# 三、深入学习循环语句

## 3.1 for循环里面的break;和continue;语句

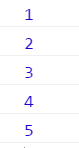
哇，我已经找到我要的答案了，我不需要进行更多的循环了！

比如，寻找第一个能被5整除的数：

### [break语句？](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Statements/break)

for循环中，如果遇见了break语句，这个for循环就会立即终止，不在进行其他的迭代了。

|  |
| --- |
| 1. for(var i = 1 ; i <= 100; i++){ 2. console.log(i); 3. if(i == 5){ 4. **break; //找到了一个，就立即终止循环** 5. } 6. } |



break语句只能跳出当前所在的最内层循环：

|  |
| --- |
| 1. //break只中断了最内层循环，外层循环还在继续 2. for(var i = 1 ; i <= 10 ; i++){ 3. for(var j = 1 ; j <= 10 ; j++){ 4. console.log(i,j); 5. if(j == 5){ 6. break; 7. } 8. } 9. } |

如果你这个break就是想终止所有的循环，那么JS中允许你给循环语句加label

|  |
| --- |
| 1. //break只能终止内层循环，但是我们就是想要终止外层循环 2. //就要给外层循环加label： 3. **waiceng :** for(var i = 1 ; i <= 10 ; i++){ 4. for(var j = 1 ; j <= 10 ; j++){ 5. console.log(i,j); 6. if(j == 5){ 7. **break waiceng;**  //break label的语法 8. } 9. } 10. } |

### [continue语句?](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Statements/continue)

呃，这个答案不是我想要的，赶紧试试下一个数字吧！

遇见continue语句，for会立即终止执行后面的语句，然后进入下一次迭代了。

|  |
| --- |
| 1. for(var i = 1 ; i <= 100 ; i++){ 2. if(i == 5){ 3. continue; //当i为5的时候，立即终止执行后面的console语句，直接i为6了。 4. } 5. console.log(i); 6. } |

同样的，continue只能终端当前最内层的for，外层for要加label。

break和continue的目的，就是优化算法的。

寻找质数的方法：

|  |
| --- |
| 1. <script type="text/javascript"> 2. //寻找2~100之内的所有质数 3. waiceng : 4. for(var i = 2 ; i <= 100 ; i++){ 5. //我们要测试i是不是质数，之前我们的算法是测试i的约数个数（不包括1和自己） 6. //约数个数如果为0，那么就是质数。 7. //凭什么我要测试约数个数？？ 8. //只要你有约数，你就不是质数！！！！ 9. for(var j = 2 ; j <= Math.sqrt(i) ; j++){ 10. if(i % j == 0){ 11. **continue** waiceng; //如果这数字，被某一个数字整除了，那么立即放弃他 12. //开始验证下一个i 13. } 14. } 15. console.log(i); //如果一个数字有约数，就会执行continue，就遇不见这个语句了。 16. } 17. </script> |

## 3.2 [do while语句?](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Statements/do...while)

|  |
| --- |
| 1. var i = 1; 2. do{ 3. console.log(i); 4. i++; 5. }while(i <= 100) |

do就是做，while就是当。

语法：

|  |
| --- |
| 1. do{ 2. ①语句 3. }while(②条件表达式) |

程序一开始就会执行一次①，然后验证②是否为真，如果是真，继续执行①……

这是一个后置验证语句。怎么着都会执行一次：

|  |
| --- |
| 1. do{ 2. console.log("哈哈"); 3. }while(**false**) |



死循环，验证条件横为true：

|  |
| --- |
| 1. do{ 2. alert("么么哒"); 3. alert("呵呵"); 4. alert("我爱你"); 5. alert("呵呵"); 6. }while(**true**); |

for语句是主力语句，也就是说，for做不到的，do……while也同样做不到。do……while能做到的，for也一定能做到。

do……while语句中没有循环变量，适合没有循环变量的情况。经常地，它是个死循环。

|  |
| --- |
| 1. do{ 2. var a = prompt("你到底爱不爱我？"); 3. if(a == "爱"){ 4. alert("我也爱你"); 5. **break;** 6. }else{ 7. alert("回答错误，请重新输入"); 8. } 9. }while(true); |

## 3.3 [while语句?](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Statements/while)

while语句是一个前置验证语句，其余和do while一样一样的。

输出1~100：同样也没有步长，自己维护一个循环变量：

|  |
| --- |
| 1. var n = 1; 2. while(n <= 100){ 3. console.log(n); 4. n++; 5. } |

while语句是一个前置验证语句

|  |
| --- |
| 1. while(false){ 2. console.log("哈哈"); 3. } |

不输出。因为while是先验证的表达式，然后执行里面的语句。

同样的，while不好用，因为没有循环变量，所以也是经常做死循环！

总结一下，JS中循环语句：

|  |
| --- |
| 1. for(){ 3. } |

|  |
| --- |
| 1. do{ 3. }while() |

|  |
| --- |
| 1. while(){ 3. } |

用死循环做一个小游戏，踩地雷游戏。

游戏一开始，系统将随机生成一个地雷数，范围1~100，比如地雷数是66。

游戏开始，系统提示，请输入一个数字，范围1~100，

用户输入50，系统提示：恭喜，你没有踩到地雷，下一个人继续猜，范围50~100

下一个人输入，输入78，系统提示：恭喜，你没有踩到地雷，下一个人继续猜，范围50~78

如果输入的数字，比地雷数大，那么上限改为用户输入的数字；

如果输入的数字，比地雷数小，那么下限改为用户输入的数字；

[随机数?](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Math/random)：

|  |
| --- |
| 1. Math.random() |

能够生成一个0~1的数字，不包括0，也不包括1。术语叫做(0,1)开区间。

random就是随机的意思。

和

Math.pow();

Math.sqrt();

一样，random也是Math对象的方法。具体什么是方法，以后说。

我们现在Math.random()拓展一下：

|  |
| --- |
| 1. parseInt(Math.random() \* 8) |

就是[0,8)的随机数 ， 能取到0，但是不能取到8。所以就是[0,7]

再进行拓展

|  |
| --- |
| 1. parseInt(Math.random() \* 8) + 3 |

就是[3,11]的随机数，能取到3，但是不能取到11。所以就是[3,10]

|  |
| --- |
| 1. parseInt(Math.random() \* 9) + 3 |

就是[3,11]的随机数，能取到3，能取到11

有公式，如果想要在[a,b]闭区间取随机数，能取到a，也能取到b：

|  |
| --- |
| 1. parseInt(Math.random() \* (b - a + 1)) + a; |

比如想取10~20的随机数，能取到10，也能取到20

|  |
| --- |
| 1. parseInt(Math.random() \* 11) + 10; |

比如想取1~2的随机数，能取到1，也能取到2

|  |
| --- |
| 1. parseInt(Math.random() \* 2) + 1; |

# 四、[函数?](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Functions)初步

## 4.1 函数的定义和调用

多条语句，组合成一个“语句军团”，集体作战。

|  |
| --- |
| 1. //定义一个函数，函数就是一组语句的集合 2. **function** **haha**(){ 3. console.log(1); 4. console.log(2); 5. console.log(3); 6. console.log(4); 7. } 9. //调用函数 10. **haha()**; |



函数必须先**定义**然后才能**调用**。

**定义一个函数，用关键字function来定义**，function就是英语“功能”的意思。表示这里面定义的语句，完成了一些功能。function后面有一个空格，后面就是函数名字，函数的名字也是关键字，命名规范和变量命名是一样的。名字后面有一对儿圆括号，里面放置参数，我们一会儿介绍。然后就是大括号，大括号里面是函数的语句。

|  |
| --- |
| 1. function 函数名(){ 3. } |

函数如果不调用，那么里面的语句就一辈子不会执行，不调用就等于白写。

**调用一个函数的方法非常简单，函数名后面加一个()**  ，() 是一个运算符，表示执行一个函数。

|  |
| --- |
| 1. 函数名(); |

一旦调用了函数，函数内部的语句就会执行。

能够感觉到，函数是一些语句的集合，让语句成为一个军团，集体作战。要不出动都不出动，要出动就全出动。得到调用才出动。

**函数的意义1：在出现大量程序相同的时候，可以封装为一个function，这样只用调用一次，就能执行很多语句。**

## 4.2 函数的参数

定义在函数内部的语句，都是相同的，但是实际上我们可以通过“参数”这个东西，来让语句有差别。

定义函数的时候，内部语句可能有一些悬而未决的量，就是变量，这些变量，我们要求在定义的时候都罗列在小括号中：

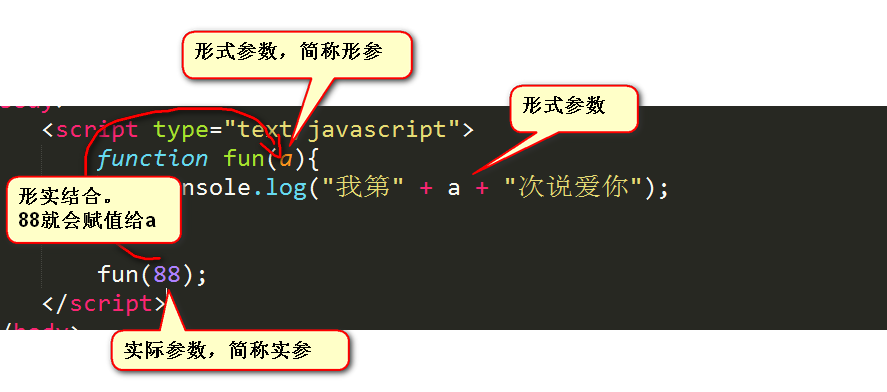
|  |
| --- |
| 1. function fun(a){ 2. console.log("我第" + **a** + "次说爱你"); 3. } |

调用的时候，要把这个变量的真实的值，一起写在括号里，这样随着函数的调用，这个值也传给了a：

|  |
| --- |
| 1. fun(**88**); |



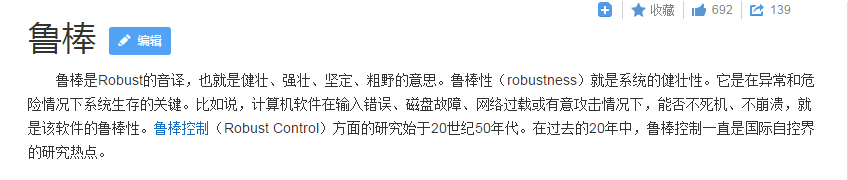
罗列在function小括号中的参数，叫做形式参数；调用时传递的数值，叫做实际参数。



参数可以有无数个，用逗号隔开。

|  |
| --- |
| 1. //有多少形式参数都可以，都罗列出来 2. function fun(a**,**b){ 3. console.log(a + b); 4. } 5. fun(3**,**5); //输出8 6. fun(8**,**11); //输出19 |

**函数的意义2：**我们在调用一个函数的时候，不用关心函数内部的实现细节，甚至这个函数是你上网抄的，可以运用。所以这个东西，给我们团队开发带来了好处。



鲁棒就是没有bug

定义函数的时候，**参数是什么类型的没写，不需要指定类型**：

|  |
| --- |
| 1. function sum(a,b){ 2. console.log(a + b); 3. } |

也就是说调用的时候，**传进去什么什么类型，就是a、b什么类型**

|  |
| --- |
| 1. sum("5",12); |

输出512，做的是连字符的运算。

我们还可以发现，**定义的时候和调用的时候参数个数可以不一样多，不报错**。

|  |
| --- |
| 1. sum(10); |

 因为我们只传了一个参数，b就没有传递，b被隐式的var了,所以值undefined。10+undefined就是NaN

|  |
| --- |
| 1. sum(10,20,32,23,22,2,4); |

 只有前两个参数被形参接收了，**后面的参数无视了**。

明天的课程，你将学习可以用arguments类数组对象，得到任何的参数。

## 4.3 函数的返回值

函数可以通过参数来接收东西，更可以通过return的语句来返回值，“吐出”东西。

|  |
| --- |
| 1. function sum(a,b){ 2. **return** a + b; //现在这个函数的返回值就是a+b的和 3. } 5. console.log(sum(3,8)); //sum没有输出功能，就要用console.log输出 6. //sum(3,8)实际上就成为了一个表达式，需要计算 7. //计算后就是11，console.log(11); |

函数只能有唯一的return，有if语句除外。

程序遇见了return，将立即返回结果，返回调用它的地方，而不执行函数内的剩余的语句。

|  |
| --- |
| 1. function fun(){ 2. console.log(1); 3. console.log(2); 4. **return**; //返回一个空值 5. console.log(3); //这行语句不执行，因为函数已经return了 6. } 7. fun(); |



函数有一个return的值，那么现在这个函数，实际上就是一个表达式，换句话说这个函数就是一个值。

所以这个函数，可以当做其他函数的参数。

|  |
| --- |
| 1. sum**(**3,sum(4,5)**)**; |

 程序从最内层做到最外层， sum(3,9) 12

函数可以接受很多值，返回一个值，

函数的意义3：模块化编程，让复杂的逻辑变得简单。

## 4.4 应用函数简化编程

我们现在做一个程序，输出2~100的所有质数。所谓的质数，就是只有1、自己两个约数，没有其他[约数](http://baike.baidu.com/link?url=FvuwvBrz4WFcE2htvduwfWg0pYMEhbLiySRVpF6_08U8mHNKhZAoUrU7qiizzLy6ne_XS04VPvlUQIAU_9YansXh2QB-NBT524iprq1AEKa)。

要把一个复杂的问题，拆分为一个个小问题。

高层的业务，就能使用底层的函数提供的API：

约数个数函数 → 判断质数函数 → 高层业务

哥德巴赫猜想

任何一个偶数，都可以拆分为两个质数的和。

现在我们要求，用户输入一个偶数，你把所有的质数拆分可能，写出来。

比如4 = 2 + 2

6 = 3 + 3

8 = 3 + 5

48 = 5 + 43

代码见案例

约数个数函数，里面的细节我们不需要关心，它足够的鲁棒，就能返回约数个数。

|  |
| --- |
| 1. function yueshugeshu(a){ 2. var count = 0; 3. for(var i = 1 ; i <= a ; i++){ 4. if(a % i == 0){ 5. count++; 6. } 7. } 8. return count; 9. } |
|  |
|  |

上层的函数，可以使用下面的API：

|  |
| --- |
| 1. function isZhishu(m){ 2. if(**yueshugeshu**(m) == 2){ 3. return true; 4. }else{ 5. return false; 6. } 7. //也可以简写 8. return yueshugeshu(m) == 2; 9. } |

小例子：

计算1!+2!+…… 10!得多少？不要发给我文件，发给我数字就行了。

## 4.5 [递归？](http://baike.baidu.com/item/%E9%80%92%E5%BD%92?fromtitle=%E9%80%92%E5%BD%92&fromid=1740695&type=syn&sefr=cr)

[示例图片1](http://read.html5.qq.com/image?src=forum&q=5&r=0&imgflag=7&imageUrl=http://mmbiz.qpic.cn/mmbiz/OuU9wKDTEicEKWIeC45O5DUfDichyqE0wHibDsArA4NLpumdKsX5iarCzXZEaJ1sT11orPLM7E8L55lXxYLpY6jaoA/0?wx_fmt=gif)

[示例图片2](https://www.google.com.hk/imgres?imgurl=http%3A%2F%2Fwiki.swarma.net%2Fimages%2Ff%2Ff7%2FSee.gif&imgrefurl=http%3A%2F%2Fwiki.swarma.net%2Findex.php%2F%25E9%2580%2592%25E5%25BD%2592%25E5%25AE%259A%25E7%2590%2586&docid=z9c4TrAqt93xbM&tbnid=KBoOoiEs0)

[递归算法详解？](http://chenqx.github.io/2014/09/29/Algorithm-Recursive-Programming/)

函数可以自己调用自己，就是递归。

斐波那契数列就是经典的递归算法：

1、1、2、3、5、8、13、21、34、55、89、144、233……

输出斐波那契数列

欣赏，对于一个搞前端的来说，没必要把程序学这么深：

|  |
| --- |
| 1. function fib(n){ 2. if(n == 1 || n == 2){ 3. return 1; 4. }else{ 5. return **fib**(n - 1) + **fib**(n - 2); 6. } 7. } 8. //一点一点计算： 9. for(var i = 1 ; i <= 55 ;i++){ 10. console.log(fib(i)); 11. } |

## 4.6 函数表达式

定义函数除了使用function之外，还有一种方法，就是函数表达式。就是函数没有名字，称为“匿名函数”，为了今后能够调用它，我们把这个匿名函数，直接赋值给一个变量。

|  |
| --- |
| 1. var haha = function(a,b){ 2. return a + b; 3. } |

以后想调用这个函数的时候，就可以直接使用haha变量来调用。

|  |
| --- |
| 1. console.log(haha(1,3)); |

如果现在这个函数表达式中的function不是匿名的，而是有名字的：

|  |
| --- |
| 1. var haha = function **xixi**(a,b){ 2. return a + b; 3. } |

那么JS表现非常的奇怪，在外部只能用haha()来调用，xixi()非引发错误！



也就是说，JS这个奇怪的特性，给我们提了个醒，定义函数，只能用这两种方法，但是不能杂糅：

|  |
| --- |
| 1. function haha(){ 2. } |

|  |
| --- |
| 1. var haha = function(){ 2. } |

错误的：

|  |
| --- |
| 1. var xixi = function **haha**(){ 2. } |

## 4.7 [函数声明的提升？](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Statements/function)（预解析）

|  |
| --- |
| 1. //先调用 2. fun(); 3. //然后定义 4. function fun(){ 5. alert("我是函数，我执行了！"); 6. } |

不会引发错误，alert能够弹出。

JS在执行前，会有一个预解析的过程，把所有的函数声明，都提升到了最最开头，然后再执行第一行语句。

所以，function定义在哪里，都不重要，程序总能找到这个函数。

**函数声明会被提升，但是函数表达式却不会被提升**

|  |
| --- |
| 1. fun(); 2. var fun = function(){ **//函数表达式，而不是function定义法** 3. alert("我是函数，我执行了！"); 4. } |

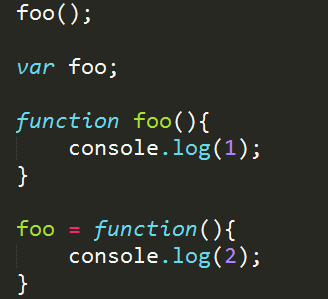


又给我们提了个醒，没有极特殊的理由，都要使用function haha(){} 来定义函数，而不要使用var haha= function(){}

函数优先

|  |
| --- |
| 1. aaa(); //现在这个aaa到底是函数，还是变量5呢？ 2. //函数优先，遇见同名标识符，预解析阶段一定把这个标识符给函数 4. var aaa = 5; //定义一个变量，是5 5. function aaa(){ 6. alert("我是aaa函数，我执行了"); 7. } |

面试题：



函数优先，现在foo这个标识符冲突了，一个函数叫做foo，一个变量也叫作foo。预解析阶段，如果遇见标识符冲突，这个标识符给函数。

具体调用哪个看是变量先被赋值成一个函数还是函数先定义

函数声明的提升，是无节操的，强制提升，即使用if语句，也会提升。

|  |
| --- |
|  |

## 4.8 函数是一个引用类型

我们之前说的，基本类型：number、string、boolean、undefined、null

引用类型也有很多种：object、function、array、RegExp、Math、Date。

|  |
| --- |
| 1. function fun(){ 2. } 3. console.log(typeof fun); |

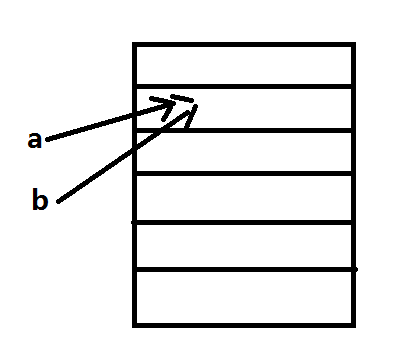


函数也是一种类型。这个类型就叫做function，是引用类型的一种。

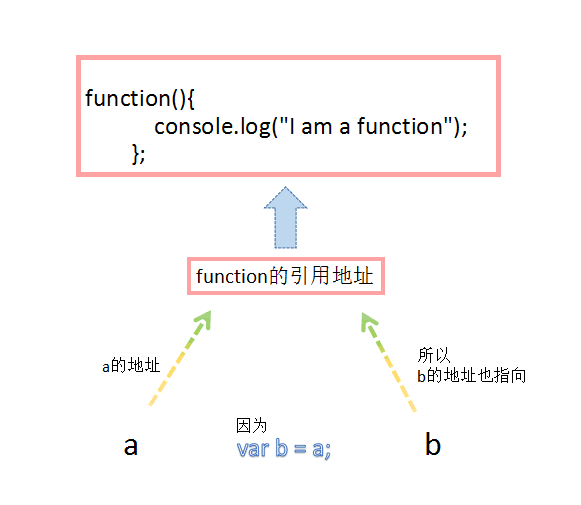
**基本类型保存值，引用类型保存地址**

我们现在变量a = 6 ； 那么这个a变量里面存储的是6这个数值；

而a = function(){} 那么这个a标签里面存储的是function的内存地址。

 a、b指向的是内存中的同一个地址，所以a就是b，b就是a。

|  |
| --- |
| 1. //定义了一个变量a，引用了一个funciton 2. //这个a变量存储的是这个匿名函数的内存地址 3. var a = function(){ 4. alert("我是一个函数，我执行了"); 5. } 6. //就是把匿名函数的地址也给了b 7. var b = a; 8. //给b添加一个属性， 以后讲对象的时候，我们再看 9. b.haha = 1; 10. //输出a的haha属性，你会发现a也有这个属性了： 11. console.log(a.haha); 12. //b的haha属性和a的同步更改的，因为都是指向的同一个对象 13. b.haha++; 14. b.haha++; 15. b.haha++; 16. console.log(a.haha); |

北京web六期班付荣同学出品

基本类型的赋值是复制数据本身

引用类型的赋值是传递存储地址

引用类型是同一个地址，改变的是地址里面的内容，地址不变，所以一个改变了另一个也就改变了！

说的通俗点，就是引用数据就是一个物体的不同的；实际上就是一个东西，相当于有别名

# 4.9、[aruguments?](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Functions/arguments)

我们之前学习函数的时候，遗漏这个知识点。

复习一下函数，我们定义函数的时候，参数要罗列在小括号里面，这个参数，叫做形参：

|  |
| --- |
| 1. function fn(a,b,c,d){ 2. } |

调用函数的时候，传参数，这个参数叫做实参。JS不要求形参数量和实参数量一样多：

|  |
| --- |
| 1. fn(1,2); //合法的，虽然少了参数，但是不报错 2. fn(1,2,3,4,5,6,7,8) //合法，多了，也不报错。 |

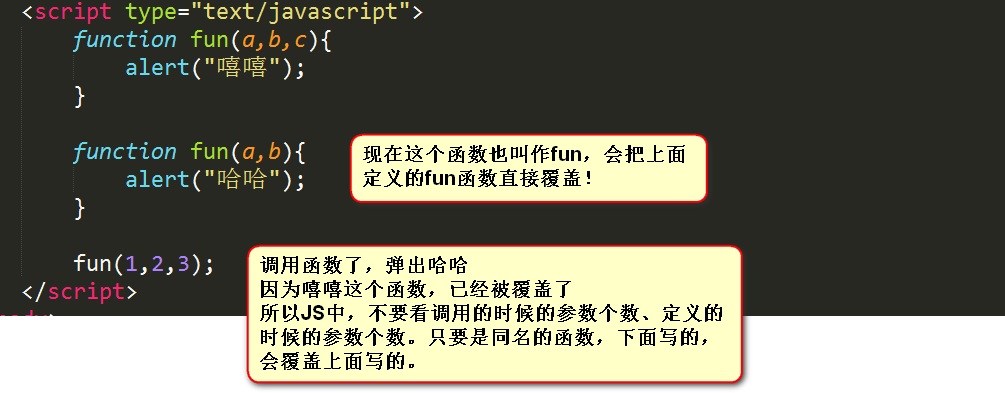
实际上，这个东西叫做方法的重载。在Java中，同一个函数名，但是参数个数不一样，视为是两个函数。

也就是说，Java中能够定义两个：

|  |
| --- |
| 1. function sum(a,b){ 2. } 3. function sum(a,b,c){ 4. } |

同名的两个function，都是sum函数，但是java中是允许这么做的，因为参数个数不一样，这种现象叫做重载。

JavaScript没有重载的概念：



JS有一个非常强大的东西，就是每一个函数内部，都可以使用一个arguments这个类数组对象。

这个arguments对象，就涵盖了所有实参。

调用函数的时候，比如：

|  |
| --- |
| 1. fun(45,436,457,34,23,12); |

此时函数内部，arguments就有一个下标，就依次等于上面调用的数：

|  |
| --- |
| 1. arguments[0] // 45 2. arguments[1] // 436 3. …… 4. arguments[5] //12 |

如果函数里面有形式参数列表，那么是和arguments同步的：

|  |
| --- |
| 1. function fun(a,b){ 2. arguments[0] = 8; //改变了第一个参数的值 3. alert(a); //8 ，弹出改变后的值 4. } 5. fun(45,436,457,34,23,12); |

arguments的功能，是模拟函数的重载，使得同一个函数，根据参数个数的不同，有不同的作用。

比如现在我们想设计一个sum函数，如果传进来一个参数，就得到这个数字的加1；如果是2个参数，那么返回两个数字的和。比如

|  |
| --- |
| 1. sum(10); //1 2. sum(3,4); //7 |

就要通过arguments.length 实际参数的个数，来进行判断：

|  |
| --- |
| 1. function sum(a,b){ 2. //如果实际参数的长度是1，说白了，你只传进来一个参数 3. switch(arguments.length){ 4. case 1: 5. return ++a; 6. break; 7. case 2: 8. return a + b; 9. break; 10. } 11. } |

更牛逼的，我们可以无限参数，设计一个函数sum，能够接受无限参数

|  |
| --- |
| 1. sum(3,4,6,5,8,2) //28 |

# 4.10、IIFE

IIFE就是immediately-invoked function expression，[即时调用函数表达式?](http://benalman.com/news/2010/11/immediately-invoked-function-expression/) [IIFE wikiPIdia](https://en.wikipedia.org/wiki/Immediately-invoked_function_expression)

如果一个函数，在定义的时候，我们就想直接调用它，就是一个IIFE。

我们试图在定义函数的后面，直接写圆括号：

|  |
| --- |
| 1. function fun(){ 2. alert("哈哈") 3. }(); |

控制台报错，这是因为函数是一个函数体，并不是表达式，只有表达式能够用()来执行。

所以就要把function fun(){}“降级”， 从函数体降级为表达式。方法有很多：

|  |
| --- |
| 1. **+**function fun(){ 2. alert("哈哈") 3. }(); |

|  |
| --- |
| 1. **-**function fun(){ 2. alert("哈哈") 3. }(); |

更通常的：

|  |
| --- |
| 1. **(**function fun(){ 2. alert("哈哈") 3. }**)**(); |

用这种方法定义的函数，名字是无效的，其他的地方想调用这个函数

|  |
| --- |
| 1. fun("嘻嘻"); |

就会报错。

所以IIFE里面的函数，都是匿名函数：

|  |
| --- |
| 1. (function(){ 2. alert("哈哈"); 3. })(); |

上面就是一个标准的IIFE。

设计一个函数，这个函数接收三个参数，比如sum(4,7,9);返回的是前两个数字大的那个数字，与第三个数字的和。

|  |
| --- |
| 1. sum(4,2,3); //7 2. sum(2,4,3); //7 3. sum(5,4,3); //8 |

|  |
| --- |
| 1. function sum(a,b,c){ 2. return (function(a,b){ 3. return a >= b ? a : b; 4. })(a,b) + c; 5. } |

红色部分是一个IIFE，本质上是一个表达式，表达式计算之后，就是值，什么值呢？a、b中大的那个数字。