

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**运行的是什么呢？”

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。唯有把这二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东西。

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

**Reference**

**base**

**name**

**strict**

**thisValue**

**value** is Language Types, include:

- Primitive values, or
- Object

or Empty

**result** is Language

- Primitive value
- Object

or Reference spe

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，你就得把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 = GetValue®`。

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水分岭。

- 3 | a.x = a = {n2}

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：

- 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）；
- 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水分岭。

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 | \${}

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`\${}`“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：

```
> `${{}}`
'[object Object]'
> eval('{}')
undefined
```

- 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。

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运行的是什么呢？”

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。唯有把这二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东西。

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

**Reference**

**base**

**name**

**strict**

**thisValue**

**value** is Language Types, include:

- Primitive values, or
- Object

or Empty

**result** is Language

- Primitive value
- Object

or Reference spe

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，你就得把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 = GetValue®`。

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水分岭。

- 3 | a.x = a = {n2}

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：

- 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）；
- 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水分岭。

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 | \${}

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`\${}`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：

```
> `${{}}`
'[object Object]'
> eval('{}')
undefined
```

- 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。

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运行的是什么呢？”

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。唯有把这二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东西。

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

**Reference**

**base**

**name**

**strict**

**thisValue**

**value** is Language Types, include:

- Primitive values, or
- Object

or Empty

**result** is Language

- Primitive value
- Object

or Reference spe

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，你就得把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 = GetValue®`。

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水分岭。

- 3 | a.x = a = {n2}

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：

- 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）；
- 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水分岭。

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 | \${}

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`\${}`“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：

```
> `${{}}`
'[object Object]'
> eval('{}')
undefined
```

- 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。

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运行的是什么呢？”

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。唯有把这二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东西。

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

**Reference**

**base**

**name**

**strict**

**thisValue**

**value** is Language Types, include:

- Primitive values, or
- Object

or Empty

**result** is Language

- Primitive value
- Object

or Reference spe

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，你就得把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 = GetValue®`。

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水分岭。

- 3 | a.x = a = {n2}

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：

- 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）；
  - 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水分岭。

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 | \${}

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`\${}`“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：

```
> `${{}}`
'[object Object]'
> eval('{}')
undefined
```

- 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。

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运行的是什么呢？”

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。唯有把这二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东西。

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

**Reference**

**base**

**name**

**strict**

**thisValue**

**value** is Language Types, include:

- Primitive values, or
- Object

or Empty

**result** is Language

- Primitive value
- Object

or Reference spe

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，它必须且必然是JavaScript语言理解的类型，也就是typeof()所识别的所有的值。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，你就得把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=GetValue®。

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水分岭。

- 3 | a.x = a = {n2}

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：

- 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）；
- 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水分岭。

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 | \${}

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`\${}`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：

```
> `${{}}`
'[object Object]'
> eval('{}')
undefined
```

- 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。

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运行的是什么呢？”

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。唯有把这二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东西。

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

**Reference**

**base**

**name**

**strict**

**thisValue**

**value** is Language Types, include:

- Primitive values, or
- Object

or Empty

**result** is Language

- Primitive value
- Object

or Reference spe

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，你就得把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 = GetValue®`。

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水分岭。

- 3 | a.x = a = {n2}

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：

- 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）；
- 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水分岭。

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 | \${}

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`\${}`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：

```
> `${{}}`
'[object Object]'
> eval('{}')
undefined
```

- 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。

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运行的是什么呢？”

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。唯有把这二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东西。

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

**Reference**

**base**

**name**

**strict**

**thisValue**

**value** is Language Types, include:

- Primitive values, or
- Object

or Empty

**result** is Language

- Primitive value
- Object

or Reference spe

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，你就得把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 = GetValue®`。

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水分岭。

- 3 | a.x = a = {n2}

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：

- 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）；
- 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水分岭。

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 | \${}

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`\${}`“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：

```
> `${{}}`
'[object Object]'
> eval('{}')
undefined
```

- 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。

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运行的是什么呢？”

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。唯有把这二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东西。

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

**Reference**

**base**

**name**

**strict**

**thisValue**

**value** is Language Types, include:

- Primitive values, or
- Object

or Empty

**result** is Language

- Primitive value
- Object

or Reference spe

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，你就得把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 = GetValue®`。

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水分岭。

- 3 | a.x = a = {n2}

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：

- 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）；
- 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水分岭。

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 | \${}

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`\${}`“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：

```
> `${{}}`
'[object Object]'
> eval('{}')
undefined
```

- 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。

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运行的是什么呢？”

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。唯有把这二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东西。

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

**Reference**

**base**

**name**

**strict**

**thisValue**

**value** is Language Types, include:

- Primitive values, or
- Object

or Empty

**result** is Language

- Primitive value
- Object

or Reference spe

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，你就得把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 = GetValue®`。

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水分岭。

- 3 | a.x = a = {n2}

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：

- 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）；
- 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水分岭。

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 | \${}

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`\${}`“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：

```
> `${{}}`
'[object Object]'
> eval('{}')
undefined
```

- 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。

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运行的是什么呢？”

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。唯有把这二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东西。

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

**Reference**

**base**

**name**

**strict**

**thisValue**

**value** is Language Types, include:

- Primitive values, or
- Object

or Empty

**result** is Language

- Primitive value
- Object

or Reference spe

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，你就得把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=GetValue®。

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水分岭。

- 3 | a.x = a = {n2}

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：

- 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）；
- 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水分岭。

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 | \${}

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`\${}`“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：

```
> `${{}}`
'[object Object]'
> eval('{}')
undefined
```

- 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。

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运行的是什么呢？”

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。唯有把这二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东西。

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

**Reference**

**base**

**name**

**strict**

**thisValue**

**value** is Language Types, include:

- Primitive values, or
- Object

or Empty

**result** is Language

- Primitive value
- Object

or Reference spe

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，你就得把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=GetValue®。

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水分岭。

- 3 | a.x = a = {n2}

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：

- 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）；
  - 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水分岭。

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 | \${}

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`\${}`“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：

```
> `${{}}`
'[object Object]'
> eval('{}')
undefined
```

- 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。

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运行的是什么呢？”

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。唯有把这二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东西。

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

**Reference**

**base**

**name**

**strict**

**thisValue**

**value** is Language Types, include:

- Primitive values, or
- Object

or Empty

**result** is Language

- Primitive value
- Object

or Reference spe

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，你就得把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 = GetValue®`。

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水分岭。

- 3 | a.x = a = {n2}

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：

- 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）；
- 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水分岭。

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 | \${}

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`\${}`“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：

```
> `${{}}`
'[object Object]'
> eval('{}')
undefined
```

- 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。

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运行的是什么呢？”

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。唯有把这二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东西。

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

**Reference**

**base**

**name**

**strict**

**thisValue**

**value** is Language Types, include:

- Primitive values, or
- Object

or Empty

**result** is Language

- Primitive value
- Object

or Reference spe

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，它必须且必然是JavaScript语言理解的类型，也就是typeof()所识别的所有的值。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，你就得把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=GetValue®。

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水分岭。

- 3 | a.x = a = {n2}

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：

- 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）；
- 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水分岭。

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 | \${}

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`\${}`“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：

```
> `${{}}`
'[object Object]'
> eval('{}')
undefined
```

- 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。

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运行的是什么呢？”

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。唯有把这二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东西。

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

**Reference**

**base**

**name**

**strict**

**thisValue**

**value** is Language Types, include:

- Primitive values, or
- Object

or Empty

**result** is Language

- Primitive value
- Object

or Reference spe

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，你就得把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 = GetValue®`。

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水分岭。

- 3 | a.x = a = {n2}

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：

- 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）；
- 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水分岭。

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：

```
> `${{}}`
'[object Object]'
> eval(`${}`)
undefined
```

- 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。

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运行的是什么呢？”

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。唯有把这二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东西。

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

**Reference**

**base**

**name**

**strict**

**thisValue**

**value** is Language Types, include:

- Primitive values, or
- Object

or Empty

**result** is Language

- Primitive value
- Object

or Reference spe

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，你就得把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 = GetValue®`。

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水分岭。

- 3 | a.x = a = {n2}

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：

- 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）；
  - 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水分岭。

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 | \${}

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`\${}`“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：

```
> `${{}}`
'[object Object]'
> eval('{}')
undefined
```

- 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。

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运行的是什么呢？”

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。唯有把这二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东西。

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

**Reference**

**base**

**name**

**strict**

**thisValue**

**value** is Language Types, include:

- Primitive values, or
- Object

or Empty

**result** is Language

- Primitive value
- Object

or Reference spe

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，你就得把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 = GetValue®`。

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水分岭。

- 3 | a.x = a = {n2}

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：

- 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）；
- 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水分岭。

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：

```
> `${{}}`
'[object Object]'
> eval('{}')
undefined
```

- 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。