[理解JS闭包（closure）的常见场景](http://blog.csdn.net/yanghua_kobe/article/details/6780181)

### 前言：

作用域的好处是内部函数可以访问定义它们的外部函数的参数和变量（除了this和argument，因为内部函数被调用时的上下文环境产生了变化）。

我是这样认为的，闭包的本质利用了作用域和变量的生命周期，从而达到保护想要公用的局部变量，避免全局污染。

很有趣的是的情形是：闭包情形下，内部函数拥有比它的外部函数更长的生命周期。

### 正文：

在清晰地理解了闭包的特性之后，我们在处理代码中的某些场景时：需要公用的变量，但是由于仅供每个模块使用，不想污染全局变量的同时又想保持模块的良好封装性（功能模块的封装可以使我们在代码移植的时候不致于漏掉关键的外部变量）如果感到捉襟见肘，这个时候就可以考虑运用到它。

##### 场景一：使用闭包和函数来构造模块

模块是一个提供接口却隐藏状态与实现的函数或对象。通过使用函数以及闭包来生产模块，我们几乎可以完全摒弃全局变量的使用，从而缓解这个Javascript的最为糟糕的特性之一所带来的影响。

模块模式的一般形式是：一个定义了私有变量和函数的函数；利用闭包创建可以访问私有变量和函数的特权函数；最后返回这个特权函数，或者把他们保存到一个可访问到的地方。

使用模块模式就可以摒弃全局变量的使用，他促进了信息隐藏和其他优秀的设计实践。对于应用程序的封装，或者构造其他单例对象（下一篇博客中来阐述我的理解），模块模式非常有效。

**var** APP={  
 keywordPop:**function**(node){  
 console.log(node.value)  
 */\*\* 此处可以调用节点的value去ajax  
 \* 保证了每一次传过来的value不重复  
 \* 重复的同参ajax浪费服务器资源且没有意义  
 \* \*/* },  
 keywordClick:**function**(){  
 **var** clickFunc=(**function**(){//利用闭包保护上一次的value  
 **var** saveValue="";//用于保存上一次的value  
 **return** (**function**(e){  
 **if** (event.keyCode == 13) {//回车键  
 //doSomething  
 **return false**;  
 }**else**{  
 **if**(saveValue!=e.target.value){//判断与闭包的inpValue是否一致  
 saveValue= e.target.value;//更换闭包的保护常量  
 APP.keywordPop(e.target);  
 }  
 }  
 })  
 }());  
 document.getElementById("Input").onclick=clickFunc;  
 }  
};

##### 场景二：封装相关的功能集

闭包可以创建额外的scope，这可以被用来组合相关的或有依赖性的代码。用这种方式可以最大限度地减少代码干扰的危害。将需要使用的变量提升为全局变量，让它不需要被再次创建也能够再次使用可以解决一部分问题。但结果并不是想的那么简单，一个全局变量关联使用的函数，那将会有第二个全局属性(函数本身也是window对象的属性)关联这个变量，这将让代码失去一定的可控性。因为如果将它使用在其他地方。这段代码的创建者不得不记住包含函数的定义以及变量的定义逻辑。它也使得代码不那么容易与其他代码整合，因为将从原来只需要确定函数名是否在全局命名空间中唯一，变成有必要确定和该函数关联的数组的名称是否在全局命名空间中保持唯一。

一个闭包可以让缓冲数组关联(干净地包含)它依赖的函数，并且同时保持缓冲数组的属性名称，像被分配在全局空间中一样，同时能够避免名称冲突以及代码交互干扰的危险。

接一下的代码创建了一个函数，将返回一个HTML字符串，其中的一部分是不变的，但那些不变的字符串需要被穿插进作为参数传递进来的变量中。

/\*   
 定义一个全局变量：getImgInPositionedDivHtml   
 被赋予对外部函数表达式一次调用返回的一个内部函数表达式   
 内部函数返回了一个HTML字符串，代表一个绝对定位的DIV   
 包裹这一个IMG元素，而所有的变量值都被作为函数调用的参数   
 \*/  
**var** getImgInPositionedDivHtml = (**function** () {  
 /\* buffAr 数组被定义在外部函数表达式中，作为一个局部变量   
 它只被创建一次。数组的唯一实例对内部函数是可见的，   
 所以它可以被用于每一次的内部函数执行   
 空字符串仅仅被用来作为一个占位符，它将被内部函数的参数代替 \*/  
 **var** buffAr = [  
 '<div id="',  
 '', //index 1, DIV ID attribute   
 '" style="position:absolute;top:',  
 '', //index 3, DIV top position   
 'px;left:',  
 '', //index 5, DIV left position   
 'px;width:',  
 '', //index 7, DIV width   
 'px;height:',  
 '', //index 9, DIV height   
 'px;overflow:hidden;\"><img src=\"',  
 '', //index 11, IMG URL   
 '\" width=\"',  
 '', //index 13, IMG width   
 '\" height=\"',  
 '', //index 15, IMG height   
 '\" alt=\"',  
 '', //index 17, IMG alt text   
 '\"><\/div>'  
 ];  
 // 返回一个内部函数对象，他是函数表达式执行返回的结果   
 **return** (**function** (url, id, width, height, top, left, altText) {  
 buffAr[1] = id;  
 buffAr[3] = top;  
 buffAr[5] = left;  
 buffAr[13] = (buffAr[7] = width);  
 buffAr[15] = (buffAr[9] = height);  
 buffAr[11] = url;  
 buffAr[17] = altText;  
 **return** buffAr.join('');  
 });  
})();

一个内联执行的函数表达式返回了内部函数对象的一个引用。并且分配了一个全局变量，让它可以被作为一个全局函数来调用。而缓冲数组作为一个局部变量被定义在外部函数表达式中。它没有被扩展到全局命名空间中，并且无论函数什么时候使用它都不需要被再次创建。

##### 场景三：封装函数

**function**(func, wait, immediate) {  
  
 **var** timeout, args, context, timestamp, result;  
  
 **var** later = **function**() {  
 **var** last = **new** Date().getTime() - timestamp;  
  
 **if** (last < wait && last >= 0) {  
 timeout = setTimeout(later, wait - last);  
 } **else** {  
 timeout = **null**;  
 **if** (!immediate) {  
 result = func.apply(context, args);  
 **if** (!timeout) context = args = **null**;  
 }  
 }  
 };  
 **return function**() {  
 context = **this**;  
 args = arguments;  
 timestamp = **new** Date().getTime();  
 **var** callNow = immediate && !timeout;  
 **if** (!timeout) timeout = setTimeout(later, wait);  
 **if** (callNow) {  
 result = func.apply(context, args);  
 context = args = **null**;  
 }  
 **return** result;  
 };  
};

该段代码封装了延时调用（防反跳）函数，首先函数内部定了一些系列变量和later函数，它们在外部无法调用。每一次调用返回函数的时候，新建了当前时间的时间戳，保存在外部函数的闭包中，通过参数判断是否调用外部函数中的later函数。