

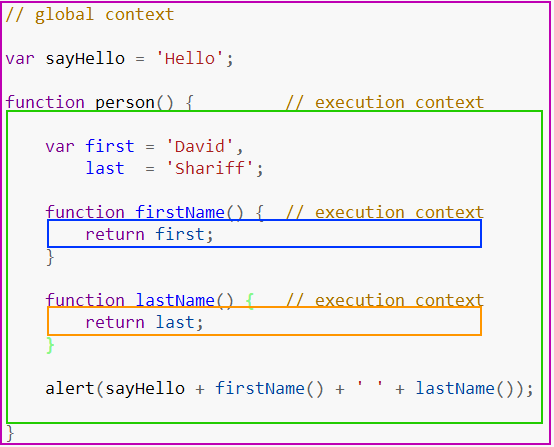
在这篇文章中，将比较深入地阐述下执行上下文 – [JavaScript](http://lib.csdn.net/base/javascript)中最基础也是最重要的一个概念。相信读完这篇文章后，你就会明白[javascript](http://lib.csdn.net/base/javascript)引擎内部在执行代码以前到底做了些什么，为什么某些函数以及变量在没有被声明以前就可以被使用，以及它们的最终的值是怎样被定义的。

**什么是执行上下文**

Javascript中代码的运行环境分为以下三种：

* 全局级别的代码 – 这个是默认的代码运行环境，一旦代码被载入，引擎最先进入的就是这个环境。
* 函数级别的代码 – 当执行一个函数时，运行函数体中的代码。
* Eval的代码 – 在Eval函数内运行的代码。

在网上可以找到很多阐述作用域的资源，为了使该文便于大家理解，我们可以将“执行上下文”看做当前代码的运行环境或者作用域。下面我们来看一个示例，其中包括了全局以及函数级别的执行上下文：

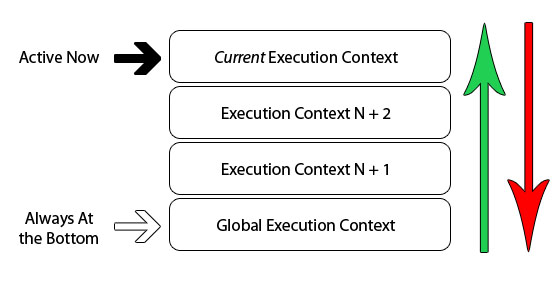


上图中，一共用4个执行上下文。紫色的代表全局的上下文；绿色代表person函数内的上下文；蓝色以及橙色代表person函数内的另外两个函数的上下文。注意，不管什么情况下，只存在一个全局的上下文，该上下文能被任何其它的上下文所访问到。也就是说，我们可以在person的上下文中访问到全局上下文中的sayHello变量，当然在函数firstName或者lastName中同样可以访问到该变量。

至于函数上下文的个数是没有任何限制的，每到调用执行一个函数时，引擎就会自动新建出一个函数上下文，换句话说，就是新建一个局部作用域，可以在该局部作用域中声明私有变量等，在外部的上下文中是无法直接访问到该局部作用域内的元素的。在上述例子的，内部的函数可以访问到外部上下文中的声明的变量，反之则行不通。那么，这到底是什么原因呢？引擎内部是如何处理的呢？

**执行上下文堆栈**

在浏览器中，javascript引擎的工作方式是单线程的。也就是说，某一时刻只有唯一的一个事件是被激活处理的，其它的事件被放入队列中，等待被处理。下面的示例图描述了这样的一个堆栈：



我们已经知道，当javascript代码文件被浏览器载入后，默认最先进入的是一个全局的执行上下文。当在全局上下文中调用执行一个函数时，程序流就进入该被调用函数内，此时引擎就会为该函数创建一个新的执行上下文，并且将其压入到执行上下文堆栈的顶部。浏览器总是执行当前在堆栈顶部的上下文，一旦执行完毕，该上下文就会从堆栈顶部被弹出，然后，进入其下的上下文执行代码。这样，堆栈中的上下文就会被依次执行并且弹出堆栈，直到回到全局的上下文。请看下面一个例子：

(function foo(i) {

if (i === 3) {

return;

}

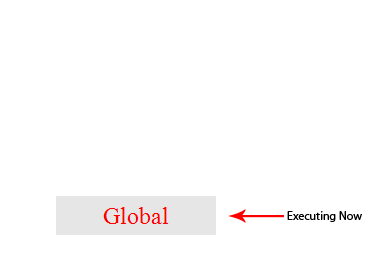
else {

foo(++i);

}

}(0));

上述foo被声明后，通过()运算符强制直接运行了。函数代码就是调用了其自身3次，每次是局部变量i增加1。每次foo函数被自身调用时，就会有一个新的执行上下文被创建。每当一个上下文执行完毕，该上上下文就被弹出堆栈，回到上一个上下文，直到再次回到全局上下文。真个过程抽象如下图:



由此可见 ，对于执行上下文这个抽象的概念，可以归纳为以下几点：

* 单线程
* 同步执行
* 唯一的一个全局上下文
* 函数的执行上下文的个数没有限制
* 每次某个函数被调用，就会有个新的执行上下文为其创建，即使是调用的自身函数，也是如此。

**执行上下文的建立过程**

我们现在已经知道，每当调用一个函数时，一个新的执行上下文就会被创建出来。然而，在javascript引擎内部，这个上下文的创建过程具体分为两个阶段:

1. 建立阶段(发生在当调用一个函数时，但是在执行函数体内的具体代码以前)
   * 建立变量，函数，arguments对象，参数
   * 建立作用域链
   * 确定this的值
2. 代码执行阶段:
   * 变量赋值，函数引用，执行其它代码

实际上，可以把执行上下文看做一个对象，其下包含了以上3个属性：

(executionContextObj = {

variableObject: { /\* 函数中的arguments对象, 参数, 内部的变量以及函数声明 \*/ },

scopeChain: { /\* variableObject 以及所有父执行上下文中的variableObject \*/ },

this: {}

}

**建立阶段以及代码执行阶段的详细分析**

确切地说，执行上下文对象（上述的executionContextObj）是在函数被调用时，但是在函数体被真正执行以前所创建的。函数被调用时，就是我上述所描述的两个阶段中的第一个阶段 – 建立阶段。这个时刻，引擎会检查函数中的参数，声明的变量以及内部函数，然后基于这些信息建立执行上下文对象（executionContextObj）。在这个阶段，variableObject对象，作用域链，以及this所指向的对象都会被确定。

上述第一个阶段的具体过程如下：

1. 找到当前上下文中的调用函数的代码
2. 在执行被调用的函数体中的代码以前，开始创建执行上下文
3. 进入第一个阶段-建立阶段:
   * 建立variableObject对象:
     1. 建立arguments对象，检查当前上下文中的参数，建立该对象下的属性以及属性值
     2. 检查当前上下文中的函数声明：

每找到一个函数声明，就在variableObject下面用函数名建立一个属性，属性值就是指向该函数在内存中的地址的一个引用

如果上述函数名已经存在于variableObject下，那么对应的属性值会被新的引用所覆盖。

* + 初始化作用域链
  + 确定上下文中this的指向对象

1. 代码执行阶段:

执行函数体中的代码，一行一行地运行代码，给variableObject中的变量属性赋值。

下面来看个具体的代码示例:

function foo(i) {

var a = 'hello';

var b = function privateB() {

};

function c() {

}

}

foo(22);

在调用foo(22)的时候，建立阶段如下:

fooExecutionContext = {

variableObject: {

arguments: {

0: 22,

length: 1

},

i: 22,

c: pointer to function c()

a: undefined,

b: undefined

},

scopeChain: { ... },

this: { ... }

}

由此可见，在建立阶段，除了arguments，函数的声明，以及参数被赋予了具体的属性值，其它的变量属性默认的都是undefined。一旦上述建立阶段结束，引擎就会进入代码执行阶段，这个阶段完成后，上述执行上下文对象如下:

fooExecutionContext = {

variableObject: {

arguments: {

0: 22,

length: 1

},

i: 22,

c: pointer to function c()

a: 'hello',

b: pointer to function privateB()

},

scopeChain: { ... },

this: { ... }

}

我们看到，只有在代码执行阶段，变量属性才会被赋予具体的值。

**局部变量作用域提升的缘由**

在网上一直看到这样的总结： 在函数中声明的变量以及函数，其作用域提升到函数顶部，换句话说，就是一进入函数体，就可以访问到其中声明的变量以及函数。这是对的，但是知道其中的缘由吗？相信你通过上述的解释应该也有所明白了。不过在这边再分析一下。看下面一段代码:

(function() {

console.log(typeof foo); // function pointer

console.log(typeof bar); // undefined

var foo = 'hello',

bar = function() {

return 'world';

};

function foo() {

return 'hello';

}

}());​

上述代码定义了一个匿名函数，并且通过()运算符强制理解执行。那么我们知道这个时候就会有个执行上下文被创建，我们看到例子中马上可以访问foo以及bar变量，并且通过typeof输出foo为一个函数引用，bar为undefined。

为什么我们可以在声明foo变量以前就可以访问到foo呢？

因为在上下文的建立阶段，先是处理arguments, 参数，接着是函数的声明，最后是变量的声明。那么，发现foo函数的声明后，就会在variableObject下面建立一个foo属性，其值是一个指向函数的引用。当处理变量声明的时候，发现有var foo的声明，但是variableObject已经具有了foo属性，所以直接跳过。当进入代码执行阶段的时候，就可以通过访问到foo属性了，因为它已经就存在，并且是一个函数引用。

为什么bar是undefined呢？

因为bar是变量的声明，在建立阶段的时候，被赋予的默认的值为undefined。由于它只要在代码执行阶段才会被赋予具体的值，所以，当调用typeof(bar)的时候输出的值为undefined。

好了，到此为止，相信你应该对执行上下文有所理解了，这个执行上下文的概念非常重要，务必好好搞懂之！

本片文章，在前人的基础上，加上自己的理解，解释一下JavaScript的代码执行过程，顺道介绍一下执行环境和闭包的相关概念。

分为两部分。第一部分是了解执行环境的相关概念，第二部分是通过实际代码了解具体执行过程中执行环境的切换。

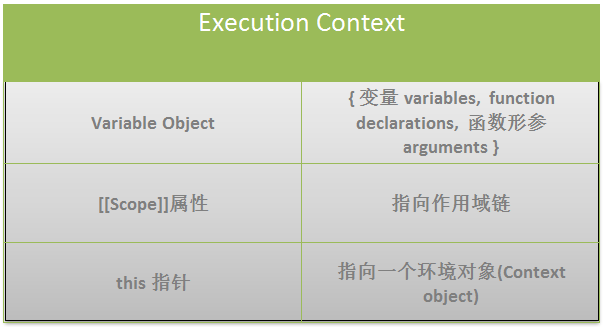
## 执行环境

### 执行环境的分类

* 1.全局执行环境  
  是JS代码开始运行时的默认环境（浏览器中为window对象）。全局执行环境的变量对象始终都是作用域链中的最后一个对象。
* 2.函数执行环境  
  当某个函数被调用时，会先创建一个执行环境及相应的作用域链。然后使用arguments和其他命名参数的值来初始化执行环境的变量对象。
* 3.使用eval()执行代码

### 执行上下文（执行环境）的组成

执行环境（execution context，EC）或称之为执行上下文，是JS中一个极为重要的概念。当JavaScript代码执行时，会进入不同的执行上下文，而每个执行上下文的组成，基本如下：



 变量对象（Variable object，VO）: 变量对象，即包含变量的对象，除了我们无法访问它外，和普通对象没什么区别

 [[Scope]]属性:数组。作用域链是一个由变量对象组成的带头结点的单向链表，其主要作用就是用来进行变量查找。而[[Scope]]属性是一个指向这个链表头节点的指针。

 this: 指向一个环境对象，注意是一个对象，而且是一个普通对象，而不是一个执行环境。

若干执行上下文会构成一个执行上下文栈（Execution context stack，ECS）。而所谓的执行上下文栈，举个例子，比如下面的代码

var a = "global var";

function foo(){

console.log(a);

}

function outerFunc(){

var b = "var in outerFunc";

console.log(b);

function innerFunc(){

var c = "var in innerFunc";

console.log(c);

foo();

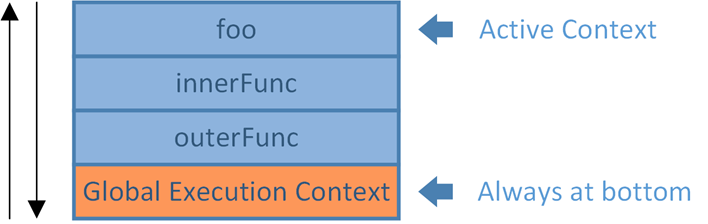
}

innerFunc();

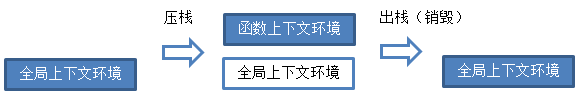
}

outerFunc()

代码首先进入Global Execution Context，然后依次进入outerFunc，innerFunc和foo的执行上下文，执行上下文栈就可以表示为：



执行全局代码时，会产生一个执行上下文环境，每次调用函数都又会产生执行上下文环境。当函数调用完成时，这个上下文环境以及其中的数据都会被消除，再重新回到全局上下文环境。处于活动状态的执行上下文环境只有一个。



## 产生执行上下文的两个阶段

当一段JS代码执行的时候，JS解释器会通过两个阶段去产生一个EC

* 创建阶段（当函数被调用，但是开始执行函数内部代码之前）
  + 创建变量对象VO
  + 设置[[Scope]]属性的值
  + 设置this的值
  + 激活/代码执行阶段
* 初始化变量对象，即设置变量的值、函数的引用，然后解释/执行代码。

### 创建变量对象VO过程

* 1.根据函数的参数，创建并初始化arguments object
* 2.扫描函数内部代码，查找函数声明（function declaration）
  + 对于所有找到的函数声明，将函数名和函数引用存入VO中
  + 如果VO中已经有同名函数，那么就进行覆盖
* 3.扫描函数内部代码，查找变量声明（Variable declaration）
  + 对于所有找到的变量声明(通过var声明)，将变量名存入VO中，并初始化为undefined
  + 如果变量名跟已经声明的形参或函数相同，则什么也不做

## 通过一段代码来了解JavaScript代码的执行

我们举例说明，假如我们有一个js文件，内容如下：

var global\_var1 = 10;

function global\_function1(parameter\_a){

var local\_var1 = 10 ;

return local\_var1 + parameter\_a + global\_var1;

}

var global\_sum = global\_function1(10);

alert(global\_sum);

下面我们来一步一步说明解释器是如何执行这段代码的：

### 1.创建全局上下文

首先，在解释器眼中，global\_var1、global\_sum叫做全局变量，因为它们不属于任何函数。local\_var1叫做局部变量，因为它定义在函数global\_function1内部。global\_function1叫做全局函数，因为它没有定义在任何函数内部。

然后，解释器开始扫描这段代码，为执行这段代码做了一些准备工作——**创建了一个全局上下文**。

全局上下文，可以把它看成一个JavaScript对象，姑且称之为global\_context。这个对象是解释器创建的，当然也是由解释器使用。（我们的JavaScript代码是接触不到这个对象的）

global\_context对象大概是这个样子的

global\_context = {

Variable\_Object :{......},

Scope :[......],

this :{......}

}

可以看到，global\_context有三个属性

* Variable\_Object（以下简称VO）  
  {  
  global\_var1：undefined  
  global\_function1：函数 global\_function1的地址  
  global\_sum：undefined  
  }

解释器在VO中记录了变量全局变量global\_var1、global\_sum，但它们的值现在是undefined的，还记录了全局函数global\_function1，但是没有记录局部变量local\_var1。VO的原型是Object.prototype。

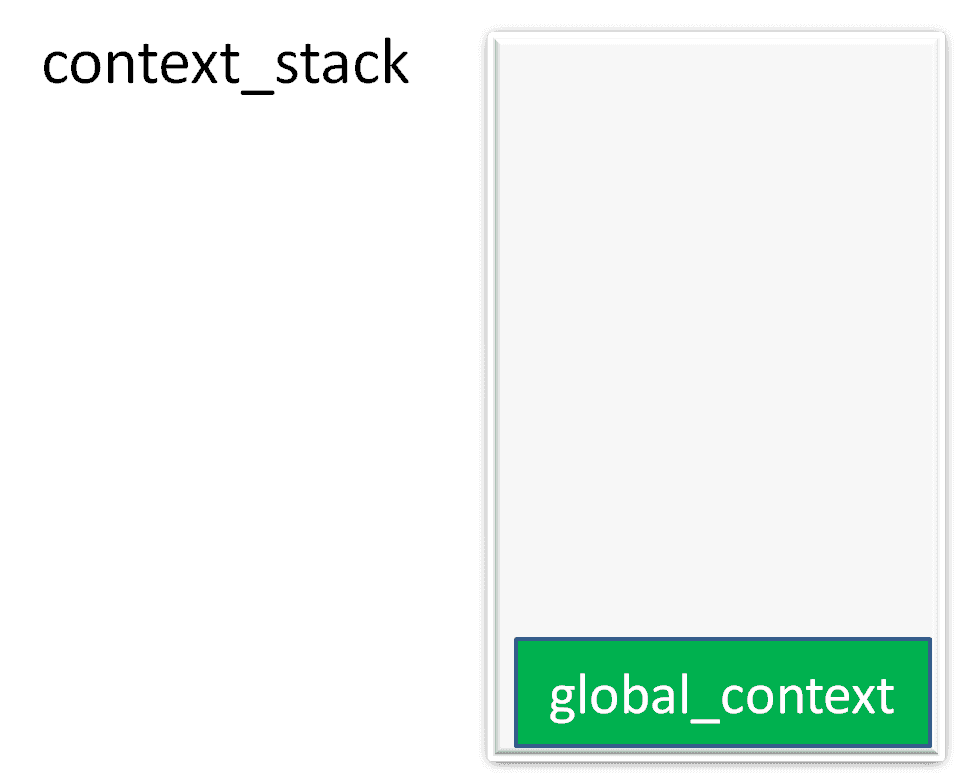
* Scope数组中的内容如下：

[ global\_context.Variable\_Object ]

* 我们看到，Scope数组中只有一个对象，就是前面刚创建的对象VO。
* this

this的值现在是undefined

global\_context对象被解释器压入一个栈中，不妨叫这个栈为context\_stack。现在的context\_stack是这样的：



创建出global\_context后，解释器又偷偷摸摸干了一件事，它给global\_function1设置了一个内部属性，也叫scope，它的值就是global\_context中的scope！也就是说，现在:

global\_function1.scope === [ global\_context.Variable\_Object ];

我们获取不到global\_function1的scope属性的，只有解释器自己能获取到。

### 2.逐行执行代码

解释器在创建了全局上下文后，就开始执行这段代码了。

#### 第一句：

var global\_var1 = 10;

解释器会把VO中的global\_var1属性的值设为10。现在global\_context对象变成了这样：

global\_context = {

Variable\_Object :{

global\_var1：10,

global\_function1：函数 global\_function1的地址,

global\_sum：undefined

},

Scope :[ global\_context.Variable\_Object ],

this :undefined

}

#### 第二句：

解释器继续执行我们的代码，它碰到了声明式函数global\_function1，由于在创建global\_context对象时，它就已经记录好了该函数，所以现在它什么也不用做。

#### 第三句：

var global\_sum = global\_function1(10);

解释器看到，我们在这里调用了函数global\_function1(解释器已经提前在global\_context的VO中记录下了global\_function1，所以它知道我们这里是一个函数调用)，并且传入了一个参数10，函数的返回结果赋值给了全局变量global\_sum。

***解释器并没有立即执行函数中的代码***，因为它要为函数global\_function1创建一个专门的context，我们叫它执行上下文（execute\_context）吧，因为每当解释器要执行一个函数时，都会创建一个类似的context。

execute\_context也是一个对象，并且与global\_context还很像，下面是它里面的内容：

execute\_context = {

Variable\_Object :{

parameter\_a：10,

local\_var1：undefined,

arguments：[10]

},

Scope :[execute\_context.Variable\_Object, global\_context.Variable\_Object ],

this :undefined

}

我们看到，execute\_context与global\_context相比，有以下几点变化：

* VO
  + 首先记录了函数的形式参数parameter\_a，并且给它赋值10，这个10就是我们调用函数时传递进去的。
  + 然后记录了函数体内的局部变量local\_var1，它的值还是undefined。
  + 然后是一个arguments属性，它的值是一个数组，里面只有一个10。

你可能疑惑，不是已经在parameter\_a中记录了参数10了吗，为什么解释器还要搞一个arguments，再来记录一遍呢？原因是如果我们这样调用函数：

global\_function1(10,20,30);

在JavaScript中是不违法的。此时VO中的arguments会变成这样：

arguments:[10,20,30]

parameter\_a的值还是10。可见，arguments是专门记录我们传进去的所有参数的。

* Scope

Scope属性仍然是一个数组，只不过里面的元素多了个execute\_context.Variable\_Object，并且排在了global\_context.Variable\_Object前面。

解释器是根据什么规则决定Scope中的内容的呢？答案非常简单：

execute\_context.Scope = execute\_context.Variable\_Object + global\_function1.scope。

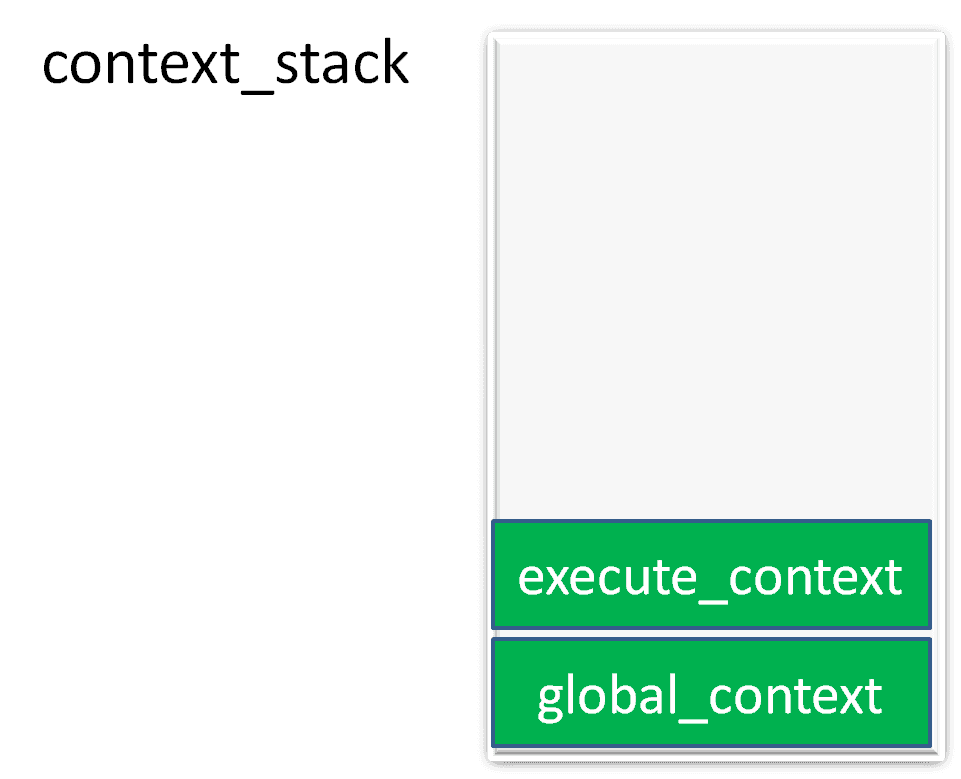
也就是说，每当要执行一个函数时，解释器都会将执行上下文（execute\_context）中Scope数组的第一个元素设为该执行上下文（execute\_context）的VO对象，然后取出函数创建时保存在函数中的scope属性（本文中则是global\_function1.scope），将其添加到执行上下文（execute\_context）Scope数组的后面。

我们知道，global\_function1是在global\_context下创建的，创建的时候，它的scope属性被设置成了global\_context的Scope，里面只有一个global\_context.Variable\_Object，于是这个对象被添加到execute\_context.Scope数组中execute\_context.Variable\_Object对象后面。

任何一个函数在创建时，解释器都会把它所在的执行上下文或者全局上下文的Scope属性对应的数组设置给函数的scope属性，这个属性是函数“与生俱来”的。

* this  
  this的值此时仍然是undefined的(但不同的解释器可能有不同的赋值)

解释器为函数global\_function1创建好了execute\_context（执行上下文）后，会把这个上下文对象压入context\_stack中，所以，现在的context\_stack是这样的：



### 准备执行函数内的代码

做好了准备工作，解释器开始执行函数里面的代码了，此时我们称函数是在执行上下文中运行的。

#### 第一句

var local\_var1 = 10 ;

它的处理办法很简单，将execute\_context的VO中的local\_var1赋值为10。这一点与在global\_context下执行的变量赋值语句的处理一样。此时的execute\_context变成这样：

execute\_context = {

Variable\_Object :{

parameter\_a：10,

local\_var1：10, //为local\_var1赋值10

arguments：[10]

},

Scope :[execute\_context.Variable\_Object, global\_context.Variable\_Object ],

this :undefined

}

#### 第二句

return local\_var1 + parameter\_a + global\_var1;

 解释器进一步考察语句，发现这是一个返回语句，于是它开始计算return 后面的表达式的值。

 在表达式中它首先碰到了变量local\_var1，它首先在execute\_context的Scope中依次查找，在第一个元素execute\_context的VO发现了local\_var1，并且知道它的值是10

 然后解释器继续前进，碰到了变量parameter\_a，它如法炮制，在execute\_context的VO中发现了parameter\_a，并且确定它的值是10。

 接着发现 global\_var1，解释器从execute\_context的Scope第一个元素execute\_context.VO中查找，没有发现global\_var1。继续查看Scope数组的第二个元素，即global\_context.VO，发现并且确定了它的值为10。

 于是，解释器将三个变量值相加得到了30，然后就返回了。

 此时，解释器知道函数已经执行完了，那么它为这个函数创建的执行上下文也没有用了，于是，它将execute\_context从context\_stack中弹出，由于没有其他对象引用着execute\_context，解释器就把它销毁了。现在context\_stack中又只剩下了global\_context。

#### 第三句

var global\_sum = 30;

现在解释器又回到全局上下文中执行代码了，这时它要把30赋值给sum，方法就是更改global\_context中的VO对象的global\_sum属性的值。

#### 第四句

alert(global\_sum);

解释器继续前进，碰到了语句alert(global\_sum);很简单，就是发出一个弹窗，弹窗的内容就是global\_sum的值30，当我们点击弹窗上的确定按钮后，解释器知道，这段代码终于执行完了，它会打扫战场，把global\_context，context\_stack等资源全部销毁。

### 再遇闭包

现在，知道了上下文，函数的scope属性的知识后，我们就可以开始学习闭包了。让我们将上面的js代码改成这样：

var global\_var1 = 10;

function global\_function1(parameter\_a){

var local\_var1 = 10 ;

function local\_function1(parameter\_b){

return parameter\_b + local\_var1 + parameter\_a + global\_var1;

}

return local\_function1 ;

}

var global\_sum = global\_function1(10);

alert(global\_sum(10));

这段代码与原先的代码最大的不同是，在global\_function1内部，我们创建了一个函数local\_function1，并且将它作为返回值。

当解释器执行函数global\_function1时，仍然会为它创建执行上下文，只不过此时execute\_context.VO中多了一个函数属性local\_function1。然后，解释器就会开始执行global\_function1中的代码。

我们直接从创建local\_function1语句开始分析，看解释器是怎么执行的，闭包的所有秘密就隐藏在其中。

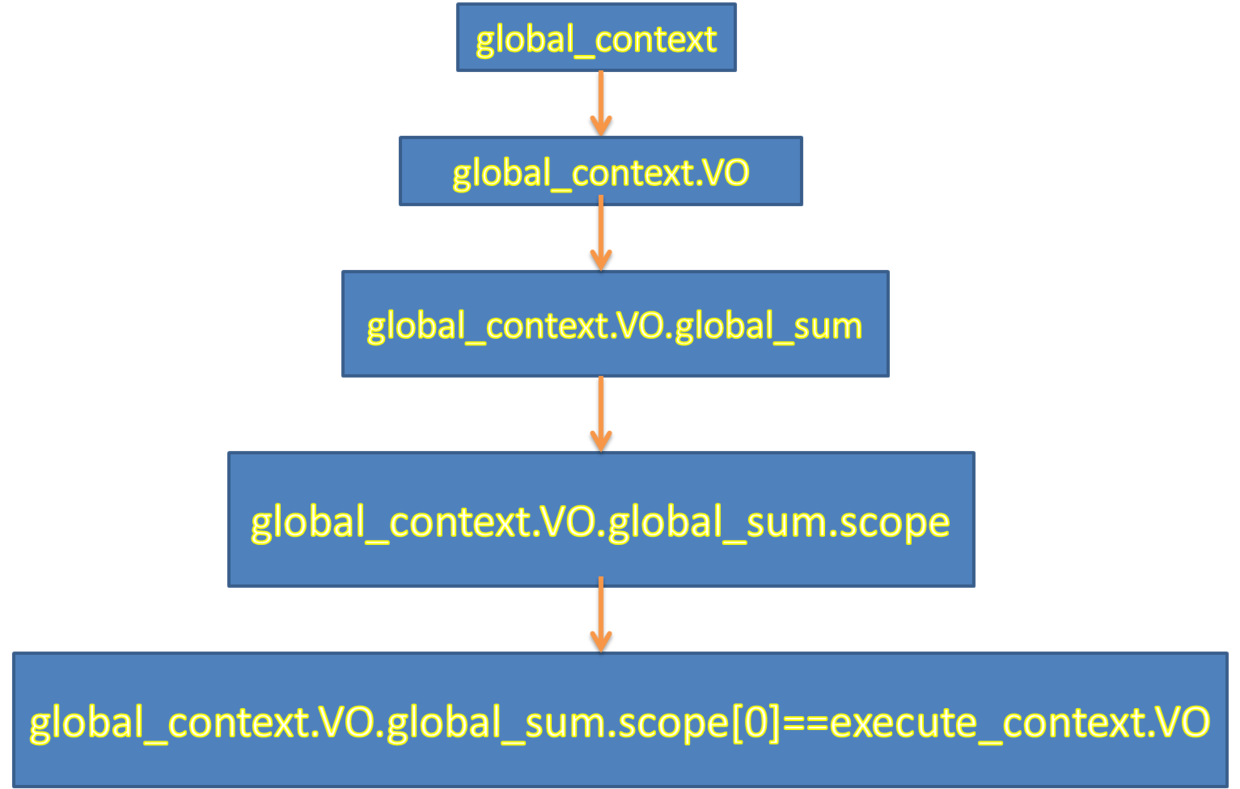
当解释器在execute\_context中执行创建local\_function1时，它仍然会将execute\_context的Scope设置给函数local\_function1的scope属性，也就是这样：

local\_function1.scope = [ execute\_context.Variable\_Object, global\_context.Variable\_Object ]

然后，解释器碰到了返回语句，把local\_function1返回并赋值给了全局变量global\_sum。此时global\_context的VO中global\_sum的值就是函数local\_function1。

此时，函数global\_function1已经执行完了，解释器会怎么处理它的execute\_context呢？

首先，解释器会把execute\_context从context\_stack中弹出，但并不把它完全销毁，而是保留了execute\_context.Variable\_Object对象，把它转移到了另一块堆内存中。为什么不销毁呢？因为还有对象引用着它呢。引用链如下：



这意味着什么呢？这说明，当global\_function1结束返回后，它的形式参数parameter\_a，局部变量local\_var1以及局部函数local\_function1都没有销毁，还仍然存在。这一点，与面向对象的语言Java中的经验完全不同，这也是闭包难以理解的根本所在。

下面我们的解释器继续执行语句alert(global\_sum(10));alert参数是对函数global\_sum的调用，global\_sum的参数为10，我们知道函数global\_sum的代码是这样的：

function local\_function1(parameter\_b){

return parameter\_b + local\_var1 + parameter\_a + global\_var1;

}

要执行这个函数，解释器仍然会为它创建一个执行上下文，我们姑且称之为local\_context2，这个对象的内容是这样的：

execute\_context2 = {

Variable\_Object :{

parameter\_b：10,

arguments：[10]

},

Scope :[execute\_context2.Variable\_Object, execute\_context.Variable\_Object, global\_context.Variable\_Object ],

this :undefined

}

这里我们重点看看Scope属性，它的第一个元素毫无疑问是execute\_context2.Variable\_Object，后面的元素是从local\_function1.scope属性中获得的，它是在local\_function1创建时所在的执行上下文的Scope属性决定的。

创建的execute\_context2压入context\_stack后，解释器开始执行语句

return parameter\_b + local\_var1 + parameter\_a + global\_var1;

对于该句中四个变量，解释器确定它们的值的办法一如既往的简单，***首先在当前执行上下文（也就是execute\_context2）的Scope的第一个元素中查找，第一个找不到就在第二个元素中查找，然后就是第三个，直至global\_context.Variable\_Object。***

然后，解释器就会将四个变量值相加后返回。弹出execute\_context2，此时execute\_context2已经没有对象引用着它，解释器就把它销毁了。

最后，alert函数会收到值40，然后发出一个弹窗，弹窗的内容就是40。程序结束

#### 说到现在，啥是闭包啊？

简单讲，当我们从函数global\_function1中返回另一个函数local\_function1时，由于local\_function1的scope属性中引用着为执行global\_function1创建的execute\_context.Variable\_Object对象，导致global\_function1在执行完毕后，它的execute\_context.Variable\_Object对象并不会被回收，此时我们称函数local\_function1是一个闭包，**因为它除了是一个函数外，还保存着创建它的执行上下文的变量信息**，使得我们在调用它时，仍然能够访问这些变量。

**函数将创建它的上下文中的VO对象封闭包含在自己的scope属性中，函数就变成了一个闭包**。从这个广泛的意义上来说，global\_function1也可以叫做闭包，因为它的scope内部属性也包含了创建它的全局上下文的变量信息，也就是global\_context.VO

作者：艾伦先生  
链接：https://www.jianshu.com/p/8f19e45fd1f1  
來源：简书  
简书著作权归作者所有，任何形式的转载都请联系作者获得授权并注明出处。

**二、如何从外部读取局部变量？**

出于种种原因，我们有时候需要得到函数内的局部变量。但是，前面已经说过了，正常情况下，这是办不到的，只有通过变通方法才能实现。

那就是在函数的内部，再定义一个函数。

　function f1(){

　　　　var n=999;

　　　　function f2(){

　　　　　　alert(n); // 999

　　　　}

　　}

在上面的代码中，函数f2就被包括在函数f1内部，这时f1内部的所有局部变量，对f2都是可见的。但是反过来就不行，f2内部的局部变量，对f1就是不可见的。这就是Javascript语言特有的"链式作用域"结构（chain scope），子对象会一级一级地向上寻找所有父对象的变量。所以，父对象的所有变量，对子对象都是可见的，反之则不成立。

既然f2可以读取f1中的局部变量，那么只要把f2作为返回值，我们不就可以在f1外部读取它的内部变量了吗！

function f1(){

　　　　var n=999;

　　　　function f2(){

　　　　　　alert(n);

　　　　}

　　　　return f2;

　　}

　　var result=f1();

　　result(); // 999

**三、闭包的概念**

上一节代码中的f2函数，就是闭包。

各种专业文献上的"闭包"（closure）定义非常抽象，很难看懂。我的理解是，闭包就是能够读取其他函数内部变量的函数。

由于在Javascript语言中，只有函数内部的子函数才能读取局部变量，因此可以把闭包简单理解成"定义在一个函数内部的函数"。

所以，在本质上，闭包就是将函数内部和函数外部连接起来的一座桥梁。

**四、闭包的用途**

闭包可以用在许多地方。它的最大用处有两个，一个是前面提到的可以读取函数内部的变量，另一个就是让这些变量的值始终保持在内存中。

怎么来理解这句话呢？请看下面的代码。

　function f1(){

　　　　var n=999;

　　　　nAdd=function(){n+=1}

　　　　function f2(){

　　　　　　alert(n);

　　　　}

　　　　return f2;

　　}

　　var result=f1();

　　result(); // 999

　　nAdd();

　　result(); // 1000

在这段代码中，result实际上就是闭包f2函数。它一共运行了两次，第一次的值是999，第二次的值是1000。这证明了，函数f1中的局部变量n一直保存在内存中，并没有在f1调用后被自动清除。

为什么会这样呢？原因就在于f1是f2的父函数，而f2被赋给了一个全局变量，这导致f2始终在内存中，而f2的存在依赖于f1，因此f1也始终在内存中，不会在调用结束后，被垃圾回收机制（garbage collection）回收。

这段代码中另一个值得注意的地方，就是"nAdd=function(){n+=1}"这一行，首先在nAdd前面没有使用var关键字，因此nAdd是一个全局变量，而不是局部变量。其次，nAdd的值是一个匿名函数（anonymous function），而这个匿名函数本身也是一个闭包，所以nAdd相当于是一个setter，可以在函数外部对函数内部的局部变量进行操作。

**五、使用闭包的注意点**

1）由于闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包，否则会造成网页的性能问题，在IE中可能导致内存泄露。解决方法是，在退出函数之前，将不使用的局部变量全部删除。

2）闭包会在父函数外部，改变父函数内部变量的值。所以，如果你把父函数当作对象（object）使用，把闭包当作它的公用方法（Public Method），把内部变量当作它的私有属性（private value），这时一定要小心，不要随便改变父函数内部变量的值。

**六、思考题**

如果你能理解下面两段代码的运行结果，应该就算理解闭包的运行机制了。代码片段一。

var name = "The Window";

　　var object = {

　　　　name : "My Object",

　　　　getNameFunc : function(){

　　　　　　return function(){

　　　　　　　　return this.name;

　　　　　　};

　　　　}

　　};

　　alert(object.getNameFunc()());//The window

var name = "The Window";

　　var object = {

　　　　name : "My Object",

　　　　getNameFunc : function(){

　　　　　　var that = this;

　　　　　　return function(){

　　　　　　　　return that.name;

　　　　　　};

　　　　}

　　};

　　alert(object.getNameFunc()());//my project;