# JS进阶——你不知道的JS

第一章——作用域是什么

变量储存在哪？

JS——编译语言

编译步骤：

1.分词/词法分析 将字符分解为有意义的代码块（词法单元）

2.解析/语法分析 将词法单元转换成抽象语法树（AST）

3.代码生成 将AST转换为可执行代码

引擎

从头到尾负责整个 JavaScript 程序的编译及执行过程。

• 编译器

引擎的好朋友之一，负责语法分析及代码生成等脏活累活（详见前一节的内容）。

• 作用域

引擎的另一位好朋友，负责收集并维护由所有声明的标识符（变量）组成的一系列查

询，并实施一套非常严格的规则，确定当前执行的代码对这些标识符的访问权限。

执行 var a = = = 2

变量的赋值操作会执行两个动作，首先编译器会在当前作用域中声明一个变量（如

果之前没有声明过），然后在运行时引擎会在作用域中查找该变量，如果能够找到就会对

它赋值。

编译器在编译过程的第二步中生成了代码，引擎执行它时，会通过查找变量 a 来判断它是

否已声明过。(RHS查找和LHS查找)

（LHS 和 RHS 的含义是“赋值操作的左侧或右侧”并不一定意味着就是“= 赋值操作符的左侧或右侧”。赋值操作还有其他几种形式，因此在概念上最 好将其理解为“赋值操作的目标是谁（LHS）”以及“谁是赋值操作的源头（RHS）”）

作用域嵌套

在当前作用域中无法找到某个变量时，引擎就会在外层嵌套的作用域中继续查找，直到找到该变量，

或抵达最外层的作用域（也就是全局作用域）为止。

异常

ReferenceError RHS 查询在所有嵌套的作用域中遍寻不到所需的变量

TypeError 作用域判别成功了，但是对结果的操作是非法或不合理的

第 2 章——词法作用域

简单地说，词法作用域就是定义在词法阶段的作用域。换句话说，词法作用域是由你在写

代码时将变量和块作用域写在哪里来决定的，因此当词法分析器处理代码时会保持作用域

不变（大部分情况下是这样的）。

查找

在多层的嵌套作用域中可以定义同名的 标识符，这叫作“遮蔽效应”（内部的标识符“遮蔽”了外部的标识符）。抛开遮蔽效应，作用域查找始终从运行时所处的最内部作用域开始，逐级向外或者说向上进行，直到遇见第一个匹配的标识符为止。

无论函数在哪里被调用，也无论它如何被调用，它的词法作用域都只由函数被声明时所处

的位置决定。

欺骗词法

eval(..) 函数可以接受一个字符串为参数，并将其中的内容视为好像在书

写时就存在于程序中这个位置的代码。

**function** foo(str, a) {

eval( str ); // 欺骗！

console.log( a, b );

}

**var** b = 2;

foo( "var b = 3;", 1 ); // 1, 3

with 通常被当作重复引用同一个对象中的多个属性的快捷方式，可以不需要重复引用对象

本身。 with 可以将一个没有或有多个属性的对象处理为一个完全隔离的词法作用域，因此这个对

象的属性也会被处理为定义在这个作用域中的词法标识符。

eval(..) 函数如果接受了含有一个或多个声明的代码，就会修改其所处的词法作用域，而

with 声明实际上是根据你传递给它的对象凭空创建了一个全新的词法作用域。

欺骗词法可能会降低代码性能

第 4 章——提升

提升：变量和函数声明从它们在代码中出现的位置被“移动”到了最上面

先声明，后赋值

函数声明会被提升，但是函数表达式却不会被提升

函数声明和变量声明都会被提升，函数会首先被提升，然后才是变量。