

## JavaScript 闭包

### 一、闭包的作用

JavaScript 中闭包是一个很难理解的概念，也是衡量 JavaScript 功力的重要标准。JavaScript 语法中的独特特性也通过闭包淋漓尽致的体现了出来。

在学习闭包的过程中，很多同学最大的困惑其实并不是语法本身，而是看着闭包的语法不知道它是干什么用的。难道闭包仅仅是一项供奉在象牙塔中供学术研究者膜拜的高冷课题吗？

其实并不是这样，闭包的本质是将一个函数中某些变量的作用域延伸到函数外部的技术。所有需要突破作用域链，在函数外部访问函数内部变量值的场合都可以使用闭包。具体的例子我们看过语法后再讲述。

### 二、作用域链

#### 1. 作用域

在 JavaScript 中，变量也可以分为全局变量和局部变量。

##### ① 全局变量

直接在 script 标签内声明的变量就是全局变量

```
var global = "Hello I am Global";  
console.log("global="+global);
```

执行结果：global=Hello I am Global

##### ② 局部变量

在函数中声明的变量是局部变量

```
function myMethod() {  
    var localVariable = "Hello I am Local";  
    console.log("localVariable="+localVariable);  
}  
  
myMethod();
```

执行结果：localVariable=Hello I am Local

但是在函数中声明变量时如果没有指定 var 关键字那么也是全局变量

```
function mySpecialMethod() {  
    specialVariable = "Hello I am Specail";  
}  
  
mySpecialMethod();  
console.log("specialVariable="+specialVariable);
```

执行结果：specialVariable=Hello I am Specail

## ③变量可见性

函数中可以使用全局变量，但函数外不能使用局部变量。

```
var global = "Hello I am Global";
console.log("global="+global);

function accessGlobal() {
    console.log("access global in function="+global);
}

accessGlobal();
```

执行结果：

global=Hello I am Global

access global in function=Hello I am Global

```
function myMethod() {
    var localVariable = "Hello I am Local";
    console.log("localVariable="+localVariable);
}

myMethod();

console.log("access localVariable in global="+localVariable);
```

执行结果：

localVariable=Hello I am Local

Uncaught ReferenceError: localVariable is not defined

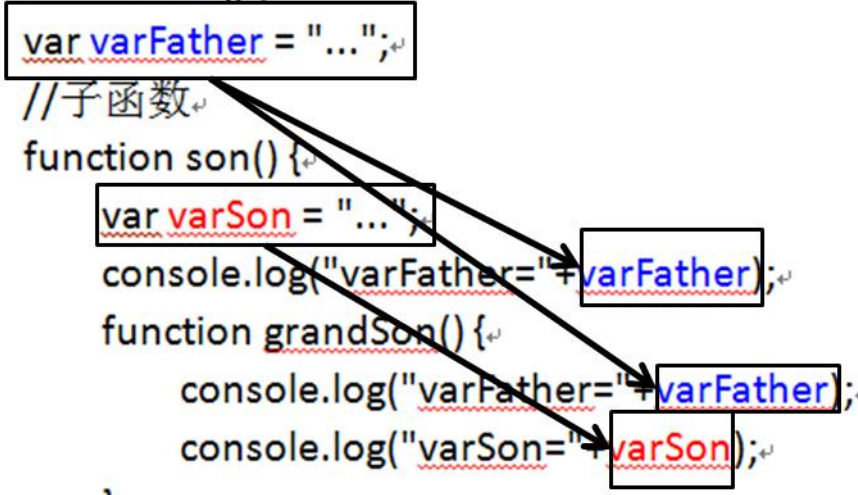
## 2.作用域链

作为一门奇葩语言，JavaScript 允许在函数内部再声明函数。以下为了称呼方便，我们把包含函数声明的函数称为父函数，把在函数内声明的函数称为子函数。子函数中当然可以再继续声明孙子辈的函数，并且“子子孙孙无穷匮也”。

```
//父函数
function father() {
    //子函数
    function son() {
        //...
        function grandSon() {
            //...
        }
    }
}
```

从作用域的角度来说，子函数可以访问父函数中声明的变量以及祖先函数中的变量，但父函数不可以访问子函数中声明的变量，这就是作用域链。

```
//父函数
function father() {
    var varFather = "...";
    //子函数
    function son() {
        var varSon = "...";
        console.log("varFather=" + varFather);
        function grandSon() {
            console.log("varFather=" + varFather);
            console.log("varSon=" + varSon);
        }
    }
}
```



### 三、突破作用域链

#### 1. 未使用闭包时

默认情况下上一级作用域(以下称父作用域)中不能访问下一级作用域(以下称子作用域)中声明的变量，这一点前面已有例证。

#### 2. 使用闭包突破作用域链

如果确实需要在父作用域中访问子作用域中声明的变量，可以通过下面这样的方法：

```
//包含特殊数据的一个函数
function variableFunction() {

    var variableForGloabl = "需要在上一级作用域中使用的变量";

    //声明一个子函数，这个函数是可以访问到 variableForGloabl 的
    function getVariable() {
        return variableForGloabl;
    }

    //将子函数的引用返回，这一点很关键
    return getVariable;
}
```

```
}

//获取 getVariable()函数的引用
var funRef = variableFunction();

//调用有权访问局部变量的 getVariable()函数获取数据
var data = funRef();
console.log("data="+data);

//可以多次调用获取
data = funRef();
console.log("data="+data);
```

执行结果：

data=需要在上一级作用域中使用的变量  
data=需要在上一级作用域中使用的变量

### 3. 闭包

其实上例中有权访问局部变量的 `getVariable()` 函数就是闭包。它所起的作用是将局部变量的作用域延伸到了全局范围——当然也可以说是从子作用域延伸到了父作用域。

这里有几个问题需要说明：

#### ① 返回函数的引用

实现闭包的过程中返回函数的引用是非常关键的一步，如果不是返回函数的引用就与改变作用域范围无关了。因为如果不返回函数的引用则可能直接返回变量值本身或闭包函数的执行结果，这样一来局部变量的作用域并不会被改变，我们仅仅是将它的值返回了，父函数执行完成后，局部变量就随之被垃圾回收机制释放掉了，对原本的作用域没有任何影响。

#### ② 缓存效果

局部变量被闭包函数返回后，即使父函数执行完毕，局部变量仍然会驻留在内存中，这是闭包技术中的一个非常鲜明的特点。

首先我们来证实这一点：

```
//声明包含闭包的函数 keepVariable()
function keepVariable() {
    //number 是要保持的数据
    var number = 1;
    //打印初始值
    console.log("init number="+number);
    //返回闭包函数
    return function(){
        //由于闭包函数的引用被返回给了上一级作用域，随时会被调用执
```

行，所以 number 要始终保持在内存中，直到父作用域也被释放

```
        number++;  
        console.log("number="+number);  
    };  
}
```

```
var closure = keepVariable();  
closure();  
closure();
```

执行结果：

```
init number=1  
number=2  
number=3
```

## 四、实例

下面我们来看几个使用闭包技术的具体例子：

### 1. 数据封装

在 Java 中，成员变量往往被设置为私有，提供对应的 public 权限的 getXxx() 和 setXxx() 方法与外界交互。而 JavaScript 中没有类的概念，也没有权限的概念，默认情况下对象的属性是可以直接读写的。

#### ① 直接读写对象属性

```
var person = {personName:"Tom"};  
console.log("person name="+person.personName);  
  
person.personName = "Jerry";  
console.log("person name="+person.personName);
```

执行结果：

```
person name=Tom  
person name=Jerry
```

#### ② 封装数据

在 JavaScript 中，我们可以借助作用域链封装数据，借助闭包提供 getXxx() 和 setXxx() 方法。

```
function Person(){  
    var personName = "init-value";  
    return {  
        getPersonName : function(){  
            return personName;  
        },  
        setPersonName : function(perName){
```

```
        personName = perName;
    }
};
}

var personObj = Person();
personObj.setPersonName("Tom");
console.log("Person's name="+personObj.getPersonName());
```

执行结果:

Person's name=Tom

看到这里可能大家会有疑问：这里返回的是一个对象并不是子函数啊？其实闭包的“官方解释”中并没有规定闭包必须是一个函数，所有封装局部变量并将作用域延伸到上一级都可以叫闭包。

## 2. 数据缓存

在较为复杂的 JavaScript 应用中，有些数据的生成和获取比较耗时。那如果每次用到数据都重新获取显示是效率十分低下的。对于获取不易的数据，将其缓存起来往往是一个提升性能的重要手段。

```
function cacheInit() {
    //缓存数据的容器对象
    var cache = {};

    //闭包方法
    return function(){

        //首先尝试从缓存中读取数据
        if('dataCached' in cache) {
            console.log("--->读取缓存");
            //如果数据已在缓存中则直接返回
            return cache['dataCached'];
        }

        console.log("--->读取缓存失败，下面主动生成数据");

        //执行一个非常耗时的创建数据的函数，返回数据
        function createData() {
            console.log("--->生成数据");
            dataCached = "Heavy Data";
            //将数据保存到缓存中
            console.log("--->将数据保存到缓存中");
            cache['dataCached'] = dataCached;
        }
    }
}
```

```
        return dataCached;
    }

    //返回数据
    return createData();
};
}

var getData = cacheInit();
var heavyData = getData();
console.log("第一次获取数据: "+heavyData);

heavyData = getData();
console.log("第二次获取数据: "+heavyData);
```

执行结果:

--->读取缓存失败，下面主动生成数据

--->生成数据

--->将数据保存到缓存中

第一次获取数据: Heavy Data

--->读取缓存

第二次获取数据: Heavy Data