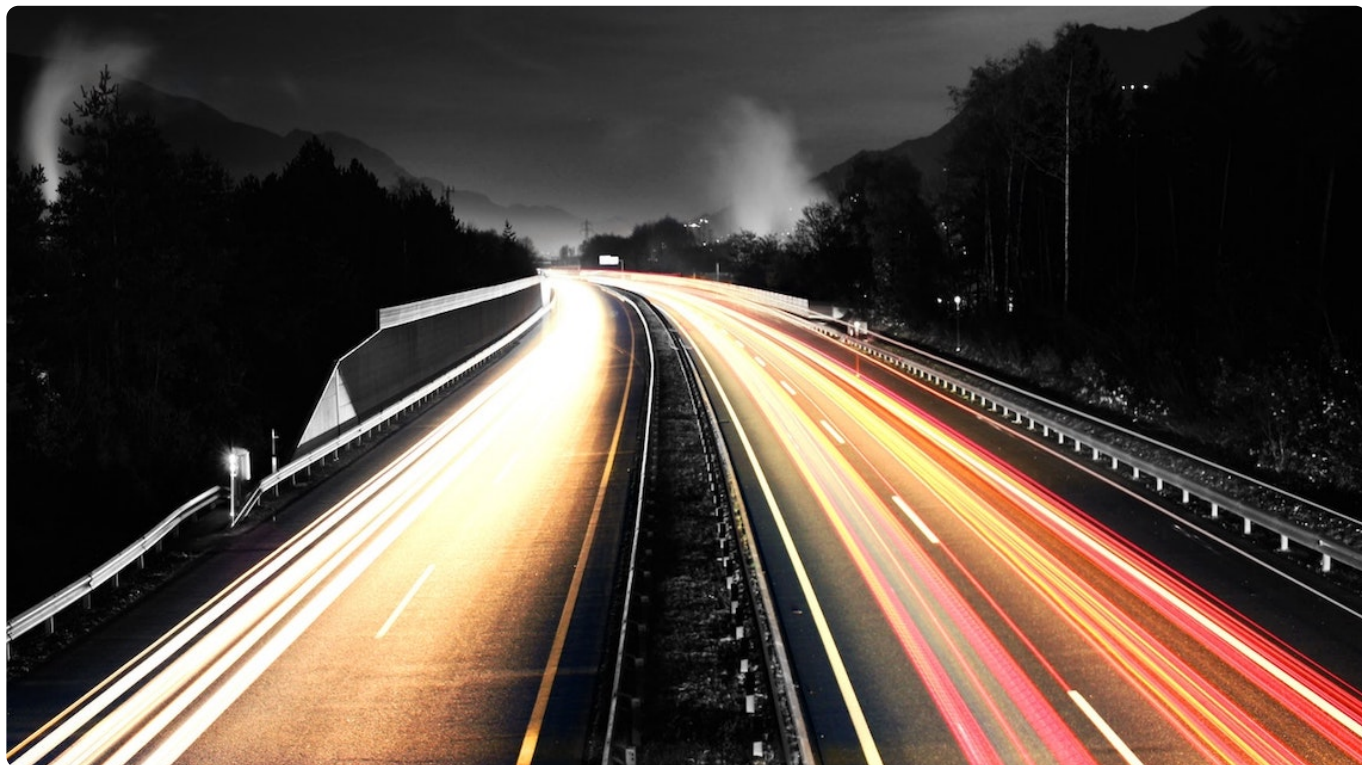


12 | 1 in 1..constructor: 这行代码的结果值，既可能是true，也可能是false

2019-12-11 周爱民

JavaScript核心原理解析

[进入课程 >](#)



讲述：周爱民

时长 16:00 大小 14.66M



你好，我是周爱民。欢迎你回到我的专栏。


如果你听过上一讲，那么你应该知道，接下来我要与你聊的是 JavaScript 的**面向对象系统**。

最早期的 JavaScript 只有一个非常非常弱的对象系统。我用过 JavaScript 1.0，甚至可能还是最早尝试用它在浏览器中写代码的一批程序员，我也寻找和收集过早期的 CEniv 和 ScriptEase，只为了探究它最早的语言特性与 JavaScript 之间的相似之处。

然而，不得不说的是，曾经的 JavaScript 在**面向对象**特性方面，在语法上更像 Java，而在实现上却是谁也不像。

JavaScript 1.0~1.3 中的对象

在 JavaScript 1.0 的时候，对象是不支持继承的。那时的 JavaScript 使用的是称为“**类抄写**”的技术来创建对象，例如：

 复制代码

```
1 function Car() {  
2   this.name = "Car";  
3   this.color = "Red";  
4 }  
5  
6 var x = new Car();
```

关于类抄写以及与此相关的性质，我会在后续的内容中详细讲述。现在，你在这里需要留意的是：在“Car()”这个函数中，事实上该函数是以“类”的身份来声明了一系列的属性（Property）。正是因此，使用`new Car()`来创建的“类的实例”（也就是对象`this`）也就具有了这些属性。

这样的“类 -> 对象”的模型其实是很简单和粗糙的。但 JavaScript 1.0 时代的**对象**就是如此，并且，重要的是，事实上直到现在 JavaScript 的对象仍然如此。ECMAScript 规范明确定义了这样一个概念：

对象是零到多个的属性的集合。

In ECMAScript, an object is a collection of zero or more properties.

你可能还注意到了，JavaScript 1.0 的对象系统是有类的，并且在语义上也是“对象创建自类”。这使得它在表面上“看起来”还是有一些继承性的。例如，一个对象必然继承了它的类所声明的那些性质，也就是“属性”。但是因为这个 1.0 版存在的时间很短，所以后来大多数人都不得记得 JavaScript “**有类，而又不支持类的继承**”这件事情，从而将从 JavaScript 1.1 才开始具有的**原型继承**作为它最主要的面向对象特征。

在这个阶段，JavaScript 中有关全局环境和全局变量的设计也已经成熟了，简单地来说，就是：

1. 向没有声明的变量名赋值，会隐式地创建一个全局变量；
2. 全局变量会被绑定为全局对象（global）的属性。

这样一来，JavaScript 的变量环境（或者全局环境）与对象系统就关联了起来。而接下来，由于 JavaScript 也实现了带有闭包性质的函数，因此“闭包”也成了环境的管理组件。也就是说，闭包与对象都具有实现变量环境的能力。

因此，在这个阶段，JavaScript 提出了“**对象闭包**”与“**函数闭包**”两个概念，并把它们用来实现的环境称为“**域（Scope）**”。这些概念和语言特性，一直支持 JavaScript 走到 1.3 版本，并随着 ECMAScript ed3 确定了下来。

在这个时代，JavaScript 语言的设计与发展还基本是以它的发明者布兰登·艾奇（Brendan Eich）为主导的，JavaScript 的语言特性也处于一个较小的集合中，并且它的应用也主要是以浏览器客户端为主。这时代的 JavaScript 深得早期设计与语言定义的精髓。这些东西，你可以从后来布兰登·艾奇的一个开源项目中读到。这个项目称为 Narcissus，是用 JavaScript 来实现的一个完整的 JavaScript 1.3。在这个项目中，对象和函数所创建的闭包都统一由一个简单的对象表示，称为 scope。例如：

 复制代码

```
1 scope = {  
2   object: < 创建本闭包的对象或函数 >,  
3   parent: < 父级的 scope >  
4 }
```

因此，所谓“**使用 with 语句创建一个对象闭包**”就简单地被实现为：

 复制代码

```
1 // code from $(narcissus)/src/jsexec.js  
2 ...  
3 // 向 x 所代表的 scope-chain 表尾加入一个新的 scope  
4 x.scope = {object: t, parent: x.scope};  
5 try {  
6   // n.body 是 with 语句中执行的语句块  
7   execute(n.body, x); // 指在该闭包（链）`x`中执行上述语句
```

```
8 }  
9 finally {  
10     x.scope = x.scope.parent; // 移除链尾的一个 scope  
11 }
```

可见 JavaScript 1.3 时代的执行环境，其实就是一个闭包链的管理。而且这种闭包既可以是对象的，也可以是函数的。尽管在静态语法说明或描述时，它们被称为**作用域或域**（Scope），或者在动态环境中它们被称为**上下文**（Context），但在本质上，它们是同样的一堆东西。

综合来看，JavaScript 中的对象本质上是**属性集**，这可以视为一个**键值列表**，而对象继承是由这样的列表构成的、称为原型的链。另一方面，执行的上下文就是函数或全局的变量表，这同样可以表达为一个键值列表，而执行环境也可以视为一个由该键值列表构成的链。

于是，在 JavaScript 1.3，以及 ECMAScript ed3 的整个时代，这门语言仅仅依赖**键值列表**和**基于它们的链**实现并完善了它最初的设计。

属性访问与可见性

但是从一开始，JavaScript 就有一个东西没有说清楚，那就是属性名的可见性。

这种可见性在 OOP（面向对象编程）中有专门的、明确的说法，但在早期的 JavaScript 中，它可以简单地理解为“**一个属性是否能用 for...in 语句列举出来**”。如果它可以被列举，那么就是可见的，否则就称为隐藏的。

你知道，任何对象都有“constructor”这个属性，缺省指向创建它的构造器函数，并且它应当是隐藏的属性。但是在早期的 JavaScript 中，这个属性如何隐藏，却是没有规范来约定的。例如在 JScript 中，它就是一个特殊名字，只要是这个名字，就隐藏；而在 SpiderMonkey 中，当用户重写这个属性后，它就变成了可见的。

后来 ECMAScript 就约定了所谓的“**属性的性质**（attributes）”这样的东西，也就是我们现在知道的**可写性**、**可列举性**（可见性）和**可配置性**。ECMAScript 约定：

“constructor”缺省是一个不可列举的属性；

使用赋值表达式添加属性时，属性的可列举性缺省为 true。

这样一来，“constructor”在可见性（这里是指可列举性）上的行为就变得可预期了。

类似于此的，ECMAScript 约定了读写属性的方法，以及在属性中访问、操作性质的全部规则，并统一使用所谓“属性描述符”来管理这些规则。于是，这使得 ECMAScript 规范进入了 5.x 时代。相较于早期的 3.x，这个版本的 ECMAScript 规范并没有太多的改变，只是从语言概念层面上实现了“大一统”，所有浏览器厂商，以及引擎的开发都遵循了这些规则，为后续的 JavaScript 大爆发——ECMAScript 6 的发布铺平了道路。

到目前为止，JavaScript 中的对象仍然是简单的、原始的、使用 JavaScript 1.x 时代的基础设计的原型继承。而每一个对象，仍然都只是简简单单的一个所谓的“属性包”。

从原型中继承来的属性

对于绝大多数对象来说，“constructor”是从它的原型继承来的一个属性，这有别于它“自有的 (Own)”属性。在原型继承中，在子类实例重写属性时，实际发生的行为是“**在子类实例的自有属性表中添加一个新项**”。这并不改变原型中相同属性名的值，但子类实例中的**属性性质**以及**值**覆盖了原型中的。这是原型继承——几乎是公开的——所有的秘密所在。

在使用原型继承来的属性时，有两种可能的行为，这取决于属性的具体性质——属性描述符的类型。

1. 如果是**数据描述符** (d)，那么 `d.value` 总是指向这个数据的值本身；
2. 如果是**存取描述符**，那么 `d.get()` 和 `d.set()` 将分别指向属性的存取方法。

并且，如果是存取描述符，那么存取方法 (get/setter) 并不一定关联到数据，也并不一定是数据的置值或取值。某些情况下，存取方法可能会用作特殊的用途，例如模拟在 VBScript 中常常出现的“无括号的方法调用”。例如：

 复制代码

```
1 excel = Object.defineProperty(new Object, 'Exit', {
2   get() {
3     process.exit();
4   }
5 });
6
7 // 类似 JScript/VBScript 中的 ActiveObject 组件的调用方法
```



```
8 excel.Exit;
```

当用户不使用属性赋值或`defineProperty()`等方法来添加自有的属性时，属性访问会（默认地）上溯原型链直到找到指定属性。这一定程度上成就了“包装类”这一特殊的语言特性。

所谓“**包装类**”是 JavaScript 从 Java 借鉴来的特性之一，它使得用户代码可以用标准的面向对象方法来访问普通的值类型数据。于是，所谓“一切都是对象”就在眨眼间变成了现实。例如，下面这个示例中使用的字符串常量 `x`：

```
1 x = "abc";
2 console.log(x.toString());
```

[复制代码](#)

当在使用 `x.toString()` 时，JavaScript 会自动将“值类型的字符串（“abc”）”通过包装类变成一个字符串对象。这类似于执行下面的代码：

```
1 console.log(Object(x).toString());
```

[复制代码](#)

这个包装的过程发生于**函数调用运算“()”**的处理过程中，或者将“`x.toString`”作为整体来处理的过程中。也就是说，仅仅是“对象属性存取”这个行为本身，并不会触发一个普通“值类型数据”向它的包装类型转换。

除了`Undefined`，基本类型中的所有值类型数据都有自己的包装类，包括符号，又或者布尔值。这使得这些值类型的数据也可以具有与之对应的包装类的原型属性或方法。这些属性与方法自己引用自原型，而不是自有数据。很显然的，值类型数据本身并不是对象，因此也不可能拥有自有的属性表。

字面量与标识符

通常情况下，开发人员会将标识符直接称为**名字**（在 ECMAScript 规范中，它的全称是“标识符名字（`IdentifierName`）”），而**字面量**是一个数据的文本表示。显然，通常


标识符就用作后者的名字标识。对于这两种东西，在 ECMAScript 中的处理机制并不太一样，并且在文本解析阶段就会把二者区分开来。例如：

```
1 // var x = 1;
2 1;
3 x;
```

 复制代码

其中“1”是字面量值，JavaScript 会直接处理它；而 x 是一个标识符，就需要建立一个“引用”来处理了。但是接下来，如果是代码（假设下面的代码是成立的）：

```
1 1.toString
```

 复制代码

那么它作为“整体”就需要被创建为一个引用，以作为后续计算的操作数（取成员值，或仅是引用该成员）。是的，就它们同是“引用”这一事实而言，“1.toString”与“x”在引擎级别有些类似。

然而在数字字面量中，“1.xxxxx”这样的语法是有含义的。它是浮点数的表示法。所以“1.toString”这样的语法在 JavaScript 中会报错，这个错误来自于浮点数的字面量解析过程，而不是“.”作为存取运算符的处理过程。在 JavaScript 中，浮点数的小位数是可以为空的，因此“1.”和“1.0”将作为相同的浮点数被解析出来。

既然“1.”表示的是浮点数，那么“1...constructor”表示的就是该浮点数字面量的“.constructor”属性。

现在我想你已经看出来了，标题中的：

```
1 1 in 1..constructor
```

 复制代码

其实是一个表达式。在语义上，它与如下的表达式是等义的：

```
1 # 检查对象“constructor”是否有属性名“1”
2 > 1 in Object(1.0).constructor
3 false
4
5 # (同上)
6 > 1 in 1..constructor
7 false
```

[复制代码](#)

属性存取的不确定性

除了**存取器**（get/setter）带来的不确定性之外，JavaScript 的属性存取结果还受到**原型继承（链）**的影响。上例中的表达式值并不恒为`false`，例如：

```
1 # 修改原型链中的对象
2 > Number[1] = true; // or anything
3
4
5 # 影响到上例中表达式的结果
6 > 1 in 1..constructor
7 true
```

[复制代码](#)

因为`Object(1.)`意味着将数字“1.0”封装成它对应的包装类的一个对象实例（x）。例如：

```
1 x = new Number(1.0);
```

[复制代码](#)

而“`x.constructor`”不是自有属性，由于`x`是“`Number()`”这个类 / 构造器的子类实例，因此该属性实际继承自原型链上的“`Number.prototype.constructor`”这个属性。然后，在缺省情况下，“`aFunction.prototype.constructor`”指向这个函数自身。

也就是说，“`Number.prototype.constructor`”与“`1...constructor`”相同，且都指向`Number()`自身。

所以上面的示例中，当我们添加了“`Number[1]`”这个下标属性之后，标题中表达式的值就变了。

知识回顾

这一讲的标题看起来像是其他语言中的循环或迭代，又或者在代码文本上看起来像是一个范围检查（语义上看起来像是“1 在某个 $1 \dots n$ ”的范围中）。但事实上，它不仅包含了 JavaScript 中从对象成员存取这样的基础话题，还一直延伸到了**包装类**这样的复杂概念的全部知识。

当然，重要的是，源于 JavaScript 中面向对象系统的独特设计，它的对象属性存取结果总是不确定的。

如果属性不是自有的，那么它的值就是原型决定的；

当属性是存取方法的，那么它的值就是求值决定的。

思考题

虽然这一讲没有太深入的内容，但是有两道练习题留给大家，非常烧脑：

1. 试述表达式 `[]` 的求值过程。
2. 在上述表达式中加上符号 `“+-*/”` 并确保结果可作为表达式求值。

NOTE：题目 1 是一个空数组的“单值表达式”，当它作为表达式来处理时，请问它是如何求值的（你得先想想它的“值”是多少）。

NOTE：题目 2 的意思，就是如何把这些字符组合在一起，仍然是一个可求值的表达式。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。


点击参与 

打卡 46 天，彻底搞定 JavaScript



扫一扫参与小程序打卡



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

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 11 | throw 1;: 它在“最简单语法榜”上排名第三

下一篇 13 | new X: 从构造器到类，为你揭密对象构造的全程

精选留言 (3)

 写留言



kittyE

2019-12-13

1. 我理解，`[]` 作为单值表达式，要`GetValue(v)`，但为啥结果是 `[]`，不太明白，ecma关于`GetValue`的描述，感觉好复杂。
2. `[]*[]/++[[]][+[]]-[+[]]` 我随便写了一个 还真的能有值，不知道这样理解对不对，求老师解惑

展开 

作者回复: 第2题你的理解是对的，不过表达式可以再简一些。^^.

关于第一个问题，思考方向不是`GetValue`，而是`toPrimitive`。还有，它的结果不是`[]`，它的求值结果是`0`。





Astrogladiator-埃蒂...

2019-12-11

试述表达式[]的求值过程。

对照<http://www.ecma-international.org/ecma-262/5.1/#sec-9.1>

<http://www.ecma-international.org/ecma-262/5.1/#sec-8.12.8>

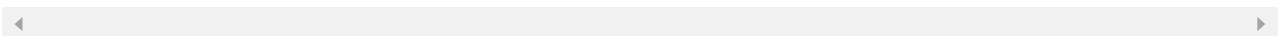
step1: []不是一个原始类型，需要转化成原始类型求值

step2: 这个隐式转换是通过宿主对象中的[[DefaultValue]]方法来获取默认值...

展开 ∨

作者回复: 第1个问题, 这样解释是不对的。[[DefaultValue]]是用在那些值类型的包装对象上的, 例如5和new Number(5)之间的关系。而preferredType是另外一个问题, 涉及JavaScript对“预期转换目标类型”的管理, 不同的运算之间还不同(但都与具体的运算操作有关), 与当前这个问题却没有太大的关系。

第2个问题的意思, 是如何使一个表达式里面只出现 “+ - * /” 和 “[]”, 并且表达式还可以通过语法检测并计算求值。



许童童

2019-12-11

老师讲得非常好, JavaScript中的面向对象设计确实很独特, 早期我们还称其为基于对象, 不过随着我们对JavaScript了解的深入, 现在都已经改口了。对象存取的结果是面向对象运行时中结果的体现, 如果属性不是自有的, 就由原型决定, 如果属性是存取方法, 就由方法求值决定。另外, 属性描述符有两种主要形式: 数据描述符和存取描述符。

展开 ∨

