上一篇文章中介绍了Execution Context中的三个重要部分：VO/AO，scope chain和this，并详细的介绍了VO/AO在JavaScript代码执行中的表现。

本文就看看Execution Context中的scope chain。

作用域

开始介绍作用域链之前，先看看JavaScript中的作用域（scope）。在很多语言中（C++，C#，Java），作用域都是通过代码块（由{}包起来的代码）来决定的，但是，在**JavaScript**作用域是跟函数相关的，也可以说成是**function-based**。

例如，当for循环这个代码块结束后，依然可以访问变量"i"。

```
for(var i = 0; i < 3; i++){
    console.log(i);
}

console.log(i); //3
```

对于作用域，又可以分为全局作用域（Global scope）和局部作用域（Local scpoe）。

全局作用域中的对象可以在代码的任何地方访问，一般来说，下面情况的对象会在全局作用域中：

- 最外层函数和在最外层函数外面定义的变量
- 没有通过关键字"var"声明的变量
- 浏览器中，window对象的属性

局部作用域又被称为函数作用域（Function scope），所有的变量和函数只能在作用域内部使用。

```
var foo = 1;
window.bar = 2;

function baz(){
    a = 3;
    var b = 4;
}
// Global scope: foo, bar, baz, a
// Local scope: b
```

作用域链

通过前面一篇文章了解到，每一个Execution Context中都有一个VO，用来存放变量，函数和参数等信息。

在JavaScript代码运行中，所有用到的变量都需要去当前AO/VO中查找，当找不到的时候，就会继续查找上层Execution Context中的AO/VO。这样一级级向上查找的过程，就是所有Execution Context中的AO/VO组成了一个作用域链。

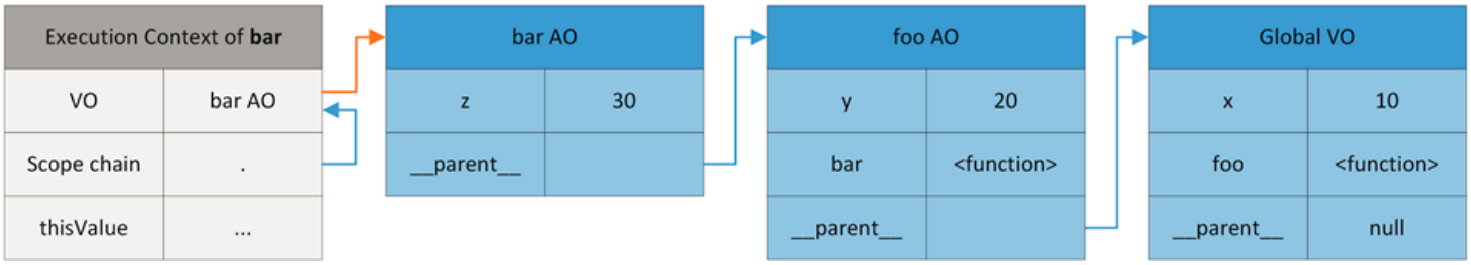
所以说，作用域链与一个执行上下文相关，是内部上下文所有变量对象（包括父变量对象）的列表，用于变量查询。

```
Scope = VO/AO + All Parent VO/AOs
```

看一个例子：

```
var x = 10;

function foo() {
    var y = 20;

    function bar() {
        var z = 30;

        console.log(x + y + z);
    };

    bar()
};

foo();
```

上面代码的输出结果为"60"，函数bar可以直接访问"z"，然后通过作用域链访问上层的"x"和"y"。



- 绿色箭头指向VO/AO
- 蓝色箭头指向scope chain（VO/AO + All Parent VO/AOs）

再看一个比较典型的例子：

```
var data = [];
for(var i = 0 ; i < 3; i++){
    data[i]=function() {
        console.log(i);
    }
}

data[0]();// 3
data[1]();// 3
data[2]();// 3
```

第一感觉（错觉）这段代码会输出"0，1，2"。但是根据前面的介绍，变量"i"是存放在"Global VO"中的变量，循环结束后"i"的值就被设置为3，所以代码最后的三次函数调用访问的是相同的"Global VO"中已经被更新的"i"。

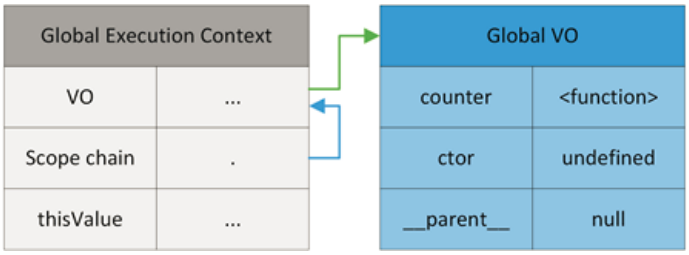
结合作用域链看闭包

在JavaScript中，闭包跟作用域链有紧密的关系。相信大家对下面的闭包例子一定非常熟悉，代码中通过闭包实现了一个简单的计数器。

```
function counter() {
    var x = 0;

    return {
        increase: function increase() { return ++x; },
        decrease: function decrease() { return --x; }
    };
}

var ctor = counter();

console.log(ctor.increase());
console.log(ctor.decrease());
```

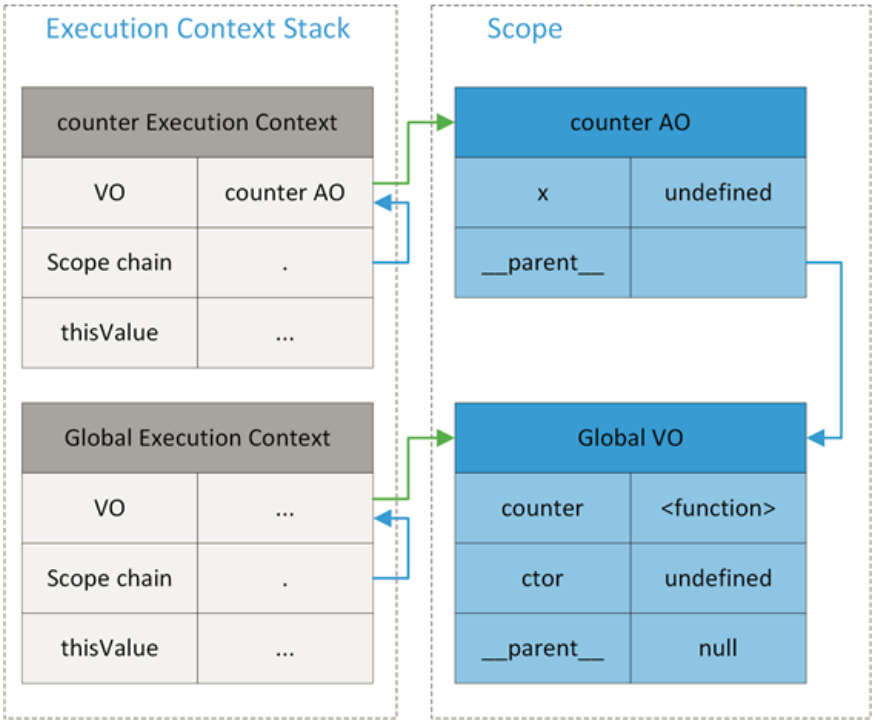
下面我们就通过Execution Context和scope chain来看看在上面闭包代码执行中到底做了哪些事情。

1. 当代码进入Global Context后，会创建Global VO

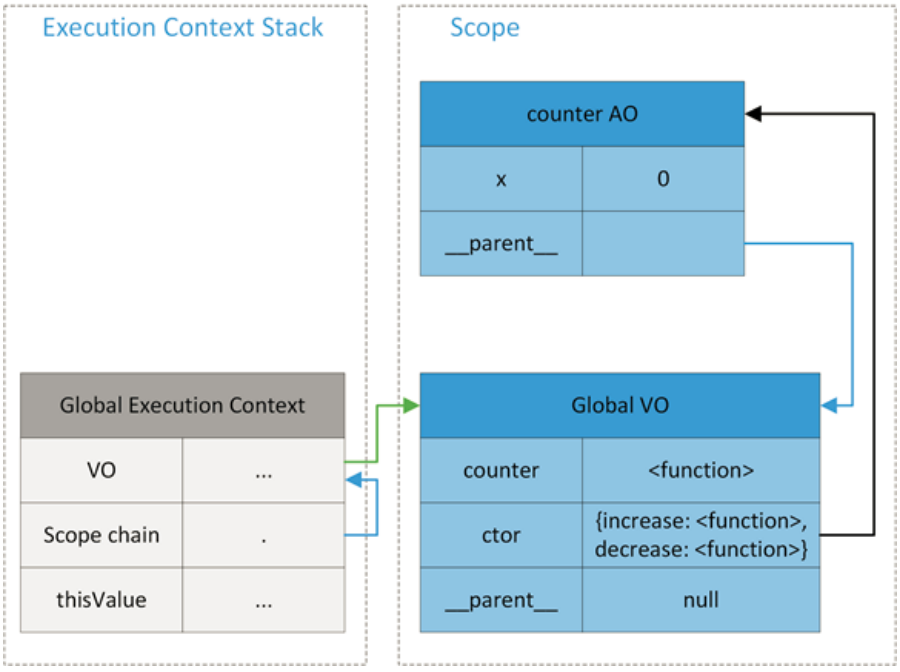


- 绿色箭头指向VO/AO
- 蓝色箭头指向scope chain（VO/AO + All Parent VO/AOs）

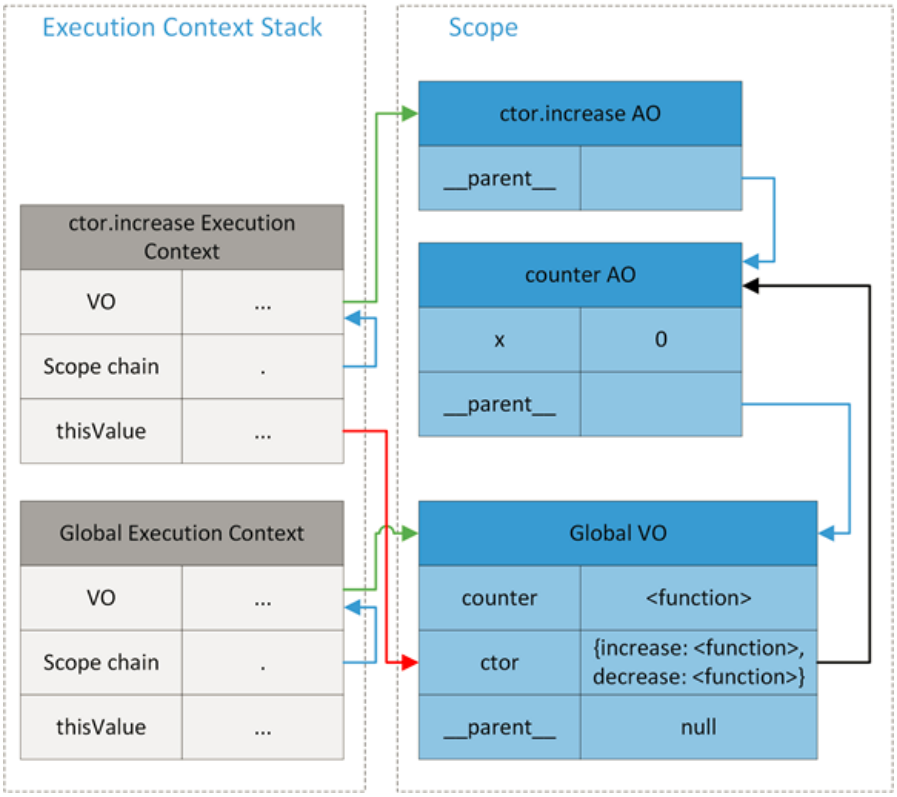
2. 当代码执行到"var cter = counter();"语句的时候，进入counter Execution Context；根据上一篇文章的介绍，这里会创建counter AO，并设置counter Execution Context的scope chain



3. 当counter函数执行的最后，并退出的时候，Global VO中的ctor就会被设置；这里需要注意的是，虽然counter Execution Context退出了执行上下文栈，但是因为ctor中的成员仍然引用counter AO（因为counter AO是increase和decrease函数的parent scope），所以counter AO依然在Scope中。



4. 当执行"ctor.increase()"代码的时候，代码将进入ctor.increase Execution Context，并为该执行上下文创建VO/AO，scope chain和设置this；这时，ctor.increase AO将指向counter AO。



- 绿色箭头指向VO/AO
- 蓝色箭头指向scope chain（VO/AO + All Parent VO/AOs）
- 红色箭头指向this
- 黑色箭头指向parent VO/AO

相信看到这些，一定会对JavaScript闭包有了比较清晰的认识，也了解为什么counter Execution Context退出了执行上下文栈，但是counter AO没有销毁，可以继续访问。

二维作用域链查找

通过上面了解到，作用域链（scope chain）的主要作用就是用来进行变量查找。但是，在JavaScript中还有原型链（prototype chain）的概念。

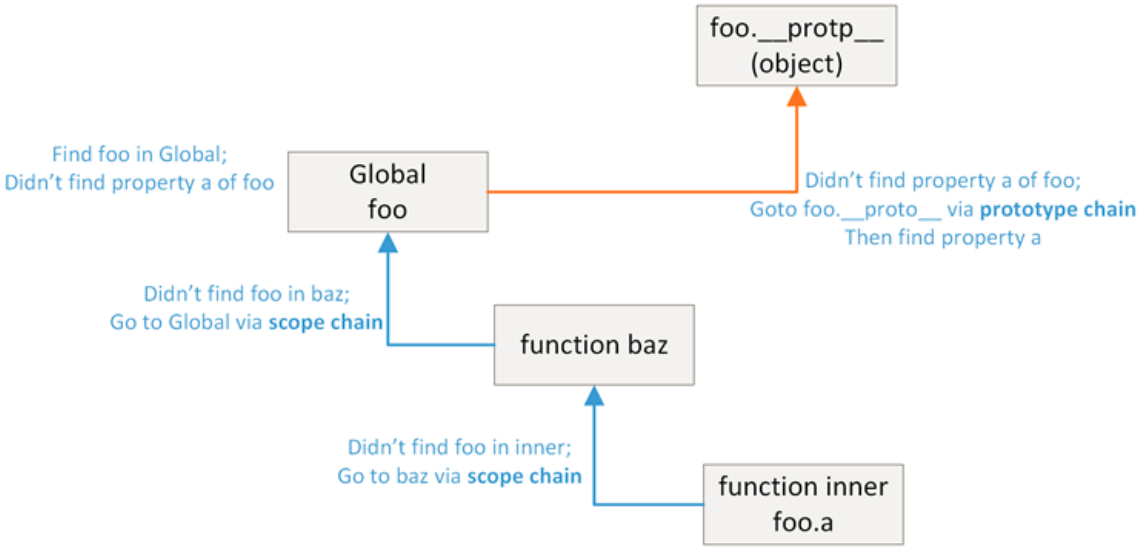
由于作用域链和原型链的相互作用，这样就形成了一个二维的查找。

对于这个二维查找可以总结为：当代码需要查找一个属性（**property**）或者描述符（**identifier**）的时候，首先会通过作用域链（**scope chain**）来查找相关的对象；一旦对象被找到，就会根据对象的原型链（**prototype chain**）来查找属性（**property**）。

下面通过一个例子来看看这个二维查找：

```
var foo = {}

function baz() {

    Object.prototype.a = 'Set foo.a from prototype';

    return function inner() {
        console.log(foo.a);
    }

}

baz()();
// Set bar.a from prototype
```

对于这个例子，可以通过下图进行解释，代码首先通过作用域链（scope chain）查找"foo"，最终在Global context中找到；然后因为"foo"中没有找到属性"a"，将继续沿着原型链（prototype chain）查找属性"a"。



- 蓝色箭头表示作用域链查找
- 橘色箭头表示原型链查找

总结

本文介绍了JavaScript中的作用域以及作用域链，通过作用域链分析了闭包的执行过程，进一步认识了JavaScript的闭包。

同时，结合原型链，演示了JavaScript中的描述符和属性的查找。

下一篇我们就看看Execution Context中的this属性。