**[js作用域与命名空间](http://segmentfault.com/a/1190000000736812)**

[[](http://segmentfault.com/u/gecko23) **gecko23**](http://segmentfault.com/u/gecko23)2014年10月23日 发布

* 推荐 **0** 推荐
* 收藏 **6** 收藏，1.3k 浏览

尽管 JavaScript 支持一对花括号创建的代码段，但是并不支持块级作用域；  
而仅仅支持 *函数作用域*。

function test() { // 一个作用域

for(var i = 0; i < 10; i++) { // 不是一个作用域

// count

}

console.log(i); // 10

}

**注意:** 如果不是在赋值语句中，而是在 return 表达式或者函数参数中，{...} 将会作为代码段解析，  
而不是作为对象的字面语法解析。如果考虑到 自动分号插入，这可能会导致一些不易察觉的错误。

[**译者注**](http://cnblogs.com/sanshi/)**：**如果 return 对象的左括号和 return 不在一行上就会出错。

// 译者注：下面输出 undefined

function add(a, b) {

return

a + b;

}

console.log(add(1, 2));

JavaScript 中没有显式的命名空间定义，这就意味着所有对象都定义在一个*全局共享*的命名空间下面。

每次引用一个变量，JavaScript 会向上遍历整个作用域直到找到这个变量为止。  
如果到达全局作用域但是这个变量仍未找到，则会抛出 ReferenceError 异常。

**隐式的全局变量**

// 脚本 A

foo = '42';

// 脚本 B

var foo = '42'

上面两段脚本效果**不同**。脚本 A 在*全局*作用域内定义了变量 foo，而脚本 B 在*当前*作用域内定义变量 foo。

再次强调，上面的效果**完全不同**，不使用 var 声明变量将会导致隐式的全局变量产生。

// 全局作用域

var foo = 42;

function test() {

// 局部作用域

foo = 21;

}

test();

foo; // 21

在函数 test 内不使用 var 关键字声明 foo 变量将会覆盖外部的同名变量。  
起初这看起来并不是大问题，但是当有成千上万行代码时，不使用 var 声明变量将会带来难以跟踪的 BUG。

// 全局作用域

var items = [/\* 数组 \*/];

for(var i = 0; i < 10; i++) {

subLoop();

}

function subLoop() {

// subLoop 函数作用域

for(i = 0; i < 10; i++) { // 没有使用 var 声明变量

// 干活

}

}

外部循环在第一次调用 subLoop 之后就会终止，因为 subLoop 覆盖了全局变量 i。  
在第二个 for 循环中使用 var 声明变量可以避免这种错误。  
声明变量时**绝对不要**遗漏 var 关键字，除非这就是*期望*的影响外部作用域的行为。

**局部变量**

JavaScript 中局部变量只可能通过两种方式声明，一个是作为函数参数，另一个是通过 var 关键字声明。

// 全局变量

var foo = 1;

var bar = 2;

var i = 2;

function test(i) {

// 函数 test 内的局部作用域

i = 5;

var foo = 3;

bar = 4;

}

test(10);

foo 和 i 是函数 test 内的局部变量，而对 bar 的赋值将会覆盖全局作用域内的同名变量。

**变量声明提升（Hoisting）**

JavaScript 会**提升**变量声明。这意味着 var 表达式和 function 声明都将会被提升到当前作用域的顶部。

bar();

var bar = function() {};

var someValue = 42;

test();

function test(data) {

if (false) {

goo = 1;

} else {

var goo = 2;

}

for(var i = 0; i < 100; i++) {

var e = data[i];

}

}

上面代码在运行之前将会被转化。JavaScript 将会把 var 表达式和 function 声明提升到当前作用域的顶部。

// var 表达式被移动到这里

var bar, someValue; // 缺省值是 'undefined'

// 函数声明也会提升

function test(data) {

var goo, i, e; // 没有块级作用域，这些变量被移动到函数顶部

if (false) {

goo = 1;

} else {

goo = 2;

}

for(i = 0; i < 100; i++) {

e = data[i];

}

}

bar(); // 出错：TypeError，因为 bar 依然是 'undefined'

someValue = 42; // 赋值语句不会被提升规则（hoisting）影响

bar = function() {};

test();

没有块级作用域不仅导致 var 表达式被从循环内移到外部，而且使一些 if 表达式更难看懂。

在原来代码中，if 表达式看起来修改了*全部变量* goo，实际上在提升规则被应用后，却是在修改*局部变量*。

如果没有提升规则（hoisting）的知识，下面的代码看起来会抛出异常 ReferenceError。

// 检查 SomeImportantThing 是否已经被初始化

if (!SomeImportantThing) {

var SomeImportantThing = {};

}

实际上，上面的代码正常运行，因为 var 表达式会被提升到*全局作用域*的顶部。

var SomeImportantThing;

// 其它一些代码，可能会初始化 SomeImportantThing，也可能不会

// 检查是否已经被初始化

if (!SomeImportantThing) {

SomeImportantThing = {};

}

[**译者注**](http://cnblogs.com/sanshi/)**：**在 Nettuts+ 网站有一篇介绍 hoisting 的[文章](http://net.tutsplus.com/tutorials/javascript-ajax/quick-tip-javascript-hoisting-explained/" \t "_blank)，其中的代码很有启发性。

// 译者注：来自 Nettuts+ 的一段代码，生动的阐述了 JavaScript 中变量声明提升规则

var myvar = 'my value';

(function() {

alert(myvar); // undefined

var myvar = 'local value';

})();

**名称解析顺序**

JavaScript 中的所有作用域，包括*全局作用域*，都有一个特别的名称 this 指向当前对象。

函数作用域内也有默认的变量 arguments，其中包含了传递到函数中的参数。

比如，当访问函数内的 foo 变量时，JavaScript 会按照下面顺序查找：

1. 当前作用域内是否有 var foo 的定义。
2. 函数形式参数是否有使用 foo 名称的。
3. 函数自身是否叫做 foo。
4. 回溯到上一级作用域，然后从 **#1** 重新开始。

**注意:** 自定义 arguments 参数将会阻止原生的 arguments 对象的创建。

**命名空间**

只有一个全局作用域导致的常见错误是命名冲突。在 JavaScript中，这可以通过 *匿名包装器* 轻松解决。

(function() {

// 函数创建一个命名空间

window.foo = function() {

// 对外公开的函数，创建了闭包

};

})(); // 立即执行此匿名函数

匿名函数被认为是 表达式；因此为了可调用性，它们首先会被执行。

( // 小括号内的函数首先被执行

function() {}

) // 并且返回函数对象

() // 调用上面的执行结果，也就是函数对象

有一些其他的调用函数表达式的方法，比如下面的两种方式语法不同，但是效果一模一样。

// 另外两种方式

+function(){}();

(function(){}());

**结论**

推荐使用*匿名包装器*（[**译者注**](http://cnblogs.com/sanshi/)**：**也就是自执行的匿名函数）来创建命名空间。这样不仅可以防止命名冲突，  
而且有利于程序的模块化。

另外，使用全局变量被认为是**不好的习惯**。这样的代码倾向于产生错误和带来高的维护成本。