**[js apply/call/caller/callee/bind使用方法与区别分析](http://www.cnblogs.com/yuzhongwusan/archive/2012/02/13/2348782.html)**

**一、call 方法**调用一个对象的一个方法，以另一个对象替换当前对象(其实就是更改对象的内部指针，即改变对象的this指向的内容)。   
Js代码   
call([thisObj[,arg1[, arg2[, [,.argN]]]]])   
参数   
thisObj   
可选项。将被用作当前对象的对象。   
arg1, arg2, , argN   
可选项。将被传递方法参数序列。   
说明   
call 方法可以用来代替另一个对象调用一个方法。call 方法可将一个函数的对象上下文从初始的上下文改变为由 thisObj 指定的新对象。如果没有提供 thisObj 参数，那么 Global 对象被用作 thisObj。   
Js代码

复制代码 代码如下:

<input type="text" id="myText" value="input text"> Code   
function Obj(){this.value="对象！";}   
var value="global 变量";   
function Fun1(){alert(this.value);}   
window.Fun1(); //global 变量   
Fun1.call(window); //global 变量   
Fun1.call(document.getElementById('myText')); //input text   
Fun1.call(new Obj()); //对象！

Js代码   
Code

复制代码 代码如下:

var first\_object = {   
num: 42   
};   
var second\_object = {   
num: 24   
};   
function multiply(mult) {   
return this.num \* mult;   
}   
multiply.call(first\_object, 5); // returns 42 \* 5   
multiply.call(second\_object, 5); // returns 24 \* 5

**二、apply方法**   
apply方法的第一个参数也是要传入给当前对象的对象，即函数内部的this。后面的参数都是传递给当前对象的参数。   
对于apply和call两者在作用上是相同的，但两者在参数上有区别的。对于第一个参数意义都一样，但对第二个参数：apply传入的是一个参数数组，也就是将多个参数组合成为一个数组传入，而call则作为call的参数传入（从第二个参数开始）。   
如 func.call(func1,var1,var2,var3)对应的apply写法为：func.apply(func1, [var1,var2,var3])同时使用apply的好处是可以直接将当前函数的arguments对象作为apply的第二个参数传入。   
Js代码

复制代码 代码如下:

var func=new function(){this.a="func"}   
var myfunc=function(x,y){   
var a="myfunc";   
alert(this.a);   
alert(x + y);   
}   
myfunc.call(func,"var"," fun");// "func" "var fun"   
myfunc.apply(func,["var"," fun"]);// "func" "var fun"

**三、caller 属性**返回一个对函数的引用，即调用了当前函数的函数体。   
functionName.caller :functionName 对象是所执行函数的名称。   
说明:   
对 于函数来说，caller 属性只有在函数执行时才有定义。 如果函数是由 JScript 程序的顶层调用的，那么 caller 包含的就是 null 。如果在字符串上下文中使用 caller 属性，那么结果和 functionName.toString 一样，也就是说，显示的是函数的反编译文本。   
Js代码

复制代码 代码如下:

function CallLevel(){   
if (CallLevel.caller == null)   
alert("CallLevel was called from the top level.");   
else   
alert("CallLevel was called by another function:\n"+CallLevel.caller);   
}   
function funCaller(){   
CallLevel();   
}   
CallLevel();   
funCaller()

**四、callee属性**返回正被执行的 Function 对象，也就是所指定的 Function 对象的正文。   
[function.]arguments.callee:可选项 function 参数是当前正在执行的 Function 对象的名称。   
说明:   
callee 属性的初始值就是正被执行的 Function 对象。   
callee 属性是 arguments 对象的一个成员，它表示对函数对象本身的引用，这有利于匿   
函数的递归或者保证函数的封装性，例如下边示例的递归计算1到n的自然数之和。而该属性   
仅当相关函数正在执行时才可用。还有需要注意的是callee拥有length属性，这个属性有时   
用于验证还是比较好的。arguments.length是实参长度，arguments.callee.length是   
形参长度，由此可以判断调用时形参长度是否和实参长度一致。   
Js代码

复制代码 代码如下:

//callee可以打印其本身   
function calleeDemo() {   
alert(arguments.callee);   
}   
//用于验证参数   
function calleeLengthDemo(arg1, arg2) {   
if (arguments.length==arguments.callee.length) {   
window.alert("验证形参和实参长度正确！");   
return;   
} else {   
alert("实参长度：" +arguments.length);   
alert("形参长度： " +arguments.callee.length);   
}   
}   
//递归计算   
var sum = function(n){   
if (n <= 0)   
return 1;   
else   
return n ＋arguments.callee(n - 1)   
}

**五、bind**   
Js代码

复制代码 代码如下:

var first\_object = {   
num: 42   
};   
var second\_object = {   
num: 24   
};   
function multiply(mult) {   
return this.num \* mult;   
}   
Function.prototype.bind = function(obj) {   
var method = this,   
temp = function() {   
return method.apply(obj, arguments);   
};   
return temp;   
}   
var first\_multiply = multiply.bind(first\_object);   
first\_multiply(5); // returns 42 \* 5   
var second\_multiply = multiply.bind(second\_object);   
second\_multiply(5); // returns 24 \* 5

**六、JS闭包(Closure)**   
所谓“闭包”，指的是一个拥有许多变量和绑定了这些变量的环境的表达式（通常是一个函数），因而这些变量也是该表达式的一部分。   
关 于闭包，最简单的描述就是 ECMAScript 允许使用内部函数－－即函数定义和函数表达式位于另一个函数的函数体内。而且，这些内部函数可以访问它们所在的外部函数中声明的所有局部变量、参数和声明 的其他内部函数。当其中一个这样的内部函数在包含它们的外部函数之外被调用时，就会形成闭包。也就是说，内部函数会在外部函数返回后被执行。而当这个内部 函数执行时，它仍然必需访问其外部函数的局部变量、参数以及其他内部函数。这些局部变量、参数和函数声明（最初时）的值是外部函数返回时的值，但也会受到 内部函数的影响。   
简而言之，闭包的作用就是在out function执行完并返回后，闭包使得Javascript的垃圾回收机制GC不会收回out function所占用的资源，因为out function的内部函数inner function的执行需要依赖out function中的变量。   
闭包的两个特点：   
1、作为一个函数变量的一个引用 - 当函数返回时，其处于激活状态。   
2、一个闭包就是当一个函数返回时，一个没有释放资源的栈区。   
例1：   
Html代码

复制代码 代码如下:

<script type="text/javascript">   
function setupSomeGlobals() {   
// Local variable that ends up within closure   
var num = 666;   
// Store some references to functions as global variables   
gAlertNumber = function() { alert(num); }   
gIncreaseNumber = function() { num++; }   
gSetNumber = function(x) { num = x; }   
}   
</script>   
<button onclick="setupSomeGlobals()">生成 - setupSomeGlobals()</button>   
<button onclick="gAlertNumber()">输出值 - gAlertNumber()</button>   
<button onclick="gIncreaseNumber()">增加 - gIncreaseNumber()</button>   
<button onclick="gSetNumber(5)">赋值5 - gSetNumber(5)</button>

例2：   
Html代码

复制代码 代码如下:

<script type="text/javascript">   
function newClosure(someNum, someRef) {   
// Local variables that end up within closure   
var num = someNum;   
var anArray = [1,2,3];   
var ref = someRef;   
return function(x) {   
num += x;   
anArray.push(num);   
alert('num: ' + num +   
' nanArray ' + anArray.toString() +   
' nref.someVar ' + ref.someVar);   
}   
}   
var closure1 = newClosure(40, {someVar:' never-online'})   
var closure2 = newClosure(99, {someVar:' BlueDestiny'})   
closure1(4)   
closure2(3)   
</script>

例3：   
Js代码

复制代码 代码如下:

<script language="javascript">   
/\* 声明一个全局变量 - getImgInPositionedDivHtml - 并将一次调用一个外部函数表达式返回的内部函数赋给它。   
这个内部函数会返回一个用于表示绝对定位的 DIV 元素包围着一个 IMG 元素 的 HTML 字符串，这样一来，   
所有可变的属性值都由调用该函数时的参数提供：   
\*/   
var getImgInPositionedDivHtml = (function(){   
/\* 外部函数表达式的局部变量 - buffAr - 保存着缓冲数组。这个数组只会被创建一次，生成的数组实例对内部函数而言永远是可用的   
因此，可供每次调用这个内部函数时使用。   
其中的空字符串用作数据占位符，相应的数据   
将由内部函数插入到这个数组中：   
\*/   
var buffAr = [   
'<div id="',   
'', //index 1, DIV ID 属性   
'" style="position:absolute;top:',   
'', //index 3, DIV 顶部位置   
'px;left:',   
'', //index 5, DIV 左端位置   
'px;width:',   
'', //index 7, DIV 宽度   
'px;height:',   
'', //index 9, DIV 高度   
'px;overflow:hidden;\"><img src=\"',   
'', //index 11, IMG URL   
'\" width=\"',   
'', //index 13, IMG 宽度   
'\" height=\"',   
'', //index 15, IMG 调蓄   
'\" alt=\"',   
'', //index 17, IMG alt 文本内容   
'\"><\/div>'   
];   
/\* 返回作为对函数表达式求值后结果的内部函数对象。   
这个内部函数就是每次调用执行的函数   
- getImgInPositionedDivHtml( ... ) -   
\*/   
return (function(url, id, width, height, top, left, altText){   
/\* 将不同的参数插入到缓冲数组相应的位置：   
\*/   
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;   
/\* 返回通过使用空字符串（相当于将数组元素连接起来）   
连接数组每个元素后形成的字符串：   
\*/   
return buffAr.join('');   
}); //:内部函数表达式结束。   
})();//自调用   
alert(getImgInPositionedDivHtml);//显示返回的函数   
alert(getImgInPositionedDivHtml("img.gif","img",100,50,0,0,"Test"));   
</script>

说 明：其中的关键技巧在于通过执行一个单行（in-line）函数表达式创建一个额外的执行环境，而将该函数表达式返回的内部函数作为在外部代码中使用的函 数。此时，缓冲数组被定义为函数表达式的一个局部变量。这个函数表达式只需执行一次，而数组也只需创建一次，就可以供依赖它的函数重复使用。   
七、原型链   
ECMAScript 为 Object 类型定义了一个内部 [[prototype]] 属性。这个属性不能通过脚本直接访问，但在属性访问器解析过程中，则需要用到这个内部[[prototype]] 属性所引用的对象链－－即原型链。可以通过一个公共的 prototype 属性，来对与内部的 [[prototype]] 属性对应的原型对象进行赋值或定义。   
例1：   
Js代码

复制代码 代码如下:

<script language="javascript">   
function NumObject(formalParameter){   
this.testNumber = formalParameter;   
}   
function StrObject(formalParameter){   
this.testString = formalParameter;   
}   
//用 NumObject类的实例替换了所有与 StrObject类的实例相关联的原型。   
StrObject.prototype =new NumObject(6);   
var objRef = new StrObject( "String\_Value" );   
//当某个属性访问器尝试读取由 objectRef 所引用的对象的属性值时，整个原型链都会被搜索。   
//不论是在对象或对象的原型中，读取命名属性值的时候只返回首先找到的属性值。而当为对象的命名属性赋值时，如果对象自身不存在该属性则创建相应的属性。   
alert(objRef.testString);//output "String\_Value"   
alert(objRef.testNumber);//output "6"   
alert(objRef.toString);   
//StrObject 的实例拥有一个原型链。该链中的第一个对象是在创建后被指定给 StrObject 构造函数的 prototype 属性的 NumObject 的一个实例。NumObject 的实例也有一个原型，即与 Object.prototype 所引用的对象对应的默认的 Object 对象的原型。最后， Object.prototype 有一个值为 null 的原型，因此这条原型链到此结束。   
objRef.testNumber = 3;//对象自身不存在该属性则创建相应的属性   
alert(objRef.testNumber);//自身有了属性，属性访问器不会再进一步搜索原型链   
alert(NumObject.prototype.isPrototypeOf(objRef));// output "true"   
</script>