闭包和作用域

作用域

每一种编程语言，它最基本的能力都是 **能够存储变量当中的值、并且允许我们对这个变量的值进行访问和修改**。那么有了变量之后，应该把它放在那里、程序如何找到它们？这是不是需要我们提前约定好一套存储变量、访问变量的规则？这套规则，就是我们常说的**作用域**。

作用域套作用域，就有了作用域链

在 JS 世界中，目前已经有了三种作用域：

- 全局作用域
- 函数作用域
- 块作用域

```
var name = 'foo'; // 全局作用域内的变量
// 函数作用域
function showName() {
    console.log(name);
}
// 块作用域
if (true) {
  name = 'BigBear'
}

showName(); // 输出 'BigBear'
```

全局作用域

声明在任何函数之外的顶层作用域的变量就是全局变量，这样的变量拥有全局作用域

函数作用域

在函数内部定义的变量，拥有函数作用域

```
var name = 'foo'; // name 是全局变量
function showName(myName) {
```

```
  // myName 是传入 showName 的局部变量
  console.log(myName);
}
function sayHello() {
  // hello 被定义成局部作用域变量
  var helloString = 'hello everyone';
  console.log(helloString);
}


showName(name); // 输出 'foo'
sayHello(); // 输出 'hello everyone'
console.log(myName); // 抛出错误：myName 在全局作用域未定义
console.log(helloString); // 抛出错误：hello 在全局作用域未定义

{
  console.log(helloString, myName) // 抛出错误
}
```

块作用域

```
{
  let a = 1;
  console(a);
}

 console(a); // 报错

function showA() {
  console.log(a) // 报错
}
```

作用域链

不止用到一种作用域。当一个块或者一个函数嵌套在另一个块或者函数中时，就发生了作用域的嵌套。比如这样：

```
function addA(a) {
  console.log(a + b)
  console.log(c) // 报错
}
var b = 1
addA(2) //3
```

闭包

```
function addABC(){
  var a = 1,b = 2;

  function add(){
    return a+b+c;
  }
  return add;
}

var c = 3

var globalAdd = addABC()

console.log(globalAdd()) // 6
```

作用域嵌套的情况展示如下：

其中 add 这个函数，它嵌套在函数 addABC 的内部，想要查找 a、b、c 三个变量，它得去上层的 addAB 作用域里找，对吧？像 a、b、c 这样**在函数中被使用，但它既不是函数参数、也不是函数的局部变量，而是一个不属于当前作用域的变量，此时它相对于当前作用域来说，就是一个自由变量**。而像 add 这样**引用了自由变量的函数，就叫闭包。**
有权访问另一个函数作用域里面变量的函数。

探索词法作用域模型

站在语言的层面来看，作用域其实有两种主要的工作模型：

- 词法作用域：也称为静态作用域。这是最普遍的一种作用域模型，也是我们学习的重点
- 动态作用域：相对“冷门”，但确实有一些语言采纳的是动态作用域，如：Bash 脚本、Perl 等

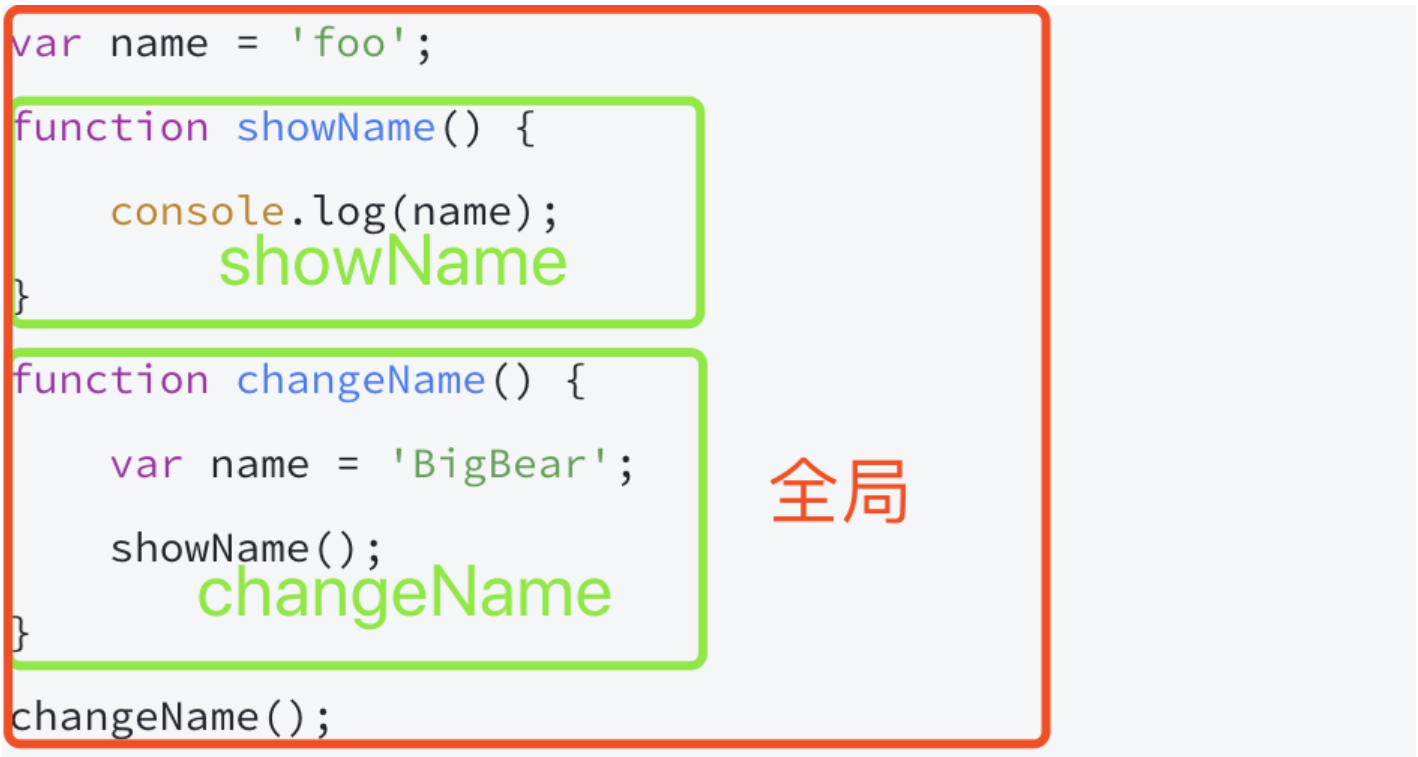
想要理解词法作用域本身，我们就不得不从 JS 的框框里跳出来，把它和它的对立面“动态作用域”放在一起来看。为了使两者的概念更加直观，我们直接来看一段代码：

```
var name = 'foo';
function showName() {
    console.log(name);
}
function changeName() {
    var name = 'BigBear';
    showName();
}
changeName();
```

这是一段 JS 代码，基于我们上节对 JS 作用域的学习，不难答出它的运行结果是 ‘foo’。这是因为 JS 采取的就是词法（静态）作用域，这段代码运行过程中，经历了这样的变量定位流程：

- 在 showName 函数的函数作用域内查找是否有局部变量 name
- 发现没找到，于是根据 **书写的位置**，查找上层作用域（全局作用域），找到了 name 的值是 foo，所以结果会打印 foo。

查找规则如下图：



如果是动态作用域：**沿着函数调用栈、在调用了 showName 的地方**继续找 name

我们总结一下，词法作用域和动态作用域的区别其实在于**划分作用域的时机：**

- 词法作用域： 在代码书写的时候完成划分，作用域链沿着它**定义的位置**往外延伸
- 动态作用域： 在代码运行时完成划分，作用域链沿着它的**调用栈**往外延伸

欺骗词法作用域

在相对高阶的前端面试中，有时面试官会抛出这样的问题：如何“欺骗”词法作用域
摸底，想知道你对词法作用域到底了解到了什么程度。

JS 遵循词法作用域模型已成定局，难道我还能把它扳成动态作用域不成？别说，还真行。

Eval

eval 拿到一个字符串入参后，它会把这段字符串的内容当做一段 js 代码（不管它是不是一段 js 代码），插入自己被调用的那个位置

```
function showName(str) {
  // var name = "BigBear
  eval(str)
  console.log(name)
}

var name = 'xiuyan'
var str = 'var name = "BigBear"'

showName(str) // 输出 BigBear
```

eval，强行插入了一个 name 的申明。

With

扩展一个语句的作用域链：
with 对大家来说可能比 eval 要陌生一些。它的作用就是帮我们 “偷懒”，当我们不想重复地写一个对象名作为前缀的时候，with 可以帮到我们：

```
var me = {
  name: 'xiuyan',
  career: 'coder',
  hobbies: ['coding', 'footbal']
}
```

```
6
7  // 假如我们想输出对象 me 中的变量，没有 with 可能会这样做：
8  console.log(me.name)
9  console.log(me.career)
10 console.log(me.hobbies)
11
12 // 但 width 可以帮我们省去写前缀的时间
13 with(me) {
14  // var name = '', career ??
15   console.log(name)
16   console.log(career)
17   console.log(hobbies)
18 }
```

'with'语句將某个对象添加到作用域链的顶部，如果在statement中有某个未使用命名空间的变量，跟作用域链中的某个属性同名，则这个变量将指向这个属性值。如果沒有同名的属性，则将抛出异常。

真题

1:

```
1  function foo(a,b){
2    console.log(b);
3    return {
4      foo:function(c){
5        return foo(c,a);
6      }
7    }
8  }
9
10 var func1=foo(0);   // undefined
11 func1.foo(1);      //  0
12 func1.foo(2);     // 0
13 func1.foo(3);     // 0
14 var func2=foo(0).foo(1).foo(2).foo(3);   // undefined 0  1  2
15 var func3=foo(0).foo(1);   // undefined 0
```

```
// func3 闭包里面的 a 一直留着，
func3.foo(2);  // 1
func3.foo(3);   // 1
```

闭包的应用

私有变量

```
// 利用闭包生成IIFE，返回 User 类
const User = (function() {
    // 定义私有变量_password
    let _password

    class User {
        constructor (username, password) {
            // 初始化私有变量_x
            _password = password
            this.username = username
        }
       // getPwd() { return _password }
       login() {
           // 这里我们增加一行 console，为了验证 login 里仍可以顺利拿到密码
           console.log(this.username, _password)
           // 使用 fetch 进行登录请求，同上，此处省略
       }
    }

    return User
})()

let user = new User('xiuyan', 'xiuyan123')

console.log(user.username)
console.log(user.password)
```

```
27 console.log(user._password)
28 user.login()
```

我大函数式编程里面一展风采

- 柯里化
- 高阶函数
- 纯函数缓存

Curry：函数的输入
纯函数：建议你定义函数的时候都定义成纯函数
Compose：组合函数功能的
高阶组件 / 高阶函数：过程抽象

// 函子