

## 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》

课程介绍 >



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?
```

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 = {};
```

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、更有迷惑性的示例：

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

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;
```



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**异常现象**，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，上面一共有是有六个操作的。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：

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

 复制代码



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：

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

 复制代码

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？

 复制代码

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

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：

 复制代码

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

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：

 复制代码

```
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（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：

 复制代码

```
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
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；
- 这里没有变量泄漏；
- 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。

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 5  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 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



# JVM + NIO + Spring

各大厂面试题及知识点详解

限时免费



## 精选留言 (56)

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，有分别的其实是变量a的值，即“变量a过去的值”和“变量a现在的值”。
- 2, 当发生第一次赋值时，“左操作数a作为一个引用被覆盖”，此时变量a产生了新的值。
- 3, 第二次赋值时，“整个语句行的最左侧‘空悬’了一个已经求值过的‘a.x’”，这是一个表达式结果，这个结果以及其中保留的“a”（即“变量a过去的值”）与变量a已经没有关系了，因为变量a已经有了新的值，即“变量a现在的值”。
- 4, 第二次赋值其实是，在“变量a过去的值”那个对象上，创建一个新属性x，x的值为变量a的值，即“变量a现在的值”。
- 5, 在第二次赋值后，因为“变量a过去的值”那个对象已经不再被任何变量持有，所以它已经无法被访问到了，它“跑丢了”。



是这样吧？

作者回复: 赞的！就是这个意思。呵呵~

共 11 条评论 >

👍 91



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}`

<sup>1</sup> <sup>2</sup>

第二次

=====

`a = { n: 2 }`

<sup>3</sup> <sup>4</sup>

第三次

=====

`{ n: 2 }`

<sup>5</sup> <sup>6</sup>

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)

$a = @3 = (^4 \leq ^5, ^6)$

第七次

=====

求值回传 (1)

$a.x = a = @3 = \dots$

共 3 条评论 >

👍 12



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。

作者回复: 第2个不太完整。不过总体满分 😊

第二个涉及的问题到20讲才开讲呢^\_^

共 2 条评论 >

👍 11



新哥

2020-06-14

画个图最好说明问题了，a和ref 指向同一块内存地址，保存的数据是{n:1};

执行第二行的时候，a下移指向新的内存地址，保存的数据是{n:2};

且第一块内存空间 添加新的属性x,因为ref.x被赋值a,所以ref.x指向新的刚添加的那个地址，数据为{n:2};

这样ref指向原始的内存地址，a指向新的内存地址；

作者回复: 是的。谢谢~ ^^.



👍 9



weineel

2019-11-15

老您好: 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。



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。

共 3 条评论 >

👍 9



🇧🇪 Hazard🔗...

2020-04-15

老师你好, 我有一些关于词法环境规范的疑问, 可能跟这一讲的内容有点出入, 希望能得到您的解答。

1. 环境记录规范有 5 种, 但是我没有找到什么资料去告诉我, 什么声明会把标识符binding到具体哪个EnvironmentRecord中; 还有就是全局变量会放在哪里?

2. ECMAScript中关于环境记录与标识符喜欢用 binding 这个词, 我不知道是什么意思? 这个变量是存储在环境记录规范中的吗? 还是存储在别的地方? 在执行上下文的结构中有一个叫 Realms 的东西, 不知道是不是跟这个有关。

3. EnvironmentRecord的内部结构其实是怎样的? 感觉听到了很多术语, 但还是感觉很抽象。

我现在看到了第9讲, 发现越来越有点看不懂, 于是从头开始学, 希望能得到老师的解答, 如果解答起来比较复杂, 能否提供一些其他资料链接。谢谢!

作者回复: 在去年的D2上, 我专门讲过一讲《JS 语言在引擎级别的执行过程》, 对你提到的问题大都有涉及。并且, 有丰富的图示讲解。所以我建议你先听听视频, 或者你的许多问题就有解了。

在这里:

[https://v.youku.com/v\\_show/id\\_XNDUwNTc3MjUzMg==.html](https://v.youku.com/v_show/id_XNDUwNTc3MjUzMg==.html)

PPT在这里找:

<https://github.com/d2forum/14th/tree/master/PPT>

还有文字版, 在“2020前端工程师必读手册”里面有收录。你搜搜~





红白十万一只

2020-02-25

老师这题我看过别的文章，不过是与运算符优先级解释。

按照运算符优先级的思路：

```
var a={n1}
```

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

=的关联性是从右到左，优先级是3，赋值运算符的返回结果是右边的值

.(成员访问)的关联性是从右到左，优先级是19

a.x的赋值等于a={n:1}，而a的赋值等于{n:1}。

按照顺序会先计算a={n:1}的值，但是a.x是成员访问优先级是19。

所以会先进行a.x的解析，解析结果就是变量a对象的引用(引用地址#001)并创建了a.x这个属性，引用被暂存。

这是表达式就是：#001.x=a={n:1}

a={n:1}时修改了变量a(例#001)的引用地址为{n:1}(例#002)。

表达式就是#001.x={n:1}(例#002)

也就是#001这个引用地址中x的值被修改为了{n:1}

#001这个引用地址的值也就是

```
{
```

```
  n:1,
```

```
  x:{n:1}
```

```
}
```

但是这个引用已经没有任何变量、属性持有了

而变量a的值就是

```
{n:1}
```

关于这种解释有没有什么问题，麻烦老师解释一下。

作者回复: 这种解释是对的。并且跟这一讲的解释是同义的。只是由于两个解释的侧重点不同，所以貌似有不同而已。

这个解释中也引入了一个#001来说明，这个在本讲中被称为“原始的a”，又或者说是“原始的a的一个引用”。其实都是相同的意思，你按照这种关联来对照着看，就明白了。但是本讲侧重于说明表达式和引用，所以是更强调基于“引用（规范类型）”的解释过程。

我刻意没有讨论优先级的问题。在课后的留言评论中提到过按优先级来演算的过程，但也不如你这里的细致。优先级是运算规则的很重要的组成部分，在设计表达式语法时也很重要，但是我们的课程并不特别关注这个部分，所以我是有意不从这个角度入手来讲的。







蓝配鸡  
2019-11-18

不明白为什么a.x 这个表达式的result是一个a的引用呢？

不应该是 undefined吗？

没明白...

作者回复: Result是引用。

value是undefined。

value = GetValue(Result)

共 3 条评论 >



7



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

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>

共 3 条评论 >



7



HoSalt

2020-05-08

老师 (test.fn)()和test.fn()的调用this都只想test，为什么前面的括号里面的内容没有返回值而是返回了引用



作者回复: 一对括号, 亦即是所谓分组表达式`()`, 这个东西是在JS中极其罕见的在执行中“返回结果(result)”的表达式。因为通常的表达式是“返回值(value)”的, 这甚至包括返回所谓ECMAScript规范引用, 这也是作为value来返回的。——另一个如此有趣的东西是表达式作为函数体的箭头函数。

在分组表达式中, “返回结果(result)”而不是“返回值(value)”其实是有着非常大的、非常有魔力的不同的。例如说:

```
> (test.fn)
```

在这个表达式里面, `test.fn`的Result是一个ECMAScript规范中的引用, 因此这个引用就被返回了, 因此`(test.fn)()`这个函数调用中, `fn()`就能得到this。也因此 `(eval)`与直接的eval没有区别, 都是eval引用。

但是你注意看,

```
> (test.fn)
```

分组表达式操作的“里面的表达式`test.fn`”是一个属性存取表达式, 它返回“ECMAScript规范的引用”。而下面:

```
> (0, test.fn)
```

代码中在“里面的表达式`0, test.fn`”是什么呢? 是用`,`号分隔开的所谓的“连续运算表达式”, 而这个连续运算表达式的第二个子表达式, 才是`test.fn`, 对吧?

连续运算表达式返回什么呢? 很不幸, 连续运算表达式返回“最后一个子表达式的值 (value)”——我们前面说过, 所有表达式中目前只有两个是直接返回Result的。其它的情况下, 其实都会返回value, 包括以value定义的规范类型, 或者GetValue(result)。

所以说, 事实上

```
> (0, test.fn)
```

表达式的返回的是连续运算的最后一个表达式的value, 也就是test.fn的getValue(result), 也就是fn这个函数。因此, 再调用

```
> (0, test.fn)()
```

的时候, 就丢失掉了test这个对象引用了。





授人以摸鱼

2019-11-24

所以我现在这么理解js中的“值”和“引用”这两个概念了：

“引用”保存了两个信息：对象的地址，和要查询的属性名(字符串或symbol)

“值”只保存了一个信息：原始值本身，或一个地址

从引用中获取值这个操作是惰性的，只有真正要使用值的时候才会执行getvalue

作者回复: 是的。都对！赞！



6



GitHubGanKai

2020-01-11

老师你好，有个问题想要请教一下你，就是MDN中：typeof 操作符返回一个字符串，表示未经计算的操作数的类型。那么这句话中的‘未经计算的操作数’是什么意思呢？这个‘未经计算的操作数’有哪些类型呢？而且这个typeof的返回值，返回的应该不是一种类型吧！因为用typeof检测类型的时候可能返回 'function'，但是function又不属于数据类型，是不是有点矛盾呢？

作者回复: 我之前并没有听过关于这个‘未经计算的操作数’的说法。因此我特地地看了一下MDN中的相关说明。

‘未经计算的操作数(unevaluated operand)’这个，其实也并没有特别的难解。例如有一个值是2的常量x，对于这个`x`，如果它“计算了”，那结果当然就是2，对吧。那么这种情况下，“未经计算时的x”是什么呢？

这个其实还是我们在文章中说的“引用（规范类型）”。“引用（规范类型）”作为左手端时，只是引用，并不求值，这种情况下它就是`unevaluated operand`。所以，一个错误的、根本不存在的引用也可以被typeof操作，因为这个“错误的、根本不存在的”并没有被“计算”，所以也就不会抛出错误。例如你试试：

```
...
```

```
typeof adfasdljkfla; // <- 随便一个变量名
```

```
...
```

之所以没有异常发生，就是后面的`adfasdljkfla`这个东东“未经计算”。同样的，如果我们尝试下面的代码：

```
...
```

```
typeof(adfasdljkfla); // 同上例
```

```
...
```

这里其实多了一个操作符，就是一对括号表示的“分组运算符（grouping）”，这个运算符也是“返回未经运算的结果”。所以同样，不会出错。——在我们这个系列的文章中，这种情况称为“引用（规范



类型)”，或者一个“(未决定操作手性rhs/lhs的)结果Result”。

还有你的另一个问题：

> 而且这个typeof的返回值，返回的应该不是一种类型吧！

这个是其它类型的语言来理解函数式语言的一个常见误区。尤其是，如果你以传统的（经典的）数据结构的知识为基础，那么更是会有误解。

在JavaScript中，以及在函数式语言中，“函数”的确是一种数据类型。它可以作为值（在函数界面上）传递，也可以作为结果（在函数返回中）传出，还可以查看类型，还可以与其它数据进行运算（例如 `1 + (function(){}))`），那么它为什么不是“一种数据类型”呢？

函数既是数据，也是运算，这个是函数式语言的核心概念。



👍 5



海绵薇薇

2019-11-20

hello 老师好：

一开始我不明白为啥要称 `var a = 1;` 是值绑定操作，看了几遍之后应该理解了，`var` 是一个申明，等号左边不是表达式。而赋值操作等号左边是一个表达式结果是引用，右边是值，这样完成的赋值操作。但是`var` 右边等号左边不是一个表达式所以不是赋值，换了名字叫绑定。

作者回复: YES! 这回依对了。^^.



👍 5



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>

^^.



👍 5



Geek\_8d73e3

2020-08-06

老师，那我还有一个疑惑

既然`let x` 为词法声明，词法声明不会初始化绑定一个`undefined`，而且js引擎拒绝访问未初始



化的词法声明

那如何解释以下代码

```
let x;
```

```
console.log(x) //这里输出undefined
```

作者回复: let x;

这是声明没错，但它有“执行期语义（Runtime Semantics）”。对于LetOrConst来说，这个执行期语义就是“绑定初始值”。

简单地说，就是“执行到这一行就初始化了”。

参见这里：

<https://tc39.es/ecma262/#sec-let-and-const-declarations-runtime-semantics-evaluation>



👍 3



Chor

2020-02-21

老师您好，我想问一下：

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”这句话是不是说，x这个本来不存在的属性仅在第二次赋值操作的时候才会被创建？
2. a.x 这个表达式计算的结果（Result）是一个引用，是否可以把这个引用看作一个“容器”，这个“容器”包含着原始的a的信息？还是说这个引用就是原始的a本身？
3. 一开始程序在分析 a.x 的时候（第二次赋值发生之前），这个表达式的Result中是否包含相关的x的信息？还是说这时候x只是一个暂时不存在、等待创建的东西？

作者回复: 1. 是的。

2. 是的。“引用（规范类型）”可以看作原始a的容器，包含原始a的信息。

3. “表达式的Result中是否包含相关的x的信息”，是的，是包含着x的相关信息。

“引用（规范类型）”是一个结构，通常有三个成员，base、name和strict。所以，无论`x`属性是否存在，`a.x`都被表达为{"base": a, "name": "x", ...}，这个结构就是`a.x`的引用，或者说是Result。直到需要读写它的时候（例如作为rhs），才会去检索a["x"]是否真实存在，并决定后续操作。

关于“引用（规范类型）”的一些细节，你可以看看这个：

<https://github.com/d2forum/14th/>

在《JS 语言在引擎级别的执行过程》中专门有一部分是讲述它的。视频在这里：



共 2 条评论 >

👍 3



itgou

2019-11-29

书读百遍，其义自见，在听读了n遍之后，终于理解了标题中的代码，但是看到链表代码，又有点晕了，亲老师解答一下。问题如下

```
var i = 10, root = {index: "NONE"}, node = root;
```

```
while (i > 0) {
```

```
  node.next = node = new Object; //本行开头的node.next未被丢弃，是因为这里大括号里面是一个闭包，而外层node=root对这里有引用吗？
```

```
  node.index = i--;
}
```

```
// 测试
```

```
node = root;
```

```
while (node = node.next) {
  console.log(node.index);
}
```

问题写在了while循环当中，请老师回答一下。

作者回复: 闭包这个概念是与函数相关的（当然对象闭包则与with相关），所以这里不适合用“闭包”这个词。

在大括号内的是一个块级作用域，你也可以叫“词法的块级作用域”或者直接叫“作用域”。

当一个“单向链表”处于系统中时，如果链表首（root）没有被引用的话，你是找不到这个完整的链表的。——很明显，你没有办法反向地检索。所以会有外层的node = root。当然，从引擎的角度上来说，如果是这样的一个链表（没有变量来引用root），那么它的确会被废弃。你从数据结构的角度上思考一下就明白了，没有办法回溯，也没有别的东西来引用任何一个“向前的”结点，只会有最后一个结点被引用（从而不被废弃）。

💬

👍 3



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 : 1}

第二句：

```
a.x = a = {m : 2};
```

// 两个等号划分了3个表达式（宏观上）；

// a.x... 要为a添加x属性的蠢蠢欲动，缓存a，a = {n : 1};

// a.x = a... 没有做赋值操作！如果代码写到这截止，事实上会报一种错，叫Error: Maximum call stack size exceeded

// a.x = a = {m : 2}; 做了两次赋值操作，首先后半段先做赋值操作，a的引用指向了新的对象 {m : 2}，第二次赋值操作完成了为之前缓存的a添加x属性的如愿已久，x的引用指向后面的这个完成了初始化的a。现在，我们去使用a，实际上使用的是后面的这个a，a = {m : 2}，那之前缓存的那个a呢？被引擎吃掉了，无法访问到。那它指向哪个对象呢？{n : 1, x : {m : 2}}，理由是一次初始化和一次属性拓展。

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。



3



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？

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！

共 2 条评论 >



3



卡尔

2020-06-11

如果a（或elemData）总是被重写的旧的变量，那么如下代码：

老师，这话是什么意思？

作者回复: 在下面的示例代码中：

node.next中的`node`，就是这个”总是重写的旧的变量“；而，



node.index中的`node`则是”新的变量“。

所以，”a（或elemData）总是被重写“，意味着在建立链表的过程中它是可以用来”暂存上一个节点（node）“的。



2

