JSinDeep1:探索执行环境(Execution Context)-ES3 篇

声明

JSinDeep 系列文章主要内容为作者对 ECMA-262 标准中一些概念的理解和探究，同时意在帮助大家快速理解。本着严谨的态度，同时又需避免 API 式的枯燥细节罗列。文章会以适当插图、例子去诠释概念，致力于通俗易懂。更具体、严谨、完整的描述建议阅读 ECMA262 文档。碍于作者水平有限，文中若有错误，欢迎大家批评指正。

*其它版本: English-Blog, English-PDF, Chinese-Blog, Chinese-PDF

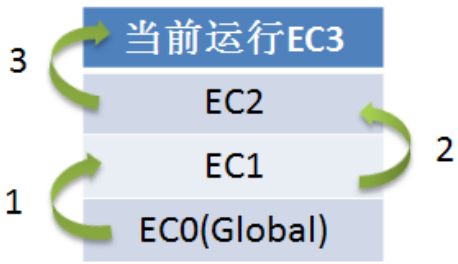
概要

在我们写 JavaScript 代码时会定义一些变量、函数等。解释器在执行这些代码时是如何处理并找到我们定义的这些数据的？在程序执行时，引用这些变量等操作的背后都发生了什么？本文主要探讨 ECMA-262-3 标准中的执行环境(Execution Context)及与之相关的一些内部机制和模型。

定义

当程序执行的控制权转移至 ECMAScript 可执行代码时，会进入到一个执行环境中（Execution Context，缩写为 EC）。在一个 EC 内也可能进入到一个新的 EC，这些 EC 逻辑上①会形成一个栈(Stack)。

EC 是程序运行时动态创建的。例如：每一个函数在被调用时都会创建一个 EC，重复调用函数（包含递归调用的情形）会重新创建新的 EC，而后放置在逻辑栈中。逻辑栈会在程序运行时随着新的函数调用、函数 return、未处理的异常抛出等情况动态变化，但逻辑栈的最顶部总是当前正运行的 EC，它的最底部总是全局 EC(Global Context)。



图：运行时的逻辑 EC 栈

①这里的“逻辑上”是因 ECMA262 标准避免限制实现者的思路，具体实现在遵循标准的前提下不受其它限制。因此这里所说的逻辑上的栈在具体实现时未必是通常意义上的栈。后面我们简称为：“逻辑栈”。

可执行代码的分类

由定义部分可知，“每一段可执行代码都有对应的 EC”，为方便按不同情况讨论，先了解几种*可执行代码*的类型。

a). 全局代码

全局代码是指在任何被解析为函数体的代码以外的最外层的代码。

```
var i = 0;          // 全局代码
function foo() {    // foo 函数定义部分为全局代码
  var j = 1;        // foo 函数体内为函数代码
}
var k = 2;          // 全局代码
```

*字符串中的动态被 eval 执行的代码除外，在下一类 Eval 代码中介绍

在程序执行前会初始化全局 EC， 逻辑栈（Logical Stack，简称为 LS）的结构类似于：

```
[伪代码]
LS = {
    globalContext
}
```

b). 函数代码

函数代码是指函数体中的代码。某一个函数体内的函数代码并不包含其内联的其它函数的函数体中的代码。

```
var i = 0;                          // 全局代码
function foo() {                    // 全局代码
  var j = 0;                        // foo 的函数代码
  function inner() {                // foo 的函数代码
    var k = 0;                      // inner 的函数代码
  }
  if (i++ == 0) foo();              // foo 的函数代码，递归调用一次 foo
}
foo();                              // 调用一次 foo
```

逻辑栈的结构类似于:

```
[伪代码]
```

```
LS = {                                  // 第一次调用 foo
    <foo>functionContext,               // 当前激活的 EC
    globalContext
}

LS = {                                  // 第二次递归调用 foo
    <foo>functionContextRecursively     // 当前激活的 EC
    <foo>functionContext,               // 等待<foo>functionContextRecursively
                                        // 返回并退出 LS 后激活
    globalContext
}
```

c). Eval 代码

当调用内置的 eval 方法且传入参数为字符串（而不是函数对象）时，该字符串中的代码为 Eval 代码。

```
eval("function foo2() {/* doSth(); */}");   // 字符串内为 eval 代码
eval(function foo3() {/* 函数代码 */});      // 多行注释外为全局代码
```

*ECMA262 原文中有精确的术语去定义该部分，本文重在辅助理解，文字表达在严谨和容易理解上可能有所折中。

对于 Eval 代码，根据 eval 语句所在位置有所区分，首先我们引入一个概念：调用环境(Calling Execution)，调用环境是指 Eval 函数调用位置所在的 EC。比如在全局 EC 中调用了 eval()，则该 Eval 代码的调用环境为该全局 EC，如果在某函数<func>functionContext 中调用了 eval(),则该 Eval 代码的调用环境为该函数<func>functionContext。

```
eval( 'var x = 10' );                   // 影响 globalContext
(function foo() {
    eval( 'var y = 20' );               //影响<foo>functionContext,
}());
alert(x);                               // 10
alert(y);                               // 运行时错误，y 未定义

[伪代码]
LS = {                                  // 调用 eval('var x = 10');
    evalCallingContext : globalContext,
    globalContext
}

LS = {                                  // eval('var x = 10');结束调用
    globalContext                       // evalCallingContext 出栈
```

```
}

LS = {                                  // 调用 foo, <foo>functionContext 入栈
    evalCallingContext :                // 调用 eval('var y = 20');
    <foo>funcitonContext,               // evalCallingContext 入栈
    globalContext
}
```

*Eval 代码中定义的变量属性不添加任何标签，可用 delete 删除。区别于正常的使用 var 关键字定义的变量会有{dontdelete}标签，delete 具有{dontdelete}标签的属性将返回 false 且不会修改该属性。由于 Firebug、Chrome Console 都采用 eval 的方式执行控制台代码，因此输入:var a = 1;console.log(delete window.a); 会返回 true，但正常写在全局代码里的该语句会返回 false。动手试一试吧。

变量初始化

每个 EC 都对应一个变量对象（Variable Object，缩写为 VO），在 EC 的初始化阶段，该 EC 范围内定义的变量、函数会作为属性添加到 VO 中。对于函数代码的情形，函数的参数也将作为属性添加到 VO 中。

*在 ES5 中 VO 会被新的词法环境(Lexical Environment)取代，在下节《2:EC in ES5》中我们再详细讨论。

更具体的，VO 的初始化按顺序依照如下步骤进行：

a). 参数处理

对于函数代码，每个函数传入的参数都会添加到 VO 中，name 为函数名，value 为传入的值。未传入的参数依然会在 VO 中，name 依然为参数名，value 为 undefined。

```
function foo(x, y, z) {
}
foo(1, 2);
```

```
[伪代码]
VO(<foo>functionContext) = {
    x : 1,
    y : 2,
    z : undefined
}
```

b). 函数定义处理

对于每一个函数定义(注意不是函数表达式)，在 VO 中创建一个属性，name 是函数名，value 为该函数对象。

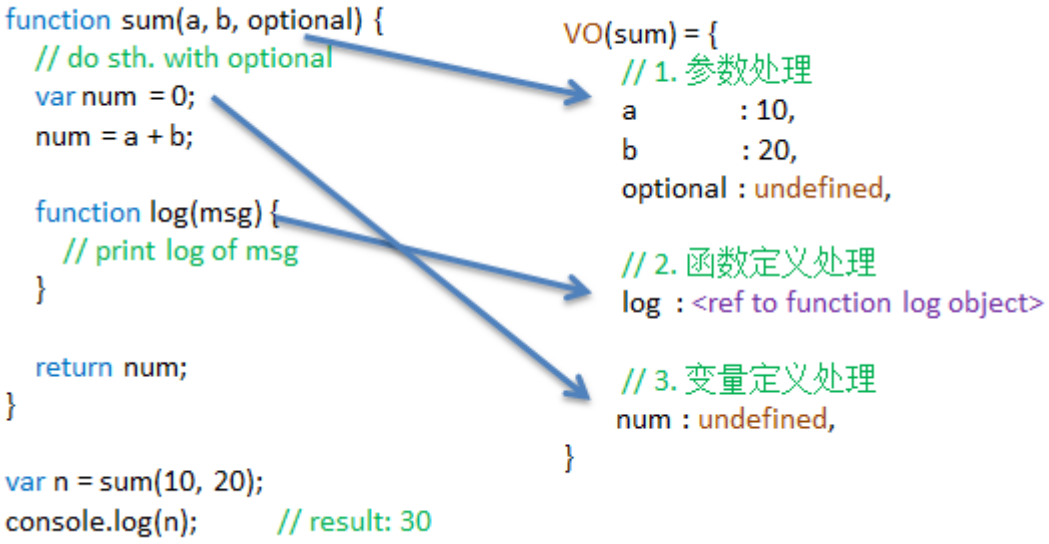
```
function foo(x, y, z) {
    function f() {
    }
}
foo(1, 2);
```

```
[伪代码]
VO(<foo>functionContext) = {
    x : 1,
    y : 2,
    z : undefined,
    f : <ref to function f object>
```

}

c). 变量定义处理

对于每一个变量定义，在 VO 中创建一个属性，name 是变量名，value 是 undefined。注意该阶段不包含变量赋值，如：var i = 0;拆分成定义和赋值两个部分。VO 初始化阶段处理的是定义部分，不包含 i = 0 赋值部分。



d). 命名冲突的解决 △

*标记△的章节对概念理解无明显作用，主要是对边界、命名冲突、具体算法的一些讨论。可选择性忽略。

解析参数时：

如定义的参数列表中有重名参数，该属性（重名的参数名作为属性）值为最右边的参数传入的值。

例：!function(x, y, y){alert(y);}(1, 2, 3) // alert 3

即使第二个 y 未传入，结果依然取自最后的 y，即：undefined

例：!function(x, y, y){alert(y);}(1, 2) // alert undefined

*function 左边的!作用是为了将 function...作为函数表达式来解析，而不是函数定义。等同于(function(){}()),当然，笔者喜欢用!更简洁一些…

解析函数定义时：

如函数名已在 VO 中存在，则采用**替换**的解决方式。

例：!function(x) {function x() {};alert(x);}(1) // alert function

解析变量定义时：

如变量名已在 VO 中存在，则采用**忽略**的解决方式。

例：`!function(x) {function x() {};var x; alert(x);}(1)  // still alert function`

注意 VO 初始化时的变量定义解析并不包括变量赋值。VO 初始化之后的 EC 变化过程在后面继续介绍。

例：`!function(x) {function x() {};var x = 1; alert(x);}(1)  // alert 1`

*正常情况下我们一般不会在定义函数时写两个相同的参数名（如 function(x, y, y)），也不会在定义一个函数后马上再定义一个重名的函数或变量。讨论该话题要么纯属好奇，要么寂寞空虚冷，极少数人为了实现 JS 引擎。

作用域链(Scope Chain)

每个 EC 在初始化时除了添加 VO 属性，还会初始化作用域链(scope chain)和 this 指针(本文主要内容非探讨 this，不作细讲)。作用域链是一个对象列表，在决定标识符含义时会通过作用域链进行变量查找。在 EC 初始化完成后，通过 with 语句和 catch 块可在运行时修改作用域链。

*例如 with 语句开始时，with 关键字紧接的括号中的对象会被放入作用域链中，影响 with 块内的变量查找，在 with 块结束时作用域链中的 with 对象会被移除，作用域链恢复到 with 语句执行前。

[伪代码]

```
activeExecutionContext = {
    VO : { },                                   // 参数、函数定义、变量定义…
    scopeChains : [VO_1, VO_2, …, VO_n ]        // 作用域链
    this : thisValue
}
```

对于全局 EC，作用域链中只有全局对象（Global Object，在下面详细说明）。对于函数 EC，作用域链类似于一个包含逻辑栈中每个 VO 对象的列表①，用来从逻辑栈的栈顶（当前激活的 EC）到栈底（全局的 EC）递归的进行变量查找。

①在 ES3 文档中并没有明确指出作用域链就是 VO 对象列表，但作用域链查找过程中每个对象都用于环境(context)间的变量查找，根据 JS 引擎具体实现可能有所不同。

全局对象(Global Object)

在进入任何 EC 前，会创建一个全局的、唯一的全局对象(Global Object)，其中包含内置的属性，如：Math, String, Date, parseInt 等等。另外，宿主(host)定义的一些属性也会初始化在全局对象中；例如，在 HTML DOM 中，会添加一个 window 属性，值为全局对象本身。

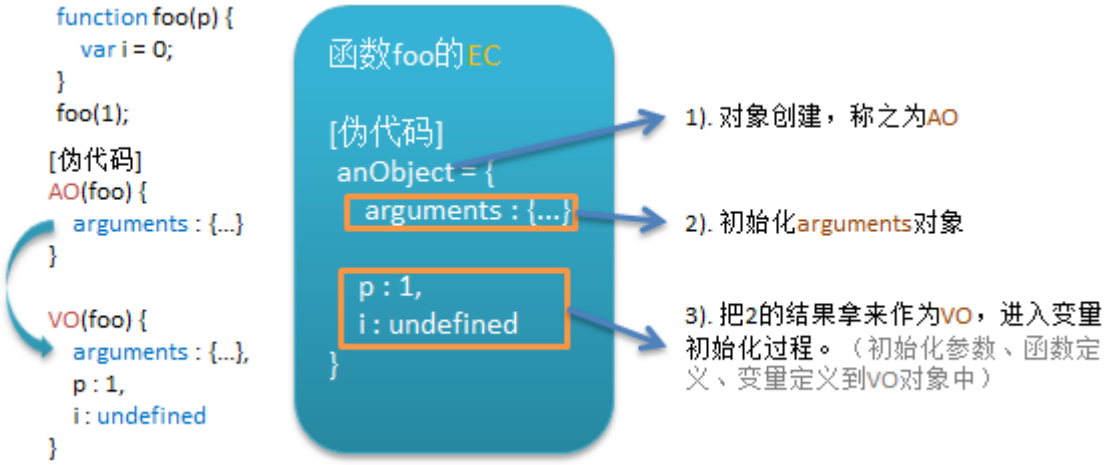
全局对象是内部机制的对象，但在HTML DOM中存在window这个特殊的自引用属性，使得全局对象可在程序中引用。

[伪代码]

```
globalObject = {
    Math : {…},
    String : {…},
    parseInt : <function>,
    …
    window : globalObject
}
```

激活对象(Activation Object)

对于函数代码的EC，在VO初始化之前会创建一个激活对象(Activation Object，缩写为AO)，该激活对象会被初始化一个arguments属性。随后，该AO对象被用作VO对象并完成前面的变量初始化阶段。



*在ES5中AO和VO都会统一的在词法环境(Lexical Environment)中定义，ES3中的AO和VO是同一个对象，只是在不同阶段（AO初始化->变量初始化）、特定环境下(只在函数EC中)的不同叫法。在下节《2:EC in ES5》中我们会详细讨论ES5词法环境。

执行环境运行

在EC的准备阶段（ES3中叫进入阶段，即：entering an execution context），VO、作用域链和this被初始化，然后开始执行EC中的代码片段。

```
function test(a, b) {
    var c = 10;
    function d() {}
    var e = function _e() {};
```

```
[伪代码] - <test>作用域链
ScopeChain(<test>functionContext) = {
    VO(<test>functionContext),
    globalObject
}
```

```
    (function x() {});
    b = 20;
}
test(10);
```

```
[伪代码] - VO 初始化：准备阶段
VO(<test>functionContext) = {
    a : 10,
    b : undefined,
    d : <ref to function d object>,
    c : undefined,
    e : undefined
}
[伪代码] - test 调用过程中的 VO 变化
VO(<test>functionContext) = {
    a : 10,
    b : 20,
    d : <ref to function d object>,
    c : 10,
    e : <ref to function _e object>
}
```

*由于篇幅有限，不对作用域链作更深入的探讨。

小结

ES3 中的执行环境按照全局代码、函数代码、Eval 代码三种情况进行区分。

1. 对于全局环境，先初始化全局对象(Global Object)，再初始化变量对象(Variable Object)，然后从全局代码开始执行。
2. 对于函数的执行环境，先初始化激活对象（Activation Object），而后该激活对象将被用作变量对象进行变量初始化过程，最后执行函数代码。
3. 对于 Eval 环境，执行环境取决于调用环境(Calling Object)，若在全局环境中调用 eval()，则 Eval 调用环境为该全局环境；若在函数的执行环境中调用 eval()，则 Eval 调用环境为该函数的执行环境。

在下一节《JSinDeep2:探索执行环境(Execution Context)-ES5 篇》，我们将共同探讨 ES5 中的词法环境(Lexical Environment)，相对于 ES3，在对性能的影响、VO&AO 等模型的统一、with 和 catch 语句的处理等多方面都会有所差异。

关于

作　者：@Bosn （花名：霍雍）
简　介：一淘数据部前端工程师
邮　箱：bosnma@live.cn
GitHub：https://github.com/bosnma/JsInDeep

参考

ECMA262-3 文档：

http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf

ECMA-262-3 in detail. Chapter 1. Execution Contexts

http://dmitrysoshnikov.com/ecmascript/chapter-1-execution-contexts/

ECMA-262-3 in detail. Chapter 2. Variable object.

http://dmitrysoshnikov.com/ecmascript/chapter-2-variable-object/