



乐 美 无 限  
nemo-tec.com

# JavaScript函数作用域链

豆连军 @八月虎baidu  
北京乐美无限科技有限公司

# 什么是函数对象

- JS的函数就是对象。
  - 对象字面量产生的对象通过 “\_\_proto\_\_” （用 `[[Prototype]]` 表示）属性隐含连接到 `Object.prototype`，而函数对象隐含连接到 `Function.prototype`，`Function.prototype` 再隐含连接到 `Object.prototype`。
- 函数是一个特殊的对象
  - 两个隐藏属性：
    - 函数的上下文（函数对象所处的作用域），用 `[[scope]]` 表示。
    - 函数体（实现函数行为的代码语句片段），可理解为计算属性，用 “()” 得到计算属性的计算值，也即函数本次运算的结果。
  - 两个附送属性：
    - `length`（形式参数长度）
    - `prototype`：函数的构造器原型对象。它的值是一个拥有 `constructor` 属性且 `constructor` 属性的值即为该函数的对象。与 `[[Prototype]]` 不同。
  - 两个缺省 `local` 局部变量，函数运行时才能被确定的变量：
    - `this`、`arguments`（实参）

# 函数的对象用法

```
var myFunc = function(){};  
myFunc.myName = “张三”;  
myFunc.age = 21 ;  
  
console.log(myFunc.myName) ;
```

# 判断对象的类型

- 判断一个对象是Arguments 还是 Arrays的方法: instanceof 和Object.prototype.toString

```
1 function bob(one, two, three) {  
2     var x = arguments;  
3     var y = [];  
4  
5     console.log ( x instanceof Array ); //false  
6     console.log ( y instanceof Array ); //true  
7 };
```

```
1 function bob(one, two, three) {  
2     var x = arguments;  
3     var y = [];  
4  
5     console.log( Object.prototype.toString.call( x ) ); //[object Argume  
6     console.log( Object.prototype.toString.call( y ) ); //[object Array]  
7 };
```

```
1 var p = new Person("Richard");  
2 p instanceof Person //true
```



# 变量的作用域

- 什么是作用域（scope）
  - JavaScript的作用域完全是一个语法概念上的，一个变量的作用域基于其声明时在代码中出现的位置。
- 两种作用域
  - Global全局作用域：全局对象window
  - Functional函数局部作用域：位于函数内部。

# 全局变量 vs 局部变量

```
var message = 'Hello';  
setMessage();  
console.log(message);
```

全局变量-  
位于Windows上下文中

```
function setMessage () {  
    var message = 'Goodbye';  
    console.log(message);  
}
```

局部变量-  
位于函数'setMessage'上下文中

Goodbye  
Hello

# ES5没有块作用域

```
for (var i = 0; i < 10; i++) {  
    /* ... */  
}  
console.log(i); // 输出结果是？
```

ES6: let关键字可以替换var来声明块作用域的变量。  
块作用域: if、switch、while、for等语句块中。

# 变量声明和函数声明的提升

- declaration hoisting
  - 函数在执行代码之前，会将函数体中的变量声明和函数声明提升到代码的最前面。
  - 函数表达式不会被提升
  - for语句中var声明的变量也会提升



# 变量和函数声明

```
<script>
  var a = 100 ; // 全局变量a
  function func1() {
    a = 1 ; // 此处为局部变量a
    func2() ; // 输出 1
    var a = 10 ;
    console.log(a) ; // 输出 10 解析为:
    function func2() {
      console.log(a) ;
    }
  }
  console.log(a) ; // 输出 100
</script>
```



```
<script>
  // 前面代码被js引擎解释为:
  var a = 100 ; // 全局变量a
  function func1() {
    var a ;
    function func2() {
      console.log(a) ;
    }

    a = 1 ; // 此处为局部变量a
    func2() ; // 输出 1
    a = 10 ;
    console.log(a) ; // 输出 10
  }
  console.log(a) ; // 输出 100
</script>
```

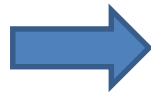
# 函数声明 vs 函数表达式

函数声明	函数表达式
出现在代码主流程中	作为表达式的一部分出现
在函数执行前被创建，在定义位置之前和之后都可以使用	仅在代码运行到所在位置时才创建，只能在被创建之后使用
	在所定义的位置即可进行调用

# 条件函数声明

解析为:

```
if (condition) {  
    function myfunction() {  
        // some code  
    }  
}
```



```
if (condition) {  
    var myfunction = function () {  
        // some code  
    }  
}
```

# 函数表达式不会被提升

```
<script>
```

```
    // 只有函数声明会被提升，而函数表达式不会被提升。
```

```
    definitionHoisted(); // 输出: "本函数被提升了!"
```

```
    definitionNotHoisted(); // TypeError(类型错误): undefined is not a  
function
```

```
    function definitionHoisted() {  
        console.log("本函数被提升了!");  
    }
```


```
    var definitionNotHoisted = function () {  
        console.log("本函数是表达式，未能被提升!");  
    };
```

```
</script>
```




# 输入参数

```
var message = 'Hello';  
setMessage('Goodbye');  
console.log(message);
```



'Goodbye' 作为一项参数值，传递给 setMessage 函数

```
function setMessage (message) {  
    console.log(message);  
}
```



message 作为参数变量名，位于函数作用域中，是本地变量。

```
Goodbye  
Hello
```

# 返回值

```
var message = 'Hello';  
console.log(setMessage());  
console.log(message);
```

message 作为函数返回值被返回时，仍然是函数作用域一部分。

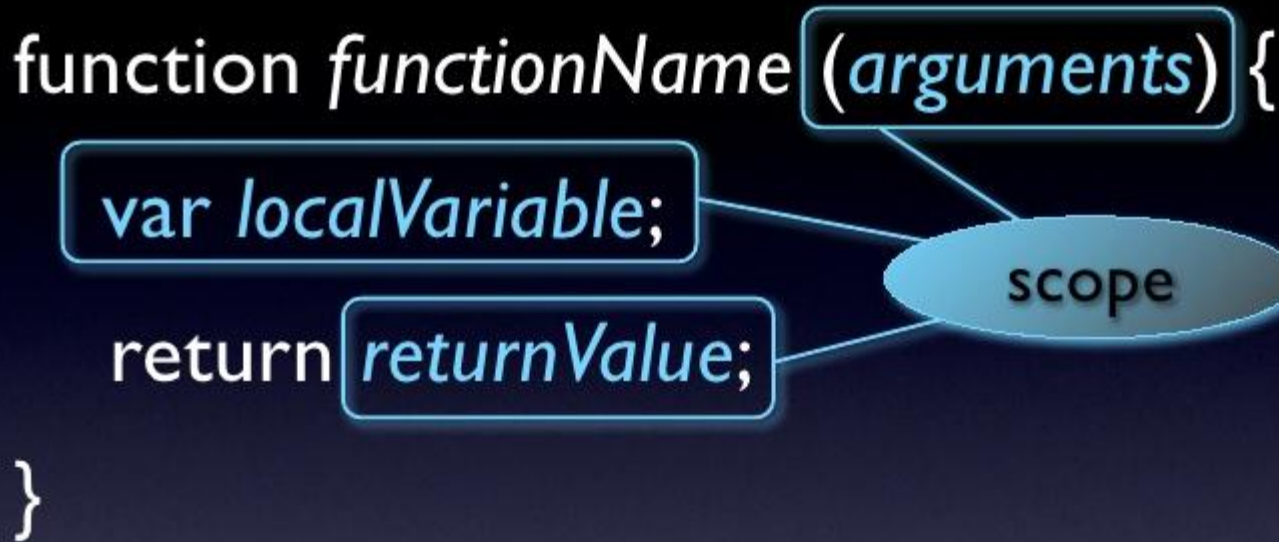
```
function setMessage () {  
    var message = 'Goodbye';  
    return message;  
}
```

message 位于函数作用域，作为函数返回值。

```
Goodbye  
Hello
```

# 函数作用域总结

- 函数的输入参数、局部变量、返回值等均位于函数作用域中。



The diagram illustrates the scope of a function. It shows a function definition: `function functionName (arguments) {`. The `arguments` parameter is enclosed in a box. Inside the function body, there is a local variable declaration `var localVariable;` and a return statement `return returnValue;`. Both `localVariable` and `returnValue` are enclosed in boxes. A central oval labeled `scope` has lines connecting it to the boxes around `arguments`, `localVariable`, and `returnValue`, indicating that these elements are all part of the function's scope. The closing brace `}` is also shown.

```
function functionName (arguments) {  
    var localVariable;  
    return returnValue;  
}
```

# 有无var声明的区别

- 声明变量时，应总是使用var

```
var message = 'Hello';  
setMessage();  
console.log(message);
```

与全局变量冲突

```
function setMessage () {  
  message = 'Goodbye';  
  console.log(message);  
}
```

没有使用var, 冲突

Goodbye  
Goodbye

```
var message = 'Hello';  
setMessage();  
console.log(message);
```

全局变量现在安全

```
function setMessage () {  
  var message = 'Goodbye';  
  console.log(message);  
}
```

局部变量使用var

Goodbye  
Hello





# 全局变量是有害的

- 全局变量容易与其他代码模块起冲突, 不容易定位故障原因。

The image shows a code editor with a function definition on the left and a global variable declaration on the right. The function is named `myStuff` and contains a line `tmp = true;`. The global variable is named `tmp` and is set to `false`. Annotations in Chinese question the purpose of the variable and the confusion it causes.

```
function myStuff () {  
  tmp = true;  
  ...  
}
```

这个变量干啥的?

tmp

false

令人纠结啊

+4½ more screens of globals

# 作用域总结

- 两种作用域：全局和函数
- 函数作用域：形式参数、局部变量和返回值
- 显式地声明变量： `var`
- 将全局变量保持到最小数量
- 对变量使用有意义的命名
- 函数对象的`[[scope]]`属性指向了函数对象所处的那个作用域。

# 函数的运行时上下文

Execution context	
Variable object	{ vars, function declarations, arguments... }
Scope chain	[ Variable object + all parent scopes ]
thisValue	Context object

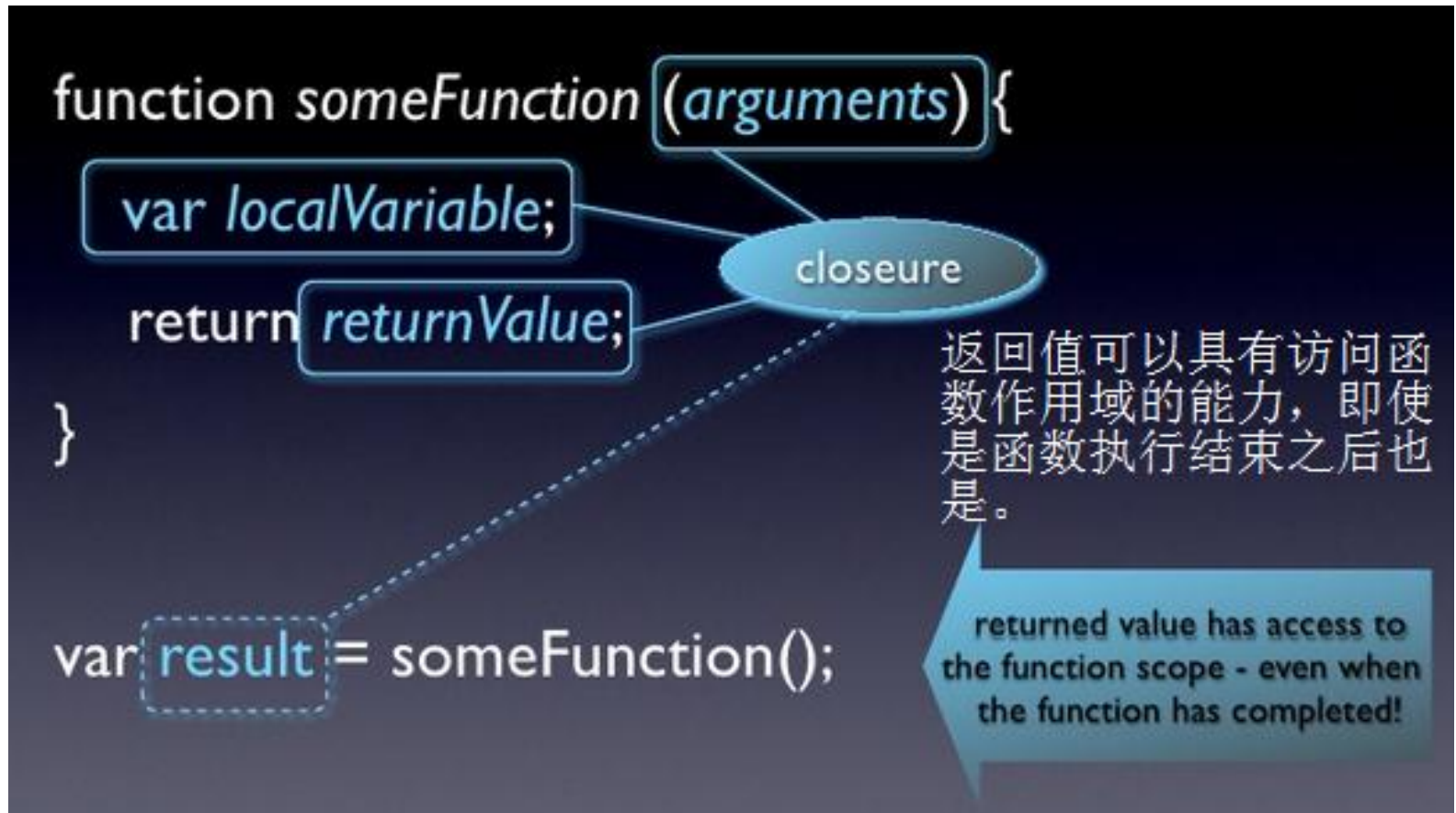
# 常见错误

```
var elements =  
document.getElementsByTagName('input');  
var n = elements.length; // 假设有10个元素  
for (var i = 0; i < n; i++) {  
    elements[i].onclick = function() {  
        console.log("当前元素编号: #" + i);  
    };  
}
```

# 正确

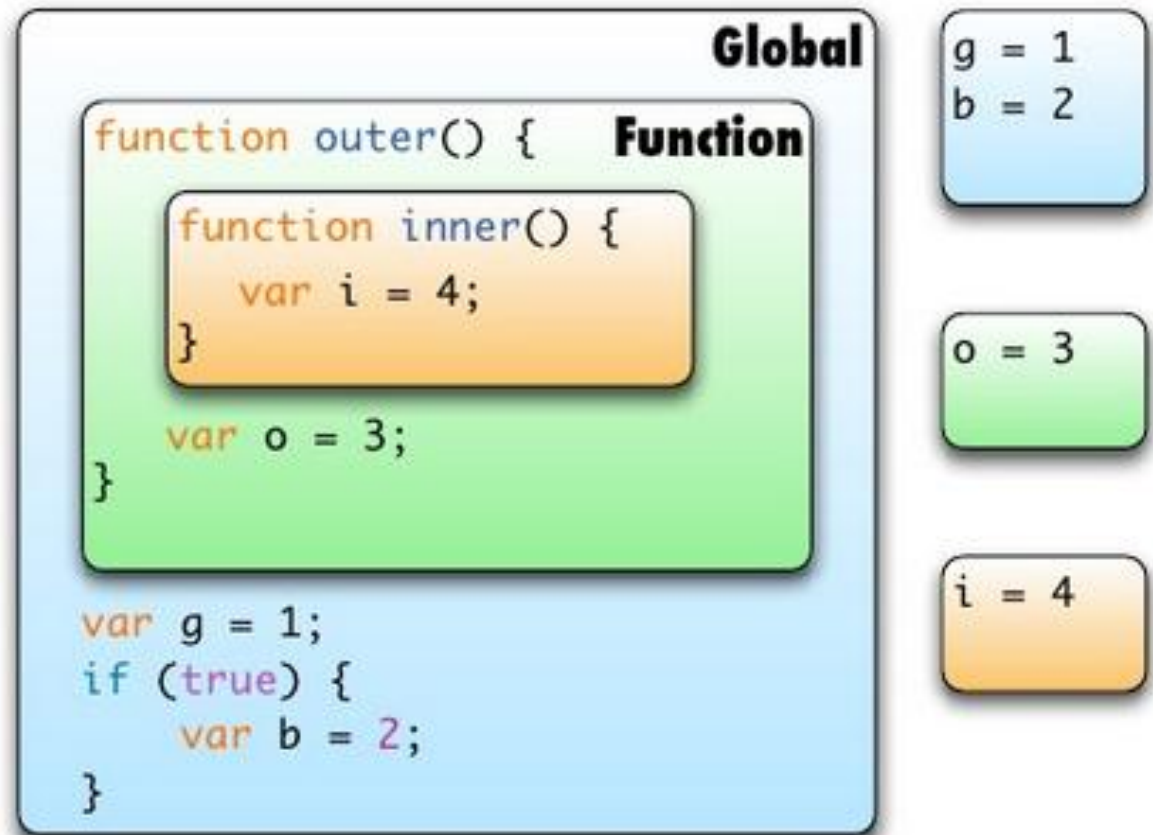
```
var elements =  
document.getElementsByTagName('input');  
var n = elements.length; // 假设有10个元素  
for (var i = 0; i < n; i++) {  
    elements[i].onclick = (function(num) {  
        return function(){  
            console.log("当前元素编号: #" + num);  
        }  
    })(i);  
}
```

# 函数作用域



# Scope (作用域)

- ▶ Scopes
  - ▶ Global
  - ▶ Function
- ▶ No block-level



# 实例演示

```
6 </head>
7 <body>
8
9 <script type="text/javascript">
10 var a = 1;
11 // function abs(x){
12 //   return Math.abs(x);
13 // }
14 setTimeout(function(){
15   alert(b);
16 }, 1000);
17
18 if(a>=1){
19   var b = 10;
20 }
21 console.log(b);
22 </script>
23 </body>
24 </html>
```

代码执行位置!

JS没有花括弧作用域!

此时, 本作用域下的变量表已经建立!

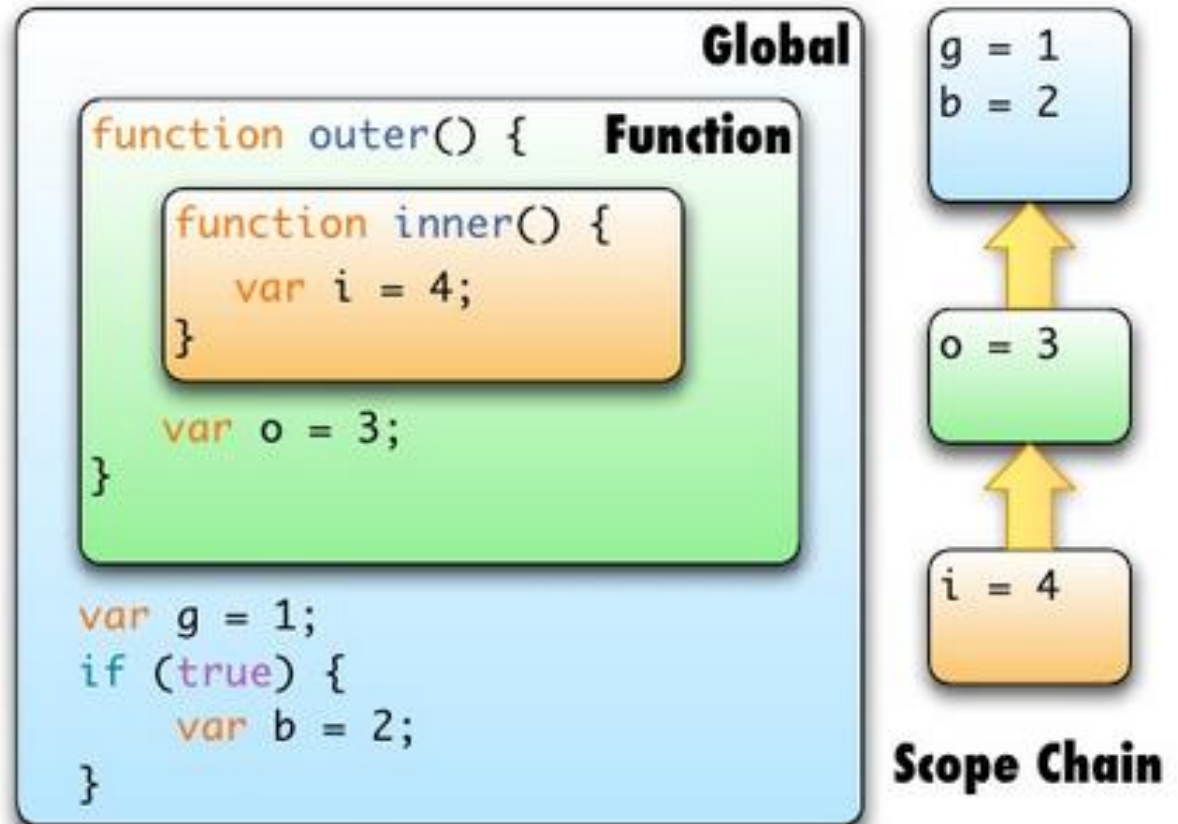
新建监控表达式...

+	this	Window js.scope.basic.html
-	全局作用域 [Window]	Window js.scope.basic.html
	a	undefined
	b	undefined
+	InstallTrigger	InstallTriggerImpl { SKIN=1, LOCALE=2, CONTENT=4, 多... }
+	applicationCache	0 items in offline cache
	closed	false
+	console	Object { log=function(), debug=function(), info=function(), 更多... }
	constructor	Object { }
+	content	Window js.scope.basic.html



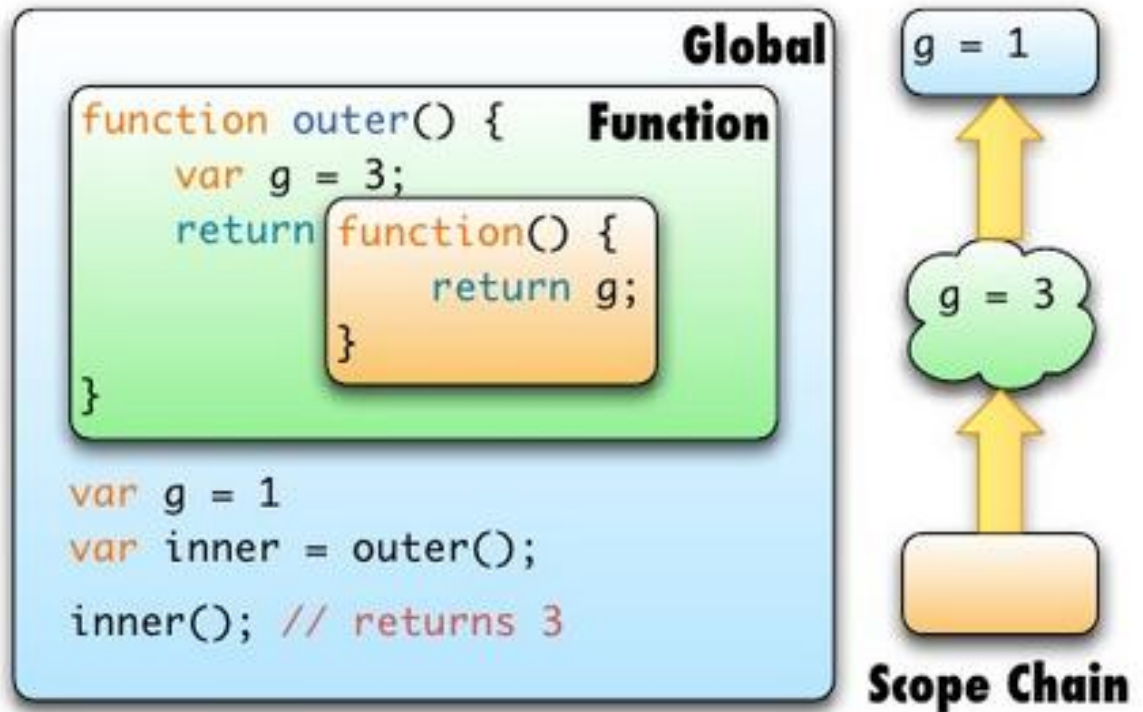
# Scope Chain（作用域链）

- ▶ Scope Chain
- ▶ Each scope “inherits” from the “previous” one



# 闭包Closure

- ▶ Closure:  
Functions  
“remember”  
their scope  
chain



# 闭包Closure

```
function something () {  
  var secretTreasure = '$$$';  
  return {  
    getTreasureLength: function () {  
      return secretTreasure.length;  
    },  
    doubleTheTreasure: function () {  
      secretTreasure += secretTreasure;  
    }  
  };  
}
```

```
var mine = something();  
console.log(mine.getTreasureLength());  
mine.doubleTheTreasure();  
console.log(mine.getTreasureLength());
```

secretTreasure 是私有的（外面看不到）

返回的对象，与局部变量secretTreasure共享同一个函数Scope作用域。

我们可以使用对象的public方法来访问私有成员

```
>>> 3
```

```
>>> 6
```

# 闭包例子

```
var buildList ;
buildList = function (list) {
    var result = [];
    for (var i = 0; i < list.length; i++) {
        var item = 'item' + i;
        result.push(
            function() {console.log(item + ' ' + list[i])}
        );
    }
    return result;
}
```

```
var testList ;
testList = function () {
    var fnlist = buildList([1,2,3]);
    // Using j only to help prevent confusio
    i.
    for (var j = 0; j < fnlist.length; j++) ;
        fnlist[j]();
    }
}
```

*testList()* //1错误: ogs "item2 undefined" 3

```
buildList = function (list) {
    var result = [];
    for (var i = 0; i < list.length; i++) {
        var item = 'item' + i;
        result.push(
            function(x) {
                return function() {console.log(item + ' ' + list[x])}
            }(i)
        );
    }
    return result;
}
```

*testList()* // 正确 !

# 闭包例子1

- 利用闭包实现Timer定时器:

```
var count = 0;

var timer = setInterval(function(){
  if ( count < 5 ) {
    log( "Timer call: ", count );
    count++;
  } else {
    assert( count == 5, "Count came via a closure, accessed each step." );
    assert( timer, "The timer reference is also via a closure." );
    clearInterval( timer );
  }
}, 100);
```



# 闭包例子2

- 利用闭包实现DOM事件监听器:

```
var count = 1;
var elem = document.createElement("li");
elem.innerHTML = "Click me!";
elem.onclick = function(){
    log( "Click #", count++ );
};
document.getElementById("results").appendChild( elem );
assert( elem.parentNode, "Clickable element appended." );
```

# 闭包例子3

- 利用闭包实现类的私有属性:

```
function Ninja() {  
    var slices = 0;  
  
    this.getSlices = function() {  
        return slices;  
    };  
    this.slice = function() {  
        slices++;  
    };  
}  
  
var ninja = new Ninja();  
ninja.slice();
```

# 闭包技术实现私有属性封装

- OO应可以支持封装，可以封装private私有成员；
- 但Object无法直接封装private成员，不过通过闭包技术可以实现私有属性的封装。

```
var radio = {  
  volume: 0,  
  frequency: 88.0,  
  
  changeVolume: function (direction) {  
    if (direction === 'up') this.volume += 1;  
    else this.volume -= 1;  
  },  
  changeTuner: function (direction) {  
    if (direction === 'up') this.frequency += 4;  
    else this.frequency -= 4;  
  }  
};
```





# 编程模式： module pattern

```
var Module = (function(){  
    var privateProperty = 'foo';  
  
    function privateMethod(args){  
        //do something  
    }  
  
    return {  
  
        publicProperty: "",  
  
        publicMethod: function(args){  
            //do something  
        },  
  
        privilegedMethod: function(args){  
            privateMethod(args);  
        }  
    }  
})();
```

# 模块技术

- 按模块封装代码块：

```
(function() {  
  var myLib = window.myLib = function() {  
    // Initialize  
  };  
  
  // ...  
})();
```

```
!function(){console.log(1);}()
```

```
var myLib = (function() {  
  function myLib() {  
    // Initialize  
  }  
  
  // ...  
  
  return myLib;  
})();
```

# 模块技术

```
(function(){  
  var count = 0;  
  
  var timer = setInterval(function(){  
    if ( count < 5 ) {  
      log( "Timer call: ", count );  
      count++;  
    } else {  
      assert( count == 5, "Count came via a closure, accessed each step." );  
      assert( timer, "The timer reference is also via a closure." );  
      clearInterval( timer );  
    }  
  }, 100);  
})();  
  
assert( typeof count == "undefined", "count doesn't exist outside the wrapper" );  
assert( typeof timer == "undefined", "neither does timer" );
```

# 模块化的好处

- 1。 清晰管理模块的数据状态和外部接口
- 2。 方便代码复用
- 3。 模块的依赖关系可以被自动管理

# 总结

- Object对象的属性是公共（public）的。
- function函数体中的局部变量是私有（private）的。
- 在函数执行完毕后，函数的闭包提供访问局部变量的能力。
- function函数可以提供带有私有数据成员模块(module)