[变量对象（Variable Object）](http://www.cnblogs.com/TomXu/archive/2012/01/16/2309728.html)

**介绍**

JavaScript编程的时候总避免不了声明函数和变量，以成功构建我们的系统，但是解释器是如何并且在什么地方去查找这些函数和变量呢？我们引用这些对象的时候究竟发生了什么？

俄文地址：http://dmitrysoshnikov.com/ecmascript/ru-chapter-2-variable-object/  
  
英文地址：http://dmitrysoshnikov.com/ecmascript/chapter-2-variable-object/

大多数ECMAScript程序员应该都知道变量与执行上下文有密切关系：

var a = 10; // 全局上下文中的变量  
   
(function () {  
 var b = 20; // function上下文中的局部变量  
})();  
   
alert(a); // 10  
alert(b); // 全局变量 "b" 没有声明

并且，很多程序员也都知道，当前ECMAScript规范指出独立作用域只能通过“函数(function)”代码类型的执行上下文创建。也就是说，相对于C/C++来说，ECMAScript里的for循环并不能创建一个局部的上下文。

for (var k in {a: 1, b: 2}) {  
 alert(k);  
}  
   
alert(k); // 尽管循环已经结束但变量k依然在当前作用域

我们来看看一下，我们声明数据的时候到底都发现了什么细节。

**数据声明**

如果变量与执行上下文相关，那变量自己应该知道它的数据存储在哪里，并且知道如何访问。这种机制称为变量对象(variable object)。

变量对象(缩写为VO)是一个与执行上下文相关的特殊对象，它存储着在上下文中声明的以下内容：  
 变量 (var, 变量声明);  
 函数声明 (FunctionDeclaration, 缩写为FD);  
 函数的形参

举例来说，我们可以用普通的ECMAScript对象来表示一个变量对象：

VO = {};

就像我们所说的, VO就是执行上下文的属性(property)：

activeExecutionContext = {  
 VO: {  
 // 上下文数据（var, FD, function arguments)  
 }  
};

只有全局上下文的变量对象允许通过VO的属性名称来间接访问(因为在全局上下文里，全局对象自身就是变量对象，稍后会详细介绍)，在其它上下文中是不能直接访问VO对象的，因为它只是内部机制的一个实现。

当我们声明一个变量或一个函数的时候，和我们创建VO新属性的时候一样没有别的区别（即：有名称以及对应的值）。

例如：

var a = 10;  
   
function test(x) {  
 var b = 20;  
};  
   
test(30);

对应的变量对象是：

// 全局上下文的变量对象  
VO(globalContext) = {  
 a: 10,  
 test: <reference to function>  
};  
   
// test函数上下文的变量对象  
VO(test functionContext) = {  
 x: 30,  
 b: 20  
};

在具体实现层面(以及规范中)变量对象只是一个抽象概念。(从本质上说，在具体执行上下文中，VO名称是不一样的，并且初始结构也不一样。

**不同执行上下文中的变量对象**

对于所有类型的执行上下文来说，变量对象的一些操作(如变量初始化)和行为都是共通的。从这个角度来看，把变量对象作为抽象的基本事物来理解更为容易。同样在函数上下文中也定义和变量对象相关的额外内容。

抽象变量对象VO (变量初始化过程的一般行为)  
 ║  
 ╠══> 全局上下文变量对象GlobalContextVO  
 ║ (VO === this === global)  
 ║  
 ╚══> 函数上下文变量对象FunctionContextVO  
 (VO === AO, 并且添加了<arguments>和<formal parameters>)

我们来详细看一下：

**全局上下文中的变量对象**

首先，我们要给全局对象一个明确的定义：

全局对象(Global object) 是在进入任何执行上下文之前就已经创建了的对象；  
这个对象只存在一份，它的属性在程序中任何地方都可以访问，全局对象的生命周期终止于程序退出那一刻。

全局对象初始创建阶段将Math、String、Date、parseInt作为自身属性，等属性初始化，同样也可以有额外创建的其它对象作为属性（其可以指向到全局对象自身）。例如，在DOM中，全局对象的window属性就可以引用全局对象自身(当然，并不是所有的具体实现都是这样)：

global = {  
 Math: <...>,  
 String: <...>  
 ...  
 ...  
 window: global //引用自身  
};

当访问全局对象的属性时通常会忽略掉前缀，这是因为全局对象是不能通过名称直接访问的。不过我们依然可以通过全局上下文的this来访问全局对象，同样也可以递归引用自身。例如，DOM中的window。综上所述，代码可以简写为：

String(10); // 就是global.String(10);  
   
// 带有前缀  
window.a = 10; // === global.window.a = 10 === global.a = 10;  
this.b = 20; // global.b = 20;

因此，回到全局上下文中的变量对象——在这里，变量对象就是全局对象自己：

VO(globalContext) === global;

非常有必要要理解上述结论，基于这个原理，在全局上下文中声明的对应，我们才可以间接通过全局对象的属性来访问它（例如，事先不知道变量名称）。

var a = new String('test');  
   
alert(a); // 直接访问，在VO(globalContext)里找到："test"  
   
alert(window['a']); // 间接通过global访问：global === VO(globalContext): "test"  
alert(a === this.a); // true  
   
var aKey = 'a';  
alert(window[aKey]); // 间接通过动态属性名称访问："test"

**函数上下文中的变量对象**

在函数执行上下文中，VO是不能直接访问的，此时由活动对象(activation object,缩写为AO)扮演VO的角色。

VO(functionContext) === AO;

活动对象是在进入函数上下文时刻被创建的，它通过函数的arguments属性初始化。arguments属性的值是Arguments对象：

AO = {  
 arguments: <ArgO>  
};

Arguments对象是活动对象的一个属性，它包括如下属性：

1. callee — 指向当前函数的引用
2. length — 真正传递的参数个数
3. properties-indexes (字符串类型的整数) 属性的值就是函数的参数值(按参数列表从左到右排列)。 properties-indexes内部元素的个数等于arguments.length. properties-indexes 的值和实际传递进来的参数之间是共享的。

例如：

function foo(x, y, z) {  
   
 // 声明的函数参数数量arguments (x, y, z)  
 alert(foo.length); // 3  
   
 // 真正传进来的参数个数(only x, y)  
 alert(arguments.length); // 2  
   
 // 参数的callee是函数自身  
 alert(arguments.callee === foo); // true  
   
 // 参数共享  
   
 alert(x === arguments[0]); // true  
 alert(x); // 10  
   
 arguments[0] = 20;  
 alert(x); // 20  
   
 x = 30;  
 alert(arguments[0]); // 30  
   
 // 不过，没有传进来的参数z，和参数的第3个索引值是不共享的  
  
 z = 40;  
 alert(arguments[2]); // undefined  
   
 arguments[2] = 50;  
 alert(z); // 40  
   
}  
   
foo(10, 20);

这个例子的代码，在当前版本的Google Chrome浏览器里有一个bug  — 即使没有传递参数z，z和arguments[2]仍然是共享的。

**处理上下文代码的2个阶段**

现在我们终于到了本文的核心点了。执行上下文的代码被分成两个基本的阶段来处理：

1. 进入执行上下文
2. 执行代码

变量对象的修改变化与这两个阶段紧密相关。

注：这2个阶段的处理是一般行为，和上下文的类型无关（也就是说，在全局上下文和函数上下文中的表现是一样的）。

**进入执行上下文**

当进入执行上下文(代码执行之前)时，VO里已经包含了下列属性(前面已经说了)：

**函数的所有形参(如果我们是在函数执行上下文中)**

    — 由名称和对应值组成的一个变量对象的属性被创建；没有传递对应参数的话，那么由名称和undefined值组成的一种变量对象的属性也将被创建。

**所有函数声明(FunctionDeclaration, FD)**

    —由名称和对应值（函数对象(function-object)）组成一个变量对象的属性被创建；如果变量对象已经存在相同名称的属性，则完全替换这个属性。

**所有变量声明(var, VariableDeclaration)**

    — 由名称和对应值（undefined）组成一个变量对象的属性被创建；如果变量名称跟已经声明的形式参数或函数相同，则变量声明不会干扰已经存在的这类属性。

让我们看一个例子：

function test(a, b) {  
 var c = 10;  
 function d() {}  
 var e = function \_e() {};  
 (function x() {});  
}  
   
test(10); // call

当进入带有参数10的test函数上下文时，AO表现为如下：

AO(test) = {  
 a: 10,  
 b: undefined,  
 c: undefined,  
 d: <reference to FunctionDeclaration "d">  
 e: undefined  
};

注意，AO里并不包含函数“x”。这是因为“x” 是一个函数表达式(FunctionExpression, 缩写为 FE) 而不是函数声明，函数表达式不会影响VO。 不管怎样，函数“\_e” 同样也是函数表达式，但是就像我们下面将看到的那样，因为它分配给了变量 “e”，所以它可以通过名称“e”来访问。 函数声明FunctionDeclaration与函数表达式FunctionExpression 的不同，将在第15章Functions进行详细的探讨，也可以参考本系列第2章[揭秘命名函数表达](http://www.cnblogs.com/TomXu/archive/2011/12/29/2290308.html)式来了解。

这之后，将进入处理上下文代码的第二个阶段 — 执行代码。

**代码执行**

这个周期内，AO/VO已经拥有了属性(不过，并不是所有的属性都有值，大部分属性的值还是系统默认的初始值undefined )。

还是前面那个例子, AO/VO在代码解释期间被修改如下：

AO['c'] = 10;  
AO['e'] = <reference to FunctionExpression "\_e">;

再次注意，因为FunctionExpression“\_e”保存到了已声明的变量“e”上，所以它仍然存在于内存中。而FunctionExpression “x”却不存在于AO/VO中，也就是说如果我们想尝试调用“x”函数，不管在函数定义之前还是之后，都会出现一个错误“x is not defined”，未保存的函数表达式只有在它自己的定义或递归中才能被调用。

另一个经典例子：

alert(x); // function  
   
var x = 10;  
alert(x); // 10  
   
x = 20;  
   
function x() {};  
   
alert(x); // 20

为什么第一个alert “x” 的返回值是function，而且它还是在“x” 声明之前访问的“x” 的？为什么不是10或20呢？因为，根据规范函数声明是在当**进入上下文**时填入的； 同意周期，在进入上下文的时候还有一个变量声明“x”，那么正如我们在上一个阶段所说，变量声明在顺序上跟在函数声明和形式参数声明之后，而且在这个**进入上下文**阶段，变量声明不会干扰VO中已经存在的同名函数声明或形式参数声明，因此，在进入上下文时，VO的结构如下：

VO = {};  
   
VO['x'] = <reference to FunctionDeclaration "x">  
   
// 找到var x = 10;  
// 如果function "x"没有已经声明的话  
// 这时候"x"的值应该是undefined  
// 但是这个case里变量声明没有影响同名的function的值  
   
VO['x'] = <the value is not disturbed, still function>

紧接着，在执行代码阶段，VO做如下修改：

VO['x'] = 10;  
VO['x'] = 20;

我们可以在第二、三个alert看到这个效果。

在下面的例子里我们可以再次看到，变量是在进入上下文阶段放入VO中的。(因为，虽然else部分代码永远不会执行，但是不管怎样，变量“b”仍然存在于VO中。)

if (true) {  
 var a = 1;  
} else {  
 var b = 2;  
}  
   
alert(a); // 1  
alert(b); // undefined,不是b没有声明，而是b的值是undefined

**关于变量**

通常，各类文章和JavaScript相关的书籍都声称：“不管是使用var关键字(在全局上下文)还是不使用var关键字(在任何地方)，都可以声明一个变量”。请记住，这是错误的概念：

任何时候，变量只能通过使用var关键字才能声明。

上面的赋值语句：

a = 10;

这仅仅是给全局对象创建了一个新属性(但它不是变量)。“不是变量”并不是说它不能被改变，而是指它不符合ECMAScript规范中的变量概念，所以它“不是变量”(它之所以能成为全局对象的属性，完全是因为VO(globalContext) === global，大家还记得这个吧？)。

让我们通过下面的实例看看具体的区别吧：

alert(a); // undefined  
alert(b); // "b" 没有声明  
   
b = 10;  
var a = 20;

所有根源仍然是VO和进入上下文阶段和代码执行阶段：

进入上下文阶段：

VO = {  
 a: undefined  
};

我们可以看到，因为“b”不是一个变量，所以在这个阶段根本就没有“b”，“b”将只在代码执行阶段才会出现(但是在我们这个例子里，还没有到那就已经出错了)。

让我们改变一下例子代码：

alert(a); // undefined, 这个大家都知道，  
   
b = 10;  
alert(b); // 10, 代码执行阶段创建  
   
var a = 20;  
alert(a); // 20, 代码执行阶段修改

关于变量，还有一个重要的知识点。变量相对于简单属性来说，变量有一个特性(attribute)：{DontDelete},这个特性的含义就是不能用delete操作符直接删除变量属性。

a = 10;  
alert(window.a); // 10  
   
alert(delete a); // true  
   
alert(window.a); // undefined  
   
var b = 20;  
alert(window.b); // 20  
   
alert(delete b); // false  
   
alert(window.b); // still 20

但是这个规则在有个上下文里不起走样，那就是eval上下文，变量没有{DontDelete}特性。

eval('var a = 10;');  
alert(window.a); // 10  
   
alert(delete a); // true  
   
alert(window.a); // undefined

使用一些调试工具(例如：Firebug)的控制台测试该实例时，请注意，Firebug同样是使用eval来执行控制台里你的代码。因此，变量属性同样没有{DontDelete}特性，可以被删除。

**特殊实现: \_\_parent\_\_ 属性**

前面已经提到过，按标准规范，活动对象是不可能被直接访问到的。但是，一些具体实现并没有完全遵守这个规定，例如SpiderMonkey和Rhino；的实现中，函数有一个特殊的属性 \_\_parent\_\_，通过这个属性可以直接引用到活动对象（或全局变量对象），在此对象里创建了函数。

例如 (SpiderMonkey, Rhino)：

var global = this;  
var a = 10;  
   
function foo() {}  
   
alert(foo.\_\_parent\_\_); // global  
   
var VO = foo.\_\_parent\_\_;  
   
alert(VO.a); // 10  
alert(VO === global); // true

在上面的例子中我们可以看到，函数foo是在全局上下文中创建的，所以属性\_\_parent\_\_ 指向全局上下文的变量对象，即全局对象。

然而，在SpiderMonkey中用同样的方式访问活动对象是不可能的：在不同版本的SpiderMonkey中，内部函数的\_\_parent\_\_ 有时指向null ，有时指向全局对象。

在Rhino中，用同样的方式访问活动对象是完全可以的。

例如 (Rhino)：

var global = this;  
var x = 10;  
   
(function foo() {  
   
 var y = 20;  
   
 // "foo"上下文里的活动对象  
 var AO = (function () {}).\_\_parent\_\_;  
   
 print(AO.y); // 20  
   
 // 当前活动对象的\_\_parent\_\_ 是已经存在的全局对象  
 // 变量对象的特殊链形成了  
 // 所以我们叫做作用域链  
 print(AO.\_\_parent\_\_ === global); // true  
   
 print(AO.\_\_parent\_\_.x); // 10  
   
})();