

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

2020-01-01 周爱民

《JavaScript核心原理解析》

课程介绍 >



讲述：周爱民

时长 23:11 大小 18.58M



你好，我是周爱民。欢迎回到我的专栏。书接上回，这一讲我们仍然讲动态执行。

之前我说过，`setTimeout` 和 `setInterval` 的第一个参数可以使用字符串，那么如果这个参数使用字符串的话，代码将会在哪里执行呢？毕竟当定时器被触发的时候，程序的执行流程“很可能”已经离开了当前的上下文环境，而切换到未知的地方去了。

所以，的确如你所猜测的那样，如果采用这种方式来执行代码，那么代码片断将在全局环境中执行。并且，这也是后来这一功能被部分限制了的原因，例如你在某些版本的 Firefox 中这样做，那么你可能会得到如下的错误提示：



复制代码

```
1 > setTimeout('alert("HI")', 1000)
2 Content Security Policy: The page's settings blocked the loading of a resource at
```

在全局环境中执行代码所带来的问题远远不止于此，接下来，我们就从这个问题开始谈起。

## 在全局环境中的 eval

早期的 JavaScript 是应用于浏览器环境中的，因此，当网页中使用<SCRIPT>标签加载.js 文件时候，代码就会在浏览器的全局环境中执行。但这个过程是同步的，将 BLOCK 掉整个网页的装载进度，因此有了defer这个属性来指示代码异步加载，将这个加载过程延迟到网页初始化结束之后。不过即使如此，JavaScript 代码仍然是执行在全局环境中的。

在那个时代，<SCRIPT>标签还支持for和event属性，用于指定将 JavaScript 代码绑定给指定的 HTML 元素或事件响应。当采用这种方式的时候，代码还是在全局环境中执行，只不过可能初始化为一个函数（的回调），并且this指向元素或事件。很不幸，有许多浏览器并不实现这些特性，尤其是for属性，它也许在 IE 中还存在，这一特性与 ActiveXObject 的集成有关。

关于脚本的动态执行，你能想象的绝大多数能在浏览器中玩的花样大概都在这里了。当然，你还可以在 DOM 中动态地插入一个SCRIPT标签来装载脚本，这在 Ajax 还没有那么流行的时候是唯二之选。另一种选择，是在 Document 初始化结束之前使用document.write()。

总而言之，为了动态执行一点什么，古典时代的 WEB 程序员是绞尽脑汁。

那么为什么不用eval()呢？

按照 JavaScript 脚本的执行机制，所有的.js 文件加载之后，它的全局代码只会执行一次。无论是在浏览器还是在 Node.js 环境中，以及它们的模块加载环境中，都是如此。这意味着放在这些全局代码中的eval()事实上也就只在初始化阶段执行一次而已。而eval()又有一个特别的性质，那就是它“总是在”当前上下文中执行代码。因此，所有其他的、放在函数中的eval()代码都只会影响函数内的、局部的上下文，而无法影响全局。

也就是说，除了初始化，eval()无法在全局执行。

不同的浏览器都有各自的内置机制来解决这个问题。IE 会允许用户代码调用window.execScript()，实现那些希望eval()执行在全局的需求。而 Firefox 采用了另外的一条道路，称为window.eval()。这个从字面上就很好理解，就是“让eval()代码执行在



window 环境中”。而window就是浏览器中的全局对象 global，也就是说，window.eval 与 global.eval 是等义的。

这带来了另外一个著名的、在 Firefox 早期实现的 JavaScript 特性，称为“对象的 eval”。

如果你试图执行obj.eval(x)，那么就是将代码文本x执行在obj的对象闭包中（类似于 with (obj) eval(x)）。因为全局环境就是使用 global 来创建的“对象环境（对象闭包）”，所以这是在实现“全局 eval()”的时候“顺手”就实现了的特性。

但这意味着用户代码可以将eval函数作为一个方法赋给任何一个 JavaScript 对象，以及任何一个属性名字。例如：

```
1 var obj = { do: eval };
2 obj.do('alert("HI")');
```

 复制代码

## 名字之争

现在，“名字”成了一个问题，在任何地方、任何位置，任何对象以及任何函数的上下文中都能“以不同的名字”来 eval() 一段代码文本。

这太不友好了！这意味着我们永远无法有效地判断、检测和优化用户代码。一方面，这对于程序员来说是灾难，另一方面，对引擎的实现者来说也非常绝望。

于是，从 ECMAScript 6 开始，ECMAScript 规定了“标准而规范地使用 eval()”的方法：你仅仅只能直接使用一个字面文本为“eval”字符串的函数名字，并且作为普通函数调用的形式来调用eval()，这样才算是“**直接调用的 eval()**”。

这个约定是非常非常罕见的。JavaScript 历史上几乎从未有过在规范中如此强调一个名字“在字面文本上的规范性”。在 ECMAScript 5 之后，一共也只出现了两个，这里的"eval"是一个，而另一个是严格模式（这个稍晚一点我们也会详细讲到）。

根据 ECMAScript 的约定，下面的这些都不是“直接调用的 eval()”：

 复制代码



```

1 // 对象属性
2 obj = { eval }
3 obj.eval(x)
4
5 // 更名的属性名或变量名（包括全局的或函数内局部的）
6 e = eval
7 var e = eval
8 e(x)
9
10 // super引用中的父类属性（包括原型方法和静态方法）
11 class MyClass { eval() { } }
12 MyClass.eval = eval;
13 class MyClassEx extends MyClass {
14     foo() { super.eval(x) }
15     static foo() { super.eval(x) }
16 }
17
18 // 作为函数（或其他大多数表达式）的返回
19 function foo() { return eval }
20 foo()(x)
21 // （或）
22 ( _=>eval ) (x)

```

总之，你所有能想到的一切——换个名字，或者作为对象属性的方式来调用 `eval`，都不再作为“直接调用的 `eval()`”来处理了。

那么，你可能会想要知道，怎样才算是“直接调用的 `eval()`”，以及它有什么效果呢？

很简单的，在全局、模块、函数的任意位置，以及一个运行中的 `eval(...)` 的代码文本的任意位置上，你使用的

`eval(x)`

这样的代码，都被称为“直接调用”。直接调用 `eval()` 意味着：

- 在代码所在位置，临时地创建一个“Eval 环境”，并在该环境中执行代码 `x`。

而反过来，其他任何将 `eval()` 调用起来，或者执行到 `eval()` 函数的方式，都称为“间接调用”。

而这两讲的标题中的写法，就是一个经典的“间接调用 `eval`”的写法：



```
1 (0, eval)(x)
```

晚一点，我们会再来详细讲述这个“间接调用”，接下来我们先说说与它相关的一点基础知识，也就是“严格模式”。

NOTE：之所以称为“经典的”写法，是因为在 ECMAScript 规范的测试项目 test262 中，所有间接调用相关的示例都是采用这种写法的。

## 严格模式是执行限制而不是环境属性

ECMAScript 5 中推出的严格模式是一项重大的革新之举，它静默无声地拉开了 ECMAScript 6~ECMAScript 10 这轰轰烈烈的时代序幕。

之所以说它是“静默无声的”，是因为这项特性刚出来的时候，大多数人并不知道它有什么用，有什么益处，以及为什么要设计成这个样子。所以，它几乎算是一个被“强迫使用”的特性，对你的团队来说是这样，对整个的 JavaScript 生态来说也是如此。

但是“严格模式”确实是一个好东西，没有它，后来的众多新特征就无法形成定论，它奠定了一个稳定的、有效的、多方一致的语言特性基础，几乎被所有的引擎开发厂商欢迎、接受和实现。

所以，我们如今大多数新写的 JavaScript 代码其实都是在严格模式环境中运行的。

对吗？

不太对。上面这个结论对于大多数开发者来说是适用的，并能理解和接受。但是，要是你在 ECMAScript 规范层面，或者在 JavaScript 引擎层面来看这句话，你会发现：咦？！“严格模式环境”是什么鬼？我们从来没见过这个东西！

是的，所谓“严格模式”，其实从来都不是一种环境模式，或者说，没有一个环境是具有“严格模式”这样的属性的。所有的执行环境——所有在执行引擎层面使用的“执行上下文（ExecuteContext）”，以及它们所引用的“环境（Environment）”，都没有“严格模式”这样的模式，也没有这样的性质。



我们所有的代码都工作在非严格模式中，而“严格模式”不过是代码执行过程中的一个限制。更确切地说，即使你用如下命令行：

 复制代码

```
1 > node --use-strict
```

来启动 Node.js，也仍然是运行在一个 JavaScript 的“非严格模式”环境中的！是的，是的，我知道，你可以立即写出来一行代码来反驳上述观点：

 复制代码

```
1 # （在上例启动的Node.js环境中测试）
2 > arguments = 1
3 SyntaxError: Unexpected eval or arguments in strict mode
```

但是请相信我：上面的示例只是一个执行限制，你绝对是运行在一个“非严格模式”环境中的！

因为所有的四种执行环境（包括 Eval 环境），在它们创建和初始化时都并没有“严格模式”这样的性质。并且，在全局环境初始化之前，在宿主环境中初始化引擎时，引擎也根本不知道所谓“严格模式”的存在。严格模式这个特性，是在环境创建完之后，在执行代码之前，从源代码文本中获取的性质，例如：

 复制代码

```
1 // (JavaScript引擎的初始化过程)
2
3 // 初始化全局, in InitializeHostDefinedRealm()
4 CALL SetRealmGlobalObject(realm, global, thisValue)
5   -> CALL NewGlobalEnvironment(globalObj, thisValue)
6
7 // 执行全局任务（含解析源代码文本等），in ScriptEvaluationJob()
8 s = ParseScript(sourceText, realm, hostDefined)
9 CALL ScriptEvaluation(s)
10
11 // 执行全局代码, in ScriptEvaluation(s)
12 result = GlobalDeclarationInstantiation(scriptBody, globalEnv)
13 if (result.[[Type]] === normal) {
14   result = ENGINING_EVALUATING(scriptBody)
15   ...
```





在这整个过程中，`ParseScript()` 解析源代码文本时，如果发现“严格模式的指示字符串”，那么就会将解析结果（例如抽象语法树 `ast`）的属性 `ast.isStrict` 置为 `true`。但这个标记仅仅只作用于抽象语法树层面，而环境中并没有相关的标识——在模块中，这个过程是类似的，只是缺省就置为 `true` 而已。

而另一方面，例如函数，它的“严格模式的指示字符串”也是在**语法解析阶段**得到的，并作为函数对象的一个内部标记。但是函数环境创建时却并不使用它，因此也不能在环境中检测到它。

我列举所有这些事实，是试图说明：“严格模式”是它们相关的可执行对象的一个属性，但并不是与之对应的执行环境的属性。因此，当“执行引擎”通过“词法环境或变量环境”来查找时，是看不到这些属性的，也就是说，执行引擎所知道的环境并没有“严格 / 不严格”的区别。

那么严格模式是怎么被实现的呢？

答案是，绝大多数严格模式特性都是在“相关的可执行对象”创建或初始化阶段就被处理掉的。例如，严格模式约定“没有 `arguments.caller` 和 `arguments.callee`”，那么，就在初始化这个对象的时候不创建这两个属性就好了。

另外一部分特性是在**语法分析阶段**识别和处理的。例如“禁止掉 8 进制字面量”，由于“严格模式的指示字符串（‘`use strict`’）”总是在第一行代码，所以在其他代码 `parser` 之前，解析器就已经根据指示字符串配置好了解析逻辑，对“8 进制字面量”可以直接抛出异常了。

从等等类似于此的情况，你能看到“严格模式”的所有限制特性，其实都并不需要执行引擎参与。进一步地来说，引擎设计者也并不愿意掺合这件事，因为这种模式识别将大幅度地降低引擎的执行效能，以及使引擎优化的逻辑复杂化。


但是，现在来到了“`eval()`”调用，怎么处理它的严格模式问题呢？

## 直接调用 VS 间接调用

绝大多数严格模式的特性都与语法分析结束后在指定对象上置的“`isStrict`”这样的标记有关，它们可以指导引擎如何创建、装配和调用代码。但是到了执行器内部，由于不可能从执行上下文开始反向查找环境，并进一步检测严格模式标识，所以 `eval()` 在原则上也不能知道“当前的”严格模式状态。



这有例外，因为“直接调用 `eval()`”是比较容易处理的，因为在使用`eval()`的时候，调用者——注意不是执行引擎——可以在当前自己的状态中得到严格模式的值，并将该值传入`eval()`的处理过程。这在 ECMAScript 中是如下的一段规范：

 复制代码

```
1 ...
2 - If strictCaller is true, let strictEval be true.
3 - Else, let strictEval be IsStrict of script.
4 ...
```

也就是说，如果 caller 的严格模式是 true，那么`eval(x)`就继承这个模式，否则就从x（也就是 script）的语法解析结果中检查 `IsStrict` 标记。

那么间接调用呢？

所谓间接调用，是 JavaScript 为了避免代码侵入，而对所有非词法方式的（即直接书写在代码文本中的）`eval()`调用所做的定义。并且 ECMAScript 约定：

- 约定 1：所有的“间接调用”的代码总是执行在“全局环境”中。

这样一来，你就没有办法向函数内传入一个对象，并用该对象来“在函数内部”执行一堆侵入代码了。

但是回到前面的问题：如果是间接调用，那么这里的`strictCaller`是谁呢？又处于哪种“严格模式”状态中呢？

答案是：不知道。因为当这样来引用全局的时候，上下文 / 环境中并没有全局的严格模式性质；反向查找源代码文本或解析过的 ast 树呢，既不经济也不可靠。所以，就有另外一个约定：

- 约定 2：所有的“间接调用”的代码将默认执行在“非严格模式”中。

也就是说，间接调用将突破引擎对严格模式的任何设置，你总是拥有一个“全局的非严格模式”并在其中执行代码。例如：





```

1 # (控制台)
2 > node --use-strict
3
4 # (Node.js环境, 严格模式的全局环境)
5 > arguments = 1
6 SyntaxError: Unexpected eval or arguments in strict mode
7 > 012
8 SyntaxError: Octal literals are not allowed in strict mode.
9
10 # 间接调用 (例1)
11 > (0, eval)('arguments = 1') // accept!
12 > arguments
13 1
14
15 # 间接调用 (例2)
16 > (0, eval)('012') // accept!
17 10
18
19 # 间接调用 (例3, 本讲的标题代码, 将创建变量x)
20 > (0, eval)('x = 100') // accept!
21 > x
22 100

```

## 为什么标题中的代码是严格模式

最后一个疑问, 就是为什么“标题中的这种写法”会是一种间接调用。并且, 更有对比性地来看, 如果是下面这种写法, 为什么就“不再是”间接调用了呢? 例如

```

1 # 直接调用
2 > (eval)('x = 100')
3 ReferenceError: x is not defined
4   at eval (eval at ...)
5
6 # 间接调用
7 > (0, eval)('x = 100')
8 100

```

在 JavaScript 中, 表达式的返回结果 (Result) 可能是值, 也可能是“引用 (规范类型)”。在“引用”的情况中, 有两个例子是比较常见、却又常常被忽略的, 包括:



```

1 # 属性存取返回的是引用
2 > obj.x

```

```
3
4 # 变量的标识符（作为单值表达式）是引用
5 > x
```

我们之前的课程中说过，所有这种“引用（规范类型）”类型的结果（Result），在作为左手端的时候，它是引用；而作为右手端的时候，它是值。所以，才会有“ $x = x$ ”这一个表达式的完整语义：

- 将右手端  $x$  的值，赋给左手端的  $x$  的引用。

好了，然而还存在一个运算符，它可以“原样返回”之前运算的结果（Result），这就是“分组运算符  $()$ ”。因为这个运算符有这样的特性，所以当它作用于属性存取和一般标识符时，分组运算返回的也仍然是后者的“运算结果（Result）”。例如：

 复制代码

```
1 # “结果 (Result)”是`100`的值
2 > (100)
3
4 # “结果 (Result)”是`{}`对象字面量（值）
5 > ({})
6
7 # “结果 (Result)”是`x`的引用
8 > (x)
9
10 # “结果 (Result)”是`obj.x`的引用
11 > (obj.x)
```

所以，从“引用”的角度上来看，`(eval)`和`eval`的效果也就完全一致，它们都是`global.eval`在“当前上下文环境”中的一个引用。但是我们接下来看，我们在这一讲的标题中写的这个分组表达式是这样的：

 复制代码

```
1 (0, eval)
```

这意味着在分组表达式内部还有一个运算，称为“连续运算（逗号运算符）”。连续运算的效果是“计算每一个表达式，并返回最后一个表达式的值（Value）”。注意，这里不是“结果（Result）”。所以它相当于执行了：



```
1 (GetValue(0), GetValue(eval))
```

因此最后一个运算将使结果从“Result→Value”，于是“引用（的信息）”丢失了。在它外层（也就是其后的）分组运算得到的、并继续返回的结果，就是“GetValue(eval)”了。这样一来，在用户代码中的(eval)(x)还是直接调用“eval 的引用”，而(0, eval)(x)就已经变成间接调用“eval 的值”了。

讲到这里，你可能已经意识到：关键在于eval是一个引用，还是一个值？是的，的确如此！不过在 ECMAScript 规范中，一个“eval 的直接调用”除了必须是一个“引用”之外，还有一个附加条件：它还必须是一个环境引用！

也就是说，属性引用的eval仍然是算着间接调用的。例如：

```
1 # （控制台，直接进入全局的严格模式）
2 > node --use-strict
3
4 # 测试用的代码（in Node.js）
5 > var x = 'arguments = 1'; // try source-text
6
7 # 作为对象属性
8 > var obj = {eval};
9
10 # 间接调用：这里的确是一个引用，并且名字是字符串文本"eval"，但它是属性引用
11 > (obj.eval)(x)
12 1
13
14 # 直接调用：eval是当前环境中的一个名字引用（标识符）
15 > eval(x)
16 SyntaxError: Unexpected eval or arguments in strict mode
17
18 # 直接调用：同上（分组运算符保留了引用的性质）
19 > (eval)(x)
20 SyntaxError: Unexpected eval or arguments in strict mode
```

所以，无论如何，只要这个函数的名字是“eval”，并且是“global.eval 这个函数在当前环境中的引用”，那么它就可以得到豁免，成为传统意义上的“直接调用”。例如：



```

1 // (一些豁免的案例，如下是直接调用)
2
3
4 // with中的对象属性 (对象环境)
5 with ({ eval }) eval(x)
6
7 // 直接名字访问 (作为缺省参数引用)
8 function foo(x, eval=eval) {
9     return eval(x)
10 }
11
12 // 不更改名字的变量名 (位于函数环境内部的词法/变量环境中)
13 function foo(x) {
14     var eval = global.eval; // 引用自全局对象
15     return eval(x)
16 }

```

## eval 怎么返回结果

那么最后一个问题，是“eval 怎么返回结果呢”？

这个问题的答案反倒非常简单。由于eval(x)是将代码文本x作为语句执行，所以它将返回语句执行的结果。所有语句执行都只返回值，而不返回引用。所以，即使代码x的运算结果 (Result) 是一个“引用 (规范类型)”，那么eval()也只返回它的值，即“GetValue(Result)”。例如：

 复制代码

```

1 # 在代码文本中直接创建了一个`eval`函数的“引用 (规范类型)”
2 > obj = { foo() { return this === obj } }
3
4 # this.foo调用中未丢失`this`这个引用
5 > obj.foo()
6 true
7
8 # 同上，分组表达式传回引用，所以`this`未丢失
9 > (obj.foo)()
10 true
11
12 # eval将返回值，所以`this`引用丢失了
13 > eval('obj.foo')()
14 false

```



## 结语

今天这一讲结束了对标题中代码的全部分析。由于标题中的代码是一个“间接调用的 eval”，因此它总是运行在一个非严格模式的全局中，于是变量x也就总是可以被创建或重写。

“间接调用 (IndirectCall)”是 JavaScript 非常非常少见的一种函数调用性质，它与“SuperCall”可以合并起来，视为 JavaScript 中执行系统中的“两大顶级疑难”。对间接调用的详细分析，涉及执行引擎的工作原理、环境和环境组件的使用、严格模式、引用（规范类型）的特殊性，以及最为特殊的“eval 是作为特殊名字来识别的”等等多个方面的基础特性。

间接调用对“严格模式”并非是一种传统意义上的“破坏”，只是它的工作机制正正好地绕过了严格模式。因为严格模式并不是环境的性质，而是代码文本层面的执行限制，所以当 eval 的间接调用需要使用全局时，无法“得到并进入”这种模式而已。

最后，间接调用其实是对传统的 window.execScript 或 window.eval 的一个保留。它有着在兼容性方面的实用意义，但对系统的性能、安全性和可靠性都存在威胁。无论如何，你应该限制它在代码中的使用。不过，它的确确实是 ECMAScript 规范中严格声明和定义过的特性，并且可称得上是“黑科技 (Hack skill)”了。

## 思考题


今天有一个作业留给你思考，问题很简单：

- 请你尝试再找出一例豁免案例，也就是直接调用 eval() 的写法。

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。

今天的课程就到这里。下一讲，我们将讨论“动态函数”，这既是“动态语言”部分的最后一小节，也将是专栏的最后一讲。

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 2

 提建议

上一篇 20 | (0, eval)("x = 100")：一行让严格模式形同虚设的破坏性设计（上）

下一篇 22 | new Function('x = 100')(); 函数的类化是对动态与静态系统的再次统一

## 学习推荐

# JVM + NIO + Spring

## 各大厂面试题及知识点详解

限时免费



## 精选留言 (7)

写留言



海绵薇薇

2020-09-05

老师好：

一开始我并不能理解函数中的eval是一个环境引用（一开始以为必须要全局环境引用）：

```
function a () {
```

```
    var eval = global.eval
```

```
}
```

这里的eval是一个环境引用，但是却和global.eval不是一个引用，因为

```
eval = global.eval
```

将右边的值赋值给左边的引用，所以eval是一个引用存储了global.eval这个引用的值。

也就是说eval是一个引用但是并不是global.eval这个引用，是函数环境引用。

所以eval要求的是环境引用并不要求全局环境引用。

另，环境可以分为四种：全局环境，对象环境，模块环境和eval环境。



另，eval的名字一定要是eval

根据上面两个直接调用eval的条件，能想到的还有如下两个额外的豁免方法。

第一（在当前环境中）：

```
try {  
    throw eval  
} catch (eval) {  
    let e = 1;  
    eval("console.log(e)")  
}
```

第二（在eval环境中）：

```
eval("let e = 1; eval(console.log(e)) ")
```

作者回复: 对的。你现在的理解就是对的。而且catch()这个豁免方法是完全出乎我意料的，但也是对的。赞~ ~

不过通常不要叫“环境引用”，就是“环境”而已，“环境（Environment）”是一个规范类型，它跟“引用（Reference）规范类型”一样，都是ECMAScript中的、同一级别的东东。



👍 3



行问

2020-01-01

2020 年好！

立一个 flag，今年要把您的书和专栏学习 2 次，不是阅读，是学习。虽然有很多的不懂，持续学习，不断积累。

也公布下本年度的 flag，有兴趣的小伙伴可以来共勉：2020 年做到 80% 以上的每一天 5:28 起床、23:00 前睡觉（2019 年只做到大概 50%，惭愧）

作者回复: ..... 给新年计划点赞！

我已经保持大概（或至少）20年的习惯，大概是每天3:00前后睡了。

不过这不是好习惯，不值得推广.....

共 2 条评论 >

👍 2



K4SHIFZ

2020-02-05

规范18.2.1.1章：



Runtime Semantics: PerformEval ( x, evalRealm, strictCaller, direct )

...

12.NOTE: If direct is true, ctx will be the execution context that performed the direct eval. If direct is false, ctx will be the execution context for the invocation of the eval function.

13.If direct is true, then

Let lexEnv be NewDeclarativeEnvironment(ctx's LexicalEnvironment).

Let varEnv be ctx's VariableEnvironment.

14.Else,

Let lexEnv be NewDeclarativeEnvironment(evalRealm.[[GlobalEnv]]).

Let varEnv be evalRealm.[[GlobalEnv]].

15.If strictEval is true, set varEnv to lexEnv.

...



kittyE

2020-01-01

```
var arr = []  
var x = 100  
arr[0] = eval  
(arr[0])(x)
```

属性引用仍然是间接调用，我这样理解对吗

作者回复: 是的。



qq

2021-03-26

```
(eval.valueOf())('a = 1')
```



undefined

2021-03-15

转眼这个课程一年了，还没有学完。  
最近开始重新翻出来学习，搭配书籍一块儿消化下。



qq



(0, eval)('this.eval("b = 1")')

作者回复: 这个可没有被豁免, "b = 1"仍然是被间接调用的eval执行的。

