**万物皆对象**

在JavaScript的世界，万物皆对象。除了null和undefined，其他基本类型数字，字符串和布尔值都有对应有包装对象。对象的一个特征是你可以在它身上直接调用方法。

对于数字基本类型，当试图在其身上调用toString方法会失败，但用括号括起来后再调用就不会失败了，内部实现是用相应的包装对象将基本类型转为对象。所以(1).toString()相当于new Number(1).toString()。因此，你的确可以把基本类型数字，字符串，布尔等当对象使用的，只是注意语法要得体。

同时我们注意到，JavaScript中数字是不分浮点和整形的，所有数字其实均是浮点类型，只是把小数点省略了而以，比如你看到的1可以写成1.，这也就是为什么当你试图1.toString()时会报错，所以正确的写法应该是这样：1..toString()，或者如上面所述加上括号，这里括号的作用是纠正JS解析器，不要把1后面的点当成小数点。内部实现如上面所述，是将1.用包装对象转成对象再调用方法。

**闭包**（函数里的函数）

①有权访问另一个函数作用域中的变量的函数。

②能够读取其他函数内部变量的函数,（定义在一个函数内部的函数）

③「函数」和「函数内部能访问到的变量」（也叫环境）的总和。

创建闭包的常见方式： 在一个函数内部创建另一个函数，并将函数通过return对象返回。

作用：

1.可以读取函数内部的变量

2.让这些变量的值始终保存在内存中（延长外部函数局部变量的生命周期）

例一：(使用闭包读取函数内部的变量)

function f1 () {  
var n=99

function f2 () {

alert(n)

}

return f2

}

var result = f1() ( f1()返回的是f2()这个函数 )

result() //99 f1()()

函数f2就被包括在函数f1内部，这时f1内部的所有局部变量，对f2都是可见的。

但是反过来就不行，f2内部的局部变量，对f1就是不 可见的。

这就是Javascript语言特有的"链式作用域"结构（chain scope），子对象会一级一级地向上寻找所有父对象的变量。所以，父对象的所有变量，对子对象都是可见的，反之则不成立。

既然f2可以读取f1中的局部 变量，那么只要把f2作为返回值，我们不就可以在f1外部读取它的内部变量了吗？

js的作用域链：函数可以嵌套，多个函数的层层嵌套构成了多个作用域的层层嵌套，

js作用域链的变量访问规则是：如果当前作用域内存在要访问的变量，则使用当前作用域的变量，否则到上一层作用域内寻找，直到全局作用域，如果找不到，则该变量为未声明。

function f1(){

var a=1

function f2(){

alert(a) //undefined

var a =2

alert(a) //2

}

f2()

}

f1()

注意，变量的声明在代码解析期完成，如果当前作用域的变量的声明和赋值语句写在变量访问语句后面，js函数会认为当前作用域已经存在要访问的变量不再向上级作用域查找，但是，由于变量的赋值发生的代码运行期，访问的到变量将是undefined.

变量的声明提前：

①使用var关键字声明的变量，会在所有的代码执行之前被声明（但是不会赋值）

②在函数中，如果不使用var关键字，变量不会被声明提前，都会成为全局变量（外部函数已经有同一个全局变量，内部函数的变量覆盖外部函数的全局变量，但是还没赋值）

function f1(){

var a=1

function f2(){

alert(a) //1

a =2 //不会声明提前

alert(a) //2

}

f2()

alert(a) //2

}

f1()

例三：变量值始终保存在内存中

function f1(){

var n =99

add =function () { return n=(n+1) } //匿名函数

function f2 (){

console.log(n)

}

return f2

}

var result = f1()

result() //99

add() //100 ( add()不加return 返回的是undefined，但实际运行了)

result() //100

在这段代码中，result实际上就是闭包f2函数。它一共运行了两次，第一次的值是99，第二次的值是100。这证明了，函数f1中的局部变量n一 直保存在内存中，并没有在f1调用后被自动清除。因为f1是f2的父函数，而f2被赋给了一个全局变量，这导致f2始终在内存中，而f2的存在依赖于 f1，因此f1也始终在内存中，不会在调用结束后，被[垃圾回收机制（garbage collection）](https://link.jianshu.com?t=http://eatpockyboy.blog.163.com/blog/static/1167346402011321423929/" \t "_blank)回收。这段代码中另一个值得注意的地方，就是"add=function(){n+=1}"这一行，首先在add前面没有使用var关键字，

因此add是一个全局变量，而不是局部变量。其次，add的值是一个[匿名函数（anonymous function）](https://link.jianshu.com?t=http://www.itxueyuan.org/view/6314.html" \t "_blank)，而这个匿名函数本身也是一个闭包，所以add相当于是一个setter，可以在函数外部对函数内部的局部变量进行操作。

常见问题： 5个 input

var btn =document.getElementByTagName(‘input’)

for( var i=0; i<btn.length; i++){

btn[i].onclick=function(){

alert(i)

}

}

问题： 冒泡事件，直接弹出最后一个数

因为for循环只是给input绑定事件，但是里面的函数代码并不会执行，

alert(i)是在你点击的时候才执行的,但是此时的i已经是5了，所以打印出5来了。

解决办法：

①通过闭包（闭包可以创建独立的环境，保存变量）

for( var i=0; i<btn.length; i++){

btn[i].onclick=a(i)

function a(i){

return function(){ alert(i) }

}

}

此时a里面的i用的是闭包里面的i，而不是for中的i，因为我们说过每个闭包的环境都是独立的。

②自执行匿名函数，开辟封闭的变量作用域环境（限制作用域）：

（立即执行的匿名函数并不是函数）

for( var i=0; i<btn.length; i++){

(function （i）{

btn[i].onclick=function(){

alert(i)

}

}) (i)

}

③使用块级作用域let

for( let i=0; i<btn.length; i++){

btn[i].onclick=function(){

alert(i)

}

}

# **[JS中for循环里面的闭包问题的原因及解决办法](https://www.cnblogs.com/ZinCode/p/5551907.html)**

一个正常的for循环，普通函数里面有一个for循环，for循环结束后最终返回结果数组

function box(){

var arr = [];

for(var i=0;i<5;i++){

arr[i] = i;

}

return arr;

}

alert(box())

//正常情况不需要闭包，就可以达到预期效果，输出结果为一个数组0,1,2,3,4

有时我们需要在for循环里面添加一个匿名函数来实现更多功能

//循环里面包含闭包函数

function box(){

var arr = [];

for(var i=0;i<5;i++){

arr[i] = function(){

return i;

}

}

return arr;

}

//alert(box()); //执行5次匿名函数本身

//alert(box()[1]);　　 //执行第2个匿名函数本身（function(){ return i;} ）

//alert(box().length); //最终返回的是一个数组，数组的长度为5

alert(box()[0]()); //数组中的第一个数返回的是5，这是为什么？

在for循环里面的匿名函数执行 return i 语句的时候，由于匿名函数里面没有i这个变量，所以这个i他要从父级函数中寻找i，而父级函数中的i在for循环中，当找到这个i的时候，是for循环完毕的i，也就是5，所以这个box得到的是一个数组[5,5,5,5,5]。

上面这段代码就形成了一个闭包：

**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。**

解决方案1：

匿名函数自我执行的写法是，在函数体外面加一对圆括号，形成一个表达式，在圆括号后面再加一个圆括号，里面可传入参数

function box(){

var arr = [];

for(var i=0;i<5;i++){

arr[i] = (function(num){

//自我执行，并传参(将匿名函数形成一个表达式)(传递一个参数)

return num; //这里的num写什么都可以

})(i);

//这时候这个括号里面的i和上面arr[i]的值是一样的都是取自for循环里面的i

}

return arr;

}

//alert(box());

//alert(box()[1]);

//alert(box().length);

alert(box()[0]);

通过给匿名函数传参，而传递的这个参数i是每次执行for循环里面的i，每次传递的参数i的值都不一样，匿名函数里面的num接收传递的参数i，所以box()最终输出结果为[0,1,2,3,4]

解决方案2

这种方案的原理就是在匿名函数1里面再写入一个匿名函数2，这个匿名函数2需要的num值会在他的父级函数匿名函数1里面去寻找，而匿名函数1里面的num值就是传入的这个参数i，和上面例子中的i是一样的

function box(){

var arr = [];

for(var i=0;i<5;i++){

arr[i] = (function(num){

//num在这里

//原理和上面一种方法一样的，所以可以实现闭包

return function(){

//在这个闭包里面再写一个匿名函数

return num;

};

})(i)

}

return arr;

}

//alert(box());

//alert(box()[1]);

//alert(box().length);

var b = box();

alert(b[0]());

alert(box()[0]());

box()最终返回结果[0,1,2,3,4],