闭包

闭包

是指有权访问另一个函数作用域中的变量的函数。

在一个函数内部创建另一个函数，当这个内部函数在包含它的外部函数之外被调用时，就会形成闭包。

当调用一个函数时，会为函数创建一个局部活动对象，包括传入的参数和在函数内部定义的变量。

一般来讲，函数中的局部活动对象只在函数执行过程中存在，函数执行完毕后就会销毁。内存中仅保存全局作用域（全局执行环境的变量对象）。

然而闭包会让函数内部变量的值始终保持在内存中，函数执行完毕后其活动对象不会被销毁，因为在函数内部定义的匿名函数的作用域链仍然在引用这个活动对象。直到匿名函数被销毁后，函数的活动对象才会被销毁。

用处

1.在函数内部模仿块级（私有）作用域

模仿块级（私有）作用域：定义一个匿名函数后立即执行。

因为函数执行完毕后，局部活动对象会被立即销毁，所以在这个匿名函数中定义的变量不能被外部访问，形成一个块级作用域。

语法：

```
(function(){
    //形成块级作用域
})()        //最后一对圆括号表示立即执行这个函数
//因为函数声明后面不能直接跟圆括号，所以给它用一对圆括号括起来，是将函数声明转换成函数表达式
```

在函数内部使用匿名函数立即执行，形成闭包

```
function outputNumbers(count){
//创造了一个块级作用域
    (function(){
        for(var i=0;i<count;i++){  alert(i);  }      //因为是闭包，所以能访问count
    })()

    alert(i);     //错误
}
```

2.能够在函数外部访问函数内部的私有变量和私有函数（特权方法）

(1)在构造函数中定义特权方法

构造函数中定义的方法有权访问内部的私有变量，并且可以在构造函数外部使用。

缺点：对每个实例都会创建同样的新方法。

(2)静态私有变量

在私有作用域中定义私有变量和私有函数，然后再定义一个构造函数及其公有方法。

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

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

```
    //构造函数
    MyObject = function(){
//定义构造函数时并没有使用函数声明，而是使用了函数表达式。函数声明只能创建局部函数，我们需要在私有作用域外面使用构造函数。
//声明MyObject时也没有使用var，这样MyObject就成了一个全局变量，能够在私有作用域之外被访问。
    }

    //特权方法
    MyObject.prototype.publicMethod = function(){
        privateVariable ++;
        return privateFunction();
    }
})();

var object = new MyObject();
console.log(object.publicMethod());//false
```

优点：所有的实例都使用同一个函数。
缺点：属性被所有实例都共享，若被修改，则所有实例的该属性都会改变。

```
(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('Tom');
var person2 = new Person('Lucy');
console.log(person1.getName());//Lucy
console.log(person2.getName());//Lucy
```

3.缓存

当一个函数的处理过程很耗时，我们可以将计算出来的值存储起来，当调用这个函数的时候，首先在缓存中查找，如果找不到，则进行计算，然后更新缓存并返回值，如果找到了，直接返回查找到的值即可。
闭包正是可以做到这一点，因为它不会释放外部的引用，从而函数内部的值可以得以保留。

```
1. var CachedSearchBox = (function(){
2.     var cache = {},
```

```
        count = [];
    return {
        attachSearchBox : function(dsid){
            if(dsid in cache){//如果结果在缓存中
               return cache[dsid];//直接返回缓存中的对象
            }
            var fsb = new uikit.webctrl.SearchBox(dsid);//新建
             cache[dsid] = fsb;//更新缓存
             if(count.length > 100){//保正缓存的大小<=100
                delete cache[count.shift()];
             }
             return fsb;
        },

        clearSearchBox : function(dsid){
            if(dsid in cache){
               cache[dsid].clearSelection();
            }
        }
    };
})();

CachedSearchBox.attachSearchBox("input1");
```

4.为函数引用设置延时

http://demo.jb51.net/js/javascript_bibao/index.htm#clSto

闭包的一个常见用法是在执行函数之前为要执行的函数提供参数。例如：将函数作为 setTimout 函数的第一个参数，这在 Web 浏览器的环境下是非常常见的一种应用。

setTimeout 用于有计划地执行一个函数（或者一串 JavaScript 代码，不是在本例中），要执行的函数是其第一个参数，其第二个参数是以毫秒表示的执行间隔。也就是说，当在一段代码中使用 setTimeout 时，要将一个函数的引用作为它的第一个参数，而将以毫秒表示的时间值作为第二个参数。但是，传递函数引用的同时无法为计划执行的函数提供参数。

然而，可以在代码中调用另外一个函数，由它返回一个对内部函数的引用，再把这个对内部函数对象的引用传递给 setTimeout 函数。执行这个内部函数时要使用的参数在调用返回它的外部函数时传递。这样，setTimeout 在执行这个内部函数时，不用传递参数，但该内部函数仍然能够访问在调用返回它的外部函数时传递的参数。

```
function callLater(paramA, paramB, paramC){
    /* 返回一个由函数表达式创建的匿名内部函数的引用:- */

    return (function(){
        /* 这个内部函数将通过 - setTimeout - 执行，
         而且当它执行时它会读取并按照传递给外部函数的参数行事：
        */
        paramA[paramB] = paramC;
    });
}

...

/* 调用这个函数将返回一个在其执行环境中创建的内部函数对象的引用。
 传递的参数最终将作为外部函数的参数被内部函数使用。
 返回的对内部函数的引用被赋给一个全局变量:-
```

```
*/

var functRef = callLater(elStyle, "display", "none");
/* 调用 setTimeout 函数，将赋给变量 - functRef -
的内部函数的引用作为传递的第一个参数:- */

hideMenu=setTimeout(functRef, 500);
```

缺点

1.内存消耗

由于闭包会携带包含它的函数的作用域，所以会比其他函数占用更多内存。

2.IE内存泄漏

在IE浏览器中，由于IE使用非原生javascript对象实现DOM对象，因此闭包会导致内存泄露问题。

例如：

```
1. function A(){
2.       var a=document.createElement("div"), //
3.             msg="Hello";
4.        a.onclick=function(){
5.           alert(msg);
6.           }
7.    }
8.  A();
```

假设A()执行时创建的作用域对象ScopeA，ScopeA引用了DOM对象a,DOM对象a
引用了function(aleert(msg))，函数function(alert(msg))引用了ScopeA，这是一个循环引用，在IE会导致内存泄露。

3.使用this会导致问题

匿名函数的执行环境具有全局性，因此this对象通常指向window。

解决方法：

把外部作用域中的this保存在一个闭包能够访问到的对象里，就可以让闭包访问该对象了。

4.闭包会在父函数外部，改变父函数内部变量的值。

5.由于返回的函数并没有立刻执行，所以不能引用循环变量。

```
function count() {
    var arr = [];
    for (var i=1; i<=3; i++) {
        arr.push(function () {
            return i * i;
        });
    }
    return arr;
}

var results = count();
var f1 = results[0];
var f2 = results[1];
var f3 = results[2];
```

```
f1(); // 16
f2(); // 16
f3(); // 16
```

修改方法：

创建一个匿名函数并立刻执行

```
(function (x) {
    return x * x;
})(3);

function count() {
    var arr = [];
    for (var i=1; i<=3; i++) {
        arr.push((function (n) {
            return function () {
                return n * n;
            }
        })(i));
    }
    return arr;
}

var results = count();
var f1 = results[0];
var f2 = results[1];
var f3 = results[2];

f1(); // 1
f2(); // 4
f3(); // 9
```