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 进入课程 >

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 的两讲合讲，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

```
# 值 1
> eval(null)
null

# 值 2
> eval(false)
false

# 字符串对象
> eval(Object('1234'))
[String: '1234']

# 字符串值
> eval(Object('1234').toString())
1234
```

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：

```
# JavaScript 在源代码层面支持 8 进制
> eval('012')
10

# 但 parseInt() 不支持 8 进制（除非显式指定 radix 参数）
> parseInt('012')
12

# Number() 也不支持 8 进制
> Number('012')
12
```

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：

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

由于第一个字符被理解为块语句，那么“abc:”就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。

> 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：

复制代码

```
1 var x = 1;
```

```
if (true) {
  var x = 2;

  with (new Object) {
    var x = 3;
  }
}
```

这个示例中的“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：

复制代码

```
#(如下示例不可执行)
> globalCtx.LexicalEnvironment === global
true

> globalCtx.VariableEnvironment === global
true
```

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：

```
> var x = 100
> let x = 200
SyntaxError: Identifier 'x' has already been declared
```

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

```
#(如下示例不可执行)
> functionCtx.LexicalEnvironment === functionCtx.VariableEnvironment
true

> moduleCtx.LexicalEnvironment === moduleCtx.VariableEnvironment
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？例如一般的块级作用域：

```
// 对象闭包
with (x) ...
```

复制代码

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。

只有前面所说过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 swith/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)`使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：

复制代码

```
var x = 'outer';
function foo() {
  console.log(x); // 'outer'
  eval('var x = 100;');
  console.log(x); // '100'
}
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，就是动态的登记“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：

复制代码

```
function foo() {
  var x = 100;
  eval('let x = 200; console.log(x);'); // 200
  console.log(x); // 100
}
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

```
#(如下示例不可执行)
> evalCtx.VariableEnvironment === fooCtx.VariableEnvironment
true

> fooCtx.VariableEnvironment === fooCtx.LexicalEnvironment
true

> evalCtx.VariableEnvironment = evalCtx.LexicalEnvironment
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

```
function foo() {
  var x = 100;
  eval('var x = 200; console.log(x);'); // 200, x 指向 foo() 中的变量 x
  console.log(x); // 200
}
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，这将不被允许，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。

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　加餐 | 捡豆吃豆的学问（上）：这门课讲的是什么？

精选留言 (4)

写留言

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...

展开

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

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...
展开

Smallfly
2019-12-30

如果改成 eval（“x = 100”）会报错，改成：

var x
eval（“x = 100”）
...
展开