

20 | (0, eval)("x = 100")：一行让严格模式形同虚设的破坏性设计（上）

2019-12-30 周爱民

《JavaScript核心原理解析》

[课程介绍 >](#)



讲述：周爱民

时长 22:38 大小 18.15M



你好，我是周爱民。

今天我们讨论动态执行。与最初的预告不同，我在这一讲里把原来的第 20 讲合并掉了，变成了 20~21 的两讲合并，但也分成了上、下两节。所以，其实只是课程的标题少了一个，内容却没有变。

动态执行是 JavaScript 最早实现的特性之一，`eval()` 这个函数是从 JavaScript 1.0 就开始内置了的。并且，最早的 `setTimeout()` 和 `setInterval()` 也内置了动态执行的特性：它们的第 1 个参数只允许传入一个字符串，这个字符串将作为代码体动态地定时执行。



NOTE：`setTimeout/setInterval` 执行字符串的特性如今仍然保留在大多数浏览器环境中，例如 Safari 或 Mozilla，但这在 Node.js/Chrome 环境中并不被允许。需要留意的是，`setTimeout/setInterval` 并不是 ECMAScript 规范的一部分。

关于这一点并不难理解，因为 JavaScript 本来就是脚本语言，它最早也是被作为脚本语言设计出来的。因此，把“装载脚本 + 执行”这样的核心过程，通过一个函数暴露出来成为基础特性既是举手之劳，也是必然之举。

然而，这个特性从最开始就过度灵活，以至于后来许多新特性在设计中颇为掣肘，所以在 ECMAScript 5 的严格模式出现之后，它的特性受到了很多的限制。

接下来，我将帮助你揭开重重迷雾，让你得见最真实的“eval()”。

eval 执行什么

最基本的、也是最重要的问题是：eval 究竟是在执行什么？

在代码 `eval(x)` 中，`x` 必须是一个字符串，不能是其他任何类型的值，也不能是一个字符串对象。如果尝试在 `x` 中传入其他的值，那么 `eval()` 将直接以该值为返回值，例如：

 复制代码

```
1 # 值1
2 > eval(null)
3 null
4
5 # 值2
6 > eval(false)
7 false
8
9 # 字符串对象
10 > eval(Object('1234'))
11 [String: '1234']
12
13 # 字符串值
14 > eval(Object('1234').toString())
15 1234
```

这里，`eval()` 会按照 JavaScript 语法规则来尝试解析字符串 `x`，包括对一些特殊字面量（例如 8 进制）的语法解析。这样的解析会与 `parseInt()` 或 `Number()` 函数实现的类型转换有所不同，例如：对 8 进制的解析，在 `eval()` 的代码中就可以使用 `'012'` 来表示十进制的 10。而使用 `parseInt()` 或 `Number()` 函数，就不支持 8 进制，会忽略前缀字符 0，得到十进制的 12。



 复制代码

```
1 # JavaScript在源代码层面支持8进制
```

```
2 > eval('012')
3 10
4
5 # 但parseInt()不支持8进制（除非显式指定radix参数）
6 > parseInt('012')
7 12
8
9 # Number()也不支持8进制
10 > Number('012')
11 12
```

另外，`eval()` 会将参数`x`强制理解为语句行，这样一来，当按照“语句 -> 表达式”的顺序解析时，“`{ }`”将被优先理解为语句中的大括号。于是，下面的代码就成了 JavaScript 初学者的经典噩梦，也就是“尝试将一个对象字面量的字符串作为代码文本执行”所导致的问题。

 复制代码

```
1 # 试图返回一个对象
2 > eval('{abc: 1}')
3 1
```

在这种情况下，由于第一个字符被理解为块语句，那么“`abc:`”将被解析成标签语句；接下来，“`1`”会成为一个“单值表达式语句”。所以，结果是返回了这个表达式的值，也就是 `1`，而不是一个字面量声明的对象。

NOTE：这一个示例就是原来用作第 20 讲的标题的一行代码。只不过，在实际写的时候发现能展开讲的内容太少，所以做了一下合并。：)

eval 在哪儿执行

`eval` 总是将代码执行在当前上下文的“当前位置”。这里的所谓的“当前上下文”并不是它字面意思中的“代码文本上下文”，而是指“（与执行环境相关的）执行上下文”。

我在之前的文章中给你提到过与 JavaScript 的执行系统相关的两个组件：环境和上下文。但我一直在尽力避免详细地讨论它们，甚至在一些场合中将它们混为一谈。

然而，在讨论 `eval()`“执行的位置”的时候，这两个东西却必须厘清，因为严格地来讲，**环境**是 JavaScript 在语言系统中的静态组件，而**上下文**是它在执行系统中的动态组件。



环境

怎么说呢？

JavaScript 中，环境可以细分为四种，并由两个类别的基础环境组件构成。这四种环境是：全局（Global）、函数（Function）、模块（Module）和 Eval 环境；两个基础组件的类别分别是：声明环境（Declarative Environment）和对象环境（Object Environment）。

你也许会问：不对啊？我们常说的词法环境到哪里去了呢？不要着急，我们马上就会讲到它的。这里先继续说清楚上面的六个东西。

首先是两个类别，它们是所有其他环境的基础，是两种抽象级别最低的、基础的环境组件。**声明环境**就是名字表，可以是引擎内核用任何方式来实现的一个“名字 -> 数据”的对照表；**对象环境**是 JavaScript 的一个对象，用来“模拟 / 映射”成上述的对照表的一个结果，你也可以把它看成一个具体的实现。所以，

- 概念：所有的“环境”本质上只有一个功能，就是用来管理“名字 -> 数据”的对照表；
- 应用：“对象环境”只为全局环境的 global 对象，或 `with (obj)...` 语句中的对象 `obj` 创建，其他情况下创建的环境，都必然是“声明环境”。

所以，所谓四种环境，其实是上述的两种基础组件进一步应用的结果。其中，全局（Global）环境是一个复合环境，它由一对“对象环境 + 声明环境”组成；其他 3 种环境，都是一个单独的声明环境。

你需要关注到的一个事实是：所有的四种环境都与执行相关——看起来它们“像是”为每种可执行的东西都创建了一个环境，但是它们事实上都不是可以执行的东西，也不是执行系统（执行引擎）所理解的东西。更加准确地说：

上述四种环境，本质上只是为 JavaScript 中的每一个“可以执行的语法块”创建了一个名字表的影射而已。

执行上下文

JavaScript 的执行系统由一个执行栈和一个执行队列构成，这在之前也讲过。关于它们的应用原理，你可以回顾一下 [🔗 第 6 讲](#) (`x: break x`)，以及 [🔗 第 10 讲](#) (`x = yield x`) 中



的内容。

在执行队列中保存的是待执行的任务，称为 Job。这是一个抽象概念，它指明在“创建”这个执行任务时的一些关联信息，以便正式“执行”时可以参考它；而“正式的执行”发生在将一个新的上下文被“推入（push）”执行栈的时候。

所以，上下文是一个任务“执行 / 不执行”的关键。如果一个任务只是任务，并没有执行，那么也就没有它的上下文；如果一个上下文从栈中撤出，那么就必须有地方能够保存这个上下文，否则可执行的信息就丢失了（这种情况并不常见）；如果一个新上下文被“推入（push）”栈，那么旧的上下文就被挂起并压向栈底；如果当前活动上下文被“弹出（pop）”栈，那么处在栈底的旧上下文就被恢复了。

NOTE：很少需要在用户代码（在它的执行过程中）撤出和保存上下文的过程，但这的确存在。比如生成器（GeneratorContext），或者异步调用（AsyncContext）。

而每一个上下文只关心两个高度抽象的信息：其一是执行点（包括状态和位置），其二是执行时的参考，也就是前面一再说到的“名字的对照表”。

所以，重要的是：每一个执行上下文都需要关联到一个对照表。这个对照表，就称为“词法环境（Lexical Environment）”。显然，它可以是上述四种环境之任一；并且，更加重要的，也可是两种基础组件之任一！

如上是一般性质的执行引擎逻辑，对于大多数“通用的”执行环境来说，这是足够的。

但对于 JavaScript 来说这还不够，因为 JavaScript 的早期有一个“能够超越词法环境”的东西存在，就是“var 变量”。所谓词法环境，就是一个能够表示标识符在源代码（词法）中的位置的环境，由于源代码分块，所以词法环境就可以用“链式访问”来映射“块之间的层级关系”。但是“var 变量”突破了这个设计限制，例如，我们常常说到的变量提升，也就是在一个变量赋值前就能访问它；又例如所有在同一个全局或函数内部的 var x 其实都是同一个，而无论它隔了多少层的块级作用域。于是你可以写出这样一个示例来：

```
1 var x = 1;
2 if (true) {
3   var x = 2;
4 }
```

 复制代码



```
5     with (new Object) {
6         var x = 3;
7     }
8 }
```

这个示例中，无论你把`var x`声明在 `if` 语句后面的块中，还是 `with` 语句后面的块中，“1、2、3”所在的“`var` 变量”`x`，都突破了它们所在的词法作用域（或对应的词法环境），而指向全局的`x`。

于是，自 ECMAScript 5 开始约定，ECMAScript 的执行上下文将有两个环境，一个称为词法环境，另一个就称为变量环境（Variable Environment）；所有传统风格的“`var` 声明和函数声明”将通过“变量环境”来管理。

这个管理只是“概念层面”的，实际用起来，并不是这么回事。

管理

为什么呢？

如果你仔细读了 ECMAScript，你会发现，所谓的全局上下文（例如 Global Context）中的两个环境其实都指向同一个！也就是：

```
1  # (如下示例不可执行)
2  > globalCtx.LexicalEnvironment === global
3  true
4
5  > globalCtx.VariableEnvironment === global
6  true
```

 复制代码

这就是在实现中的取巧之处了。

对于 JavaScript 来说，由于全局的特性就是“`var` 变量”和“词法变量”共用一个名字表，因此你声明了“`var` 变量”，那么就不能声明“同名的 `let/const` 变量”。例如：



```
1 > var x = 100
2 > let x = 200
```

 复制代码

所以，事实上它们“的确就是”同一个环境。

而具体到“var 变量”本身，在传统中，JavaScript 中只有函数和全局能够“保存 var 声明的变量”；而在 ECMAScript 6 之后，模块全局也是可以保存“var 声明的变量”的。因此，事实上也就只有它们的“变量环境（VariableEnvironment）”是有意义的，然而即使如此（也就是说即使从原理上来说它们都是“有用的”），它们仍然是指向同一个环境组件的。也就是说，之前的逻辑仍然是成立的：

[复制代码](#)

```
1 # (如下示例不可执行)
2 > functionCtx.LexicalEnvironment === functionCtx.VariableEnvironment
3 true
4
5 > moduleCtx.LexicalEnvironment === moduleCtx.VariableEnvironment
6 true
```

那么，非得要“分别地”声明这两个组件又有什么用呢？答案是：对于 `eval()` 来说，它的“词法环境”与“变量环境”存在着其他的可能性！

不用于执行的环境

环境在本质上是“作用域的映射”。作用域如果不需要被上下文管理，那么它（所对应的环境）也就不需要关联到上下文。

在早期的 JavaScript 中，作用域与执行环境是一对一的，所以也就常常混用，而到了 ECMAScript 5 之后，有一些作用域并没有对应用执行环境，所有就分开了。在 ECMAScript 5 之后，ECMAScript 规范中就很少使用“作用域（Scope）”这个名词，转而使用“环境”这个概念来替代它。

哪些东西的作用域不需要关联到上下文呢？其实，一般的块级作用域都是这样的。例如一般的块级作用域：

[复制代码](#)

```
1 // 对象闭包
2 with(x) ...
```

很显然的，这里的with语句为对象x创建了一个对象闭包，就是对象作用域，也是我们在上面讨论过的“对象环境”。然而，由于这个语句其实只需要执行在当前的上下文环境（函数 / 模块 / 全局）中，因此它不需要“被关联到”一个执行上下文，也不需要作为一个独立的可执行组件“推入（push）”到执行栈。所以，这时创建出来的环境，就是一个不用于执行的环境。

只有前面所说的四种环境是用于执行的环境，而其他的所有环境（以及反过来对应的作用域）都是不用于执行的，它们与上下文无关。并且，既然与上下文没有关联，那么也就不存在“词法环境”和“变量环境”了。

从语法上，（在代码文本中）你可以找到除了上述四种环境之外的其他任何一种块级作用域，事实上它们每个作用域都有一个对应的环境：with 语句的环境用“对象环境”创建出来，而其他的（例如 for 语句的迭代环境，又例如 switch/try 语句的块）是用“声明环境”创建出来的。

对于这些用于执行的环境中的其中三个，ECMAScript 直接约定了它们（也就是 Global/Module/Function）的创建过程。例如全局环境，就称为 NewGlobalEnvironment()。因为它们都可以在代码解析（Parser）的阶段得到，并且在代码运行之前由引擎创建出来。

而唯有一个环境，是没有独立创建过程，并且在程序运行过程中动态创建的，这就是“Eval 环境”。

所以 Eval 环境是主要用于应对“动态执行”的环境。

eval() 的环境

上面我们说到，所谓“Eval 环境”是主要用于应对“动态执行”的，并且它的词法环境与变量环境“可能会不一样”。这二者其实是相关的，并且，这还与“严格模式”这一特殊机制存在紧密的关系。

当在eval(x)用一般的方式执行代码时，如果x字符串中存在着var变量声明，那么会发生什么事情呢？按照传统 JavaScript 的设计，这意味着在它所在的函数作用域，或者全局作用域会有一个新的变量被创建出来。这也就是 JavaScript 的“动态声明（函数和 var 变量）”和“动态作用域”的效果，例如：




```
1 var x = 'outer';
2 function foo() {
3   console.log(x); // 'outer'
4   eval('var x = 100;');
5   console.log(x); // '100'
6 }
7 foo();
```

如果按照传统的设计与实现，这就会要求 `eval()` 在执行时能够“引用”它所在的函数或全局的“变量作用域”。并且进一步地，这也就要求 `eval` 有能力“总是动态地”查找这个作用域，并且 JavaScript 执行引擎还需要理解“用户代码中的 `eval`”这一特殊概念。正是为了避免这些行为，所以 ECMAScript 约定，在执行上下文中加上“变量环境 (Variable Environment)”这个东西，以便在执行过程中，仅仅只需要查找“当前上下文”就可以找到这个能用来登记变量的名字表。

也就是说，“变量环境 (VariableEnvironment)”存在的意义，就是动态地登记“var 变量”。

因此，它也仅仅只用在“Eval 环境”的创建过程中。“Eval 环境”是唯一一个将“变量环境”指向了与它自有的“词法环境”不同位置的环境。

NOTE: 其实函数中也存在一个类似的例外。但这个处理过程是在函数的环境创建之后，在函数声明实例化阶段来完成的，因此与这里的处理略有区别。由于是函数声明的实例化 (FunctionDeclaration Instantiation) 阶段来处理，因此这也意味着每次实例化（亦即是每次调用函数并导致闭包创建）时都会重复一次这个过程：在执行上下文的内部重新初始化一次变量环境与词法环境，并根据严格模式的状态来确定词法环境与变量环境是否是同一个。

这里既然提到了“Eval 自有的词法环境”，那么也稍微解释一下它的作用。

对于 Eval 环境来说，它也需要一个自己的、独立的作用域，用来确保在“`eval(x)`”的代码 `x` 中存在的那些 `const/let` 声明有自己的名字表，而不影响当前环境。这与使用一对大括号来表示的一个块级作用域是完全一致的，并且也使用相同的基础组件（即声明环境、Declarative Environment）来创建得到。这就是在 `eval()` 中使用 `const/let` 不影响它所在函数或其他块级作用域的原因，例如：



```

2  function foo(){
3    var x = 100;
4    eval('let x = 200; console.log(x);'); // 200
5    console.log(x); // 100
6  }
foo().

```

而同样的示例，由于“变量环境”指向它在“当前上下文（也就是 foo 函数的函数执行上下文）”的变量环境，也就是：

 复制代码

```

1  # (如下示例不可执行)
2  > evalCtx.VariableEnvironment === fooCtx.VariableEnvironment
3  true
4
5  > fooCtx.VariableEnvironment === fooCtx.LexicalEnvironment
6  true
7
8  > evalCtx.VariableEnvironment = evalCtx.LexicalEnvironment
9  false

```

所以，当 eval 中执行代码“var x = ...”时，就可以通过evalCtx.VariableEnvironment来访问到fooCtx.VariableEnvironment了。例如：

 复制代码

```

1  function foo() {
2    var x = 100;
3    eval('var x = 200; console.log(x);'); // 200, x指向foo()中的变量x
4    console.log(x); // 200
5  }
6  foo();

```

也许你正在思考，为什么 eval() 在严格模式中就不能覆盖 / 重复声明函数、全局等环境中的同名“var 变量”呢？

答案很简单，只是一个小小的技术技巧：在“严格模式的 Eval 环境”对应的上下文中，变量环境与词法环境，都指向它们自有的那个词法环境。于是这样一来，在严格模式中使用eval("var x...")和eval("let x...")的名字都创建在同一个环境中，它们也就自然不能重名了；并且由于没有引用它所在的（全局或函数的）环境，所以也就不能改写这些环境中的名字了。



那么一个 `eval()` 函数**所需要的**“Eval 环境”究竟是严格模式，还是非严格模式呢？

你还记得“严格模式”的使用原则么？`eval(x)` 的严格模式要么继承自当前的环境，要么就是代码 `x` 的第一个指令是字符串“`use strict`”。对于后一种情况，由于 `eval()` 是动态 parser 代码 `x` 的，所以它只需要检查一下 parser 之后的 AST（抽象语法树）的第一个节点，是不是字符串“`use strict`”就可以了。

这也是为什么“切换严格模式”的指示指令被设计成这个奇怪模样的原因了。

NOTE：按照 ECMAScript 6 之后的约定，模块默认工作在严格模式下（并且不能切换回非严格模式），所以它其中的 `eval()` 也就必然处于严格模式。这种情况下（即严格模式下），`eval()` 的“变量环境”与它的词法环境是同一个，并且是自有的。因此模块环境中的变量环境（`moduleCtx.VariableEnvironment`）将永远不会被引用到，并且用户代码也无法在其中创建新的“`var` 变量”。

最后一种情况

标题中的 `eval()` 的代码文本，说的却是最后一种情况。在这种情况下，代码文本将指向一个“未创建即赋值”的变量 `x`，我们知道，按照 ECMAScript 的约定，在非严格模式中，向这样的变量赋值就意味着在全局环境中创建新的变量 `x`；而在严格模式中，这将被不允许，并因此而抛出异常。

由于 Eval 环境通过“词法环境与变量环境分离”来隔离了“严格模式”对它的影响，因此上述约定在两种模式下实现起来其实都比较简单。

对于非严格模式来说，代码可以通过词法环境的链表逆向查找，直到 `global`，并且因为无法找到 `x` 而产生一个“未发现的引用”。我们之前讲过，在非严格模式中，对“未发现的引用”的置值将实现为向全局对象“`global`”添加一个属性，于是间接地、动态地就实现了添加变量 `x`。对于严格模式呢，向“未发现的引用”的置值触发一个异常就可以了。

这些逻辑都非常简单，而且易于理解。并且，最关键和最重要的是，这些机制与我今天所讲的内容——也就是变量环境和词法环境——完全无关。

然而，接下来你需要动态尝试一下：



- 如果你按标题中的代码去尝试写 `eval()`，那么无论如何——无论你处于严格模式还是非严格模式，你都将创建出一个变量 `x` 来。


标题中的代码突破了“严格模式”的全部限制！这就是我下一讲要为你讲述的内容了。

今天没有设置知识回顾，也没有作业。但我建议你尝试一下标题中的代码，也可以回顾一下本节课中提到的诸多概念与名词。

我相信，它与你平常使用的和理解的，有许多不一致的地方，甚至有矛盾之处。但是，相信我，这就是这个专栏最独特的地方：它讲述 JavaScript 的核心原理，而不是重复那些你可能已经知道的知识。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

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 4  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 19 | `a + b`：动态类型是灾难之源还是最好的特性？（下）

下一篇 21 | `(0, eval)("x = 100")`：一行让严格模式形同虚设的破坏性设计（下）



JVM + NIO + Spring

各大厂面试题及知识点详解

限时免费



精选留言 (10)

写留言



Smallfly

2019-12-30

看了几遍，也参考了规范文档，还是有不少疑问点，希望老师能指点一二。

1. 自 ES5 开始引入词法环境，但还没有 let 关键字，只能用 var 关键字声明变量，而 var 声明的变量又是属于变量环境的 (Variable Environment)，那 ES5 是出于什么原因考虑引入词法环境呢？
2. 词法变量和 var 变量共用一个名字表，因此不能用 let 和 var 声明同名变量，为什么用 var 声明同名变量却可以？
3. 「环境的本质是”作用域的映射“」，这句话应该怎么理解呢。我现有的理解是，环境类似于一个链表形式的作用域，变量的查找就是从当前的作用域逐级向上查找，环境不应该是作用域的集合么？

这篇文章的标题看都看不懂，期待老师的下一讲.....

作者回复: 1. 这个问题我之前没细想过。但是ES5本来也不是一个突然提出来的东西，它其实是在ES4的基础上的一次“缩减”，并为以后的ES6之类做一个奠基性的事情。所以他们应该是考虑到了今后对词法环境的需求，并且事实上严格模式的提出，也是这样的目的，都是为后面的语言设计做一些基础性的工作。此外，事实上在当时他们已经有意地在设计和使用词法环境了，例如try...catch(x)中的这



个x，在IE和Firefox里面它的作用域一直就不同，IE是用变量环境的，而Firefox是用词法环境的。这个争端早于ES5之前就有的，所以问题出在哪里，以及将来需要做成怎样，这些在当时应该已经有明确的选择了，而ES5只不过把其中的冰山一角显露出来了而已。

2. var的多次声明只算第一次，其它的都是按赋值处理的（准确地说，是按多次绑定值处理的）。

3. 环境跟作用域的关系，浅层地说，它是后者的一个扩展版本的实现。——这样理解并没有问题。但是从概念层面来说，作用域在文本解析和引擎执行中的语义并不相同，而环境更是清晰地区分了这种不同。最后，环境与作用域的关系，有点类似于“类与对象实例”之间的关系。总之，他们有差异，但不明显。关于这个概念的进化演变过程，我在给D2的演讲《JS 语言在引擎级别的执行过程》中有讲过，只是目前D2组委会那边还没有视频放出来，回头你可以找来看看。

共 2 条评论 >



6



行问

2019-12-30

其实函数中也存在一个类似的例外。但这个处理过程是在函数的环境创建之后，在函数声明实例化阶段来完成的，.....

据我的理解，函数在 JavaScript 中是一等公民，函数提升，但我不懂的是“函数每调用一次，是否函数声明的实例化一次吗”。一直以来就是“定义（声明）一个函数”，再调用，但不知道函数实际背后的逻辑是怎样。

不知道我的问题，周大看懂了没。感谢

作者回复: 是的。函数每调用一次，“函数（的声明）”就会实例化一次。

不过，这个“函数声明的实例化”与你所理解的并不同，它不是说“function f() { ... }”这个函数要被实例化，而是“函数内的那些声明（包括形式参数名和var/let/const声明名字等）”——这些东西需要被实例化。

也就是说，之所以每次调用一下这个函数f，函数内的那些变量都被初始化为undefined，且形式参数要跟调用时的实际参数绑定一下，这些都是“函数（的声明）实例化”阶段帮你做掉的。

还有，这与“函数是一等公民”并没有什么关系，以及与“函数提升”也没有关系。“函数是一等公民”与“函数提升”之间，也没有关系。这些不要混着讲，他们是不同领域或不同层面的概念。——在第22讲的时候，我会略略地提及到“一等公民”的意义的。

最后，函数（作为声明文本），与函数的引用（作为一个变量名），以及函数的实例（作为一个对象），还有作为函数的闭包（是一个环境/作用域的映射/上下文的参考），这些概念之间有相关性，但并不是一一对应的。例如我常常说的，一个函数的实例，可以有多个闭包。——关于这个示例，你



想一下下面这种调用就明白了：

```
...  
  
function f() {  
  // 当前闭包，return当前函数实例，而在外部又调用该函数实例再创建了一个闭包  
  // 因此一个函数实例就有了两个闭包（递归就是更多个了）  
  return arguments.callee;  
}  
  
// （类似递归，）与递归的效果是相同的  
f();  
...
```



5



晓小东

2019-12-31

老师我遇一个问题，我在刷一个面试题，说全局环境下let,const 没有在window属性下面，我用chrome Source面版创建一个test.js, 测试了下，代码如下：

```
const a = 100;  
let b = 200;  
var a1 = 300;
```

在右侧scope中发现两个scope

Script:

```
a:100  
b: 200
```

Global:

```
a1: 300
```

非严格模式Script貌似跟eval执行有些类似

```
> ScriptCtx.VariableEnvironment === globalCtx.LexicalEnvironment  
true  
> ScriptCtx.VariableEnvironment === ScriptCtx.LexicalEnvironment  
false
```

但在严格模式下却又与eval不同 与非严格模式貌似没有区别

可不可以介绍下浏览器执行环境下Script相关执行环境与上下文知识点。



作者回复: 是这样的, 全局环境GlobalEnv是一个复合环境, 包括一个由global构成的对象环境(objEnv)和一个一般的声明环境(declsEnv), 它是双环境组成的, 统一交付一个环境存取界面。这个是它的结构 (objEnv/declsEnv很类似或者是直接对应你在DevTools's scope中看到的Global/Script) 。

let/const声明会放在declsEnv里面, 而var的变量名会通过objEnv来声明, 这个之前也讲过了。它们组合而成的, 整个的环境叫GlobalEnv, 并且用来作为ScriptCtx中的变量环境和词法环境, 并且后面的两种环境都指向“同一个”GlobalEnv, 没有差别, 也不会分开。——再次强调, ScriptCtx的“变量环境和词法环境”是相同的。

但是你在使用eval的时候, 对“变量环境和词法环境”的使用却是有所不同的。eval会自己创建一个执行上下文, 称为evalCtx。那么这个上下文中也有“变量环境和词法环境”, 它们指向却是不同的。evalCtx.LexicalEnvironment将是新创建的、独立的, 用来存放eval(x)的`x`中声明的const/let, 并且当eval结束时, 就销毁了。而evalCtx.VariableEnvironment将指向全局的ScriptCtx.VariableEnvironment, 这样也就指向了GlobalEnv。所以在`x`中声明的var, 就丢到了全局环境中, 也就写到了global这个对象上面, 不会受eval结束的影响。这就是它的完整机制了。

你可以写一个test.js用node运行起来看看:

```
```
```

```
// 运行中x, y有值
eval('var x = 100; let y = 200; console.log(x, y)');
```

```
// eval结束后x留在了全局, 而y没有了
console.log(typeof x, typeof y)
```
```

关于在浏览器环境中的情况, 也与这个逻辑没差。只不过DevTools展示那些内部组件 (例如Script/Global/Scope) 的时候, 使用的术语或者展示方式跟ECMAScript约定不一致罢了。



4



qqq

2019-12-30

eval 的间接调用会使用全局环境的对象环境, 所有绕过了严格模式, 是不是呀

作者回复: :) 对的。

但还可以有更深入的分析, 把这些东西讲透, 也是下一讲的主要目的。





4

**Elmer**

2020-07-17

回溯链是执行环境层面还是执行上下文层面呢？ 亦或着说回溯链是怎么创建的呢

作者回复: 词法环境的链表逆向查找（回溯），是发生在执行过程中的。当前执行上下文，总是持有环境（一个词法环境+一个变量环境）链的末端，通过parent属性往上找就行了。跟对象的原型链回溯是完全一致的。——所以，使用with (x) ...的时候，也就是将对象x展开成一个环境添加到上述链的末端就可以了。

@Elmer 你的另一个问题我一直挂着没回复，是因为那个问题涉及的内容太复杂，我想另开一篇文章去写写，所以还得稍等...

共 2 条评论 >



1

**晓小东**

2020-01-03

老师我对环境理解如下您看是否正确:

环境的创建发生在代码文本的解析阶段，生成可执行代码前，引擎就创建好了。

比如：我在代码文本中，声明了同名var与let ,就会发生语法错误。

问题1：环境应该也有类似（执行上下文）作用域链类似链式的结构，可以查抄上一级环境及下一级环境。（老师我现在可以理解作用域理解上为什么会有歧义了，作用域有静态环境下，还有动态执行环境下，所谓作用域就是为了管理某个东西而存在，只是不同环境下所管理对象不同而已。不知是否可以这样理解）。

2. 对于执行上文中包含词法环境与变量环境的理解：

既然环境创建发生于代码文本解析阶段，引擎执行这段文本之前，是否就意味着，这段文本所创建的环境，执行阶段是不能发生变更的，（应该可以增加，比如动态脚本载入，会扩展全局环境节点）

而eval中文本是不能解析的，所以只能在执行阶段去动态解析。

所以既然环境无法变更，就只能在当前可执行上下文中来维护登记这么一个动态生成变量环境，与词法环境，相对应还有一个变量列表和词法列表（不知道是否属于同一个东西，相对于变量环境与词法环境，delete x 可以被清除那些，还有全局上下文变量溢出的x）

3. 引擎在编译阶段确定标识符的位置，优化标识符查找性能，是否跟环境创建有关，或者环境创建就是一个独立目的，本质就是一个名字表的影射。

4. with跟eval类似，因为with 和eval 本质都是接收一个参数，变量，所以只能动态的去创建所对应环境，引擎无法前期做标识符查找的优化，大量eval和with会拖拽引擎的执行速



率。

作者回复: 哈哈。问题的密度好大。^^.

首先，环境的确是在执行之前创建的，但比语法分析阶段要略晚。环境是“因为要执行，所以才创建的”。而语法分析，与执不执行并没有关系。当一个东西（例如全局的代码块）需要执行时，引擎才会创建它对应的环境。

当然除了4个与执行直接相关的环境之外，其它的环境是“不用于执行的”。——这个在文章中一再区分它们，就是因为历史中它们混淆得非常厉害。不用于执行的环境，只是说它们与“某个执行上下文不直接关联，不是用来作为context.VariableEnvironment或context.LexicalEnvironment的。但是，它们仍然也是“因为某个东西要被执行，才会被创建出来的”，只是“不直接关联给context”而已。

关于问题1，环境确实也是链式的，它的创建接口类似于：

```

```
aEnv = new DeclarativeEnvironment (outerEnv)
```

```

所以你在“问题2”中的推论也就正确：缺省情况下，环境内的内容是不应该改变的。但JavaScript又允许“eval()”来动态地添加（以及使用var来动态删除）变量，所以才有了“词法环境和变量环境（VariableEnvironment and LexicalEnvironment）”双设计，因为其中的VariableEnvironment就是交由eval()来动态操作的。——所以“Eval环境”才与众不同。

所以总的来说，你的“问题2”中的理解是对的，方向也没错。只是你不敢大胆确认而已。^^.

关于问题3，是独立目的。本质就是名字表的影射。事实上在引擎内部，从parser阶段得来的语法分析树是另外的一个结构，那个分析过程也“可以有（不一定要有）”标识符冲突之类的判断，与环境中的检测无关。环境，其实是基于这个语法分析结果的一个映射，是为执行服务的。

关于问题4，确实很类似。它们都创建了新的环境。不过with(x)创建的环境不用于执行，它的context（用于管理一个独立的代码片断）仍然是当前函数或全局的；而“Eval环境”是用于执行代码的，它有自己的context。eval和with都会降低效率。

共 2 条评论 >



许童童

2020-01-01

老师的文章确实是非常有深度的。和其它的文章是完全不一样的。

作者回复: 多谢多谢。^^.



G

2021-11-14

老师您好，这篇文章的后半部分没有读懂，有以下疑问。

文中讲到·不用于执行的环境·这一部分的时候。提出如下观点：

·哪些东西的作用域不需要关联到上下文呢？其实，一般的块级作用域都是这样的。·
文中做了解释：

·由于这个语句其实只需要执行在当前的上下文环境（函数 / 模块 / 全局）中，因此它不需要“被关联到”一个执行上下文，也不需要作为一个独立的可执行组件“推入（push）”到执行栈·

—— 这个解释我不是很理解，是指由于 这个语句已经执行在四种环境中的一种了，所以无需再让其拥有自己的上下文了 吗？

同时下文继续说道：

·并且，既然与上下文没有关联，那么也就不存在“词法环境”和“变量环境”了·

—— 这里如何理解，一般的块级作用域都没有对应的词法环境和变量环境吗？

—— 如果块级作用域没有词法环境，那么上文中提到的`var 变量跨域超越词法环境`如何理解呢？var变量并不能超越函数，模块或者全局中的任意一个，只能够跨越块级作用域才对啊，这么说来，跨越词法作用域，应该就是指跨越块级作用域。但是这个推和文中的描述有了矛盾。

一定是哪个概念的理解有了偏差才会出现现在的问题。希望得到老师的解答

作者回复：“上下文”是一个特定概念，即执行上下文（ExecuteContext）。这个东西只是在引擎的执行环境中才有意义，它push到执行器的栈里就执行，pop出来就抛弃。此第一。

词法环境与变量环境，是执行上下文（是一个Record格式的数据结构）的两个成员，并且只关联到这种执行上下文中。也就是说，在执行上下文之外讨论词法环境与变量环境是没有意义的。此其二。

我们通常说的“环境”，是一个“环境（规范类型）”的数据结构。它被创建出来之后，要么放到一个可执行上下文中去（这时它是词法环境或变量环境），要么就只是一个环境，能被“环境块的链（Chain）”回溯访问到，但并不是直接关联到执行上下文中的。——举例来说，可能有100个“环境（记录）”，但只有20个执行上下文，那么也就只有20个词法环境或变量环境，其它的80个，虽然也都是



环境记录，但只是在Chain上，能被访问而已。

环境是执行过程中的物理结构，作用域是我们在文本分析代码时（在形式上、人为地）可以理解的抽象结构。事实上，环境是作用域的映射，你可以理解为作用域（至少）有一个环境。也就是说，有很多作用域“是环境”，但并不是“（可执行的上下文的）变量环境或词法环境”，就如上面举的100个环境块的例子一样。

所以后面才会说：

> 从语法上，（在代码文本中）你可以找到除了上述四种环境之外的其他任何一种块级作用域，事实上它们每个作用域都有一个对应的环境.....

不过从行文措辞的角度上来说，后文中“对于这些用于执行的环境中的其中三个，.....”，其中的“这些”改成“那些”要好一点。呵呵，因为在上下文的关系中这里其实有个转折的性质~~~

不知道.....有没有再次把你说昏。:(~



晓小东

2019-12-31

老师还是接上一个问题如下（我试着在之前打印下变量b）：

```
const a = 100;
```

```
console.log(b);
```

```
let b = 200;
```

```
var a1 = 300;
```

为什么console.log(b); 不是出现暂时性死区报错 ReferenceError: Cannot access 'b' before initialization

而是 Uncaught ReferenceError: b is not defined

执行脚本前貌似只处理varDecl(变量声明) lexicalDecls（词法声明）没有处理，与函数环境有所不同，global全局环境的这种处理方式，有什么好处呢（或者出于什么目的，而这样设计，或者就是对象环境与声明环境这种复合环境的特性）。

作者回复: 'b is not defined'应该是一个不太友好的错误提示。但是ECMAScript只规范了错误的类型，并不规范错误的提示，所以.....不同的引擎在这上面的表现确实不一样哦，也没有什么道理可言哦~

“执行脚本前貌似只处理varDecl(变量声明)lexicalDecls（词法声明）没有处理”，这样的理解不太正确。因为引擎对这两种都是处理的，只不过var变量是绑定了初值的，而let/const不绑定初值（直到



执行到它们声明并赋值的那行语句为止）。由于后者不绑定初值，所以出错提示“Cannot access 'b' before initialization”才是正确的意思，并且你按照这个来理解varDecl(变量声明)lexicalDecls（词法声明）就好了。

至于“b is not defined”，老实说我之前也发现这个东东，我觉得这样提示不对，但无可奈何不是么。:(~



Astrogladiator–埃蒂...

2019-12-30

```
(0, eval)("x = 100")
```

我用typeof (0,eval) 显示这个是函数类型，应该是一个立即执行的函数

类似 (function(params){})(params);

```
"use strict" , eval)("x = 100")
```

我用typeof (0,eval) 显示这个是函数类型，应该是一个立即执行的函数

类似 (function(params){})(params);

"use strict" 是不是说只是限制了当前上下文的声明环境，但在这个闭包构造的声明环境中并不受此限制？

作者回复: 不是的，“use strict”其实并没有这样的作用。关于这些，请期待下一讲。^^.

