Exection Context

1. 定义

当控制器达到ECMAScript可执行代码的时候，控制器就进入了一个执行上下文(EC)

2. ECStack

一系列活动的EC从逻辑上形成一个栈ECStack，栈底总是全局EC，栈顶是当前的EC。在不同的EC间切换时，通过压栈或者退栈的形式修改。

3. 可执行代码的类型

- 全局代码
  这类代码是在"程序"级别上被处理的，比如加载一个js文件或者内联的代码<script>标签。程序初始化时，

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

- 函数代码
  当控制器进入函数代码时，就有新的EC被push到ECStack。当函数返回时，ECStack会pop当前的EC。当所有代码执完后，就只剩[globalContext]，直到程序结束

- **Eval**代码
  调用eval函数时的上下文，就是调用上下文[callingContext]

4. EC包含了三个属性

（使用ECMAScript中的对象来表示）：

```
activeExecutionContext = {
    VO: {...}, // 或者 AO
    this: thisValue,
    Scope: [ // 作用域链，所有变量对象的列表，用于标识符查询
    ]
};
```

Variable Object

每个EC都有与之相对应的变量对象(VO)，EC中定义的所有变量和函数都保存在这个对象中。VO实现了数据存储和获取。

1. 定义

> A variable object (in abbreviated form — VO) is a special object related with an execution context and which stores:
>
> - variables (var, VariableDeclaration);
> - function declarations (FunctionDeclaration);
> - function formal parameters declared in the context.

```
activeExecutionContext = {
    VO: {
        变量声明(var),
        函数声明(FD),
        函数形参(function arguments)
    }
};
```

声明新的变量和函数实际上就是在VO中增加变量/函数名对应的属性和属性值的过程，例如：

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

//全局上下文中的变量对象
VO(globalContext)={
    a: 10,
    test: function
};
//test函数上下文中的变量对象
VO(test functionContext)={
    x: 30,
    b: 20
}
```

2. 全局上下文中的变量对象

- 全局对象

  > 全局对象是一个在进入任何执行上下文前就创建出来的对象；此对象以单例形式存在；它的属性在程序任何地方都可以直接访问，其生命周期随着程序的结束而终止。

  全局对象在创建的时候，诸如Math,String,Date,parseInt等等属性也会被初始化，同时，其中一些对象会指向全局对象本身——比如，DOM中，全局对象上的`window`属性就指向了全局对象本身

- 全局上下文的变量对象就是全局对象本身

  ```
  VO(globalContext) === global
  ```

当在全局上下文中声明一个变量时，可以通过全局对象上的属性来间地引用该变量。而在其他的上下文中是无法直接引用VO的。

```
var a = new String('test');

alert(a); // 直接访问，在VO(globalContext)找到

alert(window['a']); // 通过global === VO(globalContext)间接访问
alert(a === this.a); // true
```

3. 函数上下文中的变量对象

- 在函数上下文中，VO是不能直接访问的，扮演**活跃对象**(Active Object)的角色。
  `VO(functionContext) === AO`
- AO在进入函数上下文时会被创建，初始化只有`arguments`属性，其值就是`Arguments`对象

处理上下文代码的两个阶段

不管是全局上下文还是函数上下文，都会经过这两个阶段

1. 进入执行上下文

进入执行上下文，即预编译过程，VO会按照顺序被一些属性填充：

- 函数的形参（仅在进入函数执行上下文）

  其属性名就是形参的名字，其值就是实参的值；对于没有传递的参数，其值为`undefined`

- 函数声明

  如果变量对象已经包含了相同名字的属性，则替换它的值

- 变量声明

  其属性名即为变量名，其值为`undefined`。如果和已经声明的函数名或者函数的参数名相同，*不会影响已存在的属性*

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

test(10);

//以10为参数进入test函数的上下文时
```

```
AO(test)= {
  a:10, //直接赋为实参的值
  b:undefined,
  c:undefined,
  d:指向函数声明d,
  e:undefined
}
```

x属于函数表达式，不会对VO造成影响。而_e虽然也是函数表达式，但由于赋值给变量，可以通过变量e访问到。

2. 执行代码

到了执行代码阶段，AO/VO就会修改为：

```
AO(test) ={
    a: 10,
    b: undefined,
    c: 10,
    d: 指向函数声明d,
    e: 指向函数声明_e
}
```

3. 变量提升hoisting

例子

```
alert(x); // function

var x = 10;
alert(x); // 10

x = 20;

function x() {};

alert(x); // 20
```

在进入上下文的时候，按照顺序，函数声明先填充VO。后虽然有变量声明x，但不会对有相同名字的函数声明和函数形参发生冲突，因此刚进入上下文时，填充为：

```
VO['x'] = 指向函数声明x
```

虽然有语句var x = 10，如果函数"x"还未定义，则 "x" 为undefined, 但是量声明并不会影响同名的函数值

而在执行代码阶段，VO被修改为

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

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

作用域

每一段代码都有一个与之关联的作用域链，这个作用域链是一个变量对象的**列表**，即EC中的`scope`属性。它的用途是保证对执行环境有权访问的所有变量和函数的有序访问。

1. 作用域链

> 作用域链是一条变量对象的链，它和执行上下文有关，用于在处理标识符时候进行变量查询。

大致表示如下：

```
activeExecutionContext = {
    VO: {...}, // 或者 AO
    this: thisValue,
    Scope: [ // 作用域链，所有变量对象的列表，用于标识符查询
    ]
};
```

Scope作用域链的抽象定义是一个**对象列表**，用数组来表示：

```
Scope = [VO1, VO2, ..., VOn]
```

2. [[scope]]

> *[[Scope]]* is a hierarchical chain of all parent variable objects, which are above the current function context; the chain is saved to the function at its ***creation***.

`[[scope]]`包含了函数创建时作用域链中的所有变量对象，在函数创建时保存在函数中并且不会变，是函数的内部属性。即使函数一直没有调用，[[scope]]属性也一直存在，直到函数销毁。

3. 作用域

> JavaScript中的函数运行在它们被定义的作用域里，而不是它们被执行的作用域里 在JS中，作用域的概念和其他语言差不多，在每次调用一个函数的时候，就会进入一个函数内的作用域，当从函数返回以后，就返回调用前的作用域。 JS作用域的实现采用列表的方式，而非堆栈，具体过程为：

- 一个函数被定义的时候, 会将它定义时刻的作用域链链接到这个函数对象的[[scope]]属性。

- 在一个函数被调用的时候，函数EC的作用域链会被初始化为该函数的[[scope]]属性，然后将AO链到作用域链的顶端，即EC的Scope属性会被填充为：

  ```
  Scope = AO + [[scope]]
  ```

  AO是Scope的第一个元素，添加在作用域链的最前端。

标识符处理过程包括了对应的变量名的属性查询，比如：在作用域链中会进行一系列的变量对象的检测，从作用域链的AO一直到最上层上下文。如果两个相同名字的变量存在于不同的上下文中时，处于底层上下文的变量会优先被找到。 例子：

```
var x = 10;
function foo(y) {
  function bar(z) {
    var z = 30;
    alert(x +  y + z);
  }
  bar();
}
foo(20); // 60
foo(10); // 50
```

全局上下文的变量对象：

```
globalContext.VO = {
    x: 10,
    foo: 指向函数声明foo
}
ECStack = [globalContext];
```

foo函数创建时，其[[scope]]属性为：

```
foo.[[scope]] = [globalContext.VO]
```

foo函数以参数20调用时（进入上下文），foo函数的AO为：

```
fooContext(20).AO = {
    y: 20,
    bar: 指向函数声明bar
}
fooContext(20).Scope = fooContext(20).AO + foo.[[scope]]
                  = [fooContext(20).AO, globalContext.VO]
ECStack = [fooContext(20), globalContext]
```

内部bar函数创建时，

```
bar.[[scope]] = [fooContext(20).AO, globalContext.VO]
```

bar函数激活时，

```
barContext.AO = {
    z: 30
}
barContext.Scope = barContext.AO + bar.[[scope]]
                 = [barContext.AO, fooContext(20).AO, globalContext.VO]
ECStack = [barContext, fooContext(20), globalContext]
```

x,y,z的表示符查询过程：

```
- "x"
-- barContext.AO // not found
-- fooContext(20).AO // not found
-- globalContext.VO // found - 10

- "y"
-- barContext.AO // not found
-- fooContext(20).AO // found - 20

- "z"
-- barContext.AO // found - 30
```

bar函数返回

```
ECStack = [fooContext(20), globalContext]
```

foo函数返回，

```
ECStack = [globalContext]
```

当以参数10再次调用foo时，会创建新的`fooContext(10)`并压栈，重复上述过程。

4. 闭包的简单介绍

[[Scope]]是在函数创建的时候就保存在函数对象上了，并且直到函数销毁的时候才消失。 闭包就是函数代码和其[[Scope]]属性的组合。[[Scope]]包含了函数创建所在的**词法环境**（上层变量对象）。上层上下文中的变量，

可以在函数激活的时候，通过变量对象的词法链（函数创建的时候就保存起来了）查询到。

```
function foo() {
  var x = 10;
  var y = 20;
  return function () {
    alert([x, y]);
  };
}
var x = 30;
var bar = foo(); // anonymous function is returned
bar(); // [10, 20]
```

词法作用域链是在函数创建的时候定义的——变量“x”的值是10，而不是30。并且，foo函数返回的匿名函数的[[scope]]属性，即使在foo函数的上下文结束后，依然存在。

栗子：

```
function factory() {
     var name = 'laruence';
     var intro = function(){
          alert('I am ' + name);
     }
     return intro;
}
function app(para){
     var name = para;
     var func = factory();
     func();
}
app('eve');
```

进入全局上下文：

```
globalContext.VO = {
    factory: 指向函数声明factory,
    app: 指向函数声明app
}
ECStack = [globalContext]
```

factory函数和app函数创建时，

```
factory.[[scope]] = [globalContext.VO]
app.[[scope]] = [globalContext.VO]
```

执行到app('eve')时，进入app函数上下文

```
appContext.AO = {
    para: 'eve',
    name: undefined,
    func: undefined,
}
appContext.Scope = [appContext.AO, globalContext.VO]
ECStack = [appContext, globalContext]
```

执行app函数内代码：

```
appContext.AO = {
    para: 'eve',
    name: 'eve',
    func:
}
```

当factory函数被调用时，进入factory函数上下文

```
factoryContext.AO = {
    name: undefined,
    intro: undefined
}
factoryContext.Scope = [factoryContext.AO, globalContext.VO]
ECStack = [factoryContext, appContext, globalContext]
```

执行factory函数代码后，

```
factoryContext.AO = {
    name: 'laruence',
    intro: 指向匿名函数
}
```

其中intro指向了一个匿名函数，该函数创建时的[[scope]]为

```
anonymousContext.[[scope]] = [factoryContext.AO, globalContext.VO] ??
```

factory函数返回intro所指向的匿名函数，回到app函数赋值给func。但由于返回的匿名函数有变量引用了factory函数中的变量，因此factory函数的上下文不能被销毁

```
appContext.AO = {
    para: 'eve',
    name: 'eve',
    func: 指向匿名函数
}
ECStack = [factoryContext, appContext, globalContext]
```

当执行func()时，又进入到匿名函数的上下文

```
anonymousContext.AO = {}
anonymousContext.Scope = [anonymousContext.AO, factoryContext.VO,
globalContext.VO]
ECStack = [anonymousContext.AO, factoryContext, appContext, globalContext]
```

匿名函数查找变量name，在factoryContext.VO中找到，因此输出'laruence'。 执行结束后，销毁

```
ECStack = [factoryContext, appContext, globalContext]
```

app函数执行结束后

```
ECStack = [globalContext]
```

```

```