[作用域链(Scope Chain)](http://www.cnblogs.com/TomXu/archive/2012/01/18/2312463.html)

**前言**

我们已经知道一个执行上下文 的数据（变量、函数声明和函数的形参）作为属性存储在变量对象中。

同时我们也知道变量对象在每次进入上下文时创建，并填入初始值，值的更新出现在代码执行阶段。

英文原文：http://dmitrysoshnikov.com/ecmascript/chapter-4-scope-chain/  
中文参考：http://www.denisdeng.com/?p=908

**定义**

如果要简要的描述并展示其重点，那么作用域链大多数与内部函数相关。

我们知道，ECMAScript 允许创建内部函数，我们甚至能从父函数中返回这些函数。

var x = 10;  
   
function foo() {   
 var y = 20;   
 function bar() {  
 alert(x + y);  
 }   
 return bar;   
}  
   
foo()(); // 30

这样，很明显每个上下文拥有自己的变量对象：对于全局上下文，它是全局对象自身；对于函数，它是活动对象。

作用域链正是内部上下文所有变量对象（包括父变量对象）的列表。此链用来变量查询。即在上面的例子中，“bar”上下文的作用域链包括AO(bar)、AO(foo)和VO(global)。

但是，让我们仔细研究这个问题。

让我们从定义开始，并进深一步的讨论示例。

作用域链与一个执行上下文相关，变量对象的链用于在标识符解析中变量查找。

函数上下文的作用域链在函数调用时创建的，包含活动对象和这个函数内部的[[scope]]属性。下面我们将更详细的讨论一个函数的[[scope]]属性。

在上下文中示意如下：

activeExecutionContext = {  
 VO: {...}, // or AO  
 this: thisValue,  
 Scope: [ // Scope chain  
 // 所有变量对象的列表  
 // for identifiers lookup  
 ]  
};

其scope定义如下：

Scope = AO + [[Scope]]

这种联合和标识符解析过程，我们将在下面讨论，这与函数的生命周期相关。

**函数的生命周期**

函数的的生命周期分为创建和激活阶段（调用时），让我们详细研究它。

**函数创建**

众所周知，在进入上下文时函数声明放到变量/活动（VO/AO）对象中。让我们看看在全局上下文中的变量和函数声明（这里变量对象是全局对象自身，我们还记得，是吧？）

var x = 10;  
   
function foo() {  
 var y = 20;  
 alert(x + y);  
}  
   
foo(); // 30

在函数激活时，我们得到正确的（预期的）结果－－30。但是，有一个很重要的特点。  
  
此前，我们仅仅谈到有关当前上下文的变量对象。这里，我们看到变量“y”在函数“foo”中定义（意味着它在foo上下文的AO中），但是变量“x”并未在“foo”上下文中定义，相应地，它也不会添加到“foo”的AO中。乍一看，变量“x”相对于函数“foo”根本就不存在；但正如我们在下面看到的——也仅仅是“一瞥”，我们发现，“foo”上下文的活动对象中仅包含一个属性－－“y”。

fooContext.AO = {  
 y: undefined // undefined – 进入上下文的时候是20 – at activation  
};

函数“foo”如何访问到变量“x”？理论上函数应该能访问一个更高一层上下文的变量对象。实际上它正是这样，这种机制是通过函数内部的[[scope]]属性来实现的。

[[scope]]是所有父变量对象的层级链，处于当前函数上下文之上，在函数创建时存于其中。

注意这重要的一点－－[[scope]]在函数创建时被存储－－静态（不变的），永远永远，直至函数销毁。即：函数可以永不调用，但[[scope]]属性已经写入，并存储在函数对象中。

另外一个需要考虑的是－－与作用域链对比，[[scope]]是函数的一个属性而不是上下文。考虑到上面的例子，函数“foo”的[[scope]]如下：

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

举例来说，我们用通常的ECMAScript 数组展现作用域和[[scope]]。

继续，我们知道在函数调用时进入上下文，这时候活动对象被创建，this和作用域（作用域链）被确定。让我们详细考虑这一时刻。

**函数激活**

正如在定义中说到的，进入上下文创建AO/VO之后，上下文的Scope属性（变量查找的一个作用域链）作如下定义：

Scope = AO|VO + [[Scope]]

上面代码的意思是：活动对象是作用域数组的第一个对象，即添加到作用域的前端。

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

这个特点对于标示符解析的处理来说很重要。

标示符解析是一个处理过程，用来确定一个变量（或函数声明）属于哪个变量对象。

这个算法的返回值中，我们总有一个引用类型，它的base组件是相应的变量对象（或若未找到则为null）,属性名组件是向上查找的标示符的名称。引用类型的详细信息在第13章.this中已讨论。

标识符解析过程包含与变量名对应属性的查找，即作用域中变量对象的连续查找，从最深的上下文开始，绕过作用域链直到最上层。

这样一来，在向上查找中，一个上下文中的局部变量较之于父作用域的变量拥有较高的优先级。万一两个变量有相同的名称但来自不同的作用域，那么第一个被发现的是在最深作用域中。

我们用一个稍微复杂的例子描述上面讲到的这些。

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

对此，我们有如下的变量/活动对象，函数的的[[scope]]属性以及上下文的作用域链：

全局上下文的变量对象是：

globalContext.VO === Global = {  
 x: 10  
 foo: <reference to function>  
};

在“foo”创建时，“foo”的[[scope]]属性是：

foo.[[Scope]] = [  
 globalContext.VO  
];

在“foo”激活时（进入上下文），“foo”上下文的活动对象是：

fooContext.AO = {  
 y: 20,  
 bar: <reference to function>  
};

“foo”上下文的作用域链为：

fooContext.Scope = fooContext.AO + foo.[[Scope]] // i.e.:  
   
fooContext.Scope = [  
 fooContext.AO,  
 globalContext.VO  
];

内部函数“bar”创建时，其[[scope]]为：

bar.[[Scope]] = [  
 fooContext.AO,  
 globalContext.VO  
];

在“bar”激活时，“bar”上下文的活动对象为：

barContext.AO = {  
 z: 30  
};

“bar”上下文的作用域链为：

barContext.Scope = barContext.AO + bar.[[Scope]] // i.e.:  
   
barContext.Scope = [  
 barContext.AO,  
 fooContext.AO,  
 globalContext.VO  
];

对“x”、“y”、“z”的标识符解析如下：

- "x"  
-- barContext.AO // not found  
-- fooContext.AO // not found  
-- globalContext.VO // found - 10  
  
- "y"  
-- barContext.AO // not found  
-- fooContext.AO // found - 20  
  
- "z"  
-- barContext.AO // found - 30

**作用域特征**

让我们看看与作用域链和函数[[scope]]属性相关的一些重要特征。

**闭包**

在ECMAScript中，闭包与函数的[[scope]]直接相关，正如我们提到的那样，[[scope]]在函数创建时被存储，与函数共存亡。实际上，闭包是函数代码和其[[scope]]的结合。因此，作为其对象之一，[[Scope]]包括在函数内创建的词法作用域（父变量对象）。当函数进一步激活时，在变量对象的这个词法链（静态的存储于创建时）中，来自较高作用域的变量将被搜寻。

例如：

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

我们再次看到，在标识符解析过程中，使用函数创建时定义的词法作用域－－变量解析为10，而不是30。此外，这个例子也清晰的表明，一个函数（这个例子中为从函数“foo”返回的匿名函数）的[[scope]]持续存在，即使是在函数创建的作用域已经完成之后。

关于ECMAScript中闭包的理论和其执行机制的更多细节，阅读16章闭包。

**通过构造函数创建的函数的[[scope]]**

在上面的例子中，我们看到，在函数创建时获得函数的[[scope]]属性，通过该属性访问到所有父上下文的变量。但是，这个规则有一个重要的例外，它涉及到通过函数构造函数创建的函数。

var x = 10;  
   
function foo() {  
   
 var y = 20;  
   
 function barFD() { // 函数声明  
 alert(x);  
 alert(y);  
 }  
   
 var barFE = function () { // 函数表达式  
 alert(x);  
 alert(y);  
 };  
   
 var barFn = Function('alert(x); alert(y);');  
   
 barFD(); // 10, 20  
 barFE(); // 10, 20  
 barFn(); // 10, "y" is not defined  
   
}  
   
foo();

我们看到，通过函数构造函数（Function constructor）创建的函数“bar”，是不能访问变量“y”的。但这并不意味着函数“barFn”没有[[scope]]属性（否则它不能访问到变量“x”）。问题在于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。考虑到这一点，如通过这种函数创建除全局之外的最上层的上下文闭包是不可能的。

**二维作用域链查找**

在作用域链中查找最重要的一点是变量对象的属性（如果有的话）须考虑其中－－源于ECMAScript 的原型特性。如果一个属性在对象中没有直接找到，查询将在原型链中继续。即常说的二维链查找。（1）作用域链环节；（2）每个作用域链－－深入到原型链环节。如果在Object.prototype 中定义了属性，我们能看到这种效果。

function foo() {  
 alert(x);  
}  
   
Object.prototype.x = 10;  
   
foo(); // 10

活动对象没有原型，我们可以在下面的例子中看到：

function foo() {  
   
 var x = 20;  
   
 function bar() {  
 alert(x);  
 }  
   
 bar();  
}  
   
Object.prototype.x = 10;  
   
foo(); // 20

如果函数“bar”上下文的激活对象有一个原型，那么“x”将在Object.prototype 中被解析，因为它在AO中不被直接解析。但在上面的第一个例子中，在标识符解析中，我们到达全局对象（在一些执行中并不全是这样），它从Object.prototype继承而来，响应地，“x”解析为10。

同样的情况出现在一些版本的SpiderMokey 的命名函数表达式（缩写为NFE）中，在那里特定的对象存储从Object.prototype继承而来的函数表达式的可选名称，在Blackberry中的一些版本中，执行时激活对象从Object.prototype继承。但是，关于该特色的更多细节在第15章函数讨论。

**全局和eval上下文中的作用域链**

这里不一定很有趣，但必须要提示一下。全局上下文的作用域链仅包含全局对象。代码eval的上下文与当前的调用上下文（calling context）拥有同样的作用域链。

globalContext.Scope = [  
 Global  
];  
   
evalContext.Scope === callingContext.Scope;

**代码执行时对作用域链的影响**

在ECMAScript 中，在代码执行阶段有两个声明能修改作用域链。这就是with声明和catch语句。它们添加到作用域链的最前端，对象须在这些声明中出现的标识符中查找。如果发生其中的一个，作用域链简要的作如下修改：

Scope = withObject|catchObject + AO|VO + [[Scope]]

在这个例子中添加对象，对象是它的参数（这样，没有前缀，这个对象的属性变得可以访问）。

var foo = {x: 10, y: 20};  
   
with (foo) {  
 alert(x); // 10  
 alert(y); // 20  
}

作用域链修改成这样：

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

我们再次看到，通过with语句，对象中标识符的解析添加到作用域链的最前端：

var x = 10, y = 10;  
   
with ({x: 20}) {  
   
 var x = 30, y = 30;  
   
 alert(x); // 30  
 alert(y); // 30  
}  
   
alert(x); // 10  
alert(y); // 30

在进入上下文时发生了什么？标识符“x”和“y”已被添加到变量对象中。此外，在代码运行阶段作如下修改：

1. x = 10, y = 10;
2. 对象{x:20}添加到作用域的前端;
3. 在with内部，遇到了var声明，当然什么也没创建，因为在进入上下文时，所有变量已被解析添加;
4. 在第二步中，仅修改变量“x”，实际上对象中的“x”现在被解析，并添加到作用域链的最前端，“x”为20，变为30;
5. 同样也有变量对象“y”的修改，被解析后其值也相应的由10变为30;
6. 此外，在with声明完成后，它的特定对象从作用域链中移除（已改变的变量“x”－－30也从那个对象中移除），即作用域链的结构恢复到with得到加强以前的状态。
7. 在最后两个alert中，当前变量对象的“x”保持同一，“y”的值现在等于30，在with声明运行中已发生改变。

同样，catch语句的异常参数变得可以访问，它创建了只有一个属性的新对象－－异常参数名。图示看起来像这样：

try {  
 ...  
} catch (ex) {  
 alert(ex);  
}

作用域链修改为：

var catchObject = {  
 ex: <exception object>  
};  
   
Scope = catchObject + AO|VO + [[Scope]]

在catch语句完成运行之后，作用域链恢复到以前的状态。