在一个语法块中，会将所有的let/const名字登记在BoundNames表中，以完成“例3”所示的名字重复检查。这在如下章节：

> <https://tc39.es/ecma262/#sec-let-and-const-declarations-static-semantics-early-errors>

但这个位置的语法查错（至少在ECMAScript中看起来）是与`var x`声明无关的。并且事实上在ECMAScript规范中，并没有对`var`语句定义任何的语法错误抛出。

然而“（在同一个块中）不能重复声明”的语法限制是真实存在的。这项限制存在于两个地方。

首先，语法parser引擎自己会处理这个重复检测（尽管ECMAScript没有定义）。parser过程会维护当前块的词法上下文，并且拒绝在forBody和forHead中出现这种重复声明。而且有趣的是，这个检测过程对于let/const，以及var来说是不同的。——具体来说，let/const是只检测当前词法作用域，而var是检测词法作用域栈（scopeStack, scope chains）。关于这一点的实现，可以在这里看到：

> <https://github.com/babel/babel/blob/master/packages/babel-parser/src/util/scope.js#L95>

所以这是一个parser在语法解析中表现出来的结果。但带来了一个更有趣的示例：

...

// 示例5，不成立

```
for (let x in {}) { var x = 100 }
```

// 示例6，成立

```
for (let x in {}) { let x = 100 }
```

...

接下来，我们需要置疑：使用eval()来执行的话，会不会产生一个“提升到外部（例如global）的变量”呢？

答案是：也不会。而这也是唯一——处在ECMAScript中对这种现象做了解释的地方，原文是：

> A direct eval will not hoist var declaration over a like-named lexical declaration.

也就是“直接的eval()是不能将对变量提升到同名的词法声明之外的”。也就是，如下代码会导致一个执行期的错误：

...

// 示例7，不成立

```
for (let x in {}) { eval('var x = 100') }
```

...



而这一段的说明是被ECMAScript写进规范，并在执行期而`eval()`来处理的。参见这里：

> <https://tc39.es/ecma262/#sec-evaldeclarationinstantiation>

共 4 条评论 >

👍 8



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>

共 2 条评论 >

👍 7



zcdll

2019-11-20

看不懂。。。第一个 switch 那个例子都看不懂。。

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。

共 5 条评论 >

👍 6



Geek_8d73e3

2020-05-26

老师，我发现，我运行这段代码的时候，并没有报错。

```
for(let i = 0;i<10;i++){
```

```
    let i = 1000;
```

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

```
}
```



作者回复: 这是因为

```
> `for (let i = 0...)`
```

和

```
> `{ let i = 1000; ...`
```

是在两个作用域里面。前者是forEnv，后者是bodyEnv。所以不冲突，不会算作重复声明。



6



Marvin

2019-11-26

如果使用let /const 声明for循环语句，会迭代创建作用域副本。那么不是和文中的：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这句话相矛盾么？

作者回复: for (let/const ...) “通常情况下”只支持一个块级作用域。

for (let/const ... in/of ...) 会迭代创建作用域副本。

有一眯眯细微的不同哦。 ^^.

共 3 条评论 >



5



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

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
```



6



🐱🐱🐱🐱🐱...

2020-05-20

看了4遍 终于看懂了



4



海绵薇薇

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。

对于forEnv和loopEnv的范围我不是很清楚, 请老师指点。

```
for(let i = 0; i < 10; i++)
```

```
  setTimeout(() => console.log(i))
```

1 如上代码, let i 声明的 i 在forEnv还是在loopEnv / iterationEnv里?

1.1 如果在loopEnv / iterationEnv里那么forEnv看起来就没啥用了

1.2 如果在forEnv (文章中说let只会执行一次, 并且forEnv是loopEnv的上级), 那么按理说console.log打印出来的都是11 (参考于: 晓小东)

2 关于单语 let a = 1 报错问题

2.1 如果是单语句中词法声明被重复有问题, 那么with({}) let b = 1 这个报错就解释不通了。上面是说with有自己的块作用域, 这个词法声明是在自己块语句中做的, 并不会和别人冲突

2.2 同样的情况存在于for(let a...) ... 中, for也有自己的作用域, 并且每次循环都会生成新的副本, 也不应该存在重复问题

3 关于上面提到的eval

```
eval('let a = 1'); console.log(a) // 报错
```

eval是不是自己也有一个作用域?

期待:)

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。



4



桔子

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。

作者回复: 是的。^^.

共 2 条评论 >



4



G

2020-10-03

老师您好，关于如何学习这门课，可否请您指点一下。

第一部分的内容我其实已经来回读了很多遍了，我接触js差不多一年，里面很多内容对我来说比较难懂。在读到后面内容的时候，我常常需要再翻回前面这些内容重新读，因为有一些前面章节我所没有理解的地方，在后续章节会讲到，这时候重新读我就会有新的收获。而且我在重新读的过程中，发现文章中的每一句话都是很有用的，少看一句话可能就会让我造成理解上的错误，每次重新读都有新收获这件事，让我开心又让我焦虑，因为这代表我并没有完全读懂任何一章。在继续学习第三部分的时候，我又发现我开始很难读懂文章，此时我不知道我应该先整体读完第三部分然后再回过头来重新读几遍，还是把每一篇文章尽量弄清楚，我目前采取的是后面这种方式，我已经学了差不多20天这个课程，但是目前也没有真正走出第二部分的内容，这个会不会是我这种学习方式不对。

希望周老师指点一下。

作者回复: 可以先“观其大略，不求甚解”，一遍两遍之后，再“务求精细，绝无遗漏”。读书也好，学习也好，不同的东西要用不同的方法来应对，这门课是适合反复研究的。



在这门课的结束语中说过：即便是同一个石狮子，在不同的层次看到的，仍然是不同的东西、不同的答案、不同的理解。所以，不要纠结于你之前的所得“是否错了”，你可能只是高度提高了，理解有了不同而已。——如果你能否定之前的所见，是提高；如果你有能力质疑它，也是提高；如果你能肯定它，还是提高。

不进则退，无论否定、质疑、肯定，皆是进步，但都不是终点。是谓学习。



👍 3



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))  
  console.log(i), setTimeout(() => console.log('Last:' + i), 30);
```

0, 1, 2

Second: 0, 1, 2

Last: 0, 1, 2

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



👍 4



二二

2020-10-12

你好老师，按照文章的解释，因为for循环中let会导致块级作用域，开销会变大，此处的开销可以粗略理解成时间。

```
var a = new Array(10000).fill(0)
console.time('var')
for(var i=0, len=a.length; i<len; i++){
}
console.timeEnd('var')
console.time('let')
for(let i=0, len=a.length; i<len; i++){
}
console.timeEnd('let')
```

在chrome devtool执行的结果，var会比let要慢许多，请问中间还发生了什么，导致var会比let慢呢？

作者回复: 这个是特难解释的，因为devtool测的也不见得是js引擎的结果，而且chrome自身也还对v8引擎有优化，不见得是ecmascript语言规范所表现出来的样子。

只以你的例子来说，由于var其实声明是在for语句之外一层的变量作用域（这里正好是全局作用域）中的，所以在for语言中访问var变量其实是要经过多一次的查找的。而let块是在for语句里，尽管每次都创建一个新的（从上一次复制而来），但是它们之间不需要嵌套，所以访问层次总是1。因此，大抵来说，是用空间（更多的作用域环境）换了时间（更少的访问层次）。

对于js引擎（包括jit优化引擎）来说，let以及它所对应的词法作用域是易于优化处理的，而var则很难，因为var中的名字是可增删的，因此不能缓存也不能做层次的消减。



👍 2



青山入我怀

2020-08-22

老师，请问既然forEnv是loopEnv的上级，而iterationEnv又是loopEnv的副本，那么按道理在iterationEnv中对i的改动，在查找i时不都是会通过环境链回溯，找到forEnv这个运用域下的i吗，那么闭包现象发生时，找到的i应该是同一个i啊，感觉增加了副本无法避免这个问题啊？

作者回复: 你忘了通常来说的代码，类似于`for (let i =0; i<x; i++) ...`，会在下一次迭代之前先`i++`一次？如果是`let i in obj`运算，那么也会发生一次将属性名提取到i的操作~

所以i是副本，但“通常”会被立即重写。当然，如果代码中没有`i++`这样类似重写的操作，那么这个复制副本的行为就浪费了。



👍 2



Geek_8d73e3

2020-06-08

所以，语句for (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。

老师, 对于这句话, 如果我运行以下代码

```
var x = 1;
```

```
let y = 2;
```

那么JavaScript也会创建两个作用域? 一个变量作用域管理x, 一个词法作用域管理y?

那么如果全局中已经存在了变量作用域和词法作用域

为什么for(let i =0....)中, 这个i不在刚才的词法作用域中声明, 而要重新再讲一个词法作用域?

作者回复: 一般的“块级作用域”所在的环境中并没有“变量作用域”。所以块级作用域只能放let/const。

同时具有“变量作用域”和“词法作用域”的只有函数、全局和ES6之后的模块。所以, 当在一个一般的“块级作用域”中声明了var的时候, 它就必须被“提升”到上面三种环境中去存放。

所以下面的代码中:

```
...  
for (...) {  
  var x = 1;  
  let y = 2;  
  ...  
}
```

在这其中`y`是放在for的forBody块中的, 而`x`是放在这个语句所在的、更外层的“变量作用域”中的。

这个变量作用域并不是为上面两行代码“专门创建”的。但是for(let i =0....)中的“词法作用域”是为for语句专门创建的, 因此它跟更外层的(例如全局)并不是同一个。



👍 2



从未止步

2020-03-21

老师, 因为在循环后边的单语句中如果出现了词法声明, 但是这时候其实单语句并没有块级作用域, 需要重复声明创建作用域副本, 来支持这个语句的执行, 所以JavaScript限制了这种情况的发生, 可以这样理解嘛? 谢谢老师~

作者回复: 作用域的价值在于“容纳变量声明”, 而代价是“每增加一级作用域, 都会导致效率变差”。

而一般的for语句, 以及`for (var ...`其实不需要在该语句的位置上声明变量, 因此没有必要付出上述的高昂的代价。但是语法`for (let/const ...`就需要一个作用域来放变量声明, 并“隔开forBody区的变量声明”, 所以就加入了这个作用域。



单语句不支持块级作用域，本质上就是尽量提高效率。——即使因此带来一些语法限制，也是在所不惜的。



2

