Javascript中的delete介绍

作者： 字体：[增加 减小] 类型：转载 时间：2012-09-02

关于JavaScript中的Delete一直没有弄的很清楚，最近看到两篇这方面的文章，现对两文中部分内容进行翻译(内容有修改和添加，顺序不完全一致，有兴趣推荐看原文)，希望能对大家有所帮助

一、问题的提出

我们先来看看下面几段代码，要注意的是，以下代码不要在浏览器的开发者工具(如FireBug、Chrome Developer tool)中运行，原因后面会说明：

为什么我们可以删除对象的属性：

代码如下: 复制代码

```
var o = { x: 1 };
delete o.x; // true
o.x; // undefined
```

但不以删除像这样声明的变量：

代码如下: 复制代码

```
var x = 1;
delete x; // false
x; // 1
```

也不能删除像这样定义的函数：

代码如下: 复制代码

```
function x(){}
delete x; // false
typeof x; // "function"
```

注意：当delete操作符返回true时表示可以删除，返回false表示不能删除

要理解这一点，我们首先需要掌握像变量实例化和属性特性这样的概念 - - 遗憾的是这些内容在一些javascript的书中很少讲到。理解它们并不难，如果你不在乎它们为什么这么运行，你可以随意的跳过这一部分。

二、代码类型

在ECMAScript中有三种类型的可执行代码：Global code（全局代码）、Function code（函数代码）和Eval code（放在Eval中执行的代码）。

代码如下: 复制代码

```
var x=1;//Global code
function test(){
var y=2;//Function Code
eval("var z=3");//Eval Code in Function
}
eval("function evalTest(){}");//Eval Code in Global
```

三、执行上下文

当ECMAScript 代码执行时，它总是在一定的上下文中运行，执行上下文是一个有点抽象的实体，它有助于我们理解作用域和变量实例化如何工作的。对于三种类型的可执行代码，每个都有执行的上下文。当一个函数执行时，可以说控制进入到函数代码（Function code）的执行上下文。全局代码执行时，进入到全局代码（Global code）的执行上下文。

正如你所见，执行上下文逻辑上来自一个栈。首先可能是有自己作用域的全局代码，代码中可能调用一个函数，它有自己的作用域，函数可以调用另外一个函数，等等。即使函数递归地调用它自身，每一次调用都进入一个新的执行上下文。

四、Activation object(激活对象)/Variable object(变量对象)

每一个执行上下文在其内部都有一个Variable Object。与执行上下文类似，Variable object是一个抽象的实体，用来描述变量实例化的机制。有趣的是在代码中声明的变量和函数实际上被当作这个变量对象的属性被添加。

当进入全局代码的执行上下文时，一个全局对象用作变量对象。这也正是为什么在全局范围中声明的变量或者函数变成了全局对象的属性。

代码如下: 复制代码

```
/* remember that `this` refers to global object when in global scope */
var GLOBAL_OBJECT = this;

var foo = 1;
```

```
GLOBAL_OBJECT.foo; // 1
foo === GLOBAL_OBJECT.foo; // true

function bar(){}
typeof GLOBAL_OBJECT.bar; // "function"
GLOBAL_OBJECT.bar === bar; // true
```

全局变量变成了全局对象的属性，但是，那些在函数代码（Function code）中定义的局部变量又会如何呢？行为其实很相似：它成了变量对象的属性。唯一的差别在于在函数代码（Function code）中，变量对象不是全局对象，而是所谓的激活对象（Activation object）。每次函数代码（Function code）进入执行作用域时，就会创建一个激活对象（Activation object）。

不仅函数代码（Function code）中的变量和函数成为激活对象的属性，而且函数的每一个参数（与形参相对应的名称）和一个特定Arguments 对象也是。注意，激活对象是一种内部机制，不会被程序代码真正访问到。

代码如下: 复制代码

```
(function(foo){

var bar = 2;
function baz(){}

/*
In abstract terms,

Special `arguments` object becomes a property of containing function's Activation object:
ACTIVATION_OBJECT.arguments; // Arguments object

...as well as argument `foo`:
ACTIVATION_OBJECT.foo; // 1

...as well as variable `bar`:
ACTIVATION_OBJECT.bar; // 2

...as well as function declared locally:
typeof ACTIVATION_OBJECT.baz; // "function"
*/

})(1);
```

最后，在Eval 代码（Eval code）中声明的变量作为正在调用的上下文的变量对象的属性被创建。Eval 代码（Eval code）只使用它正在被调用的哪个执行上下文的变量对象。

代码如下: 复制代码

```
var GLOBAL_OBJECT = this;

/* `foo` is created as a property of calling context Variable object,
which in this case is a Global object */

eval('var foo = 1;');
GLOBAL_OBJECT.foo; // 1

(function(){

/* `bar` is created as a property of calling context Variable object,
which in this case is an Activation object of containing function */

eval('var bar = 1;');

/*
In abstract terms,
ACTIVATION_OBJECT.bar; // 1
*/

})();
```

五、属性特性

现在变量会怎样已经很清楚（它们成为属性），剩下唯一的需要理解的概念是属性特性。每个属性都有来自下列一组属性中的零个或多个特性 - - ReadOnly, DontEnum, DontDelete 和Internal，你可以认为它们是一个标记，一个属性可有可无的特性。为了今天讨论的目的，我们只关心DontDelete 特性。

当声明的变量和函数成为一个变量对象的属性时 - - 要么是激活对象（Function code），要么是全局对象（Global code），这些创建的属性带有DontDelete 特性。但是，任何明确的（或隐含的）创建的属性不具有DontDelete 特性。这就是我们为什么一些属性能删除，一些不能。

代码如下: 复制代码

```
var GLOBAL_OBJECT = this;
```

```
/* `foo` is a property of a Global object.
It is created via variable declaration and so has DontDelete attribute.
This is why it can not be deleted. */

var foo = 1;
delete foo; // false
typeof foo; // "number"

/* `bar` is a property of a Global object.
It is created via function declaration and so has DontDelete attribute.
This is why it can not be deleted either. */

function bar(){}
delete bar; // false
typeof bar; // "function"

/* `baz` is also a property of a Global object.
However, it is created via property assignment and so has no DontDelete attribute.
This is why it can be deleted. */

GLOBAL_OBJECT.baz = 'blah';
delete GLOBAL_OBJECT.baz; // true
typeof GLOBAL_OBJECT.baz; // "undefined"
```

六、内置属性和DontDelete

一句话：属性中一个独特的特性(DontDelete)控制着这个属性是否能被删除。注意，对象的内置属性（即对象的预定义属性）有DontDelete 特性，因此不能被删除。特定的Arguments 变量（或者，正如我们现在了解的，激活对象的属性），任何函数实例的length属性也拥有DontDelete 特性。

代码如下: 复制代码

```
(function(){

/* can't delete `arguments`, since it has DontDelete */

delete arguments; // false
typeof arguments; // "object"
```

```
/* can't delete function's `length`; it also has DontDelete */

function f(){}
delete f.length; // false
typeof f.length; // "number"

})();
```

与函数参数相对应的创建的属性也有DontDelete 特性，因此也不能被删除。

代码如下: 复制代码

```
(function(foo, bar){

delete foo; // false
foo; // 1

delete bar; // false
bar; // 'blah'

})(1, 'blah');
```

七、未声明的赋值

简单地就是未声明的赋值在一个全局对象上创建一个可删除的属性。

代码如下: 复制代码

```
var GLOBAL_OBJECT = this;

/* create global property via variable declaration; property has DontDelete */
var foo = 1;

/* create global property via undeclared assignment; property has no DontDelete */
bar = 2;//可理解为 window.bar=2; 根据上面的第五点是可以删除的

delete foo; // false
typeof foo; // "number"

delete bar; // true
```

```
typeof bar; // "undefined"
```

请注意，DontDelete特性是在属性创建的过程中确定的，后来的赋值不会修改现有属性已经存在的特性，理解这一点很重要。

代码如下: 复制代码

```
/* `foo` is created as a property with DontDelete */
function foo(){}

/* Later assignments do not modify attributes. DontDelete is still there! */
foo = 1;
delete foo; // false
typeof foo; // "number"

/* But assigning to a property that doesn't exist,
creates that property with empty attributes (and so without DontDelete) */

this.bar = 1;
delete bar; // true
typeof bar; // "undefined"
```

八、Eval code

在Eval中创建的变量或方法比较特别，没有DontDelete特性，也就是说可以删除。

代码如下: 复制代码

```
eval("var x = 1;");
console.log(x); // 1
delete x;
console.log(typeof x); // undefined

eval("function test(){ var x=1; console.log(delete x);/* false */;return 1;}");
console.log(test()); // 1
delete test;
console.log(typeof test); // undefined
```

注意，这里说的在Eval中创建的变量或方法不包括方法内部的变量或方法，如上面代码中的红色部分，仍然跟之前讲的一致：不能被删除。

九、FireBug的困惑

我们看一段在FireBug中执行的代码结果：

代码如下: 复制代码

```
var x=1;
delete x;
console.log(typeof x);//undefined

function y(){
var z=1;
console.log(delete z);//false
}
y();
delete y;
console.log(typeof y);//undefined
```

这明明是违反上述规则的，但跟上面第八点对比后发现，这正在代码在eval中执行的效果。虽然没有证实，但我猜测FireBug（Chrome Developer tool）中控制台代码是用eval执行的。

所以，当大家在测试JS代码时，如果涉及到当前上下文环境时特别要注意。

十、delete操作符删除的对象

C++中也有delete操作符，它删除的是指针所指向的对象。例如：

代码如下: 复制代码

```
class Object {
public:
Object *x;
}

Object o;
o.x = new Object();
delete o.x; // 上一行new的Object对象将被释放
```

但Javascript的delete与C++不同，它不会删除o.x指向的对象，而是删除o.x属性本身。

代码如下: 复制代码

```
var o = {};
o.x = new Object();
delete o.x; // 上一行new的Object对象依然存在
o.x; // undefined，o的名为x的属性被删除了
```

在实际的Javascript中，delete o.x之后，Object对象会由于失去了引用而被垃圾回收， 所以delete o.x也就“相当于”删除了o.x所指向的对象，但这个动作并不是ECMAScript标准， 也就是说，即使某个实现完全不删除Object对象，也不算是违反ECMAScript标准。

“删除属性而不是删除对象”这一点，可以通过以下的代码来确认。

代码如下: 复制代码

```
var o = {};
var a = { x: 10 };
o.a = a;
delete o.a; // o.a属性被删除
o.a; // undefined
a.x; // 10, 因为{ x: 10 } 对象依然被 a 引用，所以不会被回收
```

另外，delete o.x 也可以写作 delete o["x"]，两者效果相同。

十一、其他不能被删除的属性

除了上面说过的内置属性（即预定义属性）不能被删除外，prototype中声明的属性也不能delete：

代码如下: 复制代码

```
function C() { this.x = 42; }
C.prototype.x = 12;
C.prototype.y = 13;

var o = new C();
o.x; // 42, 构造函数中定义的o.x

delete o.x; //true 删除的是自身定义的x
o.x; // 12, prototype中定义的o.x，即使再次执行delete o.x也不会被删除

delete o.y; //true，因为 o自身没有o.y属性，y存在于prototype链中，也就是说对象自身属性和prototype属性是不同的
```

```
o.y; //13
```

小结

上面说了那么多，希望对大家认识JavaScript中的Delete有所帮助。由于水平有限，不保证完全正确，如果发现错误欢迎指正。