

02 | `var x = y = 100`: 声明语句与语法改变了JavaScript语言核心性质

2019-11-13 周爱民

《JavaScript核心原理解析》

课程介绍 >



讲述：周爱民

时长 21:12 大小 19.43M



你好，我是周爱民。

如果你听过上一讲的内容，心里应该会有一个问题，那就是——在规范中存在的“引用”到底有什么用？它对我们的编程有什么实际的影响呢？


当然，除了已经提及过的`delete 0`和`obj.x`之外，在今后的课程中，我还会与你讨论这个“引用”的其它应用场景。而今天的内容，就从标题来看，若是我要说与这个“引用”有关，你说不定得跳起来说我无知；但若说是跟“引用”无关的话呢，我觉得又不能把标题中的这一行代码解释清楚。

为什么这行代码看起来与规范类型中的“引用”无关呢？因为这行代码出现的时候，连ECMAScript 这个规范都不存在。



我大概是在 JavaScript 1.2 左右的时代就接触到这门语言，在我写的最早的一些代码中就使用过它，并且——没错，你一定知道的：它能执行！

有很多的原因会促使你在 JavaScript 写出表达式连等这样的代码。从 C/C++ 走过来的程序员对这样的一行代码是不会陌生的。它能用，而且结果也与你的预期会一致，例如：

 复制代码

```
1 var x = y = 100;  
2 console.log(x); // 100  
3 console.log(y); // 100
```

它既没错、又好用，还很酷，你说我们为什么不用它呢？然而很不幸，[这行代码可能是 JavaScript 中最复杂和最容易错用的表达式了。](#)

所以今天我要和你一起好好地盘盘它。

声明

至今为止，除标签声明之外，JavaScript 中一共只有六条声明用的语句。注意，所有真正被定义“声明”的语法结构都一定是“语句”，并且都用于声明一个或多个标识符。这里的标识符包括变量、常量等。

严格意义上讲，JavaScript 只有变量和常量两种标识符，六条声明语句中：

- **let** *x* ...

声明变量 *x*。不可在赋值之前读。

- **const** *x* ...

声明常量 *x*。不可写。

- **var** *x* ...

声明变量 *x*。在赋值之前可读取到 `undefined` 值。



- **function** *x* ...

声明变量 *x*。该变量指向一个函数。

- **class** *x* ...

声明变量 *x*。该变量指向一个类（该类的作用域内部是处理严格模式的）。

- **import** ...

导入标识符并作为常量（可以有多种声明标识符的模式和方法）。

除了这六个语句之外，还有两个语句有潜在的声明标识符的能力，不过它们并不是严格意义上的声明语句（声明只是它们的语法效果）。这两个语句是指：

- **for** (*var|let|const x* ...) ...

for 语句有多种语法来声明一个或多个标识符，用作循环变量。

- **try ... catch** (*x*) ...

catch 子句可以声明一个或多个标识符，用作异常对象变量。

总的来说，除上述的语法，用户是没有其它方式来在当前的代码上下文中“声明”出一个标识符来的。而我之所以在这里严格强调这一“汇总性的”结果，是因为下面的一个简单论断，所有的“声明”：

- 都意味着 JavaScript 将可以通过“静态”语法分析发现那些声明的标识符；
- 标识符对应的变量 / 常量“一定”会在用户代码执行前就已经被创建在作用域中。

这个标题中的 `var x` 就是一个声明。在这个声明的后半部分，使用“=”这个符号引导了一个初始化语法——通常情况下可以将它理解为一个赋值运算。



从读取值到赋值

声明是在语法分析阶段就处理的，并且因此它会使得当前代码上下文在正式执行之前就拥有了被声明的标识符，例如x。

这其实非常有趣，因为这表明JavaScript 虽然被称为是“动态语言”，但确实是拥有静态语义的。而在 JavaScript 的早期，这个静态语义其实并没有处理得太好，一个典型的问题就是所谓的“变量提升”。也就是可以在变量声明之前访问该变量。例如：

 复制代码

```
1 console.log(x); // undefined
2 var x = 100;
3 console.log(x); // 100
```

这个“变量提升”还包括“变量被创建于声明它的语法块”之外的意思，但这并不是这里要讨论的内容，我会在今后再讲它。在今天的课程里，你只需要留意这个变量的读写过程就好了。那么，关于读取值，之前声明的变量与常量又有什么不同呢？

如上面已经说过的，由于标识符是在用户代码执行之前就已经由静态分析得到，并且创建在环境中，因此 let 声明的变量和 var 声明的变量在这一点上没有不同：它们都是在读取一个“已经存在的”标识符名。例如：

 复制代码

```
1 var y = "outer";
2 function f() {
3   console.log(y); // undefined
4   console.log(x); // throw a Exception
5   let x = 100;
6   var y = 100;
7   ...
8 }
```

正是由于var y所声明的那个标识符在函数 f() 创建（它自己的闭包）时就已经存在，所以才阻止了console.log(y)访问全局环境中的y。类似的，let x所声明的那个x其实也已经存在 f() 函数的上下文环境中。访问它之所以会抛出异常（Exception），不是因为它不存在，而是因为这个标识符被拒绝访问了。



在 ECMAScript 6 之后出现的let/const变量在“声明（和创建）一个标识符”这件事上，与var并没有什么不同，只是 JavaScript 拒绝访问还没有绑定值的let/const标识符而已。

回到 ECMAScript 6 之前：JavaScript 是允许访问还没有绑定值的`var`所声明的标识符的。这种标识符后来统一约定称为“变量声明（`varDecls`）”，而“`let/const`”则称为“词法声明（`lexicalDecls`）”。JavaScript 环境在创建一个“变量名（`varName` in `varDecls`）”后，会为它初始化绑定一个 `undefined` 值，而“词法名字（`lexicalNames`）”在创建之后就没有这项待遇，所以它们在缺省情况下就是“还没有绑定值”的标识符。

NOTE：6 种声明语句中的函数是按 `varDecls` 的规则声明的；类的内部是处于严格模式中，它的名字是按 `let` 来处理的，而 `import` 导入的名字则是按 `const` 的规则来处理的。所以，所有的声明本质上只有三种处理模式：`var` 变量声明、`let` 变量声明和 `const` 常量声明。

所以，标题中的`var x = ...`在语义上就是为变量 `x` 绑定一个初值。在具体的语言环境中，它将被实现为一个赋值操作。

赋值

如果是在一门其它的（例如编译型的）语言中，“为变量 `x` 绑定一个初值”就可能实现为“在创建环境时将变量 `x` 指向一个特定的初始值”。这通常是静态语言的处理方法，然而，如前面说过的，JavaScript 是门动态的语言，所以它的“绑定初值”的行为是通过动态的执行过程来实现的，也就是赋值操作。

那么请你仔细想想，一个赋值操作在语法上怎么表达呢？例如：

```
1 变量名 = 值
```

 复制代码

这样对吗？不对！在 JavaScript 中，这样讲是非常不正确的。正确的说法是：

```
1 lRef = rValue
```

 复制代码

也就是将右操作数（的值）赋给左操作数（的引用）。它的严格语法表达是：

LeftHandSideExpression **<=** | **AssignmentOperator** **>** *AssignmentExpression*



也就是说，在 JavaScript 中，一个赋值表达式的左边和右边其实“都是”表达式！

向一个不存在的变量赋值

接下来我要给你介绍的是从 JavaScript 1.0 开始就遗留下来的一个巨坑，也就是所谓的“**变量泄漏**”问题。这在早期的 JavaScript 中的确是一个好用的特性：如果你向一个不存在的变量名赋值，那么 JavaScript 会在全局范围内创建它。

也就是说，代码中不需要显式地声明一个变量了，变量可以随用随声明，也不用像后来的 let 语句一样，还要考虑在声明语句之前能不能访问的问题了。这非常简单，在少量的代码中也相当易用。

但是，如果代码规模扩大，变成百千万行代码，那么“**一个全局变量是在哪里声明和创建的**”就变成一个非常要紧的问题。

如果随时都可能泄露一个代码给全局，或者随时都可能因为忘记本地的声明而读写了全局变量，那对调试除错将是一场灾难。另外，晚一些出现的运行期优化技术也不能很好地处理这种情况。所以从 ECMAScript5 开始的严格模式就禁止了这种特性，试图避免用户将变量泄露到全局环境。

然而现实中，即使在严格模式下这种泄露也未能避免。这称为“**间接执行**”，这将是另一个巨大的议题，并且是 ECMAScript6 之后开始的一种新的机制。但是现在这里发生的事情，也就是这个“向不存在的变量赋值”的问题，是从 JavaScript 1.0 时代就遗留下来的问题，也是 ECMAScript 为 JavaScript 填补的最大设计漏洞之一。

那么，在具体技术细节上，这个变量声明是如何发生的呢？

事实上，这是因为在早期设计中，JavaScript 的全局环境是引擎使用一个称为“**全局对象**”东西管理起来的。

这个全局对象几乎类似或完全等同于一个普通对象。只不过，JavaScript 引擎将全局的一些缺省对象、运行期环境的原生对象等东西都初始化在这个全局对象的属性中，并使用这个对象创建了一个称为“**全局对象闭包**”的东西，从而得到了 JavaScript 的全局环境。



早期的 JavaScript 的引擎实现非常简洁，许多基础的技术组件都是直接复用的，例如这里的所谓全局环境、全局闭包，或者全局对象的实现方法，就与“**with 语句**”的效果完全相同——他们是相互复用的。

当向一个不存在的变量赋值的时候，由于全局对象的属性表是可以动态添加的，因此 JavaScript 将变量名作为属性名添加给全局对象。而访问所谓全局变量时，就是访问这个全局对象的属性。因此，实际效果就变成了“可以动态地向全局环境中添加一个变量”。并且，显然地，我们在第一讲已经讲过这个结果——你可以删除掉这个动态添加的“变量”，因为本质上就是在删除全局对象的属性。

那么现在（我是指 ECMAScript6 之后）的 JavaScript 的全局环境有什么不同吗？

为了兼容旧的 JavaScript 语言设计，现在的 JavaScript 环境仍然是通过将全局对象初始化为这样的一个全局闭包来实现的。但是为了得到一个“尽可能”与其它变量环境相似的声明效果（varDecls），ECMAScript 规定在这个全局对象之外再维护一个变量名列表（varNames），所有在静态语法分析期或在 eval() 中使用 var 声明的变量名就被放在这个列表中。然后约定，这个变量名列表中的变量是“直接声明的变量”，不能使用 delete 删除。

于是，我们得到了这样的一种结果：

 复制代码

```
1 > var a = 100;
2 > x = 200;
3
4 # `a`和`x`都是global的属性
5 > Object.getOwnPropertyDescriptor(global, 'a');
6 { value: 100, writable: true, enumerable: true, configurable: false }
7 > Object.getOwnPropertyDescriptor(global, 'x');
8 { value: 200, writable: true, enumerable: true, configurable: true }
9
10 # `a`不能删除, `x`可以被删除
11 > delete a
12 false
13 > delete x
14 true
15
16 # 检查
17 > a
18 100
19 > x
20 ReferenceError: x is not defin
```



所以，表面看起来“泄漏到全局的变量”与使用var声明的都是全局变量，并且都实现为 global 的属性，但事实上它们是不同的。并且当var声明发生在 eval() 中的时候，这一特性又还有所不同，例如：

 复制代码

```
1 # 使用eval声明
2 > eval('var b = 300');
3
4 # 它的性质是可删除的
5 > Object.getOwnPropertyDescriptor(global, 'b').configurable;
6 true
7
8 # 检测与删除
9 > b
10 300
11 > delete b
12 true
13 > b
14 ReferenceError: b is not define
```

这种情况下使用var声明的变量名尽管也会添加到 varNames 列表，但它也可以从 varNames 中移除（这是唯一一种能从 varNames 中移除项的特例，而 lexicalNames 中的项是不可移除的）。

发生了什么？

所以，现在回到今天讨论的这行代码var x = y = 100，在这行代码中，等号的右边是一个表达式y = 100，它发生了一次“向不存在的变量赋值”，所以它隐式地声明了一个全局变量y，并赋值为 100。

而一个赋值表达式操作本身也是有“结果（Result）”的，它是右操作数的值。注意，这里是“值”而非“引用”，例如下面的测试中的a将是一个函数，而不是带着“this 对象”信息的方法：

 复制代码



```
1 // 调用obj.f()时将检测this是不是原始的obj
2 > obj = { f: function() { return this === obj } };
3
4 // false, 表明赋值表达式的“结果(result)”只是右侧操作数的值，即函数f
5
```



```
6 > (a = obj.f)();  
f-1--
```

到现在为止，我们讲述了整个语句的过程，也就是说，由于“`y = 100`”的结果是 100，所以该值将作为初始值赋值“变量 `x`”。并且，从语义上来说，这是变量“`x`”的初始绑定。

之所以强调这一点，是因为相同的分析过程也可以用在 `const` 声明上，而 `const` 声明是只有一次绑定的，常量的初始绑定也是通过“执行赋值过程”来实现的。

知识回顾

- `var` 等声明语句总是在变量作用域（变量表）或词法作用域中静态地声明一个或多个标识符。
- 全局变量的管理方式决定了“向一个不存在的变量赋值”所导致的变量泄漏是不可避免的。
- 动态添加的“`var` 声明”是可以删除的，这是唯一能操作 `varNames` 列表的方式（不过它并不存在多少实用意义）。
- 变量声明在引擎的处理上被分成两个部分：一部分是静态的、基于标识符的词法分析和管
理，它总是在相应上下文的环境构建时作为名字创建的；另一部分是表达式执行过程，是对
上述名字的赋值，这个过程也称为绑定。
- 这一讲标题里的这行代码中，`x` 和 `y` 是两个不同的东西，前者是声明的名字，后者是一个
赋值过程可能创建的变量名。

思考题

根据今天讲解的内容，我希望你可以尝试回答以下问题：

- 严格来说，声明不是语句。但是，是哪些特性决定了声明不是“严格意义上的”语句呢？

在下一讲中我会来讲一讲 JavaScript 社区中的一个历史悬案，这桩悬案与今天讨论的这行代码的唯一区别在于：它不是声明语句，而是赋值表达式。



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

上一篇 01 | delete 0: JavaScript中到底有什么是可以销毁的

下一篇 03 | $a.x = a = \{n:2\}$: 一道被无数人无数次地解释过的经典面试题

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 (59)

💬 写留言



fatme

2019-11-14

声明和语句的区别在于发生的时间点不同，声明发生在编译期，语句发生在运行期。声明发生在编译期，由编译器为所声明的变量在相应的变量表，增加一个名字。语句是要在运行时执行的程序代码。因此，如果声明不带初始化，那么可以完全由编译器完成，不会产生运行时执行的代码。

作者回复: +1 ^^.



👍 79



hello, 老师好啊, 研读完文章和评论后还存在如下疑问:

1.

```
var y = "outer";
```

```
function f() {
```

```
    console.log(y); // undefined
```

```
    console.log(x); // throw a Exception
```

```
    let x = 100;
```

```
    var y = 100;
```

```
    ...
```

```
}
```

老师解释函数内部读取不到外部变量的原因是“函数创建的时候标识符x和y就被创建了”。因为这是一个函数声明, 也就是在编译的时候就创建了。

高程上的意思是函数执行的时候会生成一个活动对象当做变量对象, 这时候标识符才会生成, 包括arguments, 形参实参, 声明的变量, 挂在活动对象上。

两个解释好像都能说明上面的现象。

有点糊涂了。

2.

```
function a() {
```

```
    function b() {}
```

```
}
```

在代码执行前连函数b都被创建了吗?



3. 老师对一定了解闭包的本质，后面有机会说到吗？

作者回复：“因为这是一个函数声明，也就是在编译的时候就创建了”

===

这个是你混乱的根源，因为从“函数声明”到“闭包”之间还有好几个环节的。

“函数创建的时候标识符x和y就被创建了”，这个函数内的x和y都是可以被parser到的，因此在函数静态解析完之后，函数就可以确定内部有x和y了，并且相应的静态作用域也（在语法处理相关的内核逻辑部分）被创建了。但是这些东西是用户代码完全不可见的，你不会知道，也用不到。

然后是一个函数作为表达式得到它的一个实例的过程，也就是

```
> (function () {}).xxxxx
```

中“.xxxx”之前的部分作为表达式被独立处理的时候，这种情况下函数会有一个自己的环境，该环境也是“基于前面得到的静态作用域”来创建的。这个环境仍然不为用户所知，只是它能传递，例如你可以把这样的东西“返回（return）”或“赋值（=）”给别的东西，你会发现这个东西的“环境/执行上下文”并没有变，所以它们在“返回（return）”或“赋值（=）”的过程之前就被创建了，并且能被这些操作所“传递”。

最后才是闭包，闭包只发生于“f()”的这个“调用操作()”之后——注意是“之后”，所以高程等等都是说它在“执行之后”标识符才会生成，而我这一章都是在讲“静态语义”，所以我会说他在函数（作为一个静态的对象）创建的时候就已经有这些标识符了。

这两者都不矛盾。根本上来说，函数实例、函数闭包，都不过是“静态词法解析结果（函数定义/(un) Anonymous Function Definition）”的一个映像，这个“函数定义”就是第4小节讲的那个东东。

Ok，既然“x,y都是代码执行前就被（静态分析）创建了的”，那么哪种情况下“x”才不是“词法的”呢？

```
function f() { let y; return x+y }
```

注意在这个例子中，“x”就不是词法的，它在任何情况下都不被创建，不在f()的标识符列表中，也不在语法/词法分析中。

Ok. 这是第1个问题。

关于第2个问题，答案其实如上所述：如果“创建”是指静态语义中的一个“函数”，那么确实是创建了的。——但它还没有“绑定”到一个闭包的执行上下文中。

关于第3个问题，是这样的，这一整个课程（系列）其实不只20讲，大概会是40~45讲的样子。但编



辑同学不允许我公布后续内容的计划（呜……），所以……我只能告诉你，在下一个课程里面，才会讲到闭包。

共 2 条评论 >

👍 20



铭

2019-11-13

〈以下是小生愚见〉

概念纷繁，建议老师将讲解重心放到这门语言的现有特性，贯之历史脉络，是否（怎样）解决了某种设计缺陷。这样，知识纵深感更强，并可指导实际工作以避免踩坑。适当穿插示例代码和图文更佳。

作者回复: 多谢。我在后面的课程中尽量注意这个 ^^.

共 2 条评论 >

👍 18



Ppei

2020-05-02

老师你好，词法声明会有提升吗？

一些书里面会说不存在变量提升，但是文中说，是拒绝访问。

我是不是该从编译期跟运行期去理解？

作者回复: @Ppei 你提这个问题是很到位的，严格地说，这是我这一讲中跟ECMAScript描述中不一致的地方。所以我需要非常细致地回复你的问题。

严格来讲，所谓变量提升(Hoisting)指的是如下的现象：

...

```
function foo() {  
  var x = 100;  
  if (true) {  
    let y = 200;  
    var z = 300;  
    ...  
  }  
}
```

在如上示例中，变量z的声明被提升到了x相同的位置声明，相对应的、用作比较的y就没有提升，它声明在if()语句之后的块语句中。

这是严格的ECMAScript规范概念下的“变量提升”，因此它的确描述的是一种语法现象——在语法阶



段，通过自然的、表面的识别就可以理解的现象。

在ECMAScript中被明确指出的提升现象还包括函数声明。亦即是说，下面的示例：

```
...  
function foo() {  
  var x = 100;  
  if (true) {  
    let y = 200;  
    function z() {  
    }  
    ...  
  }  
}  
...
```

在这个示例中，函数z()和之前的变量z声明都同样被提升到了x的位置。

这两种现象——或这两种“严格意义上的变量提升”是JavaScript 1.x时代的早期设计带来的结果。因为没有块级作用域，因此所有的声明都必须“上浮”到函数或全局一级的作用域来处理。这也是我在《JavaScript语言精髓与编程实践》一书中，需要分析“作用域的等级”的原因。这种等级决定了作用域之间的交互关系，而关系之一，就是所谓“提升”。

除了这两种“严格意义上的变量提升”——函数声明是“隐式的变量声明”——之外，ECMAScript并没有规范其它的提升现象。

然而通常，在我的讲述中会把如下的现象也称为提升。有些时候，为了特别地指出它们，我会强调它们是一种“提升效果”。例如：

```
...  
console.log(x);  
var x = 100;  
...  
...
```

在如上的例子中，x在它的声明语句之前是能够被访问的。因此，这也是提升。与之相比较的：

```
...  
console.log(y);  
var x = 100;  
let y = 200;  
...
```

这个示例中的y就不能被访问，并且提示是：

```
> ReferenceError: y is not defined
```



所以，在语法概念上，这个`y`是“还没有声明的”。但是，在事实上呢？在事实上，`y`和`x`都是被声明过了的，只不过`y`被声明之后未被初始化，而`x`被初始化成了`undefined`。

所以，“从实现上来说”，这里的所谓

> 1、var有变量提升（效果），而let没有变量提升（效果）

其实与之前讨论的

> 2、var与function在词法作用域中的变量提升

并不一样。上述规则2是在规范层面真实的存在着的，是语法级别的、在构建作用域的阶段就被“提升”的。而规则1却不是，规则1中的var/let声明在相同的位置，只不过let声明成了未初始化的，并且ECMAScript约定：访问一个未初始化的变量（例如y）时，将错误信息显示成“... is not defined”而已。

所以我才会说，这种var提升效果，在本质上是`x`没有被拒绝访问，而相对的`y`被拒绝访问。

回到一些其它的有关Hoisting现象的说明上来，你可以参考一下MDN：

<https://developer.mozilla.org/zh-CN/docs/Glossary/Hoisting>

不过MDN对上述两大类的提升/提升效果也是混在一起讲的，并没有从实现机制的角度来阐释。另外，import和var一样也有“提升效果”，则是“第三种提升”，机制上也是有所不同的，这个我在本专栏中略有讲到，但并不详细。

共 2 条评论 >

👍 15



Mr_Liu

2019-11-13

思考题: 小白的我，没有太明确的答案，暂时还不能明确自己理解究竟是否正确，希望听老师后续的课程能够明白

读完今天的这篇理解了昨天的提问，为什么var x = '123' delete x 是false，即使是

```
var obj = {  
  a: '123',  
  b: {  
    name: '123'  
  }  
}
```

```
var z = obj.b
```

delete z 返回也是false

所以问了那么delete x 存在有什么意义。

今天老师的科普解答了




```
x = '123'
```

```
delete x 返回true
```

是因为你可以删除掉这个动态添加的“变量”，因为本质上就是在删除全局对象的属性。同时也理解了上一讲的“只有在delete x等值于delete obj.x时 delete 才会有执行意义。例如with (obj) ...语句中的 delete x，以及全局属性 global.x。”这句

但上一节关于“delete x”归根到底，是在删除一个表达式的、引用类型的结果（Result），而不是在删除 x 表达式，或者这个删除表达式的值（Value）。后一句理解了，但前一句是否可以理解为实际上是删除引用呢，希望老师解答一下

立一个flag，每个争取评论下面都有我的，不为别的，就为增加自己的思考

作者回复: 是的。

delete从不删除值。delete只能删除引用，例如obj.x，或者with(obj) delete x，这些都是以引用的方式得到的，所以delete才能删除它们。

`delete 0`这种行为并不真的能够发生，它什么也没做，只是返回了true而已。



👍 12



Elmer

2019-12-23

文中提到：ECMAScript 规定在这个全局对象之外再维护一个变量名列表（varNames）那么window是怎么取到这些变量的值的，如window.a不是平级么。在global scope中，window var let const 的关系是什么。求讲解

作者回复: 这里说到的 varNames在绝大多数情况下是没用的，与你这里想讨论的东西关系也不大。

window在浏览器环境中等同于global。你在浏览器控制台上查看一下就明白了：

```
...
```

```
# 这个globalThis是新EMAScript规范中声明的，它等义于传统的global
```

```
> window === globalThis
```

```
true
```

```
# 用下面的代码可以获得“传统的global”，并比较之
```

```
> window === Function('return this')()
```

```
true
```

```
...
```

所以所谓window，就是global，相同的东西。那么window中取变量也就是查global这个对象的属性



表，之前也都说过了，不再讲了。至于var/let/const之间的关系，在全局域（global scope）中，var声明在global的属性表中，并在varNames中有一个登记；let/const共享使用同一个称为词法环境（lexicallyEnv）的东西，所以它们不能同名。词法环境在后面会讲到。

因此从原理上来说，全局作用域中的global对象属性，与词法环境中的名字其实是可以重名的。——所以，var与let/const的重名限制，主要是来自于语法。

例如：

```
...  
  
# 添加属性  
> global.n = 100  
  
# 看起来有了全局变量的样子（实际上没有在varNames中登记）  
> n  
100  
  
# 现在可以声明let  
> let n = 200  
  
# 这是一个Let变量  
> n  
200  
  
# 如果你使用var声明，则与let/const会有重名冲突了  
> var x = 300  
> let x = 400  
SyntaxError: Identifier 'x' has already been declared  
...
```



👍 10



孜孜

2020-07-11

今天写IIFE,突然有点问题想问下老师,

1. 为什么(function f(){ return this}) 可以, (var test=1) 不可以。
2. 两种IIFE的写法, (function f(){ return this})(); 和 (function f(){ return this}()) 有何区别。
3. 函数调用 () 和表达式取值 () 如何在ECMAScript找到说明?



作者回复: 1. 第一个是“函数表达式”，(expression, ...)的语法能通过；第二个是语句（6种声明语句之一），所以这个语法通不过。

2. 第一种`function f...`()是将函数f作为一个“单值表达式”，这里的第一对括号是作为“分组运算符”来用的，它强制将`function f...`作为一个表达式来做语法解析，从而避免了它被“（优先地）解释为函数声明语句”。而第二种，也就是`function f {}`()中的f，也是因为相同的原因（第一对括号作为分组运算符），从而导致解析过程进入表达式解析，所以这个函数与第一种没区别。但是接下来，它进行了函数运算调用——也就是f()。——而这里也是两种用户的不同，第一种用法是第一对括号返回函数，第二种用法中第一对括号返回的是函数调用的结果。

3. 这分别是：

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

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

不过第二种不称为“表达式取值”，而是“分组表达式”，它的运算效果是“取Result”——也就是说不仅是取值，还可以包括取“（ECMAScript规范的）引用”。

共 2 条评论 >

👍 8



Zheng

2020-01-14

老师，我用node执行这段代码，结果是undefined，但是换成浏览器打印就可以打印出来a的配置信息，这是因为node环境和浏览器的差异还是什么，我试过好多次了，应该不是偶然：

```
var a = 100;
x = 200;
console.log(Object.getOwnPropertyDescriptor(global,"a")); //浏览器执行的时候global改为globalThis
```

作者回复：是这样的，node缺省情况下是把.js文件当成模块来加载的，它会为每一个模块（亦即是.js文件）包一层所谓的“模块封装器”。这个封装器是一个函数。所以，.js文件中的代码事实上是运行在函数中的。这样一来，`var a`就变成了声明函数内的局部变量，而不是在全局global上的。

在node的控制台里面输入的代码是作为全局环境下的代码来执行的，因此如果你的代码是在node控制台上逐行执行，就没有问题。另外，如果你使用-e参数来执行，那么也能得到这样的效果（这种情况下node不会添加模块封装器）：

```
...
> node -e "$(cat test.js)" -p
{ value: 100,
  writable: true,
  enumerable: true,
  configurable: false }
...
```



> NOTE: 模块封装器, 参见: http://nodejs.cn/api/modules.html#modules_the_module_wrapper



8



陆昱嘉

2019-11-17

老师, 一个赋值表达式的左边和右边其实“都是”表达式, 那么`var x=(var y=100);`这样就报错, 原因是什么? `varNames`里面的冲突?

作者回复: “`var y = 100`”不是表达式, 而是语句。

所以“`var x = y = 100`”里面:

- “`var x ...`”是语句语法 (Variable Statement)
- “`= ...`”是初始化器 (Initializer, 严格来说, 也不是表达式, 而是语法的一部分)
- “`y = 100`”是表达式 (expression)。

所以你列的问题, 报错的原因是“语句在语法上不支持这种写法”, 即在“`= ...`”中的“`...`”上面, 必须是一个用来做“初始器”的表达式, 而不能“再是一个语句”。

参见ECMAScript:

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

<https://tc39.es/ecma262/#prod-VariableDeclaration>



7



佳民

2019-11-13

思考题: `var`声明会声明提升, 在语法解析 (静态分析) 阶段进行, 不是在运行阶段执行, 这样理解对吗?

作者回复: 是的。不过所有的6种声明都是如此, 非独`var`声明。

共 2 条评论 >

7



Isaac

2020-06-28

「一个赋值表达式操作本身也是有“结果 (Result)”, 它是右操作数的值。注意, 这里是“值”而非“引用”」



老师，你好，这句话从“值类型”的角度可以理解，但是对于引用类型怎么理解？
比如：var x = y = { name: 'jack ma' }。

我的理解：

由于 { name: 'jack ma' } 本身是引用类型，所以 y = { name: 'jack ma' } 的赋值操作的结果也是“一个引用”，所以这里的“值”其实和类型无关，仅仅是一个运算结果。

在回到这句话：「它是右操作数的值」，用更通俗易懂话来讲，这里的“值”仅仅是一个运算结果，和类型无关。

请问老师我这样理解正确吗？如果错误的话，该怎么解释 var x = y = { name: 'jack ma' }？

作者回复：你说的“引用类型”，不是我说的“引用（规范类型）”。

以下面的代码为例：

```
...  
x = obj.foo  
...
```

右操作是`obj.foo`，它的引用是obj.foo整体，这包括“obj这个对象”的信息——这称为“引用（规范类型）”；而它的值是GetValue(obj.foo)，GetValue()是引擎的内部操作，是从“引用（规范类型）”中取值，其结果会是foo这个函数。

对于上述赋值表达式来说，`x = obj.foo` 其结果是`x`变成了函数foo，那么它就是右侧操作数的“值”，而不是右操作数“obj.foo”的全部信息。

有没有上述示例的反例呢？也就是一个运算符的结果仍然是“obj.foo的全部信息（引用）”，而不仅是它的“值（GetValue的结果）”呢？

有的。下面的示例：

```
...  
(obj.foo)  
...
```

这一对括号称为“分组运算符（也有称着强制运算符的）”，这个括号的运算结果就是“操作数的引用”。所以，在下一步的运算中：

```
...  
(obj.foo)()
```



...

这个方法调用中的foo函数可以得到this为obj。

最后汇总一下：

...

```
obj = {  
  foo() {  
    console.log(this === obj)  
  }  
}
```

// case 1, 赋值运算只得到了右侧运算数的“值”

```
x = obj.foo;  
x(); // false
```

// case 2, 分组运算得到了操作数的“引用（全部的信息）”

```
(obj.foo)(); // true  
...
```



6



Geek_baa4ad

2020-05-04

```
var x = y = 100;  
Object.getOwnPropertyDescriptor(global, 'x');  
Object.getOwnPropertyDescriptor(global, 'y');  
{value: 100, writable: true, enumerable: true, configurable: false}configurable: falseenum  
erable: truevalue: 100writable: true__proto__: Object  
Object.getOwnPropertyDescriptor(global, 'x');  
{value: 100, writable: true, enumerable: true, configurable: false}  
Object.getOwnPropertyDescriptor(global, 'y');  
{value: 100, writable: true, enumerable: true, configurable: false}
```

得到结果一样吖，x y 是一个相同的东西吧 最新的v8 实现不一样啦？

作者回复: 不要在浏览器中测试，也不要再node repl中测试。这些要么受环境的host/global设计的影响，要么受模块加载的影响。

得到纯v8引擎测试环境的方法，要么是直接编译一个v8，要么是使用下面这样方法：



```
...  
// 将下面的代码写文件test.js  
var x = y = 100;  
console.log(Object.getOwnPropertyDescriptor(global, 'x'));  
console.log(Object.getOwnPropertyDescriptor(global, 'y'));  
...  
  
然后:  
...  
  
# 在命令行上使用nodejs (在mac/linux环境)  
> node -e "$(cat test.js)"  
{ value: 100, writable: true, enumerable: true, configurable: false }  
{ value: 100, writable: true, enumerable: true, configurable: true }  
...  
  
好运。: )
```



👍 6



蓝配鸡

2019-11-13

醍醐灌顶，但是有一些疑问：

文中说，

"如果是在一门其它的（例如编译型的）语言中，“为变量 x 绑定一个初值”就可能实现为“在创建环境时将变量 x 指向一个特定的初始值”。这通常是静态语言的处理方法，然而，如前面说过的，JavaScript 是门动态的语言，所以它的“绑定初值”的行为是通过动态的执行过程来实现的，也就是赋值操作。”

为什么动态语言就不可以给变量初始化，一定要使用动态赋值呢？

我对动态语言的理解是，变量的类型可以在运行时改变，静态语言变量的类型不可以改变。但是这性质好像并不影响初始化？

作者回复: 动态语言的诸多细节，其实要到18讲之后才会讨论到。不过有个概念上的问题，并不是说有动态类型就是动态语言，或者说支持动态执行就是动态语言。有很多方面的“动态语言的特性”，这个需要详细解析。

仅是说初始化这一项，使用动态赋值的原因是因为这个值必须要到执行期环境中才能确定下来，而在静态语法分析阶段是确认不了的。——所以它不可能“先于环境”而执行。



共 2 条评论 >

👍 6

老师您好，在回过头来重新读这个课程的时候，我产生了一些新的疑惑。

在静态语法解析阶段，会在词法环境中添加所声明的标识符，那么像下面这样的代码：

```
var arr = new Array;  
for (var i=0; i<5; i++) arr.push(function f() {  
  // ...  
});
```

这段代码是在第八讲中粘贴过来的，第八讲中有说，静态函数f () 有且仅有一个。那么这个函数f是什么时候被定义的，又被定义在了什么样的词法环境下呢？

我上面的表述可能不明确，我大概就是想问这么一个问题：

```
let obj = {  
  test:function(cb){  
    cb();  
    (()=>{console.log(this);})();  
  }  
}  
obj.test(() => {console.log(this);})
```

() => {console.log(this)} 这个箭头函数，是在什么时候被定义的，定义在了哪里。

从 cb执行 this打印来看，应该是定义在了全局环境下。

但是由于它是一个匿名函数，所以我在全局无法打印出它来验证。但是我把

() => {console.log(this)}换成function f() {console.log(this)},全局下也没有办法访问到 f 。

我描述的可能不太清楚，我大概是想知道，被当做实参传入的函数，是在什么时候被声明的，声明在了哪里。

作者回复: 好问题!

其实你发现了一个有关于执行环境的、很深层面的问题。简单地表述来说是这样：如果具名函数把自己的名字登记在“函数所在的上下文（环境）”中，那么匿名函数如何登记自己呢？——对于这个问题，更深一点的问下去，就会是“如果匿名函数不能登记，那么它怎么执行呢”？

其实，整个问题的核心关键在于：这根本不是匿名/具名函数的问题，而是函数声明/函数表达式的问题。注意以下的讨论中要特别强调的是：你所有几个示例中声明的，都是“具名的函数表达式”，而不是“具名函数（声明语句）”。

好的。真实的情况是这样：1、函数表达式“不会”向当前的环境中登记自己的名字；2、你无法声明出一个匿名函数的语句（这有一个例外，就是export default ...）。



先说第1条。“（所有的）函数表达式”其实都是不会向当前环境中声明的，只是在表达式执行到的时候，这个函数才会被创建。而当它被创建时，它立即创建一个与这个函数相关的上下文，而由于它是“函数表达式”，所以不会“变动（当前的）”环境——即不改变词法的环境，也不改变变量的环境。

然后你应该会注意到这个问题，就是“具名的函数表达式”也是有名字的，对吧。是的，但是这个名字“并不注册在当前的上下文环境中”。在函数表达式的实现中，这是一个非常少有人注意到的技巧，称为“多重的环境（或这里可以称为闭包）”，也就是说这样的函数其实有两个闭包，一个外层的，登记了函数名字；一个内层的，登记了参数表等等。这样一来，在“具名的函数表达式”中，既可以让函数体内的代码用到“函数名”（包括覆盖它），又不需要这个名字被“注册到”外部的（例如当前的）环境中。

关于这一部分，可以参阅《JavaScript语言精髓与编程实践》的第“5.5.2.4 函数表达式的特殊性”一节。

第2个问题。这是一个细节，因为export是语句，因此如果它用来声明一个匿名函数，那么这个匿名函数也将是“语句声明出来的”，所以它是特例。并且在真实的情况中，这样的“匿名函数”其实也是有名字的，它被登记在一个名为“*default*”的项中。这个在本课程的第4讲中也是讲述过了。

最后，你的问题其实还涉及“函数表达式在当前作用域中到底‘需要/不需要’有名字”的问题，以及所谓的“条件化声明（语句）”的问题，等等。这些是古老时代不同js引擎的实现带来的、与规范有差异的遗产，例如JScript 5.6及之前的版本就认为“具名的函数表达需要在当前作用域中声明一个名字”（这也称为名字泄露）。不过，这些问题中绝大多数如今已经有了定论，你可以再翻翻文章。

共 3 条评论 >

👍 5



卡尔

2020-06-11

老师，你说let声明的变量不能在赋值之前使用。

这里说的赋值是不是说赋值操作呢？

```
let a;
```

```
console.log(a)
```

上面代码对a是没有赋值操作吧？

作者回复: `let a;`

是作为`let a = undefined;`处理的。

声明语句确实是在语法分析期和环境初始化阶段处理的，但当代码“执行到”相应位置时，会执行它在“运行期语义”——也就是“绑定值”。

而如上说它是作为“let a = undefined”来处理的，因此这一行代码不是“没有赋值操作”，而是“执行



了初值绑定操作”。

“绑定初值”与“赋值”是语法效果上是一致的，只是概念上的不同。

共 2 条评论 >

👍 5



家家家

2019-12-02

忘了在哪本书中还是哪篇文章中讲过，变量的生命周期：声明阶段、初始化阶段、赋值阶段。老师这里讲的静态分析阶段就是指的变量生命周期中的声明阶段对吗？

对于var，它的声明阶段和初始化阶段是一起发生的，都在静态分析中；对于let，它的声明阶段和初始化阶段是分开的，只有声明阶段在静态分析中，是这样理解的嘛？

作者回复: 这个讲法倒也不错，比较容易理解。

不过有些不严谨的地方，比如说，“var x;”这样的声明，“声明阶段和初始化阶段一齐发生”是对的，因为var声明的时候，名字声明并初始化为undefined确实是同时发生的。但是，如果是“var x = 100;”，那么就比较容易混淆了。因为“x = 100”其实是赋值，而不是引擎层面的“初始化”。

对于let来说，“声明阶段和初始化阶段是分开”其实更不严谨。确切地说，let就没有“初始化阶段”。而用户代码“let x = 100”中的“x = 100”就是赋值阶段了。“let 没有初始化阶段”，所以才会出现它在未赋值之前不能读的现象。

用这样三个阶段来解释这件事情，跟ECMAScript中的逻辑并不矛盾（而且也可以正确解释），只是细节上需要严谨一点、留意一点就行。

共 2 条评论 >

👍 5



Marvin

2019-11-14

相当于

var/let/const x = (y = 100)

再拆就是

y=100 // 变量泄漏

var x=y // 模拟表达式返回值赋值

作者回复: 是的。简洁，正确。^^.



👍 5



墨灵

2020-03-17

以前我把

...

```
console.log(x);
```

```
var x = 100;
```

...

理解成

...

```
var x;
```

```
console.log(x);
```

```
x = 100;
```

...

看来以前的理解是有误的，是更为底层的东西在起作用。

作者回复: 我很长的时间里也一直是以来这种方法来理解的，效果上也确实类似。但在语言层面，底层的设计与实现确非如此。^^。



4



Smallfly

2020-02-24

$y = 100$ 有的地方叫赋值语句，有的地方叫赋值表达式。因为它的执行结构能在代码层面获得 $x = (y = 10)$ ，所以我更倾向于认为它是表达式。

想请问下老师叫赋值语句是错误的么，还是有其它的原因？

作者回复: 在ECMAScript中，并没有“赋值语句”，它的准确说法是“赋值表达式语句”。——任何一般表达式，都可以解析为一个语句，并称为“一般表达式语句”。所以“赋值表达式（语句）”在这里并没有任何的特殊性。

“任何一般表达式”都可以通过在末尾添加一个“;”号来表明将它处理为一个语句（包括还有换行符，以及文末符等等，这些参考“自动分号插入(ASI)”这个语法特性）。

所以这里不存在是否“错误”的问题。很多地方、很多书都只是按传统的语言习惯来理解JavaScript，所以用传统的名字或概念往上面套。进一步的，也就有了很多似是而非的概念。如果真正的追求概念上的准确性，那么应该使用ECMAScript规范中的明确定义。——但是如果这样讲，那么这个课程中基础也还有稍稍有一些地方有着含混不清的概念的。

JavaScript中需要分开理解表达式和语句，它们处理机制不同，结果不同，效果也不同，应用环境还是不同。这些的入手点称为“表达式语言”，也就是所谓“函数式语言特性”。你可以尝试着将所有的语句去掉，试着用“纯粹地JavaScript表达式”来写代码，你会发现JS在这个层面上也是完备的。只有先抽离出了“表达式”，以及“表达式语言”，再反观“语句”，才能理解语句真正的用处。总而言之，语句



与表达式在JavaScript的语言设计中是很精彩而又令人迷惑的。

这个课程的后面一部分会分开讲“语句执行”和“表达式执行”，慢慢看就会明白这些东西之间的区别了。



👍 4



授人以摸鱼

2019-11-22

发现我好像对全局作用域的理解有一些偏差：

var，或者没有声明直接赋值，这样的创建标识符（引用）是作为global对象的字段存在的，可以用Object.getOwnPropertyDescriptor从global上读到。

全局作用域里用let，const创建的变量，虽然也是全局可见，但它并没有创建在global上，而是创建在了另一个地方。从作用域链的视角来看的话，这个作用域要比global低一级这样子。

作者回复: 你是对的。在JavaScript中，“全局环境”里面的var与let/const是用了两个东西来管理的，所以他们也确实是创建在不同的地方。

但是从“作用域链”的角度上来说，它们并没有级别高低（也就是parent没有相互指向）。使得它们存取的效果有差别的，是因为“全局环境”采用了词法环境优先（也就是let/const声明）的顺序。

共 2 条评论 >

👍 4

