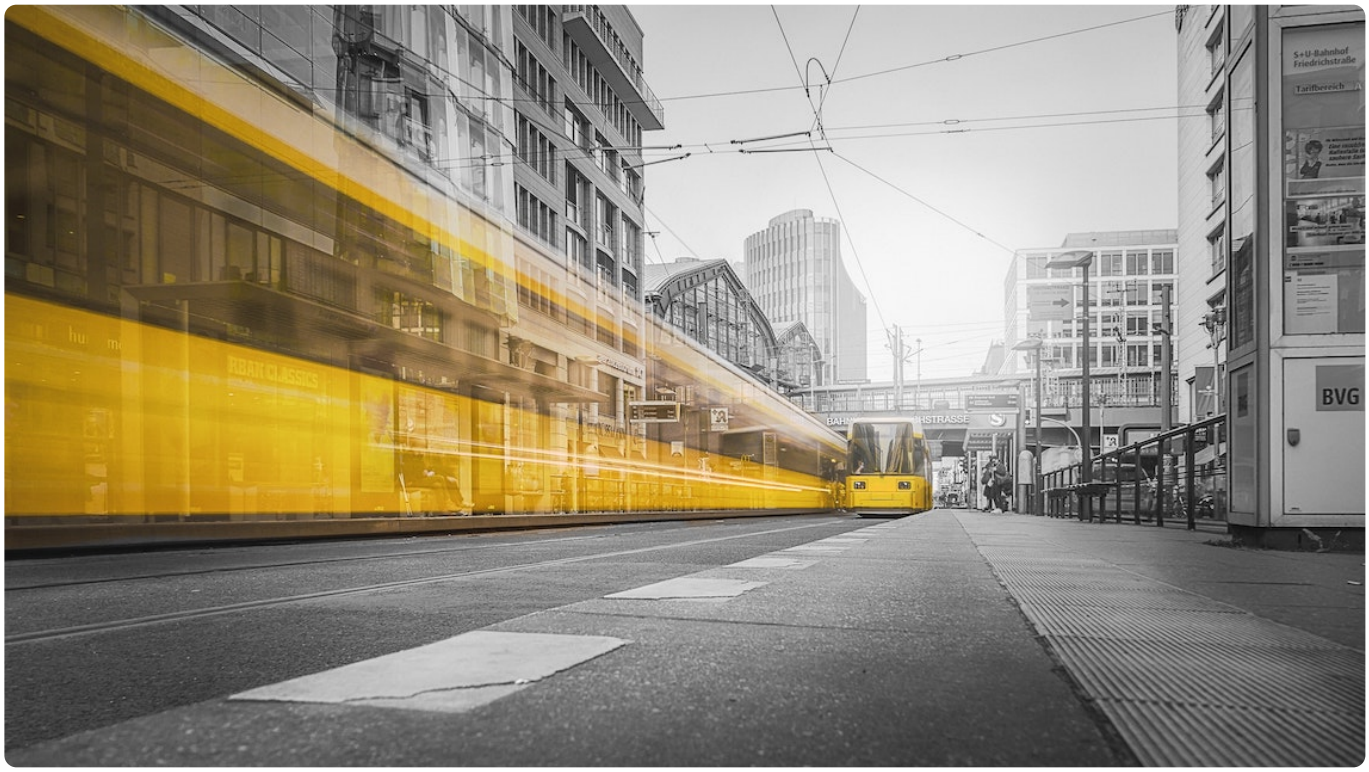
01 | delete 0：JavaScript中到底有什么是可以销毁的

2019-11-11 周爱民

JavaScript核心原理解析 进入课程 >



讲述：周爱民
时长 18:10　大小 16.65M

你好，我是周爱民，感谢你来听我的专栏。

今天是这个系列的第一讲，我将从 JavaScript 中最不起眼的、使用率最低的一个运算——delete 讲起。

你知道，JavaScript 是一门面向对象的语言。它很早就支持了 delete 运算，这是一个元老级的语言特性。但细追究起来，delete 其实是从 JavaScript 1.2 中才开始有的，与它一同出现的，是对象和数组的字面量语法。

有趣的是，JavaScript 中最具恶名的 typeof 运算其实是在 1.1 版本中提供的，比 delete 运算其实还要早。这里提及 typeof 这个声名狼籍的运算符，主要是因为 delete 的操作与类型的识别其实是相关的。

习惯中的“引用”

早期的 JavaScript 在推广时，仍然采用传统的数据类型的分类方法，也就是说，它宣称自己同时支持值类型和引用类型的数据，并且，所谓值类型中的字符串是按照引用来赋值和传递引用（而不是传递值）的。这些都是当时“开发人员的概念集”中已经有的、容易理解的知识，不需要特别地解释。

但是什么是引用类型呢?

在这件事上，JavaScript 偷了个懒，它强行定义了“Object 和 Function 就是引用类型”。这样一来，引用类型和值类型就给开发人员讲清楚了，对象和函数呢，也就可以理解了：它们按引用来传递和使用。

绝大多数情况下，这样解释起来是行得通的。但是到了 delete 运算这里，就不行。

因为这样一来，`delete 0`就是删除一个值，而`delete x`就既可能是删除一个值，也可能是删除一个引用。然而，当时 JavaScript 又同时约定：那些在 global 对象上声明的属性，就“等同于”全局变量。于是，这就带来了第三个问题：`delete x`还可能是删除一个 global 对象上的属性。而它在执行这个操作的时候，看起来却像是一个全局变量（的名字）。

这中间有哪些细节的区别呢?

delete 这个运算的表面意思，是该运算试图销毁某种东西。然而，`delete 0`中的 0 是一个具体的、字面量表示的“值”。一个字面量值“0”如何在现实世界中销毁呢？假定它销毁了，那是不是说，在这个语言当前的运行环境中，就不能使用 0 这个值了呢？显然，这不合理。

所以，JavaScript 认为“**所有删除值的 delete 就直接返回 true**”，表明该行为过程中没有异常。很不幸，JavaScript 1.2 的时代并没有结构化异常处理（即 try...catch 语句）。所以，通过函数调用中返回 true 来表明“没有异常”，其实是很常规的做法。

然而，返回值只表明执行过程中没有异常，但实际的执行行为是“什么也没发生”。你显然不可能真的将“0”从执行系统中清理出去。

那么接下来，就还剩下删除变量和删除属性。由于全局变量实际上是通过全局对象的属性来实现的，因此删除变量也就存在识别这两种行为的必要性。例如：

```
1 delete x
```

复制代码

这行代码究竟是在删除什么呢？出于 JavaScript 是动态语言这项特性，所以从根本上来说，我们是没有办法在语法分析期来判断x的性质的。所以现在，需要有一种方法在运行期来标识x的性质，以便进一步地处理它。

这就导致了一种新的“引用”类型呼之欲出。

到底在删除什么？

探索工作往往如此，是所谓“进五退一”，甚至是“进五退四”。在今后的专栏文章中，你往往会看到，我在碰触到一种新东西的时候会竭力向前，但随后又后退好几步，再来讨论一些更基础层面的东西。这是因为如果不把这些基础概念说得清楚明白，那么往前冲的那几步常常就被带偏了方向。

一如现在这个问题：`delete 0`到底是在删除什么？

对于一门编译型语言来说，所谓“0”，就是上面所述的一个值，它可以是基础值（Primitive values），也可以是数值类型。但如果将这个问题上升到编译之前的、所谓语法分析的阶段，那么“0”就会被称为一个记号（Tokens）。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”。

delete 不仅仅是要操作 0 或 x 这样的单个记号或标识符（例如变量）。因为这个语法实际起作用的是一个对象的属性，也就是“删除对象的成员”。那么它真正需要的语法其实是：

```
1 delete obj.x
```

复制代码

只不过因为全局对象的成员可以用全局变量的形式来存取，所以它才有了

```
1 delete x
```

这样的语法语义而已。所以，这正好将你之前所认识的倒转过来，是删除 x 这个成员，而不是删除 x 这个值。不过终归有一点是没错的：既然没办法表达异常，而 delete 0 又不产生异常，那么它自然就该返回 true。

然而，如果你理解了`delete obj.x`，那么就一定会想到：`obj.x`既不是之前说过的引用类型，也不是之前说过的值类型，它与`typeof(x)`识别的所有类型都无关。因为，它是一个表达式。

所以，delelet 这个操作的正式语法设计并不是“删除某个东西”，而是“删除一个表达式的结果”：

```
1 delete UnaryExpression
```

表达式的结果是什么?

在 JavaScript 中表达式是一个很独特的东西，所有一切表达式运算的终极目的都是为了得到一个值，例如字符串。然后再用另外一些操作将这个值输出出来，例如变成网页中的一个元素（element）。这是 JavaScript 语言创生的原力，也是它的基础设计。也只因为有了这种设计，它才变得既像面向对象的，又像函数式语言的样子。

表达式的执行特性，以及表达式与语句的关系等等细节，回头我放在第二阶段的内容中讲给你听。现在我们只需要关注一个要点，表达式计算的结果到底是什么？因为就像上面所说的，这个结果，才是`delete`这个操作要删除的东西。

在 JavaScript 中，有两个东西可以被执行并存在执行结果值（Result），包括语句和表达式。比如你用`eval()`来执行一个字符串，那么事实上，你执行的是一个语句，并返回了语

句的值；而如果你使用一对括号来强制一个表达式执行，那么这个括号运算得到的，就是这个表达式的值。

表达式的值，在 ECMAScript 的规范中，称为“引用”。

这是一种称为“规范类型”的东西。

规范中的“引用”

事实上这个概念出现得也很早。从 JavaScript 1.3 开始，ECMAScript 规范就在语言定义的层面，正式地将上述的天坑补起来，推出了上面说到的这个“（真正的）引用类型”。

但是，由于这个时候规范的影响力在开发人员中并不那么大，所以开发人员还是习惯性地将对象和函数称为引用，而其它类型就称为值，并且继续按照传统的理解来解释 JavaScript 中对数据的处理。

这种情况下，一个引用只是在语法层面上表达“它是对某种语法元素的引用”，而与在执行层面的值处理或引用处理没关系。于是，下面这行简短的语句

```
1 delete 0
```

复制代码

事实上是在说：JavaScript 将 0 视为一个表达式，并尝试删除它的求值结果。

所以，现在这里的 0，其实不是值（Value）类型的数据，而是一个表达式运算的结果值（Result）。而在进一步的删除操作之前，JavaScript 需要检测这个 Result 的类型：

如果它是值，则按照传统的 JavaScript 的约定返回 true；

如果它是一个引用，那么对该引用进行分析，以决定如何操作。

这个检测过程说明，ECMAScript 约定：任何表达式计算的结果（Result）要么是一个值，要么是一个引用。并且需要留意的是，在这个描述中，所谓对象，其实也是值。准确地说，是“非引用类型”。例如：

```
1 delete {}
```

复制代码

那么显然，这里要删除的一对大括号是表示一个字面量的对象，当它被作为表达式执行的时候，结果也是一个值。这也是我常常将所有这类表达式称为“单值表达式”的原因，这里并没有所谓的“引用”。

你可以像下面这样，非常细致而准确地解释这一行代码：单值表达式的运算结果返回那个“对象字面量”的单值。然后，`delete`运算发现它的操作数是“值 / 非引用类型”，就直接返回了 true。

所以，什么也没有发生。

还会发生什么

那么到底还会发生什么呢?

在 JavaScript 的内部，所谓“引用”是可以转换为“值”，以便参与值运算的。因为表达式的本质是求值运算，所以引用是不能直接作为最终求值的操作数的。这依赖于一个非常核心的、称为“GetValue()”的内部操作。所谓内部操作，也称为内部抽象操作（internal abstract operations），是 ECMAScript 描述一个符合规范的引擎在具体实现时应当处理的那些行为。

`GetValue()`是从一个引用中取出值来的行为。这有什么用呢？比如说下面这行代码：

```
1 x = x
```

复制代码

我们上面说过，所谓 x 其实是一个引用。上面的表达式其实是一个赋值表达式，那么“引用 x 赋值给引用 x”有什么意义呢？其实这在语法层面来解释是非常直接的：

> 所有赋值操作的含义，是将右边的“值”，赋给左边用于包含该值的“引用”。

那么上面的`x=x`，其实就是被翻译成：

```
1  x = GetValue(x)
```

复制代码

来执行的。而 JavaScript 识别两个 x 的不同的方法，就称为“手性”，即是所谓“左手端 (lhs, left hand side)”和“右手端 (rhs)”。它本来是用来描述自然语言的语法中，一个修饰词应该是放在它的主体的前面或是后面的。而在程序设计语言中，它用来说明一个记号（Token）是放在了赋值符号（例如“=”号）的左边或是右边。作为一个简单的结论，区别上例中的两个 x 的方法就是：

> 如果 x 放在左边作为 lhs，那么它是引用；如果放在右边作为 rhs，那么就是值。

所以`x=x`的语义并不是“x 赋给 x”，而是“**把值 x 赋给引用 x**”。

所以，“delete x”归根到底，是在**删除一个表达式的、引用类型的结果（Result）**，而不是在**删除 x 表达式**，或者这个**删除表达式的值（Value）**。

是的，在 JavaScript 中的`delete`是一个很罕见的、能直接操作“引用”的语法元素。由于这里的“引用”是在 ECMAScript 规范层面的概念，因此在 JavaScript 语言中能操作它的语法元素其实非常非常少。

然而很不幸，delete 就是其中之一。

告诉我这些有什么用

等等，我想你一定会问了：神啊，让我知道这些究竟又什么用呢？我永远也不会去执行`delete 0`这样的操作啊！

是的。但是我接下来要告诉你的事实是：`obj.x`也是一个引用。对象属性存取是 JavaScript 的面向对象的基本操作之一，所以本质上我们早就在使用“引用”这个东西了，只不过它太习以为常，所以大家都视而不见。

“属性存取（"."运算符）”返回一个关于“x”的引用，然后它可以作为下一个操作符（例如函数调用运算“()”）的左手端来使用，这才有了著名的“对象方法调用”运算：

```
1 obj.x()
```

复制代码

因为在对象方法调用的时候，函数 _x()_ 是来自于`obj.x`这个引用的，所以这个引用将`obj`这个对象传递给 x()，这才会让函数 _x()_ 内部通过 this 来访问到 obj。

根本上来说，如果`obj.x`只是值，或者它作为右手端，那么它就不能“携带”obj 这个对象，也就完成不了后续的方法调用操作。

> 对象存取 + 函数调用 = 方法调用

这是 JavaScript 通过连续表达式运算来实现新的语义 / 语法的经典示例。

而所谓“连续运算”其实是函数式运算范式的基本原则。也就是说，`obj.x()`是在 JavaScript 中集合了“引用规范类型操作”、“函数式”、“面向对象”和“动态语言”等多种特性于一体的一个简单语法。

而它对语言的基础特性的依赖，就在于：

- `delete 0`中的这个`0`是一个表达式求值；
- `delete x`中的`x`是一个引用；
- `delete obj.x`中`obj.x`是一组表达式连续运算的结果（Result/ 引用）；

于是，我们现在可以解释，当 x 是全局对象 global 的属性时，所谓`delete x`其实只需要返回`global.x`这个引用就可以了。而当它不是全局对象 global 的属性时，那么就需要从当前环境中找到一个名为`x`的引用。找到这两种不同的引用的过程，称为 ResolveBinding；而这两种不同的`x`，称为不同环境下绑定的标识符 / 名字。

知识回顾

下一讲我将给你讲述的，就是这个名字从声明到发现的全过程。至于现在，这一讲就要告一段落了。今天的内容中，有一些知识点我来带你回顾一下。

delete 运算符尝试删除值数据时，会返回 true，用于表示没有错误（Error）。

delete 0 的本质是删除一个表达式的值（Result）。

delete x 与上述的区别只在于 Result 是一个引用（Reference）。

delete 其实只能删除一种引用，即对象的成员（Property）。

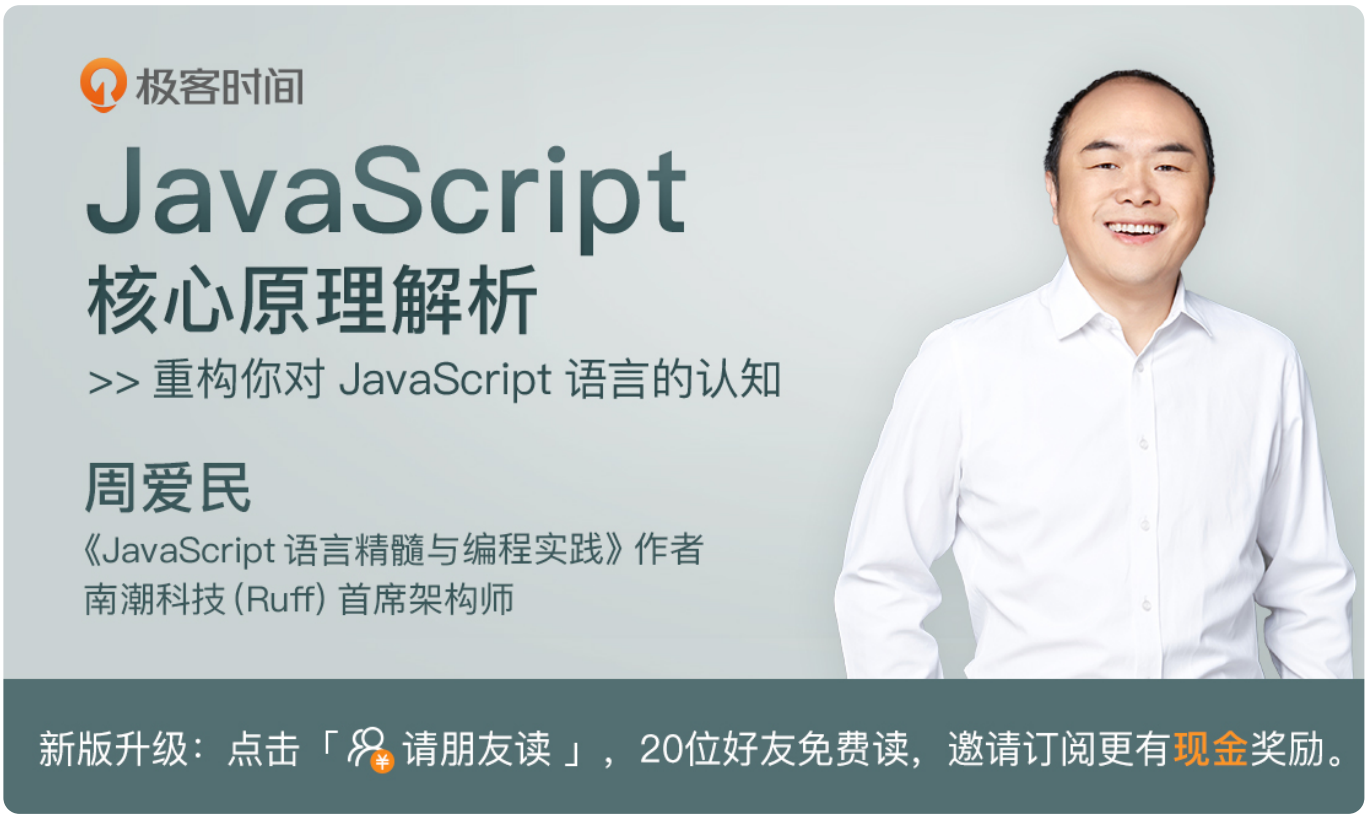
所以，只有在`delete x`等值于`delete obj.x`时 delete 才会有执行意义。例如`with (obj) ...`语句中的 delete x，以及全局属性 global.x。

思考题

delete x 中，如果 x 根本不存在，会发生什么?

delete object.x 中，如果 x 是只读的，会发生什么?

希望你喜欢我的分享。



© 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　开篇词 | 如何解决语言问题?

精选留言 (1)

写留言

文全

2019-11-11

作为一个工作几年的前端，急需这部分底层原理

展开