# Chapter 4. Scope chain

## Introduction
## 导言

执行上下文的数据(变量，方法定义，方法形参)是存放在**变量对象**的属性中的.

同时，我们也了解了**变量对象**是在**进入执行上下文时被初始化**在**代码执行阶段更新**的。

## Definition
## 定义

```
var x = 10;

function foo() {

  var y = 20;

  function bar() {
    alert(x + y);
  }

  return bar;

}

foo()(); // 30
```

每个**执行上下文**都有他自己的**变量对象**：对于全局执行上下文来说他就是**全局对象本身**，对方法来说就是**AO**。

那么作用域链就是内部上下文中一张含有所有父级变量对象的列表。这个链用于变量搜索。即上例中，bar上下文中的作用域链包含**AO(bar)**,**AO(foo)**和**VO(global)**

> 作用域链式是与执行上下文密切相关的变量对象链，它用于标示符解析时的变量查询。

方法上下文的**作用域链式**在**方法被调用时创建**，它由**活动对象**和**方法对象的内部属性[[Scope]]**组成。

```
activeExecutionContext = {
    VO: {...}, // or AO
    this: thisValue,
    Scope: [ // Scope chain
      // list of all variable objects
      // for identifiers lookup
    ]
};
```

作用域链定义:

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

## Function life cycle
## 方法生命周期

函数生命周期分为创建、调用两个阶段。

### Function creation
### 方法创建

**[[Scope]]**是含有当前方法上下文之上**所有父级上下文的变量对象的继承链**，这个链是**方法创建时添加**的。

最重要的一点 -- **[[Scope]]是函数创建时添加的** -- 它是恒定不变的直到方法销毁。也就是说，方法可以不被调用，但是[[Scope]]属性已经存在于方法对象中。

另外需要提及的是，和作用域链的区别在于：**[[Scope]]是函数**而**不是上下文的一个属性**([[Scope]]是方法的属性，正如作用域链是上下文的一个属性一样)。考虑下例，foo方法的[[Scope]]如下

```
foo.[[Scope]] = [
  globalContext.VO // === Global
];
```

当方法调用时，在进入方法上下文时，活动对象和this值被创建，这时作用域链也被创建。接着往下看。

### Function activation
### 方法调用

```
Scope = AO|VO + [[Scope]]
```

需要注意的是AO是Scope数组的第一个元素.他被添加在作用域链顶部。

```
Scope = [AO].concat([[Scope]]);
```

这点对于标示符解析非常重要：**标示符解析**是计算变量(或方法申明)是属于作用域中哪一个AO的过程。

我们来看一个稍加复杂的例子

```
var x = 10;

function foo() {

  var y = 20;

  function bar() {
    var z = 30;
    alert(x +  y + z);
  }

  bar();
}

foo(); // 60
```

bar上下文的作用域链(Scope属性)如下

```
barContext.Scope = barContext.AO + bar.[[Scope]] // i.e.:

barContext.Scope = [
  barContext.AO,
  fooContext.AO,
  globalContext.VO
];
```

## Scope features
## Scope的特性

让我们深入了解下函数的**[[Scope]]属性**和上下文**作用域链**的一些特性

### Closures
### 闭包

闭包是**函数代码**和**函数的[[Scope]]属性**的集合。

**函数的[[Scope]]属性**包含了函数创建时的**词法环境**(父级变量对象)，存放**所有父级上下文的变量对象**，用于之后**函数调用时**需要查找的变量对象的**词法链**;

由于JavaScript采用静态作用域链的方式。闭包是为了解决静态作用域这种方式下导致的函数内部访问外部变量引起的问题。

例:

```
var x = 10;

function foo() {
  alert(x);
}

(function () {
  var x = 20;
  foo(); // 10, but not 20
})();
```

我们发现，变量x在函数foo的**[[Scope]]属性中被找到**，即查找 **函数在创建阶段构造好的词法链**，而不是调用时产生的动态链(这是变量x值应该为20);

另一经典例子：

```
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]
```

我们再次确认在标示符解析时使用的是**函数创建时的词法作用域链** - 变量x解析为10而不是30.而且，本例清楚地说明打**函数的[[Scope]]属性在创建方法的上下文销毁后任然存在**(本例中匿名方法是foo方法返回的)。

## [[Scope]] of functions created via Function constructor 《Function构造函数》创建的方法的[[Scope]]属性

上例中我证实，[[Scope]]属性是在方法创建时创建，并且通过这个属性访问所有父级上下文的变量。接下来我们将了解这点的另一重要特性，这是由Function构造函数创建的方法引起的。

```
var x = 10;

function foo() {

  var y = 20;

  function barFD() { // FunctionDeclaration
    alert(x);
    alert(y);
  }

  var barFE = function () { // FunctionExpression
    alert(x);
    alert(y);
  };

  var barFn = Function('alert(x); alert(y);');

  barFD(); // 10, 20
  barFE(); // 10, 20
  barFn(); // 10, "y" is not defined

}

foo();
```

如例，通过Function构造函数创建的方法barFn不能访问变量y。这并不意味barFn方法没有[[Scope]]内部属性(否则它将不能访问x变量).而在于**通过Function构造函数创建的函数的[[Scope]]属性仅包含全局对象**。也就是说，通过这中方式创建的方法是不可能创建父级上下文的闭包环境，除全局情况外。

**Two-dimensional Scope chain lookup**
**二维作用域链查找**

作用域链查询的另一关键点在与**变量对象**(仅是VO)**的原型**(如果他们有)**也会被查询** -- **因为ECMAScript原型的特性：**如果当前对象中没有找到相关属性，那么将查询他的原型链。即二维链式查找查找:(1)作用域链,(2) 每个作用域链节点 -- 深入查找原型链.我们可以通过修改Object.prototype属性来观察他的影响。

```
function foo() {
  alert(x);
}

Object.prototype.x = 10;

foo(); // 10
```

**Scope chain of the global and eval contexts**
**全局和eval上下文的作用域链**

eval上下文和调用上下文拥有同样的作用域链

**Affecting on Scope chain during code execution**
**代码执行阶段影响作用域链**

在ECMAScipt中，有两种语法在代码执行阶段可以修改作用域链。**with**和**catch**语句。他们都将语句中的对象添加到作用域链**第一个**，用于标示符查询。

当catch语句结束后，作用域链同样恢复之前装填

## Conclusion
## 总结