在JavaScript的运行过程中，经常会遇到一些"奇怪"的行为，不理解为什么JavaScript会这么工作。

这时候可能就需要了解一下JavaScript执行过程中的相关内容了。

执行上下文

在JavaScript中有三种代码运行环境：

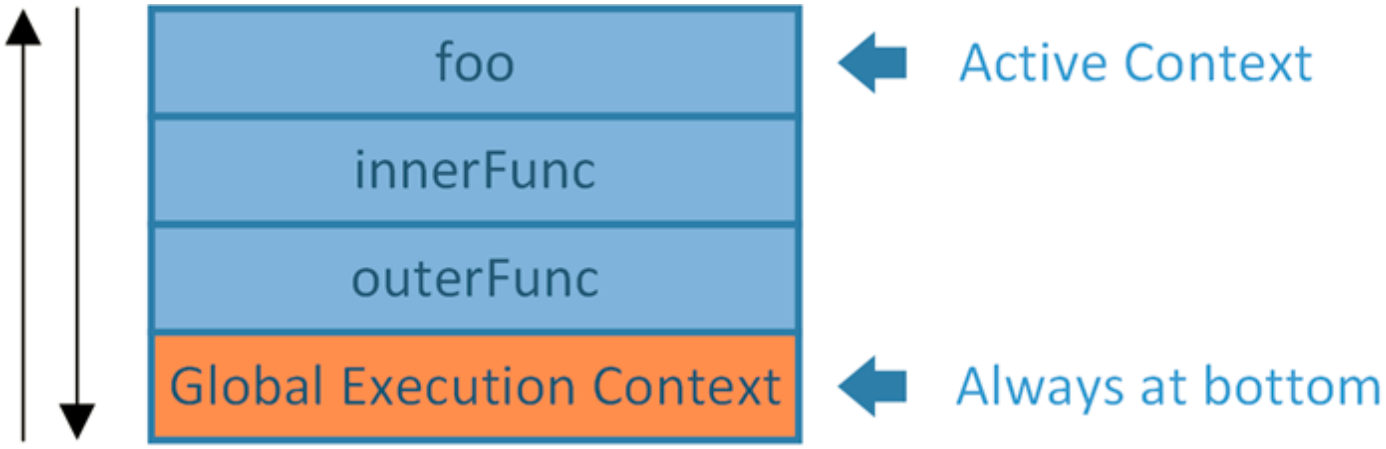
- Global Code
  - JavaScript代码开始运行的默认环境
- Function Code
  - 代码进入一个JavaScript函数
- Eval Code
  - 使用eval()执行代码

为了表示不同的运行环境，JavaScript中有一个执行上下文（**Execution context，EC**）的概念。也就是说，当JavaScript代码执行的时候，会进入不同的执行上下文，这些执行上下文就构成了一个执行上下文栈（**Execution context stack，ECS**）。

例如对如下面的JavaScript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：



当JavaScript代码执行的时候，第一个进入的总是默认的Global Execution Context，所以说它总是在ECS的最底部。

对于每个Execution Context都有三个重要的属性，变量对象（Variable object，VO），作用域链（Scope chain）和this。这三个属性跟代码运行的行为有很重要的关系，下面会一一介绍。

当然，除了这三个属性之外，根据实现的需要，Execution Context还可以有一些附加属性。

Execution context	
Variable object	{ vars, function declarations, arguments... }
Scope chain	[Variable object + all parent scopes]
thisValue	Context object

VO和AO

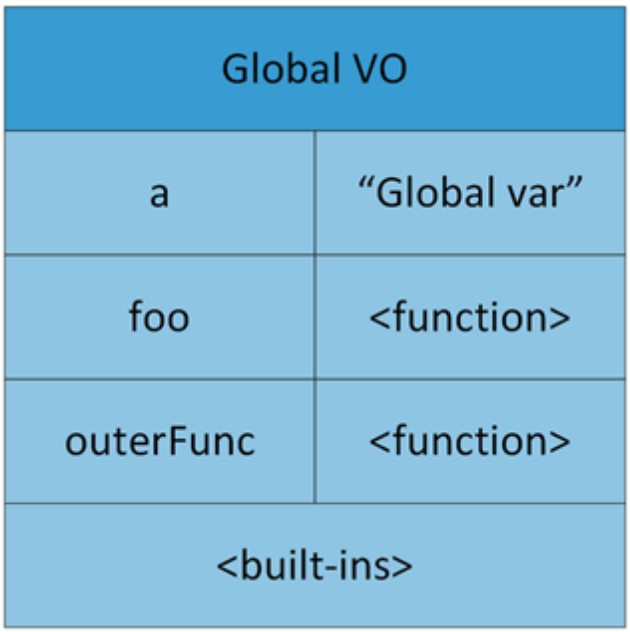
从上面看到，在Execution Context中，会保存变量对象（Variable object，VO），下面就看看变量对象是什么。

变量对象（**Variable object**）

变量对象是与执行上下文相关的数据作用域。它是一个与上下文相关的特殊对象，其中存储了在上下文中定义的变量和函数声明。也就是说，一般VO中会包含以下信息：

- 变量 (var, Variable Declaration);
- 函数声明 (Function Declaration, FD);
- 函数的形参

当JavaScript代码运行中，如果试图寻找一个变量的时候，就会首先查找VO。对于前面例子中的代码，Global Execution Context中的VO就可以表示如下：



注意，假如上面的例子代码中有下面两个语句，Global VO仍将不变。

```
(function bar(){}) // function expression, FE
baz = "property of global object"
```

也就是说，对于VO，是有下面两种特殊情况的：

- 函数表达式（与函数声明相对）不包含在VO之中
- 没有使用var声明的变量（这种变量是，"全局"的声明方式，只是给Global添加了一个属性，并不在VO中）

活动对象（**Activation object**）

只有全局上下文的变量对象允许通过VO的属性名称间接访问；在函数执行上下文中，VO是不能直接访问的，此时由激活对象(Activation Object,缩写为AO)扮演VO的角色。激活对象 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 Objects 是函数上下文里的激活对象AO中的内部对象，它包括下列属性：

1. callee：指向当前函数的引用
2. length： 真正传递的参数的个数
3. properties-indexes：就是函数的参数值(按参数列表从左到右排列)

对于VO和AO的关系可以理解为，VO在不同的Execution Context中会有不同的表现：当在Global Execution Context中，可以直接使用VO；但是，在函数Execution Context中，AO就会被创建。

```
AbstractVO (generic behavior of the variable instantiation process)
  |
  |==> GlobalContextVO
  |        (VO === this === global)
  |
  |==> FunctionContextVO
           (VO === AO, <arguments> object and <formal parameters> are added)
```

当上面的例子开始执行outerFunc的时候，就会有一个outerFunc的AO被创建：

Activation object	
arguments	{}
b	“var in outerFunc”
innerFunc	<function>

通过上面的介绍，我们现在了解了VO和AO是什么，以及他们之间的关系了。下面就需要看看JavaScript解释器是怎么执行一段代码，以及设置VO和AO了。

细看Execution Context

当一段JavaScript代码执行的时候，JavaScript解释器会创建Execution Context，其实这里会有两个阶段：

- 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Scope chain
  - 创建VO/AO（variables, functions and arguments）
  - 设置this的值
- 激活/代码执行阶段
  - 设置变量的值、函数的引用，然后解释/执行代码

这里想要详细介绍一下"创建VO/AO"中的一些细节，因为这些内容将直接影响代码运行的行为。

对于"创建VO/AO"这一步，JavaScript解释器主要做了下面的事情：

- 根据函数的参数，创建并初始化arguments object
- 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/AO中
  - 如果**VO/AO**中已经有同名的函数，那么就进行覆盖
- 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，将变量名存入VO/AO中，并初始化为"undefined"
  - 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

看下面的例子：

```
function foo(i) {
    var a = 'hello';
    var b = function privateB() {

    };
    function c() {

    }
}

foo(22);
```

对于上面的代码，在"创建阶段"，可以得到下面的Execution Context object：

```
fooExecutionContext = {
    scopeChain: { ... },
    variableObject: {
        arguments: {
            0: 22,
            length: 1
        },
        i: 22,
        c: pointer to function c()
        a: undefined,
        b: undefined
    },
    this: { ... }
}
```

在"激活/代码执行阶段"，Execution Context object就被更新为：

```
fooExecutionContext = {
    scopeChain: { ... },
    variableObject: {
        arguments: {
            0: 22,
            length: 1
        },
        i: 22,
        c: pointer to function c()
        a: 'hello',
        b: pointer to function privateB()
    },
    this: { ... }
}
```

例子分析

前面介绍了Execution Context，VO/AO等这么多的理论知识，当然是为了方便我们去分析代码中的一些行为。这里，就通过几个简单的例子，结合上面的概念来分析结果。

Example 1

首先看第一个例子：

```
(function(){
    console.log(bar);
    console.log(baz);

    var bar = 20;

    function baz(){
        console.log("baz");
    }

})()
```

在Chrome中运行代码运行后将输出：

```
undefined
function baz(){
        console.log("baz");
    }
```

代码解释：匿名函数会首先进入"创建结果"，JavaScript解释器会创建一个"Function Execution Context"，然后创建Scope chain，VO/AO和this。根据前面的介绍，解释器会扫描函数和变量声明，如下的AO会被创建：

Activation object	
arguments	{}
bar	undefined
baz	<function>

所以，对于bar，我们会得到"undefined"这个输出，表现的行为就是，我们在声明一个变量之前就访问了这个变量。这个就是JavaScript中"Hoisting"。

Example 2

接着上面的例子，进行一些修改：

```
(function(){
    console.log(bar);
    console.log(baz);

    bar = 20;
    console.log(window.bar);
    console.log(bar);

    function baz(){
        console.log("baz");
    }

})()
```

运行这段代码会得到"bar is not defined(…)"错误。当代码执行到"console.log(bar);"的时候，会去AO中查找"bar"。但是，根据前面的解释，函数中的"bar"并没有通过var关键字声明，所有不会被存放在AO中，也就有了这个错误。

注释掉"console.log(bar);"，再次运行代码，可以得到下面结果。"bar"在"激活/代码执行阶段"被创建。

```
function baz(){
        console.log("baz");
    }
20
20
```

Example 3

现在来看最后一个例子：

```
(function(){
    console.log(foo);
    console.log(bar);
    console.log(baz);

    var foo = function(){};

    function bar(){
        console.log("bar");
    }

    var bar = 20;
    console.log(bar);

    function baz(){
        console.log("baz");
    }

})()
```

代码的运行结果为：

```
undefined
function bar(){
        console.log("bar");
    }
function baz(){
        console.log("baz");
    }
20
```

代码中，最"奇怪"的地方应该就是"bar"的输出了，第一次是一个函数，第二次是"20"。

其实也很好解释，回到前面对"创建VO/AO"的介绍，在创建VO/AO过程中，解释器会先扫描函数声明，然后"foo: <function>"就被保存在了AO中；但解释器扫描变量声明的时候，虽然发现"var bar = 20;"，但是因为"foo"在AO中已经存在，所以就没有任何操作了。

但是，当代码执行到第二句"console.log(bar);"的时候，"激活/代码执行阶段"已经把AO中的"bar"重新设置了。

总结

本文介绍了JavaScript中的执行上下文（Execution Context），以及VO/AO等概念，最后通过几个例子展示了这几个概念对我们了解JavaScript代码运行的重要性。

通过对VO/AO在"创建阶段"的具体细节，如何扫描函数声明和变量声明，就可以对JavaScript中的"Hoisting"有清晰的认识。所以说，了解JavaScript解释器的行为，以及相关的概念，对理解JavaScript代码的行为是很有帮助的。

后面会对Execution Context中的Scope chain和this进行介绍。