数组创建

JavaScript中创建数组有两种方式，第一种是使用 Array 构造函数：

var arr1 = new Array(); //创建一个空数组

var arr2 = new Array(20); // 创建一个包含20项的数组

var arr3 = new Array("lily","lucy","Tom"); // 创建一个包含3个字符串的数组

创建数组的第二种基本方式是使用数组字面量表示法：

var arr4 = []; //创建一个空数组

var arr5 = [20]; // 创建一个包含1项的数组

var arr6 = ["lily","lucy","Tom"]; // 创建一个包含3个字符串的数组

在读取和设置数组的值时，要使用方括号并提供相应值的基于 0 的数字索引：

var arr6 = ["lily","lucy","Tom"]; // 创建一个包含3个字符串的数组

alert(arr6[0]); //lily

arr6[1] = "mary"; //修改第二项为mary

arr6[3] = "sean"; //增加第四项为sean

JavaScript中数组的length属性是可以修改的，看下面的示例：

var arr = ["lily","lucy","Tom"]; // 创建一个包含3个字符串的数组

arr[arr.length] = "sean"; //在下标为3处（也就是数组尾部）添加一项"sean"

arr.length = arr.length-1; //将数组的最后一项删除

如果需要判断一个对象是不是数组对象，在ECMAScript 5之前，我们可以通过 instanceof Array去判断，但是instanceof 操作符的问题在于，它假定只有一个全局执行环境。如果网页中包含多个框架，那实际上就存在两个以上不同的全局执行环境，从而存在两个以上不同版本的 Array 构造函数。如果你从一个框架向另一个框架传入一个数组，那么传入的数组与在第二个框架中原生创建的数组分别具有各自不同的构造函数。

ECMAScript 5 新增了 Array.isArray()方法。这个方法的目的是最终确定某个值到底是不是数组，而不管它是在哪个全局执行环境中创建的。

数组方法

下面开始介绍数组的方法，数组的方法有数组原型方法，也有从object对象继承来的方法，这里我们只介绍数组的原型方法，数组原型方法主要有以下这些：

join() 数组转字符串

split() 字符串转数组

push()和pop()

shift() 和 unshift()

sort()

reverse()

concat()

slice()

splice()

indexOf()和 lastIndexOf() （ES5新增）

forEach() （ES5新增）

map() （ES5新增）

filter() （ES5新增）

every() （ES5新增）

some() （ES5新增）

reduce()和 reduceRight() （ES5新增）

针对ES5新增的方法浏览器支持情况：

Opera 11+

Firefox 3.6+

Safari 5+

Chrome 8+

Internet Explorer 9+

对于支持的浏览器版本，可以通过Array原型扩展来实现。下面详细介绍一下各个方法的基本功能。

1、join()

join(separator): 将数组的元素组起一个字符串，以separator为分隔符，省略的话则用默认用逗号为分隔符，该方法只接收一个参数：即分隔符。

var arr = [1,2,3];

console.log(arr.join()); // 1,2,3

console.log(arr.join("-")); // 1-2-3

console.log(arr); // [1, 2, 3]（原数组不变）

通过join()方法可以实现重复字符串，只需传入字符串以及重复的次数，就能返回重复后的字符串，函数如下：

function repeatString(str, n) {

return new Array(n + 1).join(str);

}

console.log(repeatString("abc", 3)); // abcabcabc

console.log(repeatString("Hi", 5)); // HiHiHiHiHi

**split（）**split() 方法用于把一个字符串分割成字符串数组。

如果把空字符串 ("") 用作 separator，那么 stringObject 中的每个字符之间都会被分割。

<script type="text/javascript">

var str="How are you doing today?"

document.write(str.split(" ") + "<br />")

document.write(str.split("") + "<br />")

document.write(str.split(" ",3))

</script>

输出：

How,are,you,doing,today?

H,o,w, ,a,r,e, ,y,o,u, ,d,o,i,n,g, ,t,o,d,a,y,?

How,are,you

**注释：String.split() 执行的操作与 Array.join 执行的操作是相反的。**

2、push()和pop()

push(): 可以接收任意数量的参数，把它们逐个添加到数组末尾，并返回修改后数组的长度。

pop()：数组末尾移除最后一项，减少数组的 length 值，然后返回移除的项。

var arr = ["Lily","lucy","Tom"];

var count = arr.push("Jack","Sean");

console.log(count); // 5

console.log(arr); // ["Lily", "lucy", "Tom", "Jack", "Sean"]

var item = arr.pop();

console.log(item); // Sean

console.log(arr); // ["Lily", "lucy", "Tom", "Jack"]

3、shift() 和 unshift()

shift()：删除原数组第一项，并返回删除元素的值；如果数组为空则返回undefined 。

unshift:将参数添加到原数组开头，并返回数组的长度 。

这组方法和上面的push()和pop()方法正好对应，一个是操作数组的开头，一个是操作数组的结尾。

var arr = ["Lily","lucy","Tom"];

var count = arr.unshift("Jack","Sean");

console.log(count); // 5

console.log(arr); //["Jack", "Sean", "Lily", "lucy", "Tom"]

var item = arr.shift();

console.log(item); // Jack

console.log(arr); // ["Sean", "Lily", "lucy", "Tom"]

4、sort()

sort()：按升序排列数组项——即最小的值位于最前面，最大的值排在最后面。

在排序时，sort()方法会调用每个数组项的 toString()转型方法，然后比较得到的字符串，以确定如何排序。即使数组中的每一项都是数值， sort()方法比较的也是字符串，因此会出现以下的这种情况：

var arr1 = ["a", "d", "c", "b"];

console.log(arr1.sort()); // ["a", "b", "c", "d"]

arr2 = [13, 24, 51, 3];

console.log(arr2.sort()); // [13, 24, 3, 51]

console.log(arr2); // [13, 24, 3, 51](元数组被改变)

为了解决上述问题，sort()方法可以接收一个比较函数作为参数，以便我们指定哪个值位于哪个值的前面。比较函数接收两个参数，如果第一个参数应该位于第二个之前则返回一个负数，如果两个参数相等则返回 0，如果第一个参数应该位于第二个之后则返回一个正数。以下就是一个简单的比较函数：

function compare(value1, value2) {

if (value1 < value2) {

return -1;

} else if (value1 > value2) {

return 1;

} else {

return 0;

}

}

arr2 = [13, 24, 51, 3];

console.log(arr2.sort(compare)); // [3, 13, 24, 51]

如果需要通过比较函数产生降序排序的结果，只要交换比较函数返回的值即可：

function compare(value1, value2) {

if (value1 < value2) {

return 1;

} else if (value1 > value2) {

return -1;

} else {

return 0;

}

}

arr2 = [13, 24, 51, 3];

console.log(arr2.sort(compare)); // [51, 24, 13, 3]

5、reverse()

reverse()：反转数组项的顺序。

var arr = [13, 24, 51, 3];

console.log(arr.reverse()); //[3, 51, 24, 13]

console.log(arr); //[3, 51, 24, 13](原数组改变)

6、concat()

concat() ：将参数添加到原数组中。这个方法会先创建当前数组一个副本，然后将接收到的参数添加到这个副本的末尾，最后返回新构建的数组。在没有给 concat()方法传递参数的情况下，它只是复制当前数组并返回副本。

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

var arrCopy = arr.concat(9,[11,13]);

console.log(arrCopy); //[1, 3, 5, 7, 9, 11, 13]

console.log(arr); // [1, 3, 5, 7](原数组未被修改)

从上面测试结果可以发现：传入的不是数组，则直接把参数添加到数组后面，如果传入的是数组，则将数组中的各个项添加到数组中。但是如果传入的是一个二维数组呢？

var arrCopy2 = arr.concat([9,[11,13]]);

console.log(arrCopy2); //[1, 3, 5, 7, 9, Array[2]]

console.log(arrCopy2[5]); //[11, 13]

上述代码中，arrCopy2数组的第五项是一个包含两项的数组，也就是说concat方法只能将传入数组中的每一项添加到数组中，如果传入数组中有些项是数组，那么也会把这一数组项当作一项添加到arrCopy2中。

7、slice()

slice()：返回从原数组中指定开始下标到结束下标之间的项组成的新数组。slice()方法可以接受一或两个参数，即要返回项的起始和结束位置。在只有一个参数的情况下， slice()方法返回从该参数指定位置开始到当前数组末尾的所有项。如果有两个参数，该方法返回起始和结束位置之间的项——但不包括结束位置的项。

var arr = [1,3,5,7,9,11];

var arrCopy = arr.slice(1);

var arrCopy2 = arr.slice(1,4);

var arrCopy3 = arr.slice(1,-2);

var arrCopy4 = arr.slice(-4,-1);

console.log(arr); //[1, 3, 5, 7, 9, 11](原数组没变)

console.log(arrCopy); //[3, 5, 7, 9, 11]

console.log(arrCopy2); //[3, 5, 7]

console.log(arrCopy3); //[3, 5, 7]

console.log(arrCopy4); //[5, 7, 9]

arrCopy只设置了一个参数，也就是起始下标为1，所以返回的数组为下标1（包括下标1）开始到数组最后。

arrCopy2设置了两个参数，返回起始下标（包括1）开始到终止下标（不包括4）的子数组。

arrCopy3设置了两个参数，终止下标为负数，当出现负数时，将负数加上数组长度的值（6）来替换该位置的数，因此就是从1开始到4（不包括）的子数组。

arrCopy4中两个参数都是负数，所以都加上数组长度6转换成正数，因此相当于slice(2,5)。

8、splice()

splice()：很强大的数组方法，它有很多种用法，可以实现删除、插入和替换。

删除：可以删除任意数量的项，只需指定 2 个参数：要删除的第一项的位置和要删除的项数。例如， splice(0,2)会删除数组中的前两项。

插入：可以向指定位置插入任意数量的项，只需提供 3 个参数：起始位置、 0（要删除的项数）和要插入的项。例如，splice(2,0,4,6)会从当前数组的位置 2 开始插入4和6。

替换：可以向指定位置插入任意数量的项，且同时删除任意数量的项，只需指定 3 个参数：起始位置、要删除的项数和要插入的任意数量的项。插入的项数不必与删除的项数相等。例如，splice (2,1,4,6)会删除当前数组位置 2 的项，然后再从位置 2 开始插入4和6。

splice()方法始终都会返回一个数组，该数组中包含从原始数组中删除的项，如果没有删除任何项，则返回一个空数组。

var arr = [1,3,5,7,9,11];

var arrRemoved = arr.splice(0,2);

console.log(arr); //[5, 7, 9, 11]

console.log(arrRemoved); //[1, 3]

var arrRemoved2 = arr.splice(2,0,4,6);

console.log(arr); // [5, 7, 4, 6, 9, 11]

console.log(arrRemoved2); // []

var arrRemoved3 = arr.splice(1,1,2,4);

console.log(arr); // [5, 2, 4, 4, 6, 9, 11]

console.log(arrRemoved3); //[7]

9、indexOf()和 lastIndexOf()

indexOf()：接收两个参数：要查找的项和（可选的）表示查找起点位置的索引。其中， 从数组的开头（位置 0）开始向后查找。

lastIndexOf：接收两个参数：要查找的项和（可选的）表示查找起点位置的索引。其中， 从数组的末尾开始向前查找。

这两个方法都返回要查找的项在数组中的位置，或者在没找到的情况下返回1。在比较第一个参数与数组中的每一项时，会使用全等操作符。

?

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

console.log(arr.indexOf(5)); //2

console.log(arr.lastIndexOf(5)); //5

console.log(arr.indexOf(5,2)); //2

console.log(arr.lastIndexOf(5,4)); //2

console.log(arr.indexOf("5")); //-1

10、forEach()

forEach()：对数组进行遍历循环，对数组中的每一项运行给定函数。这个方法没有返回值。参数都是function类型，默认有传参，参数分别为：遍历的数组内容；第对应的数组索引，数组本身。

var arr = [1, 2, 3, 4, 5];

arr.forEach(function(x, index, a){

console.log(x + '|' + index + '|' + (a === arr));

});

// 输出为：

// 1|0|true

// 2|1|true

// 3|2|true

// 4|3|true

// 5|4|true

11、map()

map()：指“映射”，对数组中的每一项运行给定函数，返回每次函数调用的结果组成的数组。

下面代码利用map方法实现数组中每个数求平方。

var arr = [1, 2, 3, 4, 5];

var arr2 = arr.map(function(item){

return item\*item;

});

console.log(arr2); //[1, 4, 9, 16, 25]

12、filter()

filter()：“过滤”功能，数组中的每一项运行给定函数，返回满足过滤条件组成的数组。

var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

var arr2 = arr.filter(function(x, index) {

return index % 3 === 0 || x >= 8;

});

console.log(arr2); //[1, 4, 7, 8, 9, 10]

13、every()

every()：判断数组中每一项都是否满足条件，只有所有项都满足条件，才会返回true。

var arr = [1, 2, 3, 4, 5];

var arr2 = arr.every(function(x) {

return x < 10;

});

console.log(arr2); //true

var arr3 = arr.every(function(x) {

return x < 3;

});

console.log(arr3); // false

14、some()

some()：判断数组中是否存在满足条件的项，只要有一项满足条件，就会返回true。

var arr = [1, 2, 3, 4, 5];

var arr2 = arr.some(function(x) {

return x < 3;

});

console.log(arr2); //true

var arr3 = arr.some(function(x) {

return x < 1;

});

console.log(arr3); // false

15、reduce()和 reduceRight()

这两个方法都会实现迭代数组的所有项，然后构建一个最终返回的值。reduce()方法从数组的第一项开始，逐个遍历到最后。而 reduceRight()则从数组的最后一项开始，向前遍历到第一项。

这两个方法都接收两个参数：一个在每一项上调用的函数和（可选的）作为归并基础的初始值。

传给 reduce()和 reduceRight()的函数接收 4 个参数：前一个值、当前值、项的索引和数组对象。这个函数返回的任何值都会作为第一个参数自动传给下一项。第一次迭代发生在数组的第二项上，因此第一个参数是数组的第一项，第二个参数就是数组的第二项。

下面代码用reduce()实现数组求和，数组一开始加了一个初始值10。

var values = [1,2,3,4,5];

var sum = values.reduceRight(function(prev, cur, index, array){

return prev + cur;

},10);

console.log(sum); //25/

**find() 语法：**array.find(function(currentValue, index, arr),thisValue)

find() 方法返回通过测试（函数内判断）的数组的第一个元素的值。

find() 方法为数组中的每个元素都调用一次函数执行：

当数组中的元素在测试条件时返回 true 时, find() 返回符合条件的元素，之后的值不会再调用执行函数。

如果没有符合条件的元素返回 undefined

注意: find() 对于空数组，函数是不会执行的。

注意: find() 并没有改变数组的原始值。

**例子：**

获取数组中年龄大于 18 的第一个元素

var ages = [3, 10, 18, 20];

function checkAdult(age) {return age >= 18;}

function myFunction() {

document.getElementById("demo").innerHTML = ages.find(checkAdult);}

fruits 输出结果：18

**findIndex() 语法：array.findIndex(function(currentValue, index, arr), thisValue)**

方法返回传入一个测试条件（函数）符合条件的数组第一个元素位置。

findIndex() 方法为数组中的每个元素都调用一次函数执行：

当数组中的元素在测试条件时返回 true 时, findIndex() 返回符合条件的元素的索引位置，之后的值不会再调用执行函数。

如果没有符合条件的元素返回 -1

注意: findIndex() 对于空数组，函数是不会执行的。

注意: findIndex() 并没有改变数组的原始值。

例子：

获取数组中年龄大于等于 18 的第一个元素索引位置

var ages = [3, 10, 18, 20];

function checkAdult(age) { return age >= 18;}

function myFunction() {

document.getElementById("demo").innerHTML = ages.findIndex(checkAdult);}

// 2

includes()

两种类型中可以使用：

**字符串中**：语法string.includes(searchvalue, start)

includes() 方法用于判断字符串是否包含指定的子字符串。

如果找到匹配的字符串则返回 true，否则返回 false。

注意： includes() 方法区分大小写。

例子：

查找字符串是否包含 "Runoob":

var str = "Hello world, welcome to the Runoob。";

var n = str.includes("world");

n 输出结果: true

数组中：语法：arr.includes(searchElement)

arr.includes(searchElement, fromIndex) fromIndex可选。从该索引处开始查找 searchElement。如果为负值，则按升序从 array.length + fromIndex 的索引开始搜索。默认为 0。

例子：

includes() 方法用来判断一个数组是否包含一个指定的值，如果是返回 true，否则false。

[1, 2, 3].includes(2); // true

[1, 2, 3].includes(4); // false

[1, 2, 3].includes(3, 3); // false

[1, 2, 3].includes(3, -1); // true

[1, 2, NaN].includes(NaN); // true

**1、****说一说****javascript中有哪些数据类型?**

JavaScript 中共有七种内置数据类型，包括基本类型和应引用数据类型(对象类型)。

基本类型分为以下六种：

string（字符串）boolean（布尔值）number（数字）symbol（符号）null（空值）undefined（未定义）；基本数据类型是按值访问的,因为可以直接操作保存在变量中的实际值。

**注意：**

string 、number 、boolean 和 null undefined 这五种类型统称为原始类型（Primitive），表示不能再细分下去的基本类型

symbol是ES6中新增的数据类型，symbol 表示独一无二的值，通过 Symbol 函数调用生成，由于生成的 symbol 值为原始类型，所以 Symbol 函数不能使用 new 调用；

主要用于当前属性名，防止属性名重名，因为用字符串当属性名很容易会在不经意间造成重名。

Symbol 函数可以接收一个字符串参数，可以传一个对象作为参数（当然得这个对象要有toString() 方法才行，因为会先调用对象的toString方法来得到字符串）,也可以不传参数；

const sym1 = Symbol('adc');

const sym2 = Symbol('adc');

const sym3 = Symbol('jungle');

const sym4 = Symbol();

const sym5 = Symbol();

sym1 // Symbol(adc)

sym1.toString(); // "Symbol(adc)"

//还可以显式转换为字符串

String(sym1); // "Symbol(adc)"

sym1 === sym2 // false 参数一样，值也是不一样

sym4 === sym5 // false 不传参数，值也不一样，这就是Symbol值的独一无二性

另外，Symbol 值不可以和其他类型值进行混合运算,否则会报错；

还有就是 Symbol 值可以转换显式或者隐式转换为布尔值，同时却是不能转换成数值；

**let ab = Symbol.for('test');**

**Symbol.keyFor(ab)**  //"test"

**Symbol.for(): 用这个方法创建的Symbol值会被登记在全局环境中，调用这个方法，会先去全局环境中查看有没有以相同字符串作为参数的Symbol值，如果有就拿到这个Symbol值，没有就会创建一个新的Symbol值；**而用Symbol()创建Symbol值，不会存在全局环境供搜索，也不会去查询，直接就会创建一个新的Symbol值；

**null 和 undefined 通常被认为是特殊值，这两种类型的值唯一，就是其本身。**

**引用数据类型：** 指的是那些保存在堆内存中的对象，意思是，变量中保存的实际上只是一个指针，这个指针指向内存堆中实际的值；包括 function object array等

**2、说说你对javascript是弱类型语言的理解?**

JavaScript 是弱类型语言，而且JavaScript 声明变量的时候并没有预先确定的类型， 变量的类型就是其值的类型，也就是说变量当前的类型由其值所决定,夸张点说上一秒种的String，下一秒可能就是个Number类型了，这个过程可能就进行了某些操作发生了强制类型转换。虽然弱类型的这种不需要预先确定类型的特性给我们带来了便利，同时也会给我们带来困扰。为了能充分利用该特性就必须掌握类型转换的原理；

3、**js中的强制转换规则**

面试官问：**javascript中强制类型转换是一个非常易出现bug的点，知道强制转换时候的规则吗？**

注：规则最好配合下面什么时候发生转换使用这些规则看效果更佳。

**ToPrimitive(转换为原始值)**

ToPrimitive对原始类型不发生转换处理，只针对引用类型（object）的，其目的是将引用类型（object）转换为非对象类型，也就是原始类型。

ToPrimitive 运算符接受一个值，和一个可选的 期望类型作参数。ToPrimitive 运算符将值转换为非对象类型，如果对象有能力被转换为不止一种原语类型，可以使用可选的 期望类型 来暗示那个类型。

转换后的结果原始类型是由期望类型决定的，期望类型其实就是我们传递的type。直接看下面比较清楚。 ToPrimitive方法大概长这么个样子具体如下。

\* @obj 需要转换的对象

\* @type 期望转换为的原始数据类型，可选

ToPrimitive(obj,type)

type不同值的说明

type为string:

先调用obj的toString方法，如果为原始值，则return，否则第2步

调用obj的valueOf方法，如果为原始值，则return，否则第3步

抛出TypeError 异常

type为number:

调用obj的valueOf方法，如果为原始值，则返回，否则下第2步

调用obj的toString方法，如果为原始值，则return，否则第3步

抛出TypeError 异常

type参数为空

该对象为Date，则type被设置为String

否则，type被设置为Number

**Date数据类型特殊说明：**

对于Date数据类型，我们更多期望获得的是其转为时间后的字符串，而非毫秒值（时间戳），如果为number，则会取到对应的毫秒值，显然字符串使用更多。 其他类型对象按照取值的类型操作即可。

**ToPrimitive总结**

**ToPrimitive转成何种原始类型，取决于type，type参数可选，若指定，则按照指定类型转换，若不指定，默认根据实用情况分两种情况，Date为string，其余对象为number。那么什么时候会指定type类型呢，那就要看下面两种转换方式了。**

**toString Object.prototype.toString()**

toString() 方法返回一个表示该对象的字符串。

每个对象都有一个 toString() 方法，当对象被表示为文本值时或者当以期望字符串的方式引用对象时，该方法被自动调用。

这里先记住，valueOf() 和 toString() 在特定的场合下会自行调用。

**valueOf**

Object.prototype.valueOf()方法返回指定对象的原始值。

JavaScript 调用 valueOf() 方法用来把对象转换成原始类型的值（数值、字符串和布尔值）。但是我们很少需要自己调用此函数，valueOf 方法一般都会被 JavaScript 自动调用。

**不同内置对象的valueOf实现：**

String => 返回字符串值

Number => 返回数字值

Date => 返回一个数字，即时间值,字符串中内容是依赖于具体实现的

Boolean => 返回Boolean的this值

Object => 返回this

对照代码会更清晰一些：

var str = new String('123');

console.log(str.valueOf());//123

var num = new Number(123);

console.log(num.valueOf());//123

var date = new Date();

console.log(date.valueOf()); //1526990889729

var bool = new Boolean('123');

console.log(bool.valueOf());//true

var obj = new Object({valueOf:()=>{

return 1

}})

console.log(obj.valueOf());//1

**Number**

Number运算符转换规则：

**null 转换为 0**

**undefined 转换为 NaN**

**true 转换为 1，false 转换为 0**

**字符串转换时遵循数字常量规则，转换失败返回 NaN**

注意：对象这里要先转换为原始值，调用ToPrimitive转换，type指定为number了，继续回到ToPrimitive进行转换（看ToPrimitive）。

**String**

String 运算符转换规则

**null 转换为 'null'**

**undefined 转换为 undefined**

**true 转换为 'true'，false 转换为 'false**'

数字转换遵循通用规则，极大极小的数字使用指数形式

注意：对象这里要先转换为原始值，调用ToPrimitive转换，type就指定为string了，继续回到ToPrimitive进行转换（看ToPrimitive）。

String(null) // 'null'

String(undefined) // 'undefined'

String(true) // 'true'

String(1) // '1'

String(-1) // '-1'

String(0) // '0'

String(-0) // '0'

String(Math.pow(1000,10)) // '1e+30'

String(Infinity) // 'Infinity'

String(-Infinity) // '-Infinity'

String({}) // '[object Object]'

String([1,[2,3]]) // '1'

String(['koala',1]) //koala,1

Boolean

**ToBoolean 运算符转换规则**

**除了下述 6 个值转换结果为 false，其他全部为 true：**

Undefined、null、-0、0或+0、NaN、''（空字符串）

假值以外的值都是真值。其中包括所有对象（包括空对象）的转换结果都是true，甚至连false对应的布尔对象new Boolean(false)也是true

Boolean(undefined) // false

Boolean(null) // false

Boolean(0) // false

Boolean(NaN) // false

Boolean('') // false

Boolean({}) // true

Boolean([]) // true

Boolean(new Boolean(false)) // true

**js转换规则不同场景应用**

**面试官问：知道了具体转换成什么的规则，但是都在什么情况下发生什么样的转换呢？**

**什么时候自动转换为string类型**

在没有对象的前提下

**字符串的自动转换，主要发生在字符串的加法运算时**。**当一个值为字符串，另一个值为非字符串，则后者转为字符串。**

'2' + 1 // '21'

'2' + true // "2true"

'2' + false // "2false"

'2' + undefined // "2undefined"

'2' + null // "2null"

当有对象且与对象+时候

//toString的对象

var obj2 = {

toString:function(){ return 'a' }}

console.log('2'+obj2)

//输出结果2a

//常规对象

var obj1 = { a:1,b:2}

console.log('2'+obj1)；

//输出结果 2[object Object]

//几种特殊对象

'2' + {} // "2[object Object]"

'2' + [] // "2"

'2' + function (){} // "2function (){}"

'2' + ['koala',1] // 2koala,1

注意：不管是对象还不是对象，都有一个转换为原始值的过程，也就是ToPrimitive转换，只不过原始类型转换后不发生变化，对象类型才会发生具体转换。

**什么时候自动转换为Number类型**

**有加法运算符，但是无String类型的时候，都会优先转换为Number类型**

例子：

true + 0 // 1

true + true // 2

true + false //1

除了加法运算符，其他运算符都会把运算自动转成数值。例子：

'5' - '2' // 3

'5' \* '2' // 10

true - 1 // 0

false - 1 // -1

'1' - 1 // 0

'5' \* [] // 0

false / '5' // 0

'abc' - 1 // NaN

null + 1 // 1

undefined + 1 // NaN

//一元运算符（注意点）

+'abc' // NaN

-'abc' // NaN

+true // 1

-false // 0

**注意：null转为数值时为0，而undefined转为数值时为NaN。**

**判断等号也放在Number里面特殊说明**

== 抽象相等比较与+运算符不同，不再是String优先，而是Nuber优先。 下面列举x == y的例子

如果x,y均为number，直接比较 没什么可解释的了

1 == 2 //false

如果存在对象，ToPrimitive() type为number进行转换，再进行后面比较

var obj1 = {

valueOf:function(){return '1' }}

1 == obj1 //true

//obj1转为原始值，调用obj1.valueOf()

//返回原始值'1'

//'1'toNumber得到 1 然后比较 1 == 1

[] == ![] //true

//[]作为对象ToPrimitive得到 ''

//![]作为boolean转换得到0

//'' == 0

//转换为 0==0 //true

存在boolean，按照ToNumber将boolean转换为1或者0，再进行后面比较

//boolean 先转成number，按照上面的规则得到1

//3 == 1 false

//0 == 0 true

3 == true // false

'0' == false //true

4.如果x为string，y为number，x转成number进行比较

//'0' toNumber()得到 0

//0 == 0 true

'0' == 0 //true

**什么时候进行布尔转换**

**布尔比较时**

**if(obj) , while(obj) 等判断时或者 三元运算符只能够包含布尔值**

条件部分的每个值都相当于false，使用否定运算符后，就变成了true

**\*\*\*\*js中的数据类型判断 @@@**

**面试官问：如何判断数据类型？怎么判断一个值到底是数组类型还是对象?**

**三种方式，分别为 typeof、instanceof 和 Object.prototype.toString()**

**typeof**

通过 typeof操作符来判断一个值属于哪种基本类型。

typeof 'seymoe' // 'string'

typeof true // 'boolean'

typeof 10 // 'number'

typeof Symbol() // 'symbol'

typeof null // 'object' 无法判定是否为 null

typeof undefined // 'undefined'

typeof {} // 'object'

typeof [] // 'object'

typeof(() => {}) // 'function'

上面代码的输出结果可以看出，

**null 的判定有误差，得到的结果 如果使用 typeof，null得到的结果是object**

**操作符对对象类型及其子类型，例如函数（可调用对象）、数组（有序索引对象）等进行判定，则除了函数都会得到 object 的结果。**

**综上可以看出typeOf对于判断类型还有一些不足，在对象的子类型和null情况下。**

**instanceof**

**通过 instanceof 操作符也可以对对象类型进行判定，其原理就是测试构造函数的 prototype 是否出现在被检测对象的原型链上。**

[] instanceof Array // true

({}) instanceof Object // true

(()=>{}) instanceof Function // true

复制代码注意：instanceof 也不是万能的。 举个例子：

let arr = []

let obj = {}

arr instanceof Array // true

arr instanceof Object // true

obj instanceof Object // true

在这个例子中，**arr 数组相当于 new Array() 出的一个实例，所以 arr.proto === Array.prototype，又因为 Array 属于 Object 子类型，即 Array.prototype.proto === Object.prototype，所以 Object 构造函数在 arr 的原型链上。所以 instanceof 仍然无法优雅的判断一个值到底属于数组还是普通对象。**

还有一点需要说明下，有些开发者会说 Object.prototype.proto === null，岂不是说 arr instanceof null 也应该为 true，这个语句其实会报错提示右侧参数应该为对象，这也印证 typeof null 的结果为 object 真的只是javascript中的一个 bug 。

**Object.prototype.toString()** 可以说是判定 JavaScript 中数据类型的终极解决方法了，具体用法请看以下代码：

Object.prototype.toString.call({}) // '[object Object]'

Object.prototype.toString.call([]) // '[object Array]'

Object.prototype.toString.call(() => {}) // '[object Function]'

Object.prototype.toString.call('seymoe') // '[object String]'

Object.prototype.toString.call(1) // '[object Number]'

Object.prototype.toString.call(true) // '[object Boolean]'

Object.prototype.toString.call(Symbol()) // '[object Symbol]'

Object.prototype.toString.call(null) // '[object Null]'

Object.prototype.toString.call(undefined) // '[object Undefined]'

Object.prototype.toString.call(new Date()) // '[object Date]'

Object.prototype.toString.call(Math) // '[object Math]'

Object.prototype.toString.call(new Set()) // '[object Set]'

Object.prototype.toString.call(new WeakSet()) // '[object WeakSet]'

Object.prototype.toString.call(new Map()) // '[object Map]'

Object.prototype.toString.call(new WeakMap()) // '[object WeakMap]'

我们可以发现该方法在传入任何类型的值都能返回对应准确的对象类型。用法虽简单明了，但其中有几个点需要理解清楚：

**该方法本质就是依托Object.prototype.toString() 方法得到对象内部属性 [[Class]]**

**传入原始类型却能够判定出结果是因为对值进行了包装**

**null 和 undefined 能够输出结果是内部实现有做处理**

NaN相关总结

**NaN的概念**

**NaN 是一个全局对象的属性，NaN 是一个全局对象的属性，NaN是一种特殊的Number类型。**

**什么时候返回NaN （开篇第二道题也得到解决）**

**无穷大除以无穷大**

**给任意负数做开方运算**

**算数运算符与不是数字或无法转换为数字的操作数一起使用**

**字符串解析成数字**

一些例子：

Infinity / Infinity; // 无穷大除以无穷大

Math.sqrt(-1); // 给任意负数做开方运算

'a' - 1; // 算数运算符与不是数字或无法转换为数字的操作数一起使用

'a' \* 1;

'a' / 1;

parseInt('a'); // 字符串解析成数字

parseFloat('a');

Number('a'); //NaN

'abc' - 1 // NaN

undefined + 1 // NaN

//一元运算符（注意点）

+'abc' // NaN

-'abc' // NaN

**误区**

**toString和String的区别**

**toString**

**toString()可以将数据都转为字符串，但是null和undefined不可以转换。**

console.log(null.toString())

//报错 TypeError: Cannot read property 'toString' of null

console.log(undefined.toString())

//报错 TypeError: Cannot read property 'toString' of undefined

toString()括号中可以写数字，代表进制

二进制：.toString(2);

八进制：.toString(8);

十进制：.toString(10);

十六进制：.toString(16);

**String**

**String()可以将null和undefined转换为字符串，但是没法转进制字符串**

console.log(String(null));

// null

console.log(String(undefined));

// undefined

**循环：**

**1、for 循环**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | let arr = [1,2,3];  for (let i=0; i<arr.length; i++){   console.log(i,arr[i])  }  // 0 1  // 1 2  // 2 3 |

for 循环是 Js 中最常用的一个循环工具，经常用于数组的循环遍历。

**2、for in 循环**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | let obj = {name:'zhou',age:'\*\*'}  for(let i in obj){   console.log(i,obj[i])  }  // name zhou  // age \*\* |

for in 循环主要用于遍历普通对象，i 代表对象的 key 值，obj[i] 代表对应的 value,当用它来遍历数组时候，多数情况下也能达到同样的效果，但是你不要这么做，这是有风险的，因为 i 输出为字符串形式，而不是数组需要的数字下标，这意味着在某些情况下，会发生字符串运算，导致数据错误，比如：'52'+1 = '521' 而不是我们需要的 53。

另外 for in 循环的时候，不仅遍历自身的属性，还会找到 prototype 上去，所以最好在循环体内加一个判断，就用 obj[i].hasOwnProperty(i)，这样就避免遍历出太多不需要的属性。

**3、while 循环**

同样的遍历 cars 数组，先用 for 循环方法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | let cars=["BMW","Volvo","Saab","Ford"];  let i=0;  for (;cars[i];)  {  console.log(cars[i])  i++;  };  // BMW  // Volvo  // Saab  // Ford |

然后是 while 循环方法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | cars=["BMW","Volvo","Saab","Ford"];  var i=0;  while (cars[i])  {  console.log(cars[i] + "<br>")  i++;  }; |

我们发现，它们可以实现同样的效果，事实上它们底层的处理是一样的，不过 for 循环可以把定义、条件判断、自增自减操作放到一个条件里执行，代码看起来方便一些，仅此而已。

**4、do while 循环**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | let i = 3;  do{   console.log(i)   i--;  }  while(i>0)  // 3  // 2  // 1 |

do while 循环是 while 循环的一个变体，它首先执行一次操作，然后才进行条件判断，是 true 的话再继续执行操作，是 false 的话循环结束。

**5、Array forEach 循环**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | let arr = [1,2,3];  arr.forEach(function(i,index){   console.log(i,index)  })  // 1 0  // 2 1  // 3 2 |

forEach循环，循环数组中每一个元素并采取操作， 没有返回值， 可以不用知道数组长度,他有三个参数，只有第一个是必需的，代表当前下标下的 value。

另外请注意，forEach 循环在所有元素调用完毕之前是不能停止的，它没有 break 语句，如果你必须要停止，可以尝试 try catch 语句，就是在要强制退出的时候，抛出一个 error 给 catch 捕捉到，然后在 catch 里面 return，这样就能中止循环了，如果你经常用这个方法，最好自定义一个这样的 forEach 函数在你的库里。

**6、Array map()方法**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | let arr = [1,2,3];  let tt = arr.map(function(i){   console.log(i)   return i\*2;  })  // [2,4,6] |

map() 方法返回一个新数组，数组中的元素为原始数组元素调用函数处理后的值。  
注意：map 和 forEach 方法都是只能用来遍历数组，不能用来遍历普通对象。

**7、Array filter() 方法**

|  |  |
| --- | --- |
| 1  2  3  4  5 | let arr = [1,2,3];  let tt = arr.filter(function(i){   return i>1;  })  // [2,3] |

filter 方法是 Array 对象内置方法，它会返回通过过滤的元素，不改变原来的数组。

**8、Array some() 方法**

|  |  |
| --- | --- |
| 1  2  3  4  5 | let arr = [1,2,3];  let tt = arr.some(function(i){   return i>1;  })  // true |

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）,返回 boolean 值，不改变原数组。

**9、Array every() 方法**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | let arr = [1,2,3];  let tt = arr.some(function(i){   return i>1;  })  // 检测数组中元素是否都大于1  // false |

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），返回 boolean 值，不改变原数组。

**10、Array reduce()方法**

|  |  |
| --- | --- |
| 1  2  3  4  5 | let arr = [1,2,3];  let ad = arr.reduce(function(i,j){   return i+j;  })  // 6 |

reduce() 方法接收一个函数作为累加器，数组中的每个值（从左到右）开始缩减，最终计算为一个值。

**11、Array reduceRight()方法**

|  |  |
| --- | --- |
| 1  2  3  4  5 | let arr = [1,2,3];  let ad = arr.reduceRight(function(i,j){   return i+j;  })  // 6 |

reduceRight()方法,和 reduce() 功能是一样的，它是从数组的末尾处向前开始计算。

**12、for of 循环**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | let arr = ['name','age'];  for(let i of arr){   console.log(i)  }  // name  // age |

for of 循环是 Es6 中新增的语句，用来替代 for in 和 forEach，它允许你遍历 Arrays（数组）, Strings（字符串）, Maps（映射）, Sets（集合）等可迭代(Iterable data)的数据结构,注意它的兼容性。

**For in:**

可以看出for in 应用于数组循环返回的是数组的下标和数组的属性和原型上的方法和属性

而for in应用于对象循环返回的是对象的属性名和原型中的方法和属性。

**使用for in 也可以遍历数组，但是会存在以下问题：**

1.index索引为字符串型数字，**不能直接进行几何运算**

2.遍历顺序有可能不是按照实际数组的内部顺序

3.使用for in会遍历数组所有的可枚举属性，包括原型。例如上栗的原型方法method和name属性

//如果不想让其输出原型中的属性和方法，可以使用**hasOwnProperty**方法进行过滤 for(let index in myObject){

if(myObject.hasOwnProperty(index)){

console.log(index) }

} //输出结果为 name,age

**for of遍历的是数组元素值**。

所以**for in更适合遍历对象**，不要使用for in遍历数组。

**总结**

以上就是我总结的 Js 中常见的循环遍历方法，随着 Es6 标准的兼容性越来越好，我发现很多实现方案慢慢都不再必要了，比如 let、const 取代var 后，在某些情况下的闭包函数也就不存在了。