作用域链

2020年03月09日, 星期一 09:17

1.作用域

作用域是一套关于如何**存储**变量当中的值，并且能在之后对这个值进行**访问**和**修改**的规则

作用域的作用：

作用域指定变量与函数的可访问范围

作用域控制着变量与函数的可见性

2.作用域类型

全局作用域（global scope）

函数作用域（function scope）

块作用域（block scope）//ES5版本中没有，ES6版本中有

3.全局作用域（global scope）

- 在全局作用域下声明的变量叫做全局变量
- 全局变量在全局（代码的任何位置）下都可以使用
- 全局作用域中无法访问到局部作用域中的变量

全局变量的创建方式：

- 在全局作用域下 var 声明的变量
- 在函数内部，没有使用 var关键字声明直接赋值的变量
- 使用 window 全局对象创建的属性和方法

4.函数作用域（function scope）

- 在函数作用域下声明的变量叫做局部变量
- 局部变量只在当前函数内部中使用
- 局部作用域中可以访问到全局作用域中的变量

局部变量的创建方式

- 在函数内部通过 var 声明的变量
- 函数的形参

5.块作用域（block scope）

- 任何一对花括号 { } 中的语句集都属于一个块，在这之中定义的所有变量在代码块外都是不可见的，我们称之为块级作用域
- ES5 没有块作用域，在 ES6 中添加了块作用域

```
{
    var a = 1;
}
console.log(a);//1
```

//ES5

```
{
    let a = 1;
}
console.log(a); //Uncaught ReferenceError: a is not defined
```

//ES6

6.

```
function foo(a) {
    var b = a * 2;
    function bar(c) {
        console.log( a, b, c );
    }
    bar(b * 3);
}
foo( 2 ); // 2, 4, 12
```

//1：全局作用域 2：foo函数作用域 3.bar函数作用域

7.作用域模型

作用域共有两种主要的工作模型：

词法作用域（静态性）***// JS属于词法作用域（静态性）***

- 是由函数定义的**书写位置**决定的，与**调用位置**无关
-

```
//静态性
function foo() {
    console.log(a);//2
}
function bar() {
    var a = 3;
    foo();
}
var a = 2;
bar();
```

动态作用域（动态性）

- 由**调用位置**决定，不关心变量和函数的定义的**书写位置**
-

```
//动态性
function foo() {
    console.log(a);//3
}
function bar() {
    var a = 3;
    foo();
}
var a = 2;
bar();
```

8.JavaScript 作用域 —— 词法作用域（lexical scope）

词法作用域具有**静态性**，静态结构决定了一个变量的作用域

词法作用域与**定义位置**有关，与调用形式无关

```
var a = 3;
function sum() {
    var c = 5;
    console.log(a);
    console.log(c);
    console.log(e);
    return function() {
        var e = 6;
        console.log(a);
        console.log(c);
        console.log(e);
    };
}
```

★ 9.词法作用域补充部分

– 通过 **new Function** 创建的函数对象**不遵从**静态词法作用域

– 通过 **new Function** 创建的函数对象总是在**全局作用域**中执行

```
var scope = "global";

function checkScope() {
    var scope = "local";
    return function() {
        return scope;
    }
}
console.log(checkScope()());
```

local

```
var scope = "global";

function checkScope() {
    var scope = "local";
    return new Function("return scope;");
}
console.log(checkScope()());
```

global

10.[[scope]]属性

- 虚拟属性，无法访问和修改
- 函数**创建（定义）**时生成的属性，保存着这个函数**所有父级执行上下文环境**中的**变量对象**的集合

```
//静态性
function foo() {
    console.log(a);//2
}
function bar() {
    var a = 3;
    foo();
}
var a = 2;
bar();
```

分析foo bar函数的[[scope]]属性的属性值：

foo在定义时就决定了他的scope属性（foo.[[scope]] = [window对象]

bar也同样

11.分析

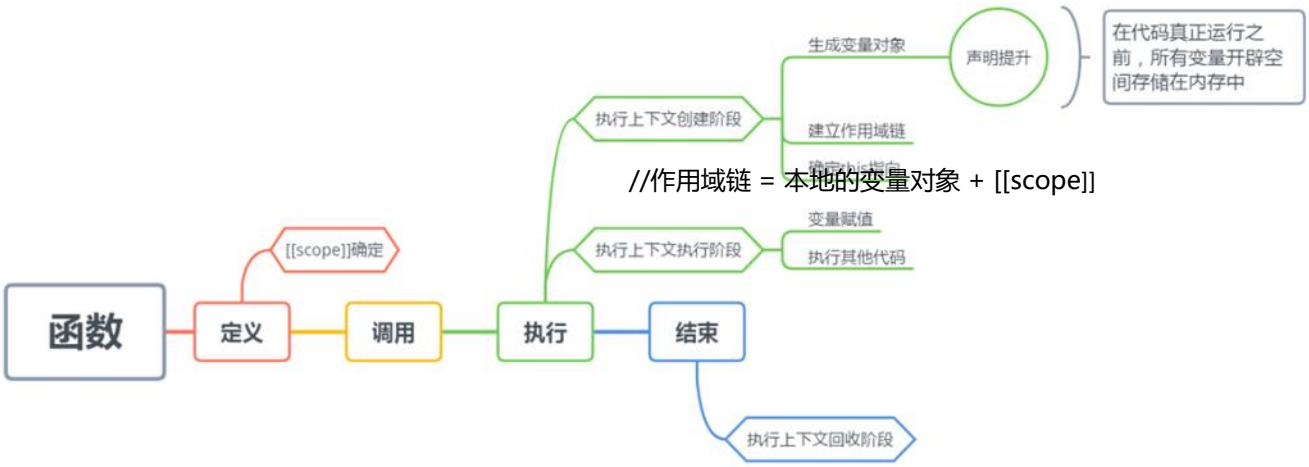
```
var x = 10;

function foo() {
    console.log(x);
}
foo(); // 10

function fun() {
    var x = 20;
    var foo1 = foo;
    foo1(); // 10还是20?
}
fun();
```

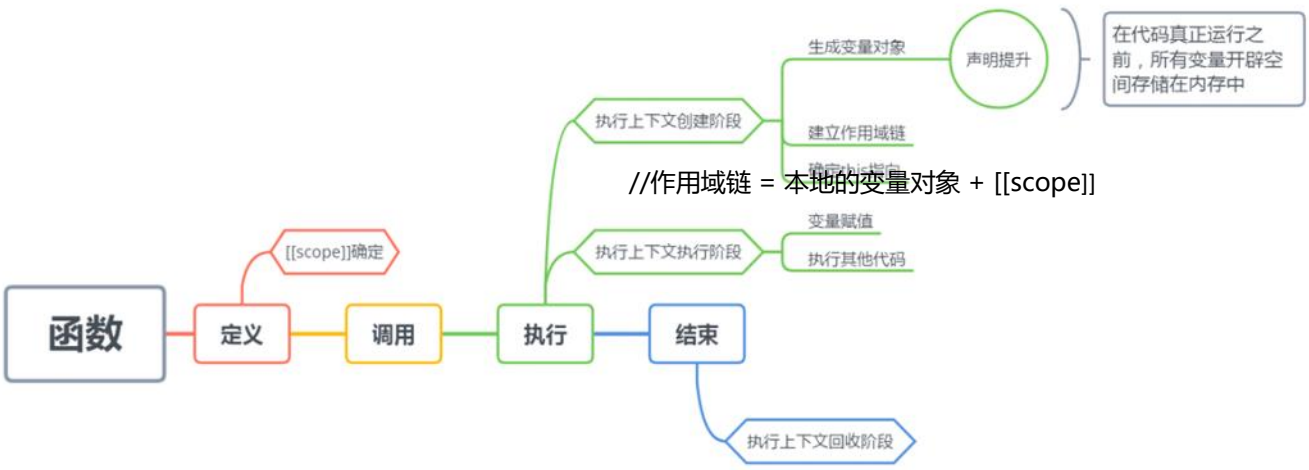
执行foo1( )就是执行foo函数，与fun无关

foo1( )输出结果为10

因此，可以验证，一个函数的作用域的访问过程只跟定义有关，与调用位置无关

12.





//作用域链 = 本地的变量对象 + [[scope]]

13.作用域链（Scope Chain）

- 由当前执行环境与所有父级执行环境的一系列变量对象组成

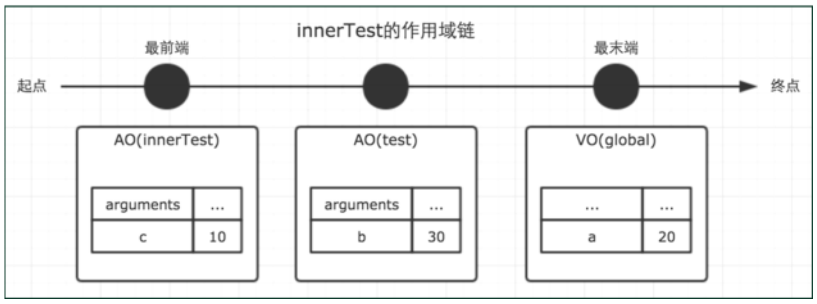
```
ScopeChain = VO + [[scope]]
```

- 提供对变量和函数访问的权限和顺序的规则 //现在自己的作用域(VO)中找，找不到再去他的所有的父级(即[[scope]])去找

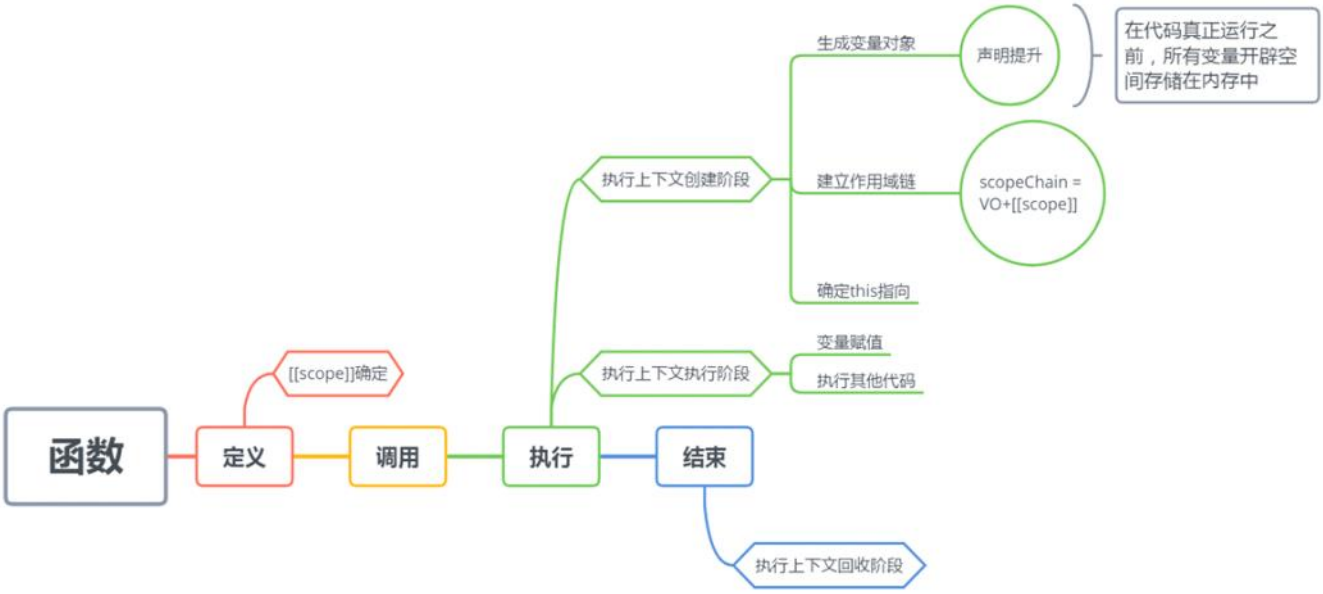
14.

```
var a = 20;

function test() {
    var b = a + 10;

    function innerTest() {
        var c = 10;
        return b + c;
    }
    return innerTest();
}
test();
```

```
innerTestEC = {
    VO: { ...
    },
    scopeChain: [VO(innerTest), VO(test), VO(global)],
}
```

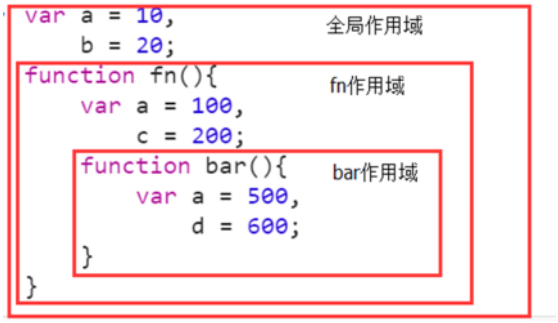


15.



16.变量函数访问规则

- 沿着作用域链从里向外查找
- 查找会在找到第一个匹配的标识符时停止
- 查找只会找一级标识符



左侧实例中：
变量 d 只能在 bar 作用域中被访问到，
变量 c 只能在 fn 和 bar 作用域中被访问到

在 bar 中访问 a 时为 500（遮蔽效应）
在 bar 中访问 c 时为 200（链式查找）

17.

```
//案例1
var name = "global";
var a = 1;
function fn1(a) {
  a++;
  var obj = {    //obj是一级标识符
    name: "fn1"    //这里的name就是二级标识符
  };
  console.log(name);
  console.log(a);
}
fn1(a);
fn1(2);
console.log(a);

  //案例2
  var a = 1;
  function fn() {
      var a = 10;
      function fn1() {
          var a = 20;
          console.log(a);
      }
      function fn2() {
          a++;
          console.log(a);
      }
      fn1(); //20
      fn2(); //11
      fn1(); //20
      fn2(); //12
  }
  fn();
  console.log(a); //1
```