什么是高阶函数？

高阶函数就是函数嵌套函数

简单总结为高阶函数是一个接收函数作为参数或者将函数作为返回输出的函数。

高阶函数是至少满足下列一个条件的函数:

①接受一个或多个函数作为输入；

②输出一个函数

第一个条件就是：函数作为参数传入另一个函数，那就是回调函数。例如：

var arr = [1,2,3,4];

arr.map(function(item,index,arr){

arr[index] = arr[index]\*10;

});

console.log(arr);//[ 10, 20, 30, 40 ]

第二种条件是函数内返回一个函数，比如闭包：下面就随便举一个例子

function a (x){

return x\*10;

}

function b(f,h){

return f(h)\*f(h)

}

var cc=b(a,5);

console.log(cc);//2500

常用js高阶函数：

一、map/reduce

如果你读过Google的那篇大名鼎鼎的论文“MapReduce: Simplified Data Processing on Large Clusters”，你就能大概明白map/reduce的概念。由于map()方法定义在JavaScript的Array中，我们调用Array的map()方法，传入我们自己的函数，就得到了一个新的Array作为结果：

1、map()：

function pow(x) {

return x \* x;

}

var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9];

arr.map(pow); // [1, 4, 9, 16, 25, 36, 49, 64, 81]

//map()传入的参数是pow，即函数对象本身。

//不需要map()，写一个循环，也可以计算出结果：

var f = function (x) {

return x \* x;

};

var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9];

var result = [];

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

result.push(f(arr[i]));

}

//的确可以，但是，从上面的循环代码，我们无法一眼看明白“把f(x)作用在Array的每一个元素并把结果生成一个新的Array”。

　　所以，map()作为高阶函数，事实上它把运算规则抽象了，因此，我们不但可以计算简单的f(x)=x2，还可以计算任意复杂的函数，比如，把Array的所有数字转为字符串：

var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9];

arr.map(String); // ['1', '2', '3', '4', '5', '6', '7', '8', '9']

//只需要一行代码。

**2、reduce()：**

　　再看reduce的用法。Array的reduce()把一个函数作用在这个Array的[x1, x2, x3...]上，这个函数必须接收两个参数，reduce()把结果继续和序列的下一个元素做累积计算，其效果就是：

[x1, x2, x3, x4].reduce(f) = f(f(f(x1, x2), x3), x4)

//比方说对一个Array求和，就可以用reduce实现：

var arr = [1, 3, 5, 7, 9];

arr.reduce(function (x, y) {

return x + y;

}); // 25

**二、filter**

　　filter也是一个常用的操作，它用于把Array的某些元素过滤掉，然后返回剩下的元素。和map()类似，Array的filter()也接收一个函数。和map()不同的是，filter()把传入的函数依次作用于每个元素，**然后根据返回值是true还是false决定保留还是丢弃该元素**。

//例如，在一个Array中，删掉偶数，只保留奇数，可以这么写：

var arr = [1, 2, 4, 5, 6, 9, 10, 15];

var r = arr.filter(function (x) {

return x % 2 !== 0;

});

r; // [1, 5, 9, 15]

//把一个Array中的空字符串删掉，可以这么写：

var arr = ['A', '', 'B', null, undefined, 'C', ' '];

var r = arr.filter(function (s) {

return s && s.trim(); // 注意：IE9以下的版本没有trim()方法

});

arr; // ['A', 'B', 'C']

　　可见用filter()这个高阶函数，关键在于正确实现一个“筛选”函数。

　　回调函数：filter()接收的回调函数，其实可以有多个参数。通常我们仅使用第一个参数，表示Array的某个元素。回调函数还可以接收另外两个参数，表示元素的位置和数组本身：

var arr = ['A', 'B', 'C'];

var r = arr.filter(function (element, index, self) {

console.log(element); // 依次打印'A', 'B', 'C'

console.log(index); // 依次打印0, 1, 2

console.log(self); // self就是变量arr

return true;

});

//利用filter，可以巧妙地去除Array的重复元素：

'use strict';

var r,

arr = ['apple', 'strawberry', 'banana', 'pear', 'apple', 'orange', 'orange', 'strawberry'];

r = arr.filter(function (element, index, self) {

return self.indexOf(element) === index;

});

alert(r.toString());

//去除重复元素依靠的是indexOf总是返回第一个元素的位置，后续的重复元素位置与indexOf返回的位置不相等，因此被filter滤掉了。

**三、sort排序算法**

　　因为Array的sort()方法默认把所有元素先转换为String再排序，结果'10'排在了'2'的前面，因为字符'1'比字符'2'的ASCII码小。如果不知道sort()方法的默认排序规则，直接对数字排序，绝对栽进坑里！

　　幸运的是，sort()方法也是一个高阶函数，它还可以接收一个比较函数来实现自定义的排序。

//要按数字大小排序，我们可以这么写：

var arr = [10, 20, 1, 2];

arr.sort(function (x, y) {

if (x < y) {

return -1;

}

if (x > y) {

return 1;

}

return 0;

}); // [1, 2, 10, 20]

//如果要倒序排序，我们可以把大的数放前面：

var arr = [10, 20, 1, 2];

arr.sort(function (x, y) {

if (x < y) {

return 1;

}

if (x > y) {

return -1;

}

return 0;

}); // [20, 10, 2, 1]

//**默认情况下，对字符串排序，是按照ASCII的大小比较的**，现在，排序应该忽略大小写，按照字母序排序。

//要实现这个算法，不必对现有代码大加改动，只要我们能定义出忽略大小写的比较算法就可以：

var arr = ['Google', 'apple', 'Microsoft'];

arr.sort(function (s1, s2) {

x1 = s1.toUpperCase();

x2 = s2.toUpperCase();

if (x1 < x2) {

return -1;

}

if (x1 > x2) {

return 1;

}

return 0;

}); // ['apple', 'Google', 'Microsoft']

//忽略大小写来比较两个字符串，实际上就是先把字符串都变成大写（或者都变成小写），再比较。

//sort()方法会直接对Array进行修改，它返回的结果仍是当前Array：

var a1 = ['B', 'A', 'C'];

var a2 = a1.sort();

a1; // ['A', 'B', 'C']

a2; // ['A', 'B', 'C']

a1 === a2; // true, a1和a2是同一对象

四：柯里化函数

函数柯里化，是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数而且返回结果的新函数的技术。

// 实现一个add方法，使计算结果能够满足如下预期：

add(1)(2)(3) = 6;

add(1, 2, 3)(4) = 10;

add(1)(2)(3)(4)(5) = 15;

代码如下：

function add(){

// 第一次执行时，定义一个数组专门用来存储所有的参数

var \_args =Array.prototype.slice.call(arguments); //[...arguments]

// 在内部声明一个函数，利用闭包的特性保存\_args并收集所有的参数值

var \_adder = function(){

\_args.push(...arguments)

return \_adder

}

// 利用toString隐式转换的特性，当最后执行时隐式转换，并计算最终的值返回

\_adder.toString = function(){

return \_args.reduce(function(a,b){

return a+b;

},0)

}

return \_adder

}

add(1)(2)(3) // 6

add(1, 2, 3)(4) // 10

add(1)(2)(3)(4)(5) // 15

add(2, 6)(1) // 9

测试：

function add(){

// 第一次执行时，定义一个数组专门用来存储所有的参数

var \_args = [...arguments]

console.log(\_args)

// 在内部声明一个函数，利用闭包的特性保存\_args并收集所有的参数值

var \_adder = function(){

\_args.push(...arguments)

return \_adder

}

// 利用toString隐式转换的特性，当最后执行时隐式转换，并计算最终的值返回

\_adder.toString = function(){

console.log(\_args)

return \_args.reduce(function(a,b){

return a+b;

},0)

}

return \_adder

}

add(1)(2)(3) // f 6

let f = add(1)(2)(3)

console.log(typeof f) //function

f(4) //10

五：forEach

语法

array.forEach(function(currentValue, index, arr), thisValue)

没有返回值，返回值为undefined

|  |  |
| --- | --- |
| **返回值:** | undefined |

forEach() 的 continue 与 break

forEach() 本身是不支持的 continue 与 break 语句的

使用 **return** 语句实现 **continue** 关键字的效果：

**continue 实现**

实例

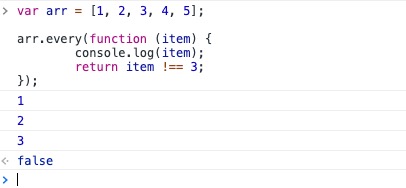
**var** arr = [1, 2, 3, 4, 5];  
  
arr.forEach(**function** (item) {  
    **if** (item === 3) {  
        **return**; *// 不能为 return false*  
    }  
    console.log(item);  
});



#### **break 实现**

实例

**var** arr = [1, 2, 3, 4, 5];  
  
arr.every(**function** (item) {  
        console.log(item);  
        **return** item !== 3;  
});



**六，some，every**

forEach() 本身是不支持的 continue 与 break 语句的，我们可以通过 [some](https://www.runoob.com/jsref/jsref-some.html" \t "https://www.runoob.com/jsref/_blank) 和 [every](https://www.runoob.com/jsref/jsref-every.html" \t "https://www.runoob.com/jsref/_blank) 来实现。

some() 方法用于检测数组中的元素是否满足指定条件（函数提供）。

some() 方法会依次执行数组的每个元素：

* 如果有一个元素满足条件，则表达式返回true , 剩余的元素不会再执行检测。
* 如果没有满足条件的元素，则返回false。

**注意：** some() 不会对空数组进行检测。

**注意：** some() 不会改变原始数组。

语法

array.some(function(currentValue,index,arr),thisValue)

every() 方法用于检测数组所有元素是否都符合指定条件（通过函数提供）。

every() 方法使用指定函数检测数组中的所有元素：

* 如果数组中检测到有一个元素不满足，则整个表达式返回 false ，且剩余的元素不会再进行检测。
* 如果所有元素都满足条件，则返回 true。

**注意：** every() 不会对空数组进行检测。

**注意：** every() 不会改变原始数组。

语法

array.every(function(currentValue,index,arr), thisValue)