深入了解JavaScript，从作用域链开始（1）

地址：http://blog.xieliqun.com/2016/10/05/scope-chain/

> 作用域是每种计算机语言最重要的基础之一，当然它也是JavaScript最重要的概念之一。要想真正的深入了解JavaScript，了解JavaScript的作用域链非常必要。现在让我们深入了解JavaScript作用域和作用域链的工作原理。

JavaScript的作用域是什么

作用域，在维基百科上解释是：在电脑程序设计中，作用域（scope，或译作有效范围）是名字（name）与实体（entity）的绑定（binding）保持有效的那部分计算机程序。
简单的说，作用域就是变量与函数的可访问范围，即作用域控制着变量与函数的可见性和生命周期。在JavaScript中，变量的作用域有全局作用域和局部作用域两种，局部作用域又称为函数作用域。

全局作用域

在代码中任何地方都能访问到的对象拥有全局作用域，一般来说以下几种情形拥有全局作用域：

（1）程序最外层定义的函数或者变量

```
var a = "tsrot";
function hello(){
    alert(a);
}

function sayHello(){
    hello();
}

alert(a);      //能访问到tsrot
hello();       //能访问到tsrot
sayHello();    //能访问到hello函数，然后也能访问到tsrot
```

（2）所有未定义直接赋值的变量（不推荐）

```
function hello(){
    a = "tsrot";
    var b = "hello tsrot";
}

alert(a);  //能访问到tsrot
alert(b);  //error 不能访问
```

（3）所有window对象的属性和方法

一般情况下，window对象的内置属性都拥有全局作用域，例如window.name、window.location、window.top等等。

局部作用域（函数作用域）

局部作用域在函数内创建，在函数内可访问，函数外不可访问。

```
function hello(){
    var a = "tsrot";
    alert(a);
}

hello(); //函数内可访问到tsrot
alert(a); //error not defined
```

作用域链是什么

了解作用域链之前我们要知道一下几个概念：

- 变量和函数的声明
- 函数的生命周期
- Activetion Object（AO）、Variable Object（VO）

变量和函数的声明

在JavaScript引擎解析JavaScript代码的时候，首先，JavaScript引擎会把变量和函数的声明提前进行预解析，然后再去执行其他代码。

变量声明：变量的声明只有一种方式，那就是用`var`关键字声明，直接赋值不是一种声明方式。这仅仅是在全局对象上创建了新的属性（而不是变量）。它们有一下区别：
（1）因为它只是一种赋值，所以不会声明提前

```
alert(a); // undefined
alert(b); // error "b" is not defined
b = 10;
var a = 20;
```

（2）直接赋值形式是在执行阶段创建

```
alert(a); // undefined, 这个大家都知道
b = 10;
alert(b); // 10, 代码执行阶段创建

var a = 20;
alert(a); // 20, 代码执行阶段修改
```

（3）变量不能删除（delete），属性可以删除

```
a = 10;
alert(window.a); // 10

alert(delete a); // true

alert(window.a); // undefined

var b = 20;
alert(window.b); // 20

alert(delete b); // false

alert(window.b); // 仍然为 20，因为变量是不能够删除的。
```

但是，这里有一个意外情况，就是在“eval”的上下文中，变量是可以删除的：

```
eval('var a = 10;');
alert(window.a); // 10

alert(delete a); // true

alert(window.a); // undefined
```

有些debug工具也是可以删除的，因为它们使用了 eval()方法来执行代码的。

函数声明：函数的声明有三种方式
（1）function name( ){ }直接创建方式

```
function add(a,b){
    return a+b;
}

add(5,4);
```

（2）new Funtion构建函数创建

```
var add=new Function("a", "b", "return a+b;");

add(4,5);
```

（3）给变量赋值匿名函数方法创建

```
var add = function(a,b){
    return a+b;
}

add(4,5);
```

后面两种方法，在声明前访问时，返回的都是一个undefined的变量。当然，在声明后访问它们都是一个function的函数。

注意：如果变量名和函数名声明时相同，函数优先声明。

```
alert(x); // function

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

x = 20;

function x() {};

alert(x); // 20
```

函数的生命周期

函数的的生命周期分为创建和执行两个阶段。

在函数创建阶段，JS解析引擎进行预解析，会将函数声明提前，同时将该函数放到全局作用域中或当前函数的上一级函数的局部作用域中。

在函数执行阶段，JS引擎会将当前函数的局部变量和内部函数进行声明提前，然后再执行业务代码，当函数执行完退出时，释放该函数的执行上下文，并注销该函数的局部变量。

什么是AO、VO

英文解释：
AO：Activetion Object（活动对象）
VO：Variable Object（变量对象）

VO对应的是函数创建阶段，JS解析引擎进行预解析时，所有的变量和函数的声明，统称为Variable Object。该变量与执行上下文相关，知道自己的数据存储在哪里，并且知道如何访问。VO是一个与执行上下文相关的特殊对象，它存储着在上下文中声明的以下内容：

- 变量 (var, 变量声明);
- 函数声明 (FunctionDeclaration, 缩写为FD);
- 函数的形参

举个例子：

```
function add(a,b){
    var sum = a + b;
    function say(){
        alert(sum);
    }
    return sum;
}
// sum,say,a,b 组合的对象就是VO，不过该对象的值基本上都是undefined
```

AO对应的是函数执行阶段，当函数被调用执行时，会建立一个执行上下文，该执行上下文包含了函数所需的所有变量，该变量共同组成了一个新的对象就是Activetion Object。该对象包含了：

- 函数的所有局部变量
- 函数的所有命名参数
- 函数的参数集合
- 函数的this指向

举个例子：

```
function add(a,b){
    var sum = a + b;
    function say(){
        alert(sum);
    }
    return sum;
}

add(4,5);
//  我用JS对象来表示AO
//  AO = {
//      this : window,
//      arguments : [4,5],
//      a : 4,
//      b : 5,
//      say : <reference to function>,
//      sum : undefined
//  }
```

JavaScript作用域链

现在我们回到主题，作用域链。

当代码在一个环境中执行时，会创建变量对象的一个作用域链（scope chain）来保证对执行环境有权访问的变量和函数的有序访问。作用域第一个对象始终是当前执行代码所在环境的变量对象（VO）。

```
function add(a,b){
    var sum = a + b;
    return sum;
}
```

假设函数是在全局作用域中创建的，在函数a创建的时候，它的作用域链填入全局对象,全局对象中有所有全局变量，此时的全局变量就是VO。此时的作用域链就是：

```
此时作用域链（Scope Chain）只有一级,就为Global Object

scope(add) -> Global Object(VO)

VO = {
    this : window,
    add : <reference to function>
}
```

如果是函数执行阶段，那么将其activation object（AO）作为作用域链第一个对象，第二个对象是上级函数的执行上下文AO，下一个对象依次类推。

```
add(4,5);
```

例如，调用add后的作用域链是：

```
此时作用域链（Scope Chain）有两级，第一级为AO，然后Global Object（VO）

scope(add) -> AO -> VO

AO = {
    this : window,
    arguments : [4,5],
    a : 4,
    b : 5,
    sum : undefined
}

VO = {
    this : window,
    add : <reference to function>
}
```

在函数运行过程中标识符的解析是沿着作用域链一级一级搜索的过程，从第一个对象开始，逐级向后回溯，直到找到同名标识符为止，找到后不再继续遍历，找不到就报错。

看过上面的内容后，可能还有人不懂，我再通熟易懂的解释一遍，先举个例子：

```
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”。此时的作用域链为：

```
此时作用域链（Scope Chain）有三级，第一级为bar AO，第二级为foo AO,然后Global Object（VO）

scope -> bar.AO -> foo.AO -> Global Object

bar.AO = {
    z : 30,
    __parent__ : foo.AO
}

foo.AO = {
    y : 20,
    bar : <reference to function>,
    __parent__ : <Global Object>
}

Global Object = {
    x : 10,
    foo : <reference to function>,
    __parent__ : null
}
```

未完待续。。。