第七章 函数表达式

函数声明和函数表达式

- 函数表达式构造出来的函数本质上是一个匿名函数（即它的name属性是空字符串）

```
//函数声明
sayHi(); //可以 因为会函数声明提升
function sayHi(){
    alert('Hi');
}

//函数表达式
sayBye();//Uncaught TypeError: sayBye is not a function
var sayBye = function(){
    alert('Bye');
}
```

递归 自己调用自己

```
//递归
// function factorial(num){
//     if(num <= 1){
//         return 1
//     }else{
//         return num * factorial(num-1);
//     }
// }

var anotherFactorial = factorial;//此时anotherFactorial 是保存了原阶乘函数的函数
factorial = null;//把原函数变空
alert(anotherFactorial(3));//出错！原因是新函数保存的是原函数 而原函数里面调用了
factorial 但此时 该函数已经为null 故报错

//解决方法
function factorial(num){
    if(num <= 1){
        return 1
    }else{
        return num * arguments.callee(num-1);
    }
}
```

闭包 有权访问另一个函数作用域中的变量的函数

创建闭包的方法 在一个函数内创建另一个函数

```
//闭包
function createComparisonFunction(propertyName){
    return function(obj1,obj2){
        var val1 = obj1[propertyName];
        var val2 = obj2[propertyName];

        if(val1 < val2){
            return -1;
        }else if(val1 > val2){
            return1
        }else{
            return 0;
        }
    }
}

//作用域链
function compare(val1,val2){
    if(val1 < val2){
        return -1;
    }else if(val1 > val2){
        return1
    }else{
        return 0;
    }
}

var result = compare(5,10);

/*  1 先定义了compare () 函数
    2 在全局作用域中掉调用它 此时会创建一个包含 this arguments val1 val2 的活动对象
      而全局执行环境的变量对象 (this result compare)  在compare () 函数的执行环境的作用域链中处于第二位
    3 这个作用域链会被保存在内部的[[Scope]]属性中
*/
```

闭包与变量

```
//闭包与变量
function createFunctions(){
    var result = new Array();

    for(var i=0;i<10;i++){
        result[i] = function(){
            console.log(i)
            return i;
        }
    }
```

```
    return result;
}

createFunctions();//该函数会返回一个函数数组 表面上看每个函数都会返回自己的下标索引值 但事实上全都是10
//原因是每个函数都保存着createFunctions()函数的活动对象，所有他们的引用都是同一个变量i
当createFunctions函数返回后 i==10

//强制闭包 让行为符合预期
function createFuncitions2(){
    var result = new Array();

    for(var i=0;i<10;i++){
        result[i] = function(num){
            return function(){
                return num
            }
        }(i);//立即函数 将循环中的i传入
    }
    return result;
}
```

关于this对象

- 匿名函数执行环境具有全局性 因此this通常指向window

```
//this对象
var name = 'the window';
//闭包中的this 由于使用了匿名函数 所以this的指向为window
var obj = {
    name:'the obj',
    getName:function() {//该方法返回一个匿名函数
        return function() {
            return this.name;
        }
    }
}
alert(obj.getName()());//the window

var name = 'the window';
//不闭包this就指向对象本身
var obj = {
    name:'the obj',
    getName:function() {
            return this.name;

    }
}
alert(obj.getName());//the obj
```

- 解决闭包中this的指向问题

```
//让闭包访问对象本身 需要一个中间变量来保存this
var name = 'the window';
var obj = {
    name:'the obj',
    getName:function() {//该方法返回一个匿名函数
        var that = this;
        return function() {
            return that.name;
        }
    }
}

alert(obj.getName()());//the obj
```

模仿块级作用域

- JS中没有块级作用域 因此语句块中定义的变量是在函数中而非语句中创建的

```
//模仿块级作用域
function outputNumber(count){
    for(var i=0;i<count;i++){
        alert(i);// 0 1 2
    }//在java中 i 在循环结束后就会被销毁
    //可是在js中 函数内部还是能访问到i
    alert(i);//计数 3
}
outputNumber(3);

//解决方法1 let关键字
function outputNumber2(count){
    for(let i=0;i<count;i++){
        alert(i);// 0 1 2
    }//在java中 i 在循环结束后就会被销毁
    //其实只要把var关键字改成let即可
    alert(i);// i is not defined
}
outputNumber2(3);

//解决方法2 块级作用域（匿名函数立即执行 执行结束后会被销毁）
function outputNumber3(count){
    (function(){
        for(var i=0;i<count;i++){
            alert(i);//0 1 2
        }
    })();
    alert(i);// i is not defined
```

```
}
outputNumber3(3);
```

私有变量

- 严格来讲 js没有私有成员的概念 但有私有变量

静态私有变量

- 通过在私有作用域中定义私有变量或函数，同样也可以创建特权方法（get set之类的）

```
//私有变量
(function(){
    var privateVal = 10;//私有变量

    function privateFunction(){
        return false;
    }//私有函数

    //构造函数
    MyObj = function(){
    }//由于初始化时没声明 这将会是一个全局变量 即在着函数之外是可以访问到的

    //特权方法 定义在原型对象上
    MyObj.prototype.publicMethod = function(){
        privateVal++;
        return privateFunction();
    }
})();

var obj1 = new MyObj();
alert(obj1.publicMethod());//false
alert(obj1.privateVal);//undefined

//例子
(function(){
    var name = '';
    Person = function(value){
        name = value;
    }

    Person.prototype.getName = function(){
        return name;
    }

    Person.prototype.setName = function(value){
        name = value;
    }
})();

var person1 = new Person('jack');
```

```
alert(person1.getName());//jack
person1.setName('rose');
alert(person1.getName());//rose

var person2 = new Person('tony');
alert(person1.getName());//tony 由于name属性是一个静态的共享属性 所以每调用一次就会保存一个新的值
alert(person2.getName());//tony
```

模块模式 单例创建私有变量和特权方法

- 单例：只有一个实例的对象
- 通过一个返回一个对象的匿名函数来实现对外的接口

```
//模块模式
var singleton = function(){
    var privateVal = 10;
    function privateFunction(){
        return false;
    }
    //特权/公有方法和属性
    return {
        publicProperty : true,
        publicMethod : function(){
            privateVal ++ ;
            return privateFunction();
        }
    }
}();


//例子
var application = function(){
    //私有函数与变量
    var components = new Array();
    //初始化
    components.push(new BaseComponent());

    //公有
    return{
        getComponentCount : function(){
            return components.length;
        },
        registerComponent : function(component){
            if(typeof component == 'object'){
                components.push(component);
            }
        }
    }
}();
```

增强的模块模式