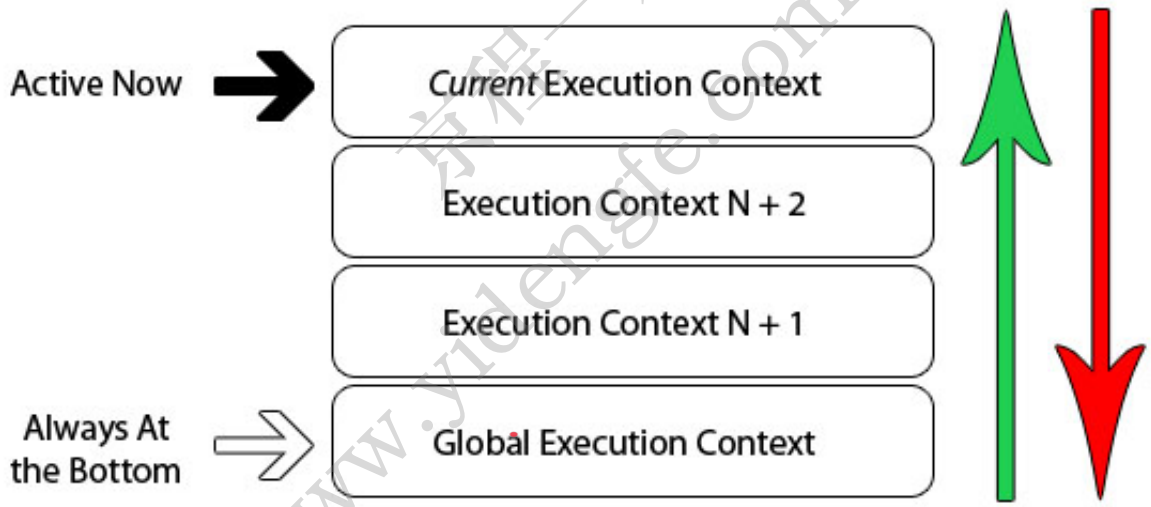
JavaScript执行堆栈详细解读

当Javascript代码执行的时候会将不同的变量存于内存中的不同位置：堆（heap）和栈（stack）中来加以区分。其中，堆里存放着一些对象。而栈中则存放着一些基础类型变量以及对象的指针。 但是我们这里说的执行栈和上面这个栈的意义却有些不同。 js 在执行可执行的脚本时，首先会创建一个全局可执行上下文globalContext，每当执行到一个函数调用时都会创建一个可执行上下文（execution context）EC。当然可执行程序可能会存在很多函数调用，那么就会创建很多EC，所以 JavaScript 引擎创建了执行上下文栈（Execution context stack，ECS）来管理执行上下文。当函数调用完成，js会退出这个执行环境并把这个执行环境销毁，回到上一个方法的执行环境... 这个过程反复进行，直到执行栈中的代码全部执行完毕，如下是以上的几个关键词，我们来一次分析一下：

> - 执行栈（Execution Context Stack）
> - 全局对象（GlobalContext）
> - 活动对象（Activation Object）
> - 变量对象（Variable Object）



[执行栈（Execution Context Stack）]

> 浏览器解释器执行 js 是单线程的过程，这就意味着同一时间，只能有一个事情在进行。其他的活动和事件只能排队等候，生成出一个等候队列执行栈（Execution Stack）。

执行栈压栈顺序

一开始执行代码的时候，变确定了一个全局执行上下文`global execution context`作为默认值。如果在你的全局环境中，调用了其他的函数，程序将会再创建一个新的 EC，然后将此 EC推入进执行栈中`execution stack`

如果函数内再调用其他函数，相同的步骤将会再次发生：创建一个新的EC -> 把EC推入执行栈。一旦一个EC执行完成，变回从执行栈中推出（pop）。

```
ECStack = [
•    globalContext
];
```

1. 继续分析压栈过程

```
function fun3() {
    console.log('fun3')
}
function fun2() {
    fun3();
}
function fun1() {
    fun2();
}
fun1();
//执行fun1 结果如下
ECStack = [
    fun1,
    globalContext
];
```

2. 变量对象（Variable Object）

变量对象VO是与执行上下文相关的特殊对象,用来存储上下文的函数声明，函数***形参***和变量。

```
//变量对象VO存储上下文中声明的以下内容
{
    //1-1 函数声明FD(如果在函数上下文中),--不包含函数表达式
    //1-2 函数形参function arguments,
    //1-3 变量声明-注意b=10不是变量，但是var b = 10;是变量，有变量声明提升
    //alert(a); // undefined
    //alert(b); // “b” 没有声明
    //b = 10;
    //var a = 20;
}
var a = 10;

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

test(30);

// 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
};
//VO分为 全局上下文的变量对象VO，函数上下文的变量对象VO
VO(globalContext) === global;
```

3. 活动对象（Activation Object）

在函数上下文中，变量对象被表示为活动对象AO,当函数被调用后，这个特殊的活动对象就被创建了。它包含普通参数与特殊参数对象（具有索引属性的参数映射表）。活动对象在函数上下文中作为变量对象使用。

```
//1.在函数执行上下文中，VO是不能直接访问的，此时由活动对象扮演VO的角色。
//2.Arguments对象它包括如下属性：callee 、length
//3.内部定义的函数
//4.以及绑定上对应的变量环境；
//5.内部定义的变量
VO(functionContext) === AO;
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
AO(test) = {
  a: 10,
  b: undefined,
  c: undefined,
  d: <reference to FunctionDeclaration "d">
  e: undefined
};
```

4. 深度活动对象（Activation Object）

```
//Activation Object 分为创建阶段和执行阶段
function foo(i) {
    var a = 'hello';
```

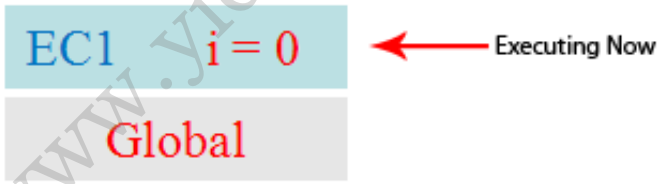
```
    var b = function privateB() {
    };
    function c() {
    }
}
foo(22);
//当我们执行foo(22)的时候，EC创建阶段会类似生成下面这样的对象：
fooExecutionContext = {
    scopeChain: { Scope },
    AO: {
        arguments: {
            0: 22,
            length: 1
        },
        i: 22,
        c: pointer to function c()
        a: undefined,
        b: undefined
    },
    VO:{..}
    Scope: [AO, globalContext.VO],
}
//在创建阶段，会发生属性名称的定义，但是并没有赋值(变量提升阶段)。一旦创建阶段（creation stage）结束，变进入了激活 / 执行阶段，那么fooExecutionContext便会完成赋值，变成这样：
//【 运行函数内部的代码，对变量复制，代码一行一行的被解释执行 】
fooExecutionContext = {
    scopeChain: { ... },
    AO: {
        arguments: {
            0: 22,
            length: 1
        },
        i: 22,
        c: pointer to function c()
        a: 'hello',
        b: pointer to function privateB()
    },
    VO:{..}
    Scope: [AO, globalContext.VO],
    this: { 运行时确认 }
}
```


5. 补充活动对象（Activation Object）

```
var x = 10;
function foo() {
  var barFn = Function('alert(x); alert(y);');
  barFn(); // 10, "y" is not defined
}
foo();
//1.通过函构造函数创建的函数的[[scope]]属性总是唯一的全局对象（LexicalEnvironment）。
//2.Eval code - eval 函数包含的代码块也有同样的效果
```

6. 整合体运行流程如下

```
//VO函数上下文的链接 AO是函数自身的
ECStack = [
        fun3
        fun2,
        fun1,
        globalContext
    ];
```

7. 写到最后

当一个异步代码（如发送ajax请求数据）执行后会如何呢？接下来需要了解的另一个概念就是：事件队列（Task Queue）。 当js引擎遇到一个异步事件后，其实不会说一直等到异步事件的返回，而是先将异步事件进行挂起。等到异步事件执行完毕后，会被加入到事件队列中。（注意，此时只是异步事件执行完成，其中的回调函数并没有去执行。）当执行队列执行完毕，主线程处于闲置状态时，会去异步队列那抽取最先被推入队列中的异步事件，放入执行栈中，执行其中的回调同步代码。如此反复，这样就形成了一个无限的循环。这就是这个过程被称为“事件循环（Event Loop）”的原因。

7. 一个小实战

```
//=====分割线======
```

```
function test() {
  var result = []
  for (var i = 0; i < 10; i++) {
    result[i] = function() {
      return i
    }
  }
  return result
}

let r = test()
r.forEach(fn => {
  console.log('fn',fn())
})
//1.函数test执行完出栈 留下AO(test)有个i的指向
//2.函数在执行的时候，函数的执行环境才会生成。所以fn执行的时候生成作用域链条指向如下
//3.AO(result[i]) --> AO(kun) --> VO(G) 加个闭包就立马创建了执行环境
<ul>
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>4</li>
    <li>5</li>
    <li>6</li>
</ul>
var list_li = document.getElementsByTagName("li");
for (var i = 0; i < list_li.length; i++) {
    list_li[i].onclick = function () {
        console.log(i);
    }
}
```

那么其实一切也就迎刃而解了。闭包的原理是Scope，this的原理是动态绑定，作用域链的原理是Scope: [AO, globalContext.VO],eval不能回收的原理是推不进AO,变量提升的原理是AO的准备阶段，异步队列的原理是ECS.

作者 [@志佳老师] 2018 年 12月 21日