# 第 16 章 匿名函数和闭包

学习要点：

1.匿名函数

2.闭包

主讲教师：李炎恢

合作网站：http://www.ibeifeng.com

讲师博客：http://hi.baidu.com/李炎恢

匿名函数就是没有名字的函数，闭包是可访问一个函数作用域里变量的函数。声明：本节内容需要有面向对象和少量设计模式基础，否则无法听懂.(所需基础 15 章的时候已经声明过了)。

## 一. 匿名函数

```
//普通函数
function box() {                        //函数名是 box
    return 'Lee';
}

//匿名函数
function () {                           //匿名函数，会报错
    return 'Lee';
}

//通过表达式自我执行
(function box() {                       //封装成表达式
    alert('Lee');
})();                                   //()表示执行函数，并且传参

//把匿名函数赋值给变量
var box = function () {                 //将匿名函数赋给变量
    return 'Lee';
};
alert(box());                           //调用方式和函数调用相似

//函数里的匿名函数
function box () {
    return function () {                //函数里的匿名函数，产生闭包
        return 'Lee';
    }
}
alert(box()());                         //调用匿名函数
```

完全等价于function box() {return 'Lee';}，只是这个定义显示的指定了函数的名字，函数的名字其实就是变量

## 二．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，创建闭包的常见的方式，就是在一个函数内部创建另一个函数，通过另一个函数访问这个函数的局部变量。

```
//通过闭包可以返回局部变量
function box() {
    var user = 'Lee';
    return function () {                    //通过匿名函数返回 box()局部变量
        return user;
    };
}
alert(box()());                             //通过 box()()来直接调用匿名函数返回值

var b = box();
alert(b());                                 //另一种调用匿名函数返回值
```

不要使用this，this指代执行函数的作用域

使用闭包有一个优点，也是它的缺点：就是可以把局部变量驻留在内存中，可以避免使用全局变量。(全局变量污染导致应用程序不可预测性，每个模块都可调用必将引来灾难，所以推荐使用私有的，封装的局部变量)。

```
//通过全局变量来累加
var age = 100;                              //全局变量
function box() {
    age ++;                                 //模块级可以调用全局变量，进行累加
}
box();                                      //执行函数，累加了
alert(age);                                 //输出全局变量

//通过局部变量无法实现累加
function box() {
    var age = 100;
    age ++;                                 //累加
    return age;
}

alert(box());                               //101
alert(box());                               //101，无法实现，因为又被初始化了

//通过闭包可以实现局部变量的累加
function box() {
    var age = 100;
    return function () {
        age ++;
        return age;
```

```
        }
    }
    var b = box();                          //获得函数
    alert(b());                             //调用匿名函数
    alert(b());                             //第二次调用匿名函数，实现累加
```

PS：由于闭包里作用域返回的局部变量资源不会被立刻销毁回收，所以可能会占用更多的内存。过度使用闭包会导致性能下降，建议在非常有必要的时候才使用闭包。

作用域链的机制导致一个问题，在循环中里的匿名函数取得的任何变量都是最后一个值。

```
    //循环里包含匿名函数
    function box() {
        var arr = [];

        for (var i = 0; i < 5; i++) {
            arr[i] = function () {
                return i;
            };
        }
        return arr;
    }

    var b = box();                          //得到函数数组
    alert(b.length);                        //得到函数集合长度
    for (var i = 0; i < b.length; i++) {
        alert(b[i]());                      //输出每个函数的值，都是最后一个值
    }
```

上面的例子输出的结果都是 5，也就是循环后得到的最大的 i 值。因为 b[i]调用的是匿名函数，匿名函数并没有自我执行，等到调用的时候，box()已执行完毕，i 早已变成 5，所以最终的结果就是 5 个 5。

```
    //循环里包含匿名函数-改 1，自我执行匿名函数
    function box() {
        var arr = [];

        for (var i = 0; i < 5; i++) {
            arr[i] = (function (num) {          //自我执行
                return num;
            })(i);                              //并且传参
        }
        return arr;
```

```
}

var b = box();
for (var i = 0; i < b.length; i++) {
    alert(b[i]);                                    //这里返回的是数组，直接打印即可
}
```

改 1 中，我们让匿名函数进行自我执行，导致最终返回给 a[i]的是数组而不是函数了。最终导致 b[0]-b[4]中保留了 0,1,2,3,4 的值。

```
//循环里包含匿名函数-改 2，匿名函数下再做个匿名函数
function box() {
    var arr = [];

    for (var i = 0; i < 5; i++) {
        arr[i] = (function (num) {
            return function () {                    //直接返回值，改 2 变成返回函数
                return num;                         //原理和改 1 一样
            }
        })(i);
    }
    return arr;
}

var b = box();
for (var i = 0; i < b.length; i++) {
    alert(b[i]());                                  //这里通过 b[i]()函数调用即可
}
```

改 1 和改 2 中，我们通过匿名函数自我执行，立即把结果赋值给 a[i]。每一个 i，是调用方通过按值传递的，所以最终返回的都是指定的递增的 i。而不是 box()函数里的 i。

关于 this 对象

在闭包中使用 this 对象也可能会导致一些问题，this 对象是在运行时基于函数的执行环境绑定的，如果 this 在全局范围就是 window，如果在对象内部就指向这个对象。而闭包却在运行时指向 window 的，因为闭包并不属于这个对象的属性或方法。

```
var user = 'The Window';

var obj = {
    user : 'The Object',
    getUserFunction : function () {
        return function () {                        //闭包不属于 obj，里面的 this 指向 window
            return this.user;
```

```
            };
        }
    };

    alert(obj.getUserFunction()());                    //The window

    //可以强制指向某个对象
    alert(obj.getUserFunction().call(obj));            //The Object

    //也可以从上一个作用域中得到对象
    getUserFunction : function () {
        var that = this;                               //从对象的方法里得对象
        return function () {
            return that.user;
        };
    }
```

内存泄漏

由于 IE 的 JScript 对象和 DOM 对象使用不同的垃圾收集方式，因此闭包在 IE 中会导致一些问题。就是内存泄漏的问题，也就是无法销毁驻留在内存中的元素。以下代码有两个知识点还没有学习到，一个是 DOM，一个是事件。

```
    function box() {
        var oDiv = document.getElementById('oDiv');    //oDiv 用完之后一直驻留在内存
        oDiv.onclick = function () {
            alert(oDiv.innerHTML);                     //这里用 oDiv 导致内存泄漏
        };
    }
    box();
```

那么在最后应该将 oDiv 解除引用来避免内存泄漏。

```
    function box() {
        var oDiv = document.getElementById('oDiv');
        var text = oDiv.innerHTML;
        oDiv.onclick = function () {
            alert(text);
        };
        oDiv = null;                                   //解除引用
    }
```

PS：如果并没有使用解除引用，那么需要等到浏览器关闭才得以释放。

模仿块级作用域

JavaScript 没有块级作用域的概念。

```
function box(count) {
    for (var i=0; i<count; i++) {}
    alert(i);                                    //i 不会因为离开了 for 块就失效
}
box(2);

function box(count) {
    for (var i=0; i<count; i++) {}
    var i;                                       //就算重新声明，也不会前面的值
    alert(i);
}
box(2);
```

以上两个例子，说明 JavaScript 没有块级语句的作用域，if () {} for () {}等没有作用域，如果有，出了这个范围 i 就应该被销毁了。就算重新声明同一个变量也不会改变它的值。

JavaScript 不会提醒你是否多次声明了同一个变量；遇到这种情况，它只会对后续的声明视而不见(如果初始化了，当然还会执行的)。使用模仿块级作用域可避免这个问题。

```
//模仿块级作用域(私有作用域)
(function () {
    //这里是块级作用域
})();

//使用块级作用域(私有作用域)改写
function box(count) {
    (function () {
        for (var i = 0; i<count; i++) {}
    })();
    alert(i);                                    //报错，无法访问
}
box(2);
```

使用了块级作用域(私有作用域)后，匿名函数中定义的任何变量，都会在执行结束时被销毁。这种技术经常在全局作用域中被用在函数外部，从而限制向全局作用域中添加过多的变量和函数。一般来说，我们都应该尽可能少向全局作用域中添加变量和函数。在大型项目中，多人开发的时候，过多的全局变量和函数很容易导致命名冲突，引起灾难性的后果。如果采用块级作用域(私有作用域)，每个开发者既可以使用自己的变量，又不必担心搞乱全局作用域。

```
(function () {
    var box = [1,2,3,4];
    alert(box);                                  //box 出来就不认识了
})();
```

在全局作用域中使用块级作用域可以减少闭包占用的内存问题，因为没有指向匿名函数的引用。只要函数执行完毕，就可以立即销毁其作用域链了。

私有变量

JavaScript 没有私有属性的概念；所有的对象属性都是公有的。不过，却有一个私有变量的概念。任何在函数中定义的变量，都可以认为是私有变量，因为不能在函数的外部访问这些变量。

```
function box() {
    var age = 100;                          //私有变量，外部无法访问
}
```

而通过函数内部创建一个闭包，那么闭包通过自己的作用域链也可以访问这些变量。而利用这一点，可以创建用于访问私有变量的公有方法。

```
function Box() {
    var age = 100;                          //私有变量
    function run() {                        //私有函数
        return '运行中...';
    }
    this.get = function () {                //对外公共的特权方法
        return age + run();
    };
}

var box = new Box();
alert(box.get());
```

可以通过构造方法传参来访问私有变量。

```
function Person(value) {
    var user = value;                       //这句其实可以省略
    this.getUser = function () {
        return user;
    };
    this.setUser = function (value) {
        user = value;
    };
}
```

但是对象的方法，在多次调用的时候，会多次创建。可以使用静态私有变量来避免这个问题。

静态私有变量

通过块级作用域(私有作用域)中定义私有变量或函数，同样可以创建对外公共的特权方法。

```
(function () {
```

```
    var age = 100;
    function run() {
        return '运行中...';
    }
    Box = function () {};                    //构造方法
    Box.prototype.go = function () {         //原型方法
        return age + run();
    };
})();


var box = new Box();
alert(box.go());
```

上面的对象声明，采用的是 Box = function () {} 而不是 function Box() {} 因为如果用后面这种，就变成私有函数了，无法在全局访问到了，所以使用了前面这种。

```
(function () {
    var user = '';
    Person = function (value) {
        user = value;
    };
    Person.prototype.getUser = function () {
        return user;
    };
    Person.prototype.setUser = function (value) {
        user = value;
    }
})();
```

使用了 prototype 导致方法共享了，而 user 也就变成静态属性了。(所谓静态属性，即共享于不同对象中的属性)。

模块模式

之前采用的都是构造函数的方式来创建私有变量和特权方法。那么对象字面量方式就采用模块模式来创建。

```
var box = {                                  //字面量对象，也是单例对象
    age : 100,                               //这是公有属性，将要改成私有
    run : function () {                      //这时公有函数，将要改成私有
        return '运行中...';
    };
};
```

私有化变量和函数：

```
var box = function () {
    var age = 100;
    function run() {
        return '运行中...';
    }
    return {                                    //直接返回对象
        go : function () {
            return age + run();
        }
    };
}();
```

上面的直接返回对象的例子，也可以这么写：

```
var box = function () {
    var age = 100;
    function run() {
        return '运行中...';
    }
    var obj =   {                               //创建字面量对象
        go : function () {
            return age + run();
        }
    };
    return obj;                                 //返回这个对象
}();
```

字面量的对象声明，其实在设计模式中可以看作是一种单例模式，所谓单例模式，就是永远保持对象的一个实例。

增强的模块模式，这种模式适合返回自定义对象，也就是构造函数。

```
function Desk() {};
var box = function () {
    var age = 100;
    function run() {
        return '运行中...';
    }
    var desk = new Desk();                      //可以实例化特定的对象
    desk.go = function () {
        return age + run();
    };
    return desk;
}();
alert(box.go());
```

# 感谢收看本次教程！

本课程是由北风网(ibeifeng.com)

瓢城 Web 俱乐部(yc60.com)联合提供：

本次主讲老师：李炎恢

我的博客：hi.baidu.com/李炎恢/

我的邮件：yc60.com@gmail.com