

## 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

2019-11-15 周爱民

JavaScript核心原理解析

[进入课程 >](#)



讲述：周爱民

时长 25:23 大小 23.26M



你好，我是周爱民。


在前端的历史中，有很多人都曾经因为同一道面试题而彻夜不眠。这道题出现在 9 年之前，它的提出者“蔡 mc（蔡美纯）”曾是 JQuery 的提交者之一，如今已经隐去多年，不复现身于前端。然而这道经典面试题仍然多年长挂于各大论坛，被众多后来者一遍又一遍地分析。

在 2010 年 10 月，[@Snandy](#)于 iteye/cnblogs 上发起对这个话题的讨论之后，淘宝的玉伯（lifesinger）也随即成为这个问题早期的讨论者之一，并写了一篇“ $a.x = a = \{ \}$ ”，深入

**理解赋值表达式**”来专门讨论它。再后来，随着它在各种面试题集中频繁出现，这个问题也就顺利登上了知乎，成为一桩很有历史的悬案。

蔡 mc 最初提出这个问题时用的标题是 **“赋值运算符:”=”, 写了 10 年 javascript 未必全了解的”=“”**，原本的示例代码如下：

```
1 var c = {};  
2 c.a = c = [];  
3 alert(c.a); //c.a 是什么?
```

 复制代码


蔡 mc 是在阅读 JQuery 代码的过程中发现了这一使用模式：

```
1 elemData = {}  
2 ...  
3 elemData.events = elemData = function(){};  
4 elemData.events = {};
```

 复制代码

并质疑，**为什么elemData.events需要连续两次赋值**。而 Snandy 在转述的时候，换了一个更经典和更有迷惑性的示例：

```
1 var a = {n:1};  
2 a.x = a = {n:2};  
3 alert(a.x); // --> undefined
```

 复制代码

Okay，这就是今天的主题。


接下来，我就为你解释一下，**为什么在第二行代码之后a.x成了 undefined 值**。

## 与声明语句的不同之处

你可能会想，三行代码中出问题的，为什么不是第 1 行代码？

在上一讲的讨论中，声明语句也可以是一个连等式，例如：

```
1 var x = y = 100;
```

 复制代码

在这个示例中，“var”关键字所声明的，事实上有且仅有“x”一个变量名。

在可能的情况下，变量“y”会因为赋值操作而导致 JavaScript 引擎“**意外**”创建一个全局变量。所以，声明语句“var/let/const”的一个关键点在于：语句的关键字 var/let/const 只是用来“声明”变量名 x 的，去除掉“var x”之后剩下的部分，并不是一个严格意义上的“赋值运算”，而是被称为“初始器（Initializer）”的语法组件，它的词法描述为：


Initializer: =AssignmentExpression

在这个描述中，“=”号并不是运算符，而是一个语法分隔符号。所以，之前我在讲述这个部分的时候，总是强调它“被实现为一个赋值操作”，而不是直接说“它是一个赋值操作”，原因就在这里。

如果说在语法“var x = 100”中，“= 100”是向 x 绑定值，那么“var x”就是单纯的标识符声明。这意味着非常重要的一点——**“x”只是一个表达名字的、静态语法分析期作为标识符来理解的字面文本，而不是一个表达式。**

而当我们从相同的代码中去除掉“var”关键字之后：

```
1 x = y = 100;
```

 复制代码

其中的“x”却是一个表达式了，它被严格地称为“赋值表达式的左手端（lhs）操作数”。

所以，关键的区别在于：（赋值表达式左侧的）操作数可以是另一个表达式——这在专栏的第一讲里就讲过了，而“var 声明”语句中的等号左边，绝不可能是一个表达式！


也许你会质疑：难道 ECMAScript 6 之后的模板赋值的左侧，也不是表达式？确实，答案是：如果它用在声明语句中，那么就“不是”。

对于声明语句来说，紧随于“var/let/const”之后的，一定是变量名（标识符），且无论是一个或多个，都是在 JavaScript 语法分析阶段必须能够识别的。

如果这里是赋值模板，那么“var/let/const”语句也事实上只会解析那些用来声明的变量名，并在运行期使用“初始器（Initializer）”来为这些名字绑定值。这样，“变量声明语句”的语义才是确定的，不至于与赋值行为混淆在一起。


因此，根本上来说，在“var 声明”语法中，变量名位置上就是写不成`a.x`的。例如：

```
1 var a.x = ... // <- 这里将导致语法出错
```

 复制代码

所以，在最初蔡 mc 提出这个问题时，以及其后 Sanady 和玉伯的转述中，都不约而同地在代码中绕过了第一行的声明，而将问题指向了第二行的连续赋值运算。

```
1 var a = {n:1}; // 第一行
2 a.x = a = {n:2}; // 第二行
3 ...
```

 复制代码

## 来自《JavaScript 权威指南》的解释

有人曾经引述《JavaScript 权威指南》中的一段文字（4.7.7 运算顺序），来解释第二行的执行过程：

JavaScript 总是严格按照从左至右的顺序来计算表达式。

并且还举了一个例子：

例如，在表达式`w = x + y * z`中，将首先计算子表达式`w`，然后计算`x`、`y`和`z`；然后，`y`的值和`z`的值相乘，再加上`x`的值；最后将其赋值给表达式`w`所指代的变量或属

性。


《JavaScript 权威指南》的解释是没有问题的。首先，在这个赋值表达式的右侧  $x + y * z$  中， $x$  与  $y * z$  是求和运算的两个操作数，任何运算的操作数都是严格从左至右计算的，因此  $x$  先被处理，然后才会尝试对  $y$  和  $z$  求乘积。这里所谓的“ $x$  先被处理”是 JavaScript 中的一个特异现象，即：

一切都是表达式，一切都是运算。

这一现象在语言中是函数式的特性，类似“一切被操作的对象都是函数求值的结果，一切操作都是函数”。

这对于以过程式的，或编译型语言为基础的学习者来说是很难理解的，因为在这些传统的模式或语言范型中，所谓“标识符 / 变量”就是一个计算对象，它可能直接表达为某个内存地址、指针，或者是一个编译器处理的东西。对于程序员来说，将这个变量直接理解为“操作对象”就可以了，没有别的、附加的知识概念。例如：

```
1 a = 100
2 b * c
```

 复制代码

这两个例子中， $a$ 、 $b$ 、 $c$  都是确定的操作数，我们只需要

将第一行理解为“ $a$  有了值 100”；

将第二行理解为“ $b$  与  $c$  的乘积”

就可以了，至于引擎怎么处理这三个变量，我们是不管的。

然而在 JavaScript 中，上面一共有六个操作的。以第二行为例，包括：

将  $b$  理解为单值表达式，求值并得到 `GetValue(evaluate('b'))`；

将  $c$  理解为单值表达式，求值并得到 `GetValue(evaluate('c'))`；

将上述两个值理解为求积表达式 `*` 的两个操作数，计算

```
evaluate('*', GetValue(evaluate('b')), GetValue(evaluate('c')))
```

所以，关键在于b和c在表达式计算过程中都并不简单的是“一个变量”，而是“一个单值表达式的计算结果”。这意味着，在面对 JavaScript 这样的语言时，你需要关注“变量作为表达式是什么，以及这样的表达式如何求值（以得到变量）”。

那么，现在再比较一下今天这一讲和上一讲的示例：

 复制代码

```
1 var x = y = 100;
2 a.x = a = {n:2}
```

在这两个例子中，

x 是一个标识符（不是表达式），而 y 和 100 都是表达式，且  $y = 100$  是一个赋值表达式。


a.x 是一个表达式，而  $a = \{n:2\}$  也是表达式，并且后者的每一个操作数（本质上）也都是表达式。

这就是“语句与表达式”的不同。正如上一讲的所强调的：“**var x**”从来都不进行计算求值，所以也就不能写成“**var a.x ...**”。

所以严格地说，在上一讲的例子中，并不存在连续赋值运算，因为“var x = ...”是**值绑定操作**，而不是“将...赋值给 x”。在代码 `var x = y = 100;` 中实际只存在一个赋值运算，那就是“ $y = 100$ ”。

## 两个连续赋值的表达式

所以，今天标题中的这行代码，是真正的、两个连续赋值的表达式：

 复制代码

```
1 a.x = a = {n:2}
```

并且，按照之前的理解，`a.x`总是最先被计算求值的（从左至右）。

回顾第一讲的内容，你也应该记得，所谓“`a.x`”也是一个表达式，其结果是一个“引用”。这个表达式“`a.x`”本身也要再计算它的左操作数，也就是“`a`”。完整地讲，“`a.x`”这个表达式的语义是：

计算单值表达式`a`，得到`a`的引用；


将右侧的名字`x`理解为一个标识符，并作为“.”运算的右操作数；

计算“`a.x`”表达式的结果（Result）。

表达式“`a.x`”的计算结果是一个引用，因此通过这个引用保存了一些计算过程中的信息——例如它保存了“`a`”这个对象，以备后续操作中“可能会”作为`this`来使用。所以现在，在整行代码的前三个表达式计算过程中，“`a`”是作为一个**引用**被暂存下来了。


那么这个“`a`”现在是什么呢？

```
1 var a = {n:1};  
2 a.x = ...
```

 复制代码

从代码中可见，保存在“`a.x`”这个引用中的“`a`”是当前的“`{n:1}`”这个对象。好的，接下来再继续往下执行：

```
1 var a = {n:1};  
2 a.x =      // <- `a` is {n:1}  
3     a =    // <- `a` is {n:1}  
4 ...
```

 复制代码

这里的“`a = ...`”中的`a`仍然是当前环境中的变量，与上一次暂存的值是相同的。这里仍然没有问题。

但接下来，发生了赋值：



```

1  ...
2  a.x =           // <- `a` is {n:1}
3      a =         // <- `a` is {n:1}
4      {n:2};      // 赋值，覆盖当前的左操作数（变量`a`）

```

于是，左操作数`a`作为一个引用被覆盖了，这个引用仍然是当前上下文中的那个变量`a`。因此，这里真实地发生了一次`a = {n:2}`。

那么现在，表达式最开始被保留在“一个结果值（Result）”中的引用`a`会更新吗？

不会的。这是因为那是一个“**运算结果**（Result）”，这个结果有且仅有引擎知道，它现在是一个引擎才理解的“**引用**（规范对象）”，对于它的可能操作只有：

取值或置值（GetValue/PutValue），以及

作为一个引用向别的地方传递等。

当然，如同第一讲里强调的，它也可以被 `typeof` 和 `delete` 等操作引用的运算来操作。但无论如何，在 JavaScript 用户代码层面，能做的主要还是**取值**和**置值**。

现在，在整个语句行的最左侧“**空悬**”了一个已经求值过的“`a.x`”。当它作为赋值表达式的左操作数时，它是一个被赋值的引用（这里是指将`a.x`的整体作为一个引用规范对象）。而它作为结果（Result）所保留的“`a`”，是在被第一次赋值操作覆盖之前的、那个“原始的变量`a`”。也就是说，如果你试图访问它的“`a.n`”，那应该是值“1”。

这个被赋值的引用“`a.x`”其实是一个未创建的属性，赋值操作将使得那个“原始的变量`a`”具有一个新属性，于是它变成了下面这样：

```

1  // a.x 中的“原始的变量`a`”
2  {
3    x: {n: 2}, // <- 第一次赋值“a = {n:2}”的结果值
4    n: 1
5  }

```



这就是第二次赋值操作的结果。

## 复现现场

上面发生了两次赋值，第一次赋值发生于“`a = {n: 2}`”，它覆盖了“原始的变量`a`”；第二次赋值发生于被“`a.x`”引用暂存的“原始的变量`a`”。

我可以给出一段简单的代码，来复现这个现场，以便你看清这个结果。例如：

 复制代码

```
1 // 声明“原始的变量 a”
2 var a = {n:1};
3
4 // 使它的属性表冻结（不能再添加属性）
5 Object.freeze(a);
6
7 try {
8     // 本节的示例代码
9     a.x = a = {n:2};
10 }
11 catch (x) {
12     // 异常发生，说明第二次赋值“a.x = ...”中操作的`a`正是原始的变量 a
13     console.log('第二次赋值导致异常. ');
14 }
15
16 // 第一次赋值是成功的
17 console.log(a.n); //
```

第二次赋值操作中，将尝试向“原始的变量`a`”添加一个属性“`a.x`”，且如果它没有冻结的话，属性“`a.x`”会指向第一次赋值的结果。

## 回到标题中的示例

那标题中的这行代码的最终结果是什么呢？答案是：

有一个新的`a`产生，它覆盖了原始的变量`a`，它的值是`{n:2}`；


最左侧的“`a.x`”的计算结果中的“原始的变量`a`”在引用传递的过程中丢失了，且“`a.x`”被同时丢弃。

所以，第二次赋值操作 “a.x = ...” 实际是无意义的。因为它所操作的对象，也就是 “原始的变量a” 被废弃了。但是，如果有其它的东西，如变量、属性或者闭包等，持有了这个 “原始的变量a” ，那么上面的代码的影响仍然是可见的。

事实上，由于 JavaScript 中支持属性读写器，因此向 “a.x” 置值的行为总是可能存在 “某种执行效果” ，而与 “a” 对象是否被覆盖或丢弃无关。

例如：

```
1 var a = {n:1}, ref = a;
2 a.x = a = {n:2};
3 console.log(a.x); // --> undefined
4 console.log(ref.x); // {n:2}
```

 复制代码

这也解释了最初 “蔡 mc” 的疑问：连续两次赋值 `elemData.events` 有什么用？


如果a（或`elemData`）总是被重写的旧的变量，那么如下代码：

```
1 a.x = a = {n:2}
```

 复制代码

意味着给**旧的变量**添加一个**指向新变量的属性**。因此，一个链表是可以像下面这样来创建的：

```
1 var i = 10, root = {index: "NONE"}, node = root;
2
3 // 创建链表
4 while (i > 0) {
5     node.next = node = new Object;
6     node.index = i--; // 这里可以开始给新 node 添加成员
7 }
8
9 // 测试
10 node = root;
11 while (node = node.next) {
12     console.log(node.index);
13 }
```

 复制代码

最后，我做这道面试题做一点点细节上的补充：

这道面试题与运算符优先级无关；

这里的运算过程与“栈”操作无关；

这里的“引用”与传统语言中的“指针”没有可比性；

这里没有变量泄露；

这行代码与上一讲的例子有本质的不同；

上一讲的例子“`var x = y = 100`”严格说来并不是连续赋值。

## 知识回顾

前三讲中，我通过对几行特殊代码的分析，希望能帮助你理解“引用（规范类型）”在 JavaScript 引擎内部的基本运作原理，包括：

引用在语言中出现的历史；

引用与值的创建与使用，以及它的销毁（delete）；

表达式（求值）和引用之间的关系；

引用如何在表达式连续运算中传递计算过程的信息；

仔细观察每一个表达式（及其操作数）计算的顺序；

所有声明，以及声明语句的共性。

## 复习题

下面有几道复习题，希望你尝试解答一下：

1. 试解析 `with ({x:100}) delete x;` 将发生什么。
2. 试说明 `(eval)()` 与 `(0, eval)()` 的不同。
3. 设“`a.x === 0`”，试说明“`(a.x) = 1`”为什么可行。

4. 为什么`with (obj={}) x = 100;` 不会给 `obj` 添加一个属性 `'x'` ?

如果你喜欢我的分享，也欢迎你把文章分享给你的朋友。

小程序学习打卡邀请

打卡 46 天，彻底搞定  
JavaScript

  
扫一扫参与小程序打卡



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

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

下一篇 04 | `export default function() {}`: 你无法导出一个匿名函数表达式

## 精选留言 (13)

 写留言



**blackhole**

2019-11-15

从内容上其实已经说清楚了，不过在内容表达上还是会让人产生困惑，我觉得问题是出在“当前上下文中的那个变量a”和“原始的变量a”这样的表述方式上。或许如下表述在语意上会更加清晰：

1, 这里其实只有一个变量，就是a，不存在那个变量a和这个变量a之分，有分别的其实...  
展开 ▾

作者回复: 赞的！就是这个意思。呵呵~



6



**Summer**

2019-11-15

有的地方描述有点晕，看了好几遍才明白表述的意思，要是有一些动态的图演示的话可能效果更好点

作者回复: 这个.....确实实现起来有难度。我通常在做讲演稿的时候才会用这种方式，但讲演稿的讲法，跟这里的课程的讲课方法区别还是很大的。

当然，即使不用动态的图，使用流程图或框线图其实也挺好的。不过，总之，以极客时间的“语音课程”来说，很难讲。——话说回来，如果是需要更深的阅读，以及更丰富的图例，以及表格等表现形式，那么可以看我的书哦。《JavaScript语言精髓与编程实践》这本书的第三版.....快要出版了啦~ ^^.



2



**sprinty**

2019-11-15

老是您好：我理解的指针和引用是，指针是存储的地址，引用是存储的别名。

在js 中的“引用”与传统语言中的“指针”有哪些根本性的区别。

展开 ▾

作者回复: 其实我早期也是这么理解的。好象大家理解事物的方式都差不多，就是从相似性出发，从差异性辨别。

但是我后来发现，与其如此，不如为新东西建个体系，然后在新体系中来看待这个新事物。这一下子就不同了。

以至于我现在对引用的认识，就不太依赖与比较或比拟。引用就是引用，它就是一个计算的结果，它存放结果中包括的那几个东西。它是一个数据结构，用在引擎层面来存储计算过程的中间信息，以及在连续计算中传递这些信息。



2



**青史成灰**

2019-11-16

老师上面引用《JavaScript权威指南》中说“JavaScript总是严格按照从左到右的顺序计算表达式”，那为什么下文的2次赋值操作`a.x = a = {n:2}`，是先赋值`a={n:2}`，然后才是`a.x = a`呢

作者回复: 这个顺序是这样来读的（你仔细看看顺序是不是从左至右）：

第一次

=====

a.x = a = {n:2}

^1 ^2

第二次

=====

a = { n: 2 }

^3 ^4

第三次

=====

{ n: 2 }

^5 ^6

第四次（以下求值然后回传）

=====

求值传回(4)

@4 <= ^5, ^6

第五次

=====

求值回传(3)

@3 = (^4 <= ^5, ^6)

第六次

=====

求值回传 (2)

a = @3 = (^4 <= ^5, ^6)

第七次

=====

求值回传 (1)

a.x = a = @3 = ...



1

**Lambert**

2019-11-15

“a.x” 这个表达式的语义是：  
计算单值表达式a，得到a的引用；  
将右侧的名字x理解为一个标识符，并作为 “.” 运算的右操作数；  
计算 “a.x” 表达式的结果（Result）。  
老师请问一下 这个时候 的 Result 是 undefined吗？ 因为还没有进行赋值  
展开 ∨

作者回复: 这个时候的Result是一个 “引用（Reference）” 。

如果它在后续运算中被作为lhs，例如 `a.x = ...`，那么它就是作为 “引用” 来使用，这样就可以访问到 `x` 这个属性，并置值；如果它在后续运算中被作为rhs，例如 `console.log(a.x)`，那么它就会被 `GetValue()` 取值（并作为值来使用），于是 `console.log()` 就能打印出它的值来。

`a.x` 整体被作为 “一个操作数”，它的用法与它被使用的位置是有关的。但是 “得到它（亦即是对 `a.x` 这个表达式求Result）” 的过程并没有什么不同。

你可以读一下这个 “.” 操作在ECMAScript中的规范：

<https://tc39.es/ecma262/#sec-property-accessors-runtime-semantics-evaluation>

2

1

**早起不吃虫**

2019-11-15

作为一名前端，看的一头雾水。。。  
展开 ∨



1

**天方夜**

2019-11-18

1. `with ({x:100}) delete x` 中 `delete` 删除的是对象的成员，即 `property x`；
  2. `(0, eval)` 之中有了一步逗号运算；
  3. 表达式 `(a.x)` 的计算结果是 `a` 对象的属性 `x` 这个引用，所以可行；
  4. `with` 只指定属性查找的优先级，所以 `with` 里面 `x = 100` 还是会泄漏到全局。
- 展开 ∨







铭

2019-11-16

反反复复看了几遍，留言区里帮我厘清了思路。

第一句：

```
var a = {n : 1};
```

```
// 变量声明，变量a作为引用，最终指向了等号右侧表达式的计算结果，即一个对象{n : ...
```

展开 ▾

作者回复: 除了“a.x = a”导致栈异常之外，这个好象不太对。其它应该没什么问题了。



旺旺

2019-11-15

JavaScript果然太灵活，然后感觉好难啊

展开 ▾



许童童

2019-11-15

1. delete返回true 可以删除

2.会先计算表达式(0, eval)

3.没懂

4.obj={} 是一个表达式，返回的是{}这个值，所以with还是在全局上下文，x会被赋值为window的属性

展开 ▾



许童童

2019-11-15

老师讲得真细啊，学到了很多，谢谢老师。

展开 ▾



Wiggle Wiggle

2019-11-15

那么“引用”这个数据结构究竟是什么样子呢？在引擎内部是如何实现的呢？老师可否讲一下或者给个链接？

作者回复: <https://tc39.es/ecma262/#sec-reference-specification-type>

^^.



**Smallfly**

2019-11-15

文章读起来挺吃力的，可能是 JS 很多设计跟固有思维不一致，也可能是对 ECMAScript 规范不了解，老师能否考虑下放文章中涉及到的规范地址？

展开 ▾

作者回复: 好主意！我问问编辑能怎么改。

后面的内容我尽量都加上。多谢提议！

