

# 加餐 | 让JavaScript运行起来

2019-12-09 周爱民

《JavaScript核心原理解析》

课程介绍 >



讲述：周爱民

时长 20:09 大小 18.46M



你好，我是周爱民。欢迎回到我的专栏。今天，是传说中的加餐时间，我将与你解说前 11 讲内容的整体体系和结论。

我们从一个问题讲起，那就是：**JavaScript 到底是怎么运行起来的呢？**

看起来这个问题最简单的答案是“解析→运行”。然而对于一门语言来说，“引擎解释与运行”都是最终结果的表象，真正处于原点的问题其实是：“**JavaScript 运行的是什么？**”

在前 11 讲中，我是试图将 JavaScript 整个的运行机制摊开在你的面前，因此我们有两条线索可以抓：

1. 表面上，它是讲引用和执行过程；
2. 在底下，讲的是引擎对“JavaScript 是什么”的理解。



## 从文本到脚本

我们先从第二条线索，也就是更基础层面的线索讲起。

JavaScript 的所谓“脚本代码”，在引擎层面看来，首先就是一段文本。在性质上，装载`a.js`执行与`eval('...')`执行并没有区别，它们的执行对象都被理解为一个“字符串”，也就是字符串这一概念本身所表示的、所谓的“字符序列”。

在字符序列这个层面上，最简单和最经济的处理逻辑是**正向遍历**，这也是为什么“语句解析器”的开发者总是希望“语言的设计者”能让他们“一次性地、不需要回归地”解析代码的原因。

回归（也就是查看之前“被 parser 过的代码”）就意味着解析器需要暂存旧数据，无法将解析器做得足够简洁，进而无法将解析器放在小存储的环境中。根本上来说，JavaScript 解析引擎是“逐字符”地处理代码文本的。

JavaScript 从“逐字符处理”得到的引擎可以理解的对象，称为记号（Tokens）。这个概念，是从第一讲就开始提的，你回顾第一讲的内容，在提出Tokens这个概念的时候，有这样一句话：

一个记号是没有语义的，记号既可以是语言能识别的，也可以是语言不能识别的。唯有把这两者同时纳入语言范畴，那么这个语言才能识别所谓的“语法错误”。

我之所以用“**delete 运算**”作为《JavaScript 核心原理解析》的开篇，是因为在我看来，这讲的是一种“不知死，即不知生”的道理。如果你不知道一个东西是如何被毁灭的，那么你也知道它创生的意义。

然而，这个理解也可以倒过来，是所谓的“不知生，亦不知死”。也就是说，如果你都不知道它被创造出来的时候是什么，那么你也知道你毁灭了什么。

而这个记号（Tokens），就是引擎从文本到脚本，JavaScript 引擎也好、语言也好，它们创造出来的第一个东西——也是在创世原点唯一的東西。

记号，要么是可识别的，要么是不能识别的。并且，它们必须同时纳入语言范畴。这个“必须同时纳入”，决定了二者不是相互孤立的元素，而是同一体系下的东西，也就是所谓的“体系的完整性”。



## 引用与静态语言的处理

看完底层的线索，我们再来看看 JavaScript 运行机制的表面线索。

引用（References）是静态语言与引擎之间的桥梁，它是 ECMAScript 规范中最大的一个挑战，你理解了“规范层面的引用（References）”，也就基本上理解了 ECMAScript 规范整个的叙述框架。这个框架的核心在于——ECMAScript 的目的是描述“引擎如何实现”，而不是“描述语言是什么”。

规范层面中的引用与引擎的核心设计有关。

在 JavaScript 语言层面，它希望引擎是一个执行器，更具体的描述是：引擎的核心是一个表达式计算的、连续的执行过程。表达式计算是整个 JavaScript 语言中最核心的预设，一旦超出这个预设，JavaScript 语言的结构体系就崩溃了。

所以，本质上来说，JavaScript 的所谓“语句能执行”也是一个或一组表达式计算过程，而且所有的计算都必须能描述成一个基本的模式：`opCode -> opData`，也就是用操作符去处理操作数。

这个相信你也明白了，这回到了我们计算理论最初的原点，是我们学习计算机这门课程最初的那个设定：**计算实现的就是“计算求解”的过程**。它的另一个公式化的表达就是著名的“算法 + 数据结构 = 程序”。

当然，这个说得有点远了，在这个概念集合中，最关键的点在于“**执行过程最终是表达式计算**”。因此，语句执行也是表达式计算，函数调用也是表达式计算，各种特殊执行结果还是表达式计算。

这些“计算”总会有一个返回值，是什么呢？

你可以参考文章里的这张图，它说明了 JavaScript 中最核心的两种执行过程（它们都被称为 evaluating）是如何最终被统一的。



result of evaluating XXX, when xxx is *statement*

## Completion Specification Type

type

value

target

GetValue(ref)

result of evaluating XXX, when xxx is *expression*

## Reference Specification Type

base

name

strict

thisValue

value is Language Types, include:

- Primitive values, or
- Object

or Empty

result is Language Types, include:

- Primitive values, or
- Object

or Reference specification type.



在语句执行的层面，它返回一个语句的完成状态，这个状态中包括了一个“value”域，它必须且必然会是 JavaScript 语言理解的类型，也就是 `typeof()` 所识别的所有值。这样一来，任何“语句”“代码”或“代码文本”就都可以被执行了，并且都可以使用 `console.log()` 输出结果给你了。

这其中最重要的一件事是，在任何语句执行并得到结果时，如果它“当时”是一个所谓的“引用”，那么这个引用就必须先调用“`GetValue(x)`”来得到值，然后放到这个“value”域中去。因为“引用”是一个规范层面的东西，它不是 JavaScript 语言能理解的，也无法展示给开发者。

最后，ECMAScript 约定：可以在“value”域中放上 `Empty`，这表明语句执行“没有值”。它能表明有值，也能表明无值，仍然是“概念完整性”。

而到了表达式执行时（注意函数调用也是表达式执行的一种），这个过程又被重来了一回。不过表达式执行会返回两个东西：它要么直接返回一个“上面的完成结果所理解的值”，要么返回一个包含这样的值的“引用”。

你可能会说了，不对呀——你刚才还说所谓“概念的完整性”，是“要么返回东西，要么返回没有东西”啊。



对的，在表达式执行这个体系里面，“没有东西”是所谓的“不可发现的引用（UnresolvableReference）”。

所以，完整的概念集是：值（value）、引用（Reference）和不可发现的引用（UnresolvableReference）。

一个不可发现的引用是能被处理的，例如`delete x`，或者`typeof x`。所有“能处理引用的”运算符都能处理它。当然，在严格模式中，会在语法分析阶段就报异常，那是另一个层面的东西，有机会的时候我们再聊。这里，在 JavaScript 语言层面，它仍然在维护一种简单的完整性。

那么，为什么要有“引用”这么个东西呢？

你想想，如果没有引用，你就得将所有的东西都直接当成一个被处理的对象，例如用 1G 的内存来处理一个 1G 文本的记号。这显然不可行。我们可以用一个简单的法子来解决，就是加一个指针指向它，在不需要访问它的“内容”时，我们就访问这个指针好了。而引用，也就是所有在“不访问内容”的情况下，用于指向这个内容的一个结构。它叫什么名字其实都好、都行，重点的是：

1. 它代表这个东西， $r(x)$ 。
2. 它包含这个东西，所以可以  $x = \text{GetValue}(r)$ 。

所以本质上，引用还是指向值、代表值的一个概念，它只是“获得值的访问能力”的一个途径。最终的结果仍然指向原点：计算值、求值。

## 结构与体系的回顾

讲完 JavaScript 整个运行机制的两条线索后，就是加餐的最后一部分内容了，我会直接为你解说前 11 讲的主题。

### 模块一：体系 1

- 1 | delete 0



讲述的是“规范引用”，将“规范引用”与传统概念中的引用区别开来。用 `Result` 来指代执行结果的“引用状态和值状态未区分”。同时指明，“状态未区分”的原因是：同一个标识符，在作为 `_lhs_` 和 `_rhs_` 的时候意义是不同的；并且，在计算没有“推进到”下一步之前，上一步的 `Result` 是无法确知“将作为”`lhs/_rhs_` 的哪一种操作数的。

JavaScript 确实有一部分表达式（或操作）是能处理“规范引用”的，例如 `delete x` 就是其中之一。有关哪些运算能处理“规范引用”，建议你自己翻阅 ECMAScript，并从中归纳。

- 2 | `var x = y = 100`

这一讲的核心是讲六种声明。所有声明（语句）都是没有返回值的（返回 `Empty`），因为它没有返回值，所以它对其他执行过程没有影响。也就是说，声明语句必须能被理解为“静态分析的结果”，而不是“动态执行的结果”。

前者称为“声明语义”，后者称为“执行语义”。声明语义就是静态语言的处理，执行语义就是动态语言的处理。这是两种语言范型的分水岭。

- 3 | `a.x = a = {n:2}`

这一讲的核心是讲表达式执行与（看起来跟它相似的）语句声明之间的区别。虽然两种看起来都相似，但其实只有这一讲的才是“表达式连等”。

在这一讲结尾的部分，我做了一个总结：有关“引用”的介绍，以及“语句”和“表达式”之间的差异与分别，自此暂告段落。

- 4 | `export default function() {}`

这一讲的核心是讲“名字”的使用。“有名字 / 没有名字”是一对概念，而“没有名字”就称为“default”，那就是将概念收敛到了唯一一个：名字。所有有关 `export/import` 的处理，就是名字与它所代表的东西之间的关系映射。

而“模块装载的过程”必须发生在用户代码之前，一共包括了两个意思：

1. 引擎必须有一个依赖顺序来“初始化”那些名字，这个与 `export` 语句是“声明”有关，声明意味着它是静态完成的（名字总是被静态声明的）；





2. 用户代码需要依赖那些名字，这与 `import` 语句不是“声明”有关，它不是声明，那么它需要通过“执行”来得到结果的，而这些“执行”必须在用户代码之前。其顺序，就是所谓模块装载树的遍历。

- 5 | `for (let x of [1,2,3]) ...`

这一讲的要点不是讲语句执行，而是讲**块级作用域**，更进一步的，它是在讲作用域的“识别”与处理。它颠覆读者认知的地方在于提出：绝大多数语句并没有块级作用域，因为它们**不需要**。

而需要块级作用域的 `for` 语句，根本的需求是需要处理多次迭代中的变量暂存。这个是有很大开销的，这与“计算机语言”的一个核心原理有关：迭代需要循环控制变量，这是命令式语言有变量的根源（之一），也是函数式语言需要处理递归的根源。

“需不需要存储计算过程中的变量”，也是命令式语言与函数式语言的分水岭。

以上是前 5 讲的内容。到现在为止，在第一模块中，我们主要提出的是语言的三个层面的概念：

- 第一层概念：记号
- 第二层概念：引用、值
- 第三层概念：表达式、语句、名字、环境 / 作用域 、（顺序执行的三种基础）逻辑

NOTE: 这主要是在 [🔗 《程序原本》](#) 前三章中的概念，包括“数、逻辑和抽象”。部分涉及到第四章，也就是“语言”中的概念。

这些概念其实基本上都是在“代码的静态组织”过程中就完成 / 实现了的。你使用一门语言，其实本质上就是在跟第三层概念打交道，而 ECMAScript 或者引擎是工作在第二个层面的。第一个层面，则是物理层面与逻辑层面的、最初的映射。

## 模块二：体系 2

接下来，我们讨论第 6~11 讲。

- 6 | `x: break x;`



这一讲是讲了真正的语句执行。仍然是“不知死，即不知生”的讲法，`break x`与语句的关系，同`delete x`与引用的关系其实差不多。

而且这一讲也提出了“语句以执行的完成状态”为结果，这个伏笔要留到第 8 讲来解开。

- 7 | `${1}`

讲述了特殊的可执行结构。如果按照第一讲中所表达的“JavaScript 引擎的核心是一个表达式计算的、连续的执行过程”，那么将所有显式的、隐式的“执行行为”合起来看，才是“执行逻辑”的全体。正如你不了解每一种特殊的可执行结构，也就不了解“`${1}`”为什么是最“晚”出现的语言特性之一。因为它是对其他执行结构的“集大成者”。

当然还有一点特殊之处也是你需要了解的，`eval(str)` 是执行语句，而`\${str}` 是执行表达式。本质上来说，JavaScript 为这两种执行都找到了“执行一个字符串”的模式，这仍然是“概念完整性”。

NOTE：试试如下代码：

```
1 > `${{}}`  
2 '[object Object]'  
3  
4 > eval('{}')  
5 undefined
```

 复制代码

- 8 | `x => x`

表面上看是讲一个箭头函数，实际上是在讲函数式语言。关键处是解开第 6 讲伏笔的这句话：

语句执行是命令式范型的体现，而函数执行代表了 JavaScript 中的对函数式范型的理解。

另外，这一讲把函数分成了三个语法组件：参数、执行体、结果。这是非常重要的一个点，它引导了后面两讲的讨论方式。





- 9 | (...x)

这一讲说的是如何改造函数的三个语法组件中的“执行体”。这一讲提出了“改造三个语法组件”的意义，也就是说，函数式语言无论如何变、语法如何处理，其实本质上，就是在这三个点上做手脚、玩花样。

- 10 | x = yield x

这一讲说的是如何改造函数的三个语法组件中的“参数”和“结果”。

NOTE: 这一讲也为将来“再讲循环”留了一个伏笔，不过这并不是前 20 讲的内容，这是“更远的将来”。^^.

- 11 | throw 1;

这一讲其实讲的是怎么读 ECMAScript 规范。

不过它是以“最小化的”三个规范说明，来讲述了 ECMAScript 层面是如何一步一步地将 JavaScript 搭建出来的。这一讲里面有很多概念和观念，一旦你弄明白了，对 ECMAScript 也好，JavaScript 也好，都能起到“点化”的作用。

其实这里有很重要的一点引导，是这样一句话：

其中的“result of evaluating...”基本上算是 ECMAScript 中一个约定俗成的写法，不管是执行语句还是表达式，都是如此。

这句话很重要，它从 ECMAScript 规范层面、从语句叙述的层面“一致化了”语句执行和表达式执行。注意：这就是上面那张图的出处！

这是第二模块的内容。根本上来说，承接我们这一模块的总标题“JavaScript 是如何运行的”，我主要为你讲述了三层概念：

- 第三层概念：表达式执行、函数执行、函数执行的扩展。
- 第二层概念：在规范层面如何统一“表达式执行和函数执行”。



- 第一层概念：语言体系的建立。

参考前面的图，既然执行结果被统一为“result”，且执行被统一为“evaluating”，那么运算就被统一成“result of evaluating...”，并且结果（如果返回给计算系统的外部的话）就是一个能被理解的 result.value。

NOTE: 这个概念层次的构建，以及最终对它要达到的效果的预期，你可以参考阅读《程序原本》第 4.6 节，它的标题是：将“计算机程序设计”教成语言课，是本末倒置的。

## 模块三：体系 3

回顾上面的内容，

- “体系 1”说的是“物理到逻辑”的映射
- “体系 2”说的是“语言体系的建立”

总体上来看，它们是在陈述一件事情：“抽象的语言”如何处理“物理的代码”。

这仍然是一个体系。

NOTE: 回顾前两大模块的标题，其实这个“体系 3”我是一开始就告诉了你的：  
从零开始：JavaScript 语言是如何构建起来的  
从表达式到执行引擎：JavaScript 是如何运行的

## 最后

本来课程设计中并没有今天这一讲的加餐。按原定的计划，就是用第 11 讲最后的“小结”算作引导你的、对之前内容最终回顾了。

但是考虑到课程进度和实际上的难度，才有了上一次的和今天的加餐。尤其是今天的内容，其实就是对上一讲——第 11 讲的小结内容的展开，希望你能对照着，重新来理解和梳理这门课程。

希望这份加餐会让你后续的课程变得轻松一些。今天就到这里，下一讲我们开始讲面向对象。



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 6 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 11 | throw 1;：它在“最简单语法榜”上排名第三

下一篇 12 | 1 in 1..constructor：这行代码的结果，既可能是true，也可能是false

## 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 (16)

写留言



kkxue

2020-05-21

个人觉得老师说明下您理解编程语言背后的哲学逻辑或者体系，我们上道才更快。

作者回复: 关于这个，还真的是有的。请参考如下：

> <https://github.com/aimingoo/my-ebooks>



其中的《程序原本》一书，是可以自由下载分享的电子书。

共 2 条评论 >

👍 3



刘长锋

2020-03-12

D2 大会上听老师说，要有新书发布，很是期待！

作者回复: 谢谢🙏已经交稿了，在等出版社～可这不肺炎疫情闹的嘛😓



👍 2



潇潇雨歇

2019-12-09

结合《程序原本》重新回顾前几讲，有趣

共 3 条评论 >

👍 2



孜孜

2020-06-18

这两章的东西，虽说是js的，但是我感觉这是不限制在某个语言层面的。



👍 1



westfall

2020-06-09

“所以本质上，引用还是指向值、代表值的一个概念，它只是“获得值的访问能力”的一个途径。最终的结果仍然指向原点：计算值、求值。”

不明白引用与指针的区别。

作者回复: 引用是在“设计语言的规范”中用的，而指针是“程序代码”中用的。  
在js中的“引用与值”，与在ECMAScript规范中的“引用与值”不是相同的概念。



👍 1



K4SHIFZ

2020-03-25

老师，下个课程什么时候出啊，迫不及待想学了

作者回复: :) )



这个真的要等很久可能。^^.

共 2 条评论 >



1



**Kids See Ghost**

2022-01-17

关于GetValue最后再请教一下

规范里的是

...

#### 6.2.4.5 GetValue ( V )

The abstract operation GetValue takes argument V. It performs the following steps when called:

1. ReturnIfAbrupt(V).
  2. If V is not a Reference Record, return V.
  3. If IsUnresolvableReference(V) is true, throw a ReferenceError exception.
  4. If IsPropertyReference(V) is true, then
    - a. Let baseObj be ? ToObject(V.[[Base]]).
    - b. If IsPrivateReference(V) is true, then
      - i. Return ? PrivateGet(baseObj, V.[[ReferencedName]]).
    - c. Return ? baseObj.[[Get]](V.[[ReferencedName]], GetThisValue(V)).
  5. Else,
    - a. Let base be V.[[Base]].
    - b. Assert: base is an Environment Record.
    - c. Return ? base.GetBindingValue(V.[[ReferencedName]], V.[[Strict]]) (see 9.1).
- ...

我以为这个被get的value就是跟base有关。我看了您说的prepack-core里的代码，这部分貌似是叫\_dereference。代码大概是

...

```
_dereference(realm: Realm, V: Reference | Value, deferenceConditionals?: boolean = true): Value {
```

```
    // This step is not necessary as we propagate completions with exceptions.
```

```
    // 1. ReturnIfAbrupt(V).
```

```
    // 2. If Type(V) is not Reference, return V.
```

```
    if (!(V instanceof Reference)) return V;
```

```
    // 3. Let base be GetBase(V).
```

```
    let base = this.GetBase(realm, V);
```

```
    //....
```

```
    if (this.HasPrimitiveBase(realm, V)) {
```



```

// i. Assert: In this case, base will never be null or undefined.
invariant(base instanceof Value && !HasSomeCompatibleType(base, UndefinedValue, NullValue));

// ii. Let base be To.ToObject(base).
base = To.ToObject(realm, base);
}
invariant(base instanceof ObjectValue || base instanceof AbstractObjectValue);

// b. Return ? base.[[Get]](GetReferencedName(V), GetThisValue(V)).
return base.$GetPartial(this.GetReferencedNamePartial(realm, V), GetThisValue(realm, V));
...

```

这里怎么看都是跟base有关系。实在是看不懂到底怎么样把一个value用getValue取出来。您的课也我也没有看到有直接讲到这个。所以能不能麻烦最后再指点一下，到底规范里面的哪一步是把像obj = {x: 'abcdef'}，当【obj.x】作为一个引用时，我们把字符串给用getValue取出来的？

作者回复: 对于这个例子，由于是obj.x,所以r.Base指向obj，r.ReferencedName指向'x'。

然后在第2~3步的判断中都是false，直到第4步，IsPropertyReference(r)将判断为True，这样会在4.a中r.Base中取出对象obj，置入baseObj。

接下来到4.c，会调用

```
baseObj.[[Get]](V.[[ReferencedName]], GetThisValue(V))
```

也就是说，会调用obj的内部方法槽[[Get]]，然后传入name和this。

来到对象标准的内部方法[[Get]]的定义，在这里：

<https://tc39.es/ecma262/#sec-ordinary-object-internal-methods-and-internal-slots-get-p-receiver>

注意按照上面的传参，这里的P就是name，就是GetReferencedName(V)。于是进入OrdinaryGet()，在这里你能看到一个过程，就是地扫描obj或递归它的parent的属性表，直到找到P所传入的这个名字'x'（记得原型链上怎么读属性obj.x吧？）。

找到P对应的名字'x'之后，检测一下，如果它是数据属性，就返回这个对象的属性列表中的值，亦即是return desc.[[Value]]。否则就调用存取器方法desc.[[Get]]来取。——记得属性有两种声明方式吧？

所以，对于obj.x来说，调用GetValue(r)的过程，最终会变成：

...

```
obj = r.[[Base]]
```



```
name = r[[ReferencedName]] // 'x'
desc = obj[name] 的属性描述符
...
```

并最终取得desc.[[Value]]的结果，例如你之前例子中的"abcdef"。



Kids See Ghost

2022-01-17

再请教一下：在JS里做equality check的时候，比如用 === – 这个时候我们是在比value还是比reference？也就是说  $x === y$  – 等于是  $\text{getValue}(x) === \text{getValue}(y)$  吗？

相应地，是不是说在JS里面，能拿来比较的东西只能是 value，两个reference是肯定不一样的？

请问规范里面有相应的章节吗？

作者回复：比值。所以是  $\text{getValue}(x) === \text{getValue}(y)$  。规范中在这里：

<https://tc39.es/ecma262/#sec-equality-operators-runtime-semantics-evaluation>

你自己看，确实就是对两个操作数都做了 GetValue。

另外，你对 GetValue 的理解是错的。我在另一个回复里说。

再另外，要自己查规范。慢慢看，你会找到的，不要偷懒。



Kids See Ghost

2022-01-17

1. 请教一下，您说“例如  $\text{obj} = \{x: \text{'abcdef'}\}$ ，当 **【obj.x】** 作为一个引用时，base是obj，而不是那个字符串'abcdef'。又例如全局的变量  $\text{let } x = \text{'abcdef'}$ ，base将是全局词法环境，指向Global.lexEnv。”

假如我现在把obj.x赋值给一个变量，那  $\text{getValue}(\text{obj.x})$  拿出来的值不应该是字符串'abcdef'吗？<https://tc39.es/ecma262/#sec-getvalue> 我理解的规范里的getValue 最终拿出来的value就是base，请问这个理解是错的吗？如果是错的，那getValue里拿出来的value到底是什么？

换一个问法，如果当 **【obj.x】** 作为一个引用时，错误地把base是当成了那个字符串'abcdef'，会造成什么样的理解错误呢，请问您能举个例子吗？

2. 另外想请教一下在<https://tc39.es/ecma262/#sec-reference-record-specification-type> 里定义的 Reference Record Specification Type里，[[ReferencedName]] 提到 "The name of the binding. Always a String if [[Base]] value is an Environment Record."这句话





的意思是什么？意思是如果base不是一个Environment Record，那ReferencedName可以不是string？请问Environment Record又是啥。。

作者回复: 规范里的getValue 最终拿出来的value就是base，请问这个理解是错了吗？如果是错的，那getValue里拿出来的value到底是什么？

-----

当然是错的。“引用（规范类型）”里面根本不保存value，它没有value字段，也不存放值的内容。它是一个值的索引器，所以你应该仔细阅读GetValue()这个内部过程，才能弄明白如何利用这个东西找到“每一种不同类型的引用”所对应的值。

同样的原因，base肯定不可能“错误的理解为那个字符串”。那个字符串根本不会在“引用（规范类型）”这个记录的任何地方。

仔细阅读规范。另外，如果你确实对规范理解有疑问，我建议你自己git一个prepack-core项目下来，然后在IDE环境（例如vscode）里调试来看。之所以推荐prepack-core，是因为它是按照规范逐行翻译成js代码的，读起来易懂。prepack-core项目在这里：

<https://github.com/aimingoo/prepack-core>

Environment Record~~ 看规范吧。里面有，是非常关键的类型。或者我之前给过你的视频地址里也有。在这里：

<https://www.bilibili.com/video/BV1Wy4y1b7PG>

如果你才开始读规范，那么至少也要先把规范的大概目录看完，知道它怎么写以及有什么吧。这些东西在结构上看明白大概也得花些功夫，带着问题先看吧，不要着急找答案。问题没问对，答案是没意义的；问题问对了，答案就几乎跃跃欲出了。



**Kids See Ghost**

2022-01-15

几个问题想向老师请教一下：

1. Reference Record Specification Type (<https://tc39.es/ecma262/#sec-reference-record-specification-type>) 就是您一直提到的“规范层面的引用（References）”吗？
  2. 引用reference里的Base，我们是理解为真正的那个数字/字符串/object etc. 吗？而不是指针/引用（内存层面上的）或者了另一个reference？
  3. 最想问的一个问题是，想知道在call function的时候，规范里有规定是pass by value还是pass by reference吗？假如说我们有
- ...

```
function fn(x) {}  
const a = {}  
const b = 1
```



fn(a)  
fn(b)  
...

这里第一次给fn传入`a`的时候，整个过程我们有`x = getValue(a)`吗还是说我们直接把`a`的reference给了`x`？fn(b)也一样吗？

我在规范里找了很久都找不到相应的章节，请问您能给一个链接到专门讲\*\*函数传参\*\*的章节吗？

4.最后，您说“这个框架的核心在于——ECMAScript 的目的是描述“引擎如何实现”，而不是“描述语言是什么”。”这句话确定没有说反了吗？难道不是 ECMAScript的目的是设计一个语言，具体语言怎么实现是javascript engine的事情。比如ECMAScript里面不会讲memory layout应该会怎么样，什么东西放到stack上，什么放到heap上，因为这个是implementation details，由引擎自己决定。

感谢！

作者回复: 1. 是。

2. 不是。例如obj = {x: 'abcdef'}, 当【obj.x】作为一个引用时，base是obj，而不是那个字符串'abcdef'。又例如全局的变量let x = 'abcdef', base将是全局词法环境，指向Global.lexEnv。

3. 是传值。例如f(obj.x)，那么实际传入的是f(GetValue(obj.x))。在执行Call()内部运算之前，传入参数列表就被处理过了。规范上也讲了，只是一个叙述。如下：

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

原文：and argumentsList is the value passed to the corresponding argument of the internal method.

4. 没说反。ECMAScript当然也描述了语言是什么样子，比如语法、静态词法等等，但这在规范中十不占一。真正多的内容是如何实现它。ECMAScript当然没讲你说的这些，因为那些是那些是执行环境。——“引擎如何实现一个语言”不单单是说栈如何操作、内存如何分配，也包括一个具体的“+”运算符如何实现，对吧？ECMAScript只约定了“+”如何实现，没说它的运算数在栈上如何分配，很正常啊，具体执行环境实现的时候用不用栈都还另说呢。



Amundsen

2020-03-20

每次反复阅读都能收获，感觉看到了不一样的自己：)



伪装

2019-12-31

按照ecma规范写一套解析器就是js了



青史成灰

2019-12-14

老师，关于上面“为什么要有‘引用’这么个东西呢”的解释，读下来感觉和C++的指针很像，指针是内存的地址，指向堆内存中的对象，需要访问指针指向的成员时，直接解引用这个指针， $v = *p$ ，就和此处的 $x = \text{GetValue}(r)$ 一样。不知道这样理解是否正确？

作者回复: 这是不一样的。

在第一讲的回复内容中，我给leslee的回复里面讲过“JavaScript中的引用”，与“ECMAScript中的引用”不是同一个东西。你这里所谈的指针概念，与“JavaScript中的引用”类似，它的细节和作用，你可以看看上面这一讲关于leslee的回复。

另外，在给Smallfly的回复中，我详细讲了ECMAScript中的引用是怎样的一个结构。你也可以阅读一下。

在这里：<https://time.geekbang.org/column/article/164312>

(不过因为这个工具的设计问题，我没办法直接指到他们的评论回复，请查找一下)

共 2 条评论 >



weineel

2019-12-09

知识密度太大。



许童童

2019-12-09

老师还是很良心的，时不时就来给我们一个加餐。



行问

2019-12-09

`eval(str)` 是执行语句，而`{str}`是执行表达式

这里是 `{str}` 正确，还是 `${str}` 正确？



作者回复: 这里是有排版错误, 后一个是`\${str}`。我的意思是, 同一个字符串(仅指它的字面文本), 这里是它作为语句和表达式执行的两种方式。但是, 如果str理解成“变量”, 而不是“变量的字面文本”, 那么就不是我的原意了。

共 2 条评论 >

