# **Javascript语言规范**

## **var关键字**

总是用 var 关键字定义变量。

### **描述**

如果不显式使用 var 关键字定义变量，变量会进入到全局上下文中，可能会和已有的变量发生冲突。另外，如果不使用var声明，很难说变量存在的作用域是哪个（可能在局部作用域里，也可能在document或者window上）。所以，要一直使用 var 关键字定义变量。

## **常量**

* 使用字母全部大写（如 NAMES\_LIKE\_THIS ）的方式命名
* 可以使用 @const 来标记一个常量 指针 （指向变量或属性，自身不可变）
* 由于IE的兼容问题，不要使用 [const关键字](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const?redirectlocale=en-US&redirectslug=JavaScript/Reference/Statements/const)

### **描述**

#### **常量值**

如果一个值是恒定的，它命名中的字母要全部大写（如 CONSTANT\_VALUE\_CASE ）。字母全部大写意味着这个值不可以被改写。

原始类型（ number 、 string 、 boolean ）是常量值。

对象的表现会更主观一些，当它们没有暴露出变化的时候，应该认为它们是常量。但是这个不是由编译器决定的。

#### **常量指针（变量和属性）**

用 @const 注释的变量和属性意味着它是不能更改的。使用const关键字可以保证在编译的时候保持一致。使用 [const](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const?redirectlocale=en-US&redirectslug=JavaScript/Reference/Statements/const) 效果相同，但是由于IE的兼容问题，我们不使用const关键字。

另外，不应该修改用 @const 注释的方法。

#### **例子**

注意， @const 不一定是常量值，但命名类似 CONSTANT\_VALUE\_CASE 的 一定 是常量指针。

/\*\*\* 以毫秒为单位的超时时长\* **@type** {number}\*/goog.example.TIMEOUT\_IN\_MILLISECONDS = 60;

1分钟60秒永远也不会改变，这是个常量。全部大写的命名意味其为常量值，所以它不能被重写。 开源的编译器允许这个符号被重写，这是因为 没有 @const 标记。

/\*\*\* Map of URL to response string.\* **@const**\*/MyClass.fetchedUrlCache\_ = new goog.structs.Map();

在这个例子中，指针没有变过，但是值却是可以变化的，所以这里用了驼峰式的命名，而不是全部大写的命名。

## **分号**

一定要使用分号。

依靠语句间隐式的分割，可能会造成细微的调试的问题，千万不要这样做。

很多时候不写分号是很危险的：

// 1.

MyClass.prototype.myMethod = function() {

return 42;

} // 这里没有分号.

(function() {

// 一些局部作用域中的初始化代码

})();

var x = {

'i': 1,

'j': 2

} //没有分号.

// 2. 试着在IE和firefox下做一样的事情.

//没人会这样写代码，别管他.

[normalVersion, ffVersion][isIE]();

var THINGS\_TO\_EAT = [apples, oysters, sprayOnCheese] //这里没有分号

// 3. 条件语句

-1 == resultOfOperation() || die();

### **发生了什么？**

1. js错误。返回42的函数运行了，因为后面有一对括号，而且传入的参数是一个方法，然后返回的42被调用，导致出错了。
2. 你可能会得到一个“no sush property in undefined”的错误，因为在执行的时候，解释器将会尝试执行 x[normalVersion, ffVersion][isIE]() 这个方法。
3. die 这个方法只有在 resultOfOperation() 是 NaN 的时候执行，并且 THINGS\_TO\_EAT 将会被赋值为 die() 的结果。

### **为什么？**

js语句要求以分号结尾，除非能够正确地推断分号的位置。在这个例子当中，函数声明、对象和数组字面量被写在了一个语句当中。右括号（”)”、”}”、”]”）不足以证明这条语句已经结束了，如果下一个字符是运算符或者”(”、”{”、”[”，js将不会结束语句。

这个错误让人震惊，所以一定要确保用分号结束语句。

### **澄清：分号和函数**

函数表达式后面要分号结束，但是函数声明就不需要。例如：

var foo = function() {

**return** true;}; // 这里要分号

function foo() {

**return** true;} // 这里不用分号

## **嵌套函数**

可以使用。

嵌套函数非常有用，比如在创建持续任务或者隐藏工具方法的时候。可以放心的使用。

## **块内函数声明**

不要使用块内函数声明。

不要这样做：

**if** (x) {

function foo() {}}

虽然大多数脚本引擎支持功能区块内声明，但ECMAScript并未认可（见 [ECMA-262](http://www.ecma-international.org/publications/standards/Ecma-262.htm) ，第13条和第14）。若与他人的及EcmaScript所建议的不一致，即可视为不好的实现方式。ECMAScript只允许函数声明语句列表, 在根语句列表脚本或者函数。相反，使用一个变量初始化函数表达式在块内定义一个函数块：

**if** (x) {

var foo = function() {}}

## **异常**

可以抛出异常。

如果你做一些比较复杂的项目你基本上无法避免异常，比如使用一个应用程序开发框架。可以大胆试一试。

## **自定义异常**

可以自定义异常。

如果没有自定义异常，返回的错误信息来自一个有返回值的函数是难处理的，是不雅的。坏的解决方案包括传递引用的类型来保存错误信息或总是返回有一个潜在的错误成员的对象。这些基本上为原始的异常处理hack。在适当的时候使用自定义的异常。

## **标准功能**

总是优先于非标准功能。

为了最大的可移植性和兼容性，总是使用标准功能而不是非标准功能（例如，采用 string.charAt(3)而非 string[3] ，用DOM的功能访问元素而不是使用特定于一个具体应用的简写）。

## **原始类型的包装对象**

没有理由使用原始类型的包装对象，更何况他们是危险的：

var x = new Boolean(false);

if (x) {

alert('hi'); //显示“hi”。

}

不要这样做！

然而类型转换是可以的。

var x = Boolean(0);

if (x) {

alert('hi'); //永远都不显示。

}

typeof Boolean(0) == 'boolean';

typeof new Boolean(0) == 'object';

这是非常有用的进行数字、字符串和布尔值转换的方式。

## **多重的原型继承**

不可取。

多重原型继承是Javascript实现继承的方式。如果你有一个以用户定义的class B作为原型的用户自定义class D，则得到多重原型继承。这样的继承出现容易但难以正确创造！

出于这个原因，最好是使用 [Closure库](https://developers.google.com/closure/library/?csw=1) 中的 goog.inherits() 或类似的东西。

function D() {

goog.base(this)}goog.inherits( D, B );

D.prototype.method =function() {

...};

## **方法和属性定义**

/\*\*构造函数\*/ function SomeConstructor() { this.someProperty = 1; } Foo.prototype.someMethod = function() { ... };

虽然有多种使用“new”关键词来创建对象方法和属性的途径，首选的创建方法的途径是：

Foo.prototype.bar = function() {

/\* ... \*/};

其他特性的首选创建方式是在构造函数中初始化字段：

/\*\* **@constructor** \*/function Foo() {

this.bar = value;}

### **为什么？**

当前的JavaScript引擎优化基于一个对象的“形状”， [给对象添加一个属性（包括覆盖原型设置的值）改变了形式，会降低性能](https://developers.google.com/v8/design" \l "prop_access) 。

## **删除**

请使用 this.foo = null 。

o.prototype.dispose = function() {

this.property\_ = null;};

而不是：

Foo.prototype.dispose = function() {

delete his.property\_;};

在现代的JavaScript引擎中，改变一个对象属性的数量比重新分配值慢得多。应该避免删除关键字，除非有必要从一个对象的迭代的关键字列表删除一个属性，或改变 if (key in obj) 结果。

## **闭包**

可以使用，但是要小心。

创建闭包可能是JS最有用的和经常被忽视的功能。在 [这里](http://jibbering.com/faq/notes/closures/) 很好地描述说明了闭包的工作。

要记住的一件事情，一个闭包的指针指向包含它的范围。因此，附加一个闭包的DOM元素，可以创建一个循环引用，所以，内存会泄漏。例如，下面的代码：

function foo(element, a, b) {

element.onclick = function() { /\* 使用 a 和 b \*/ };}

闭包能保持元素a和b的引用即使它从未使用。因为元素还保持对闭包的一个引用，我们有一个循环引用，不会被垃圾收集清理。在这些情况下，代码的结构可以如下：

function foo(element, a, b) {

element.onclick = bar(a, b);}

function bar(a, b) {

**return** function() { /\* 使用 a 和 b \*/ }}

## **eval()函数**

只用于反序列化（如评估RPC响应）。

若用于 eval() 的字符串含有用户输入，则 eval() 会造成混乱的语义，使用它有风险。通常有一个更好 更清晰、更安全的方式来编写你的代码，所以一般是不会允许其使用的。然而，eval相对比非eval使反序列化更容易，因此它的使用是可以接受的（例如评估RPC响应）。

反序列化是将一系列字节存到内存中的数据结构转化过程。例如，你可能会写的对象是：

users = [

{

name: 'Eric',

id: 37824,

email: 'jellyvore@myway.com'

},

{

name: 'xtof',

id: 31337,

email: 'b4d455h4x0r@google.com'

},

...];

将这些数据读入内存跟得出文件的字符串表示形式一样容易。

同样， eval() 函数可以简化解码RPC的返回值。例如，您可以使用 XMLHttpRequest 生成RPC，在响应时服务器返回JavaScript：

var userOnline = false;

var user = 'nusrat';

var xmlhttp = new XMLHttpRequest();

xmlhttp.open('GET', 'http://chat.google.com/isUserOnline?user=' + user, false);

xmlhttp.send('');

// 服务器返回：

// userOnline = true;

if (xmlhttp.status == 200) {

eval(xmlhttp.responseText);

}

// userOnline 现在为 true

## **with() {}**

不建议使用。

使用 with 会影响程序的语义。因为 with 的目标对象可能会含有和局部变量冲突的属性，使你程序的语义发生很大的变化。例如，这是做什么用？

**with** (foo) {

var x = 3;

**return** x;}

答案：什么都有可能。局部变量 x 可能会被 foo 的一个属性覆盖，它甚至可能有setter方法，在此情况下将其赋值为3可能会执行很多其他代码。不要使用 with 。

## **this**

只在构造函数对象、方法，和创建闭包的时候使用。

this 的语义可能会非常诡异。有时它指向全局对象（很多时候）、调用者的作用域链（在 eval里）、DOM树的一个节点（当使用HTML属性来做为事件句柄时）、新创建的对象（在一个构造函数中）、或者其他的对象（如果函数被 call() 或 apply() 方式调用）。

正因为 this 很容易被弄错，故将其使用限制在以下必须的地方：

* 在构造函数中
* 在对象的方法中（包括闭包的创建）

## **for-in 循环**

只使用在对象、映射、哈希的键值迭代中。

for-in 循环经常被不正确的用在元素数组的循环中。由于并不是从 0 到 length-1 进行循环，而是遍历对象中和它原型链上的所有的键，所以很容易出错。这里有一些失败的例子：

function printArray(arr) {

**for** (var key **in** arr) {

print(arr[key]);

}}

printArray([0,1,2,3]); //这样可以

var a = new Array(10);printArray(a); //这样不行

a = document.getElementsByTagName('\*');printArray(a); //这样不行

a = [0,1,2,3];a.buhu = 'wine';printArray(a); //这样不行

a = new Array;a[3] = 3;printArray(a); //这样不行

在数组循环时常用的一般方式：

function printArray(arr) {

var l = arr.length;

**for** (var i = 0; i < l; i++) {

print(arr[i]);

}}

## **关联数组**

不要将映射，哈希，关联数组当作一般数组来使用。

不允许使用关联数组……确切的说在数组，你不可以使用非数字的索引。如果你需要一个映射或者哈希，在这种情况下你应该使用对象来代替数组，因为在功能上你真正需要的是对象的特性而不是数组的。

数组仅仅是用来拓展对象的（像在JS中你曾经使用过的 Date 、 RegExp 和 String 对象一样的）。

## **多行的字符串字面量**

不要使用。

不要这样做：

var myString = 'A rather long string of English text, an error message **\** actually that just keeps going and going -- an error **\** message to make the Energizer bunny blush (right through **\** those Schwarzenegger shades)! Where was I? Oh yes, **\** you**\'**ve got an error and all the extraneous whitespace is **\** just gravy. Have a nice day.';

在编译时每一行头部的空白符不会被安全地去除掉；斜线后的空格也会导致棘手的问题；虽然大部分脚本引擎都会支持，但是它不是ECMAScript规范的一部分。

使用字符串连接来代替：

var myString = 'A rather long string of English text, an error message ' +

'actually that just keeps going and going -- an error ' +

'message to make the Energizer bunny blush (right through ' +

'those Schwarzenegger shades)! Where was I? Oh yes, ' +

'you**\'**ve got an error and all the extraneous whitespace is ' +

'just gravy. Have a nice day.';

## **数组和对象字面量**

建议使用。

使用数组和对象字面量来代替数组和对象构造函数。

数组构造函数容易在参数上出错。

// 长度为3var a1 = new Array(x1, x2, x3);

// 长度为 2var a2 = new Array(x1, x2);

// If x1 **is** a number **and** it **is** a natural number the length will be x1.// If x1 **is** a number but **not** a natural number this will throw an exception.// Otherwise the array will have one element **with** x1 **as** its value.var a3 = new Array(x1);

// 长度为0var a4 = new Array();

由此，如果有人将代码从2个参数变成了一个参数，那么这个数组就会有一个错误的长度。

为了避免这种怪异的情况，永远使用可读性更好的数组字面量。

var a = [x1, x2, x3];var a2 = [x1, x2];var a3 = [x1];var a4 = [];

对象构造函数虽然没有相同的问题，但是对于可读性和一致性，还是应该使用对象字面量。

var o = new Object();

var o2 = new Object();o2.a = 0;o2.b = 1;o2.c = 2;o2['strange key'] = 3;

应该写成：

var o = {};

var o2 = {

a: 0,

b: 1,

c: 2,

'strange key': 3};

## **修改内置对象原型**

不建议。

强烈禁止修改如 Object.prototype 和 Array.prototype 等对象的原型。修改其他内置原型如 Function.prototype 危险性较小，但在生产环境中还是会引发一些难以调试的问题，也应当避免。

## **Internet Explorer中的条件注释**

不要使用。

不要这样做：

var f = function () {

/\***@cc\_on** **if** (**@\_jscript**) { **return** 2\* @\*/ 3; /\*@ } @\*/};

条件注释会在运行时改变JavaScript语法树，阻碍自动化工具。