

JavaScript 闭包究竟是什么

<http://www.cnblogs.com/dolphinX/archive/2012/09/29/2708763.html>

用JavaScript一年多了，闭包总是让人二丈和尚摸不着头脑。陆陆续续接触了一些闭包的知识，也犯过几次因为不理解闭包导致的错误，一年多了资料也看了一些，但还是不是非常明白，最近偶然看了一下 jQuery基础教程 的附录，发现附录A对JavaScript的闭包的介绍简单易懂，于是借花献佛总结一下。

1.简单的例子

首先从一个经典错误谈起，页面上有若干个div，我们想给它们绑定一个onclick方法，于是有了下面的代码

```
<div id="divTest">
  <span>0</span> <span>1</span> <span>2</span> <span>3</span>
</div>
<div id="divTest2">
  <span>0</span> <span>1</span> <span>2</span> <span>3</span>
</div>
```

```
$(document).ready(function() {
  var spans = $("#divTest span");
  for (var i = 0; i < spans.length; i++) {
    spans[i].onclick = function() {
      alert(i);
    }
  }
});
```

很简单的功能可是却偏偏出错了，每次alert出的值都是4，简单的修改就好使了

```
var spans2 = $("#divTest2 span");
$(document).ready(function() {
  for (var i = 0; i < spans2.length; i++) {
    (function(num) {
      spans2[i].onclick = function() {
        alert(num);
      }
    })(i);
  }
});
```


2.内部函数

复制代码


让我们从一些基础的知识谈起,首先了解一下内部函数。内部函数就是定义在另一个函数中的函数。例如：


```
function outerFn () {  
    function innerFn () {}  
}
```

innerFn就是一个被包在outerFn作用域中的内部函数。这意味着，在outerFn内部调用innerFn是有效的，而在outerFn外部调用innerFn则是无效的。下面代码会导致一个JavaScript错误：

 `function outerFn() {`
复制代码 `document.write("Outer function
");`
 `function innerFn() {`
 `document.write("Inner function
");`
 `}`
`}`
`innerFn();`

 不过在outerFn内部调用innerFn，则可以成功运行：


复制代码  `function outerFn() {`
复制代码 `document.write("Outer function
");`
 `function innerFn() {`
 `document.write("Inner function
");`
 `}`
 `innerFn();`
`}`
`outerFn();`


复制代码

2.1伟大的逃脱

JavaScript允许开发人员像传递任何类型的数据一样传递函数，也就是说，JavaScript中的内部函数能够逃脱定义他们的外部函数。

逃脱的方式有很多种，例如可以将内部函数指定给一个全局变量：

 `var globalVar;`
复制代码 `function outerFn() {`
 `document.write("Outer function
");`
 `function innerFn() {`
 `document.write("Inner function
");`
 `}`
 `globalVar = innerFn;`
`}`

```
outerFn();  
globalVar();
```



复制代码

调用outerFn时会修改全局变量globalVar，这时候它的引用变为innerFn，此后调用globalVar和调用innerFn一样。这时在outerFn外部直接调用innerFn仍然会导致错误，这是因为内部函数虽然通过把引用保存在全局变量中实现了逃脱，但这个函数的名字依然只存在于outerFn的作用域中。

也可以通过在父函数的返回值来获得内部函数引用



```
function outerFn() {
```

复制代码

```
    document.write("Outer function<br/>");
```

```
    function innerFn() {
```

```
        document.write("Inner function<br/>");
```

```
    }
```

```
    return innerFn;
```

```
}
```

```
var fnRef = outerFn();
```

```
fnRef();
```



复制代码

这里并没有在outerFn内部修改全局变量，而是从outerFn中返回了一个对innerFn的引用。通过调用outerFn能够获得这个引用，而且这个引用可以保存在变量中。

这种即使离开函数作用域的情况下仍然能够通过引用调用内部函数的事实，意味着**只要存在调用内部函数的可能，JavaScript就需要保留被引用的函数。而且JavaScript运行时需要跟踪引用这个内部函数的所有变量，直到最后一个变量废弃，JavaScript的垃圾收集器才能释放相应的内存空间（红色部分是理解闭包的关键）。**

说了半天总算和闭包有关系了，**闭包是指有权限访问另一个函数作用域的变量的函数**，创建闭包的常见方式就是在一个函数内部创建另一个函数，就是我们上面说的内部函数，所以刚才说的不是废话，也是闭包相关的 ^_^

1.2变量的作用域

内部函数也可以有自己的变量，这些变量都被限制在内部函数的作用域中：



```
function outerFn() {
```

复制代码

```
    document.write("Outer function<br/>");
```

```
    function innerFn() {
```

```
        var innerVar = 0;
```

```
        innerVar++;
```

```
        document.write("Inner function\t");
```

```
document.write("innerVar = "+innerVar+"<br/>");
```

```
}
```

```
return innerFn;
```

```
}
```

```
var fnRef = outerFn();
```

```
fnRef();
```

```
fnRef();
```

```
var fnRef2 = outerFn();
```

```
fnRef2();
```

```
fnRef2();
```



复制代码

每当通过引用或其它方式调用这个内部函数时，就会创建一个新的innerVar变量，然后加1，最后显示



Outer function

复制代码

Inner function innerVar = 1

Inner function innerVar = 1

Outer function

Inner function innerVar = 1

Inner function innerVar = 1



复制代码

内部函数也可以像其他函数一样引用全局变量：



```
var globalVar = 0;
```

复制代码

```
function outerFn() {
```

```
document.write("Outer function<br/>");
```

```
function innerFn() {
```

```
globalVar++;
```

```
document.write("Inner function\t");
```

```
document.write("globalVar = " + globalVar + "<br/>");
```

```
}
```

```
return innerFn;
```

```
}
```

```
var fnRef = outerFn();
```

```
fnRef();
```

```
fnRef();
```

```
var fnRef2 = outerFn();
```

```
fnRef2();
```

```
fnRef2();
```



复制代码

现在每次调用内部函数都会持续地递增这个全局变量的值：



复制代码

```
Outer function
Inner function    globalVar = 1
Inner function    globalVar = 2
Outer function
Inner function    globalVar = 3
Inner function    globalVar = 4
```



复制代码

但是如果这个变量是父函数的局部变量又会怎样呢？因为内部函数会引用到父函数的作用域（有兴趣可以了解一下作用域链和活动对象的知识），内部函数也可以引用到这些变量



```
function outerFn() {
    var outerVar = 0;

    document.write("Outer function<br/>");

    function innerFn() {
        outerVar++;
        document.write("Inner function\t");
        document.write("outerVar = " + outerVar + "<br/>");
    }

    return innerFn;
}

var fnRef = outerFn();
fnRef();
fnRef();

var fnRef2 = outerFn();
fnRef2();
fnRef2();
```



复制代码

这一次结果非常有意思，也许或出乎我们的意料



```
Outer function
复制代码Inner function    outerVar = 1
Inner function    outerVar = 2
Outer function
Inner function    outerVar = 1
Inner function    outerVar = 2
```



我们看到的是前面两种情况合成的效果，通过每个引用调用innerFn都会独立的递增outerVar。也就是说第二次调用outerFn没有继续沿用outerVar的值，而是在第二次函数调用的作用域创建并绑定了一个一个新的outerVar实例，两个计数器完全无关。

当内部函数在定义它的作用域的外部被引用时，就创建了该内部函数的一个闭包。这种情况下我们称既不是内部函数局部变量，也不是其参数的变量为自由变量，称外部函数的调用环

境为封闭闭包的环境。从本质上讲，如果内部函数引用了位于外部函数中的变量，相当于授权该变量能够被延迟使用。因此，当外部函数调用完成后，这些变量的内存不会被释放（最后的值会保存），闭包仍然需要使用它们。

3.闭包之间的交互

当存在多个内部函数时，很可能出现意料之外的闭包。我们定义一个递增函数，这个函数的增量为2

```
function outerFn() {
    var outerVar = 0;

    document.write("Outer function<br/>");

    function innerFn1() {
        outerVar++;
        document.write("Inner function 1\t");
        document.write("outerVar = " + outerVar + "<br/>");
    }

    function innerFn2() {
        outerVar += 2;
        document.write("Inner function 2\t");
        document.write("outerVar = " + outerVar + "<br/>");
    }

    return { "fn1": innerFn1, "fn2": innerFn2 };
}

var fnRef = outerFn();
fnRef.fn1();
fnRef.fn2();
fnRef.fn1();

var fnRef2 = outerFn();
fnRef2.fn1();
fnRef2.fn2();
fnRef2.fn1();
```

我们映射返回两个内部函数的引用，可以通过返回的引用调用任一个内部函数，结果：

```
Outer function
Inner function 1    outerVar = 1
Inner function 2    outerVar = 3
Inner function 1    outerVar = 4
Outer function
Inner function 1    outerVar = 1
```

```
Inner function 2    outerVar = 3
Inner function 1    outerVar = 4
```



复制代码

innerFn1和innerFn2引用了同一个局部变量，因此他们共享一个封闭环境。当innerFn1为outerVar递增一时，久违innerFn2设置了outerVar的新的起点值，反之亦然。我们也看到**对outerFn的后续调用还会创建这些闭包的新实例，同时也会创建新的封闭环境，本质上是创建了一个新对象，自由变量就是这个对象的实例变量，而闭包就是这个对象的实例方法**，而且这些变量也是私有的，因为不能在封装它们的作用域外部直接引用这些变量，从而确保了了面向对象数据的专有性。

3.解惑

现在我们可以回头看看开头写的例子就很容易明白为什么第一种写法每次都会alert 4了。

```
for (var i = 0; i < spans.length; i++) {
    spans[i].onclick = function() {
        alert(i);
    }
}
```

上面代码在页面加载后就会执行，当i的值为4的时候，判断条件不成立，for循环执行完毕，但是因为每个span的onclick方法这时候为内部函数，所以i被闭包引用，内存不能被销毁，i的值会一直保持4，直到程序改变它或者所有的onclick函数销毁（主动把函数赋为null或者页面卸载）时才会被回收。这样每次我们点击span的时候，onclick函数会查找i的值（作用域链是引用方式），一查等于4，然后就alert给我们了。而第二种方式是使用了一个立即执行的函数又创建了一层闭包，函数声明放在括号内就变成了表达式，后面再加上括号括号就是调用了，这时候把i当参数传入，函数立即执行，num保存每次i的值。

这一通下来想必大家也和我一样，对闭包有所了解了吧，当然完全了解的话需要把函数的执行环境和作用域链搞清楚 ^_^