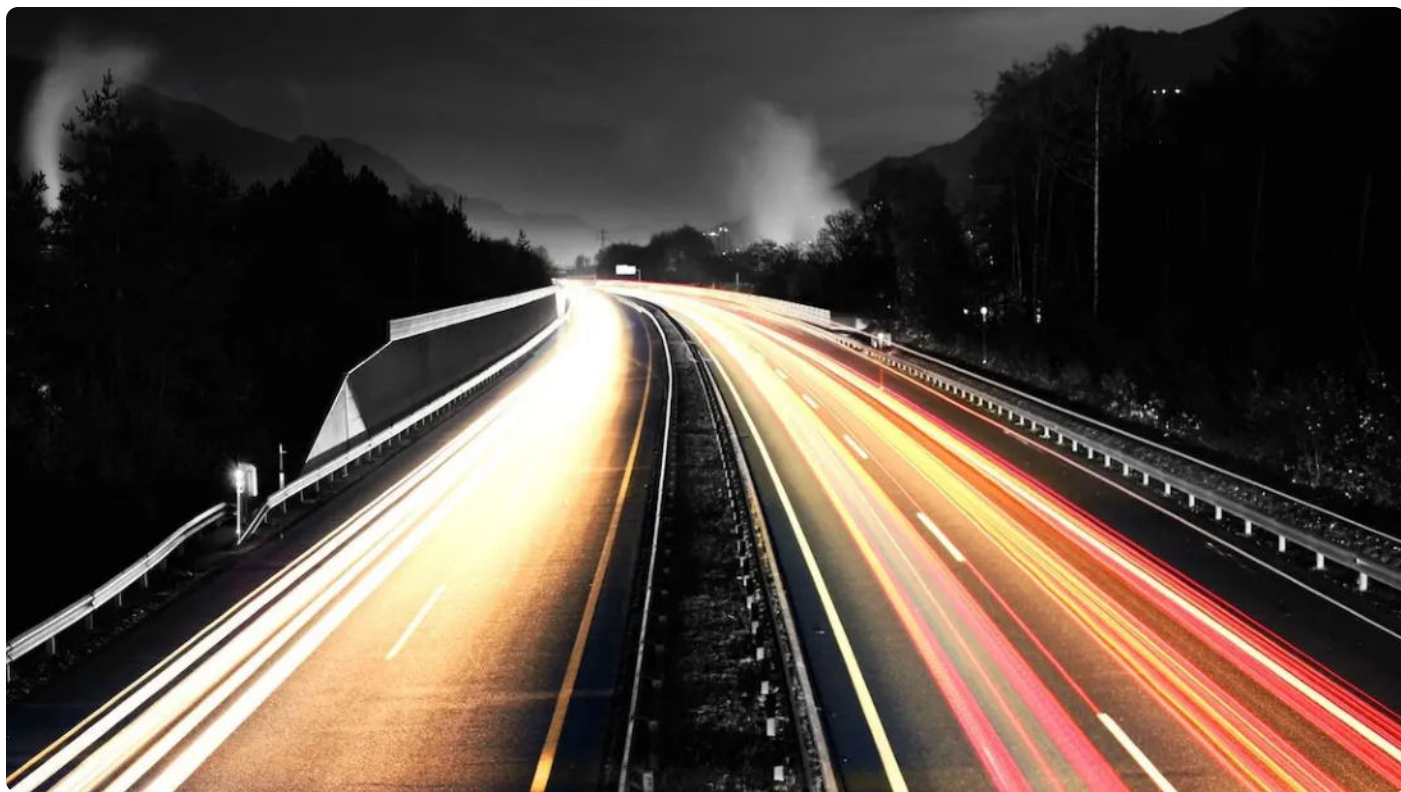


12 | 1 in 1..constructor: 这行代码的结果，既可能是true，也可能是false

2019-12-11 周爱民

《JavaScript核心原理解析》

课程介绍 >



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，就是“在一个函数中将this引用添加属性，并且使用new运算来创建对象实例”，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的一个概念：

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忘记了 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，它包括“object”和“parent”两个成员，分别表示本闭包的對象，以及父一级的作用域。例如：

 复制代码

```
1 scope = {
2   object: <创建本闭包的對象或函数>,
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“向既有的作用域链尾加入一个新的 scope”。

 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，以及引擎的开发者都遵循了这些规则，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”。

 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`，它的值是`"abc"`：

```
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，使用函数 `Object()` 来“将这个值显式地转换为对象”。

```
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（例如作为一个 ECMAScript 规范引用类型来处理的过程）。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。



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

[复制代码](#)

比如在这个例子中，如果其中“1”是字面量值，JavaScript 会直接处理它；而 x 是一个标识符（哪怕它只是一个值类型数据的变量名），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，“1..constructor”与“Object(1.0).constructor”这样的表达式是等义的，且它们的使用效果也是一样的。



```
1 # 检查对象“constructor”是否有属性名“1”  
2 > 1 in Object(1.0).constructor
```

[复制代码](#)


```
3 false
4
5 # (同上)
6 > 1 in 1..constructor
7 false
```

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，例如我们给 Number 加一个下标值为 1 的属性（我们不用管这个属性的值是什么），那么标题中的表达式“1 in 1...constructor”的值就会是true了。

 复制代码

```
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
```

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），我们假设这个对象是 x，那么“1...constructor”也就指向 x.constructor。

 复制代码

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

而“x.constructor”不是自有属性，并且，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...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。

分享给需要的人，Ta购买本课程，你将得 20 元

 生成海报并分享

 赞 3

 提建议



学习推荐

JVM + NIO + Spring

各大厂面试题及知识点详解

限时免费



精选留言 (12)

写留言



Smallfly

2020-02-08

1. [] 的求值过程

一开始没明白题目的意思, 看到留言区的提示才理解, 题目考察的是 JS 的类型转换, [] 属于对象类型, 对象类型到值类型的转换过程是什么样的?

对象转值类型的规范过程称为: ToPrimitive

分为三个步骤:

1. 判断对象是否实现 [Symbol.toPrimitive] 属性, 如果实现调用它, 并判断返回值是否为值类型, 如果不是, 执行下一步。
2. 如果转换类型为 string, 依次尝试调用 toString() 和 valueOf() 方法, 如果 toString() 存在, 并正确返回值类型就不会执行 valueOf()。
3. 如果转换类型为 number/default, 依次尝试调用 valueOf() 和 toString(), 如果 valueOf() 存在, 并正确返回值类型就不会执行 toString()。

数组默认没有实现 [Symbol.toPrimitive] 属性, 因此需要考察 2、3 两步。



[] + '' 表达式为 string 转换，会触发调用 toString() 方法，结果为空字符，等价于 '' + '' 结果为 ''。

+[] 表达式是 number 转换，会先触发调用 valueOf() 方法，该方法返回的是空数组本身，它不属于值类型，因此会再尝试调用 toString() 方法，返回空字符，+'' 结果为 0；

作者回复: 加分加分。呵呵~ ^^.



👍 25



墨灵

2020-03-20

有个小问题，当一个函数使用了函数外部的变量时，这种情况就能称为“闭包”吗？

...

```
// 函数f只是使用了全局的变量x
```

```
let x = 0;
```

```
function f (y) {
```

```
  return x + y;
```

```
}
```

```
// z引用g函数内的匿名函数，而匿名函数使用了g的参数x，而造成g的函数作用域无法释放。
```

```
function g (x) {
```

```
  return (y) => x + y;
```

```
}
```

```
const z = g(0);
```

...

在第一种情况，f函数调用之后就可以释放作用域，而第二种情况，无论z调用多少次，只要z不指向别一个值，函数g的作用域就不会释放。这就是我所理解的函数闭包，但对象闭包是什么样子的？

作者回复: 对象闭包是在非严格模式中才能用的，例如：

...

```
let f, obj = new Object;
```

```
with (obj) f = function() {};
```

...

这种情况下，with语句为对象obj创建了一个块级作用域，这个作用域（以及它的链）就被作为一个对象闭包放在函数f的作用域链上了。



在ES6之后已经不再使用对象闭包这样的说法，统一用块级作用域和环境来描述这些语法效果了。在ES6之前，由于函数的作用域（的实例）被称为闭包，所以对象的with作用域（的实例）也就称为对象闭包。

最后，事实上JavaScript的全局global也是一个对象闭包。它总是在其它所有闭包（作用域链）的顶端。

再再再补充一下，闭包跟作用域其实是不完全相同的。作用域通常是语法所对应的块，是静态概念的，而闭包是运行期才使用的概念，函数被调用一次就有一个闭包出现，但函数自身其实只有一个作用域。——所以，看起来作用域像是“类”，而闭包像是“对象”，闭包是作用域的“实例”。

并且，确实的，在ECMAScript规范中，闭包就是一个作用域“实例化”的结果。——并且“实例化”是在“函数调用”时实时地创建和发生的，是动态的、运行期的概念。



👍 8



墨灵

2020-04-09

An object is a collection of properties and has a single prototype object. The prototype may be the null value.

昨天查了一下，ECMAScript规范更新对object的描述了。



👍 4



青史成灰

2020-01-12

老师，关于这句话有个疑问：“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”

只是前半句好理解，但是后半句“对象属性存取这个行为本身”，这个对象是发生包装转化后的对象吗？如果是，那怎么感觉就变成了“先有鸡还是先有蛋”的问题了。。。如果不是，那么原始类型不存在属性存取这一说法啊

作者回复: 关键确实就在“对象属性存取这个行为本身”。

原始类型（中的值类型）的对象属性存取行为，例如“`true.toString`”会发生什么呢？它仍然是一次有效的属性存取，但它的结果（Result）并不会被立即求值。之前我们说过了，必须等到决定它是作为rhs/lhs之后，才能确定它是用来“求值”，还是只是“作为一个引用”，对不对？

那么，当得到操作`true.toString`的结果（Result）之后，在决定下一个可能的操作之前，它是一个什么状态呢？——这种情况下，它是作为一个“引用（规范类型）”来传递的。考虑到“引用（规范类



型)”作为一个原始语言（例如C）中的结构/记录类型，那么它的ref.base域存放的，将是“值true”，而ref.name域中存放的，将是属性名“toString”。

所以你看，在这个阶段中，“包装（boxing）”这个行其实并没有发生。true还是true值，并没有“变成”Object(true)，对不对？

所以说，确实存在一个“将对象属性存取这个行为（的结果）——作为一个整体”的阶段，这个存取行为并没有发生包装。但是，如果如下发生后一步的行为（也就是“作为整体来处理的过程中），那么，“包装”就会发生了。例如，GetValue(ref)，那么就会先将ref.base中的值转换成对象；又例如，true.toString()，就会先将ref.true转换为对象然后作为this值传入toString()。

所以，“包装（亦即是‘转换为对象’）”这个行为，其实是发生在`GetValue()`这个内部操作`或`()这个运算符`等等这样的运算过程中的。

所以回到最开始的，总之，“对象属性存取”这个行为本身，就是还没有触发“包装”。但它得到了包装要用的材料，亦即是ref.base和ref.name，不过还得需要“下一步”的具体操作，才能决定“包装是否会发生”。



👍 3



K4SHIFZ

2020-03-28

抱歉老师我杠一下，自动分号插入在规范11.9章：When, as the source text is parsed from left to right, a token (called the offending token) is encountered that is not allowed by any production of the grammar, then a semicolon is automatically inserted before the offending token if one or more of the following conditions is true:

The offending token is separated from the previous token by at least one Line Terminator.

The offending token is }.

...

它说是before the offending token，在}之前插入，所以应该变为{;}+{}，而不是{};+{}。这么理解对吗？虽然分号插在哪，不影响引擎会首先作为语句执行第一个{}

作者回复: 是这样的，我们通常讨论ASI的时候，是有两种语境的，一种是“尽量少写;号”，另一种是“尽量所有语句都写;号”。

ECMAScript在讨论ASI的时候，是直接认为第一种语境的。也就是说，ASI是“为了那些不想写分号的人准备的工具”。这种情况下，才会有了它的第一条规则（这个来自 Isaac Schlueter 的描述）：



> 在一个 \n 字符总是一个语句的结尾总是“自动加上 ;号”

但是我正好是“总是尽可能写分号派”的。呵呵，真的。因为我是从Pascal语言过来的，所以多数情况下我会为每个语句后面加一个;号。这带来了一种习惯，也就是“语句总是以";"号结束”。比如语句：

```
if (true) {  
}; // <- 事实上这个分号可以不写
```

如果你遵循语句以";"号结束的原则，那么你可以规避ASI的效果，因为任何情况下都有正确的";"号。但是另外的问题是，多数情况下我们在大括号后面是不用";"号的，因为大括号本身就是块语句，块语句最后的"}"本来也是语句结束符，所以一般我们会写成“{...}”而不是“{...};”。

因为是在这种语境讨论ASI，所以我才会说，可以将"};"后面的这个分号写出来，从而看到：

```
> {}+{}
```

被解析成了

```
> {}+{}
```

这个样子。OK，好吧，无论如何，我得承认，ASI是为了解决“尽量使用\n而不是使用;来结束语句”的问题的。所以尽管都是";"号的问题，但我上面讨论的并不是在ECMAScript所说的那个ASI，我这样归为ASI的问题并不正确。

共 2 条评论 >

👍 1



红白十万一只

2020-03-01

关于[]求值过程无非是隐式类型转换，隐式调用toString
来看看{}+{}这道题

可能有两种结果

1, "[Object Object] [Object Object]"

2, NAN

首先一种理解，代码块{}，而不是对象

符合Firefox的结果

```
{};
```

```
+{}
```

```
+{}.toString()
```



```
+ "[Object Object]"  
Number("[Object Object]")  
NAN
```

第二种把{}当场一个字面量

结果也就是 "[Object Object][Object Object]"

符合谷歌结果

查了ES规范，{}什么时候是字面量，什么时候是代码块

1, {}前面有运算符时，当成字面量

2, {}后面有；或隐式插入；时当场代码块

老师能更详细讲解一下{}，什么时候是代码块，什么时候是字面量么

作者回复: 这个涉及到“语句的语法规则”，也就是JS解析时处理“识别语句”的问题。

比较简单的说法是，

1. 某些情况下回车和上一语句的自然终结可以作为语句结束符。
2. 除了上一特例之外，分号和文末结束符（EOF）将被理解为语句结束符。
3. 任何情况下，从语句开始解析整个文本块。

然而比较麻烦的就是第1条规则。因为它意味着JavaScript的一个称为“自动分号插入（ASI）”规则生效，这就是有些人不赞同写行末分号的原因。——事实上是因为JavaScript读到那些特定位置的回车符或自然终结，然后自动插入了分号。

以{}为例，如果它正好在上一行的结尾之后（例1），或者是一段代码文本的最开始，那么它就被理解为语句（例2）。如下两例：

```
...
```

```
// 例1: 类声明语句的最后一个}`被理解为上一语句的结束。所以返回是一个空语句的值: undefined  
x = eval(`class foo {}`)
```

```
// 例2: eval执行的代码块以语句开始解析。所以这里将有一个空语句和一个单值表达式语句，而返回值为1
```

```
x = eval(`${1}`)  
...
```



1



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

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`。



1



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

step3: 一般在没有指定`preferredType`的情况下，会隐式转换为`number`类型的默认值

step4: `[]`默认值为`0`

可以这么理解？这个`preferredType`在什么设置？

在上述表达式中加上符号“`+-* /`”并确保结果可作为表达式求值。

这个是不是只要保证表达式中是对象或者`number`类型或者设置了`preferredType`的其他`!类型`（除了`null`, `undefined`, `NaN`）

作者回复: 第1个问题，这样解释是不对的。`[[DefaultValue]]`是用于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。



1



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。





1

**言川**

2021-07-19

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

其中第一个 'constructor' 拼错了

作者回复: 谢谢🙏已提交编辑老师处理

**仿生狮子**

2020-12-23

第二题，感觉像是在玩 JSFxcK hhhhha~~，试验发现，`[]+[]` 得空字符串，`+[]` 得 0，`[]**[]` 得 1，`++[]**[]+[]` 得 2，以此类推可获得任何数字，比如“1023”可表示为“2 的 10 次方减 1”，即 `++[]**[]+[]**([]+[]+([]**[])+(+[]))-1`。

JSFxcK 的逻辑要复杂一些，先从字符串 `undefined` 拿到 `find`，再由 `{}.find + []` 拿到更多字母，以此类推，拼出 `constructor` 拿到大写字母 `S`，在拼字符串 `toString`，可获得任何小写字母。有了数字和许多字母，就（几乎）可以愉快的编程了。（不过题目没有给非运算符，所以字符串 `true` 和 `false` 拿不到，在前几步就挂了。🔙 BACK)

**Elmer**

2020-01-03

我觉得两题的本质都是再说对象如何转换为值类型。`[]` 的求值过程在于 `[]` 所处的表达式环境需要 `number` 还是 `string`，然后执行 `array.prototype` 上对应的方法转换。题二中 `+-*/` 都是需要 `number`，所以只要不出现 `0/0` 的情况即可。

作者回复: 题二其实是没有标准答案的。不过多做一点提示，就是 `+/-` 其实也是一正值和负值运算符，不一定非得当成加减号来用。正值和负值运算符一样也会导致类型转换。

