[7000字]JavaScript数组所有方法基础总结

基础决定一个人的上限，很多时候我们感叹别人在实现一个功能时使用方法的精妙，并且反思，为什么别人想的出来自己却想不出来？我觉得主要是因为对于基础的掌握上有很大的差距。本文总结数组的所有方法的基础使用，希望对你有帮助，如果有错误，请指出。

ES6新增构造方法

Array.form()

语法

> Array.from(arrayLike[, mapFn[, thisArg]])

作用

将类数组与可迭代对象转化为数组

参数

```
arrayLike
    想要转换成数组的伪数组对象或可迭代对象。
mapFn 可选
    如果指定了该参数，新数组中的每个元素会执行该回调函数。
thisArg 可选
    可选参数，执行回调函数 mapFn 时 this 对象。
复制代码
```

返回值

转化后的新数组

注意事项

```
1、该类数组对象必须具有 length 属性，用于指定数组的长度。如果没有 length 属性，那么  转换后的数组是一个空数组。
    // 没有length属性返回空数组
    let obj = {
        0:'a',
        1:'b',
        2:'c'
    }
    Array.from(obj); // []
复制代码
2、该类数组对象的属性名必须为数值型或字符串型的数字，且数字必须是以0开始
    // key不是数字或包含数字的字符串返回undefined
    let obj = {
        name:'九九欧',
```

```
        age:18,
        length:2
    }
    Array.from(obj); // [undefined, undefined]

    // key是数字但不是以0开始
    let obj = {
        2:'a',
        3:'b',
        4:'c',
        length:3
    }
    Array.from(obj); // [undefined, undefined, "a"]

    // 满足所有条件
    let obj = {
        0:'a',
        1:'b',
        2:'c',
        length:3
    }
    Array.from(obj); // ["a","b","c"]
复制代码
Array.from()在转化对象时，要求过于苛刻，因此不适用于转化对象，它的应用场景主要是以下几个
从类数组对象(arguments)生成数组
let fn = function(){
        console.log(Array.from(arguments));
    }
    fn(1,2,3) // [1,2,3]


从 String 生成数组
    Array.from('九九欧'); // ["九","九","欧"]


从Set生成数组
    Array.from(new Set(["九","九","欧","欧"])); // ["九","欧"]


从Map生成数组
    Array.from(new Map([[1, 'a'], [2, 'b']])); // [[1, 'a'], [2, 'b']]


生成一个从0开始到指定数字的定长连续数组
    Array.from({length: 10}, (v, i) => i); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
复制代码
```

Array.of()

语法

> Array.of(element0[, element1[, ...[, elementN]]])

作用

将参数依次转化为数组中的项

参数

任意个参数，将按顺序成为返回数组中的元素

返回值

转化后的新数组

Array.of()与Array()的区别

当传入多个参数时，两者之间没有区别

当传入一个参数且参数为数字时，区别如下

```
    Array.of(9);
    // [9]
    Array(9);
    // [empty × 9]
复制代码
```

判断数组的方法

Array.isArray()

语法

> Array.isArray(obj)

作用

判断传递的值是否是一个数组

参数

需要判断的值

返回值

布尔值

实例

```
    Array.isArray([])           // true
    Array.isArray({})           // false
    Array.isArray('a')          // false
    Array.isArray(1)            // false
    Array.isArray(null)         // false
    Array.isArray(undefined)    // false
```
复制代码

改变自身值的方法

pop

语法

> arr.pop()

作用

删除数组中最后一个元素

参数

无

返回值

从数组中删除的元素（数组为空时，返回`undefined`）

实例

```
    let ary = [1,2,3,4];
    console.log(ary.pop()); // 4
    console.log(ary);       // [1,2,3]
```
复制代码

push

语法

> arr.push(element1, ..., elementN)

作用

将一个或多个元素添加到数组的末尾

参数

被添加到数组末尾的元素

返回值

```
新数组的长度
```

实例

```
    let ary = [1,2,3];
    ary.push(4);
    console.log(ary); // [1,2,3,4]
复制代码
```

应用

```
合并两个数组
```

思路：利用apply改变this的指向，以及apply传递参数时会将数组中的项以此传递

```
    let ary1 = [1,2,3];
    let ary2 = [4,5,6];
    let ary3 = Array.prototype.push.apply(ary1,ary2)
    console.log(ary1); // [1,2,3,4,5,6]
    console.log(ary3); // 6
复制代码
```

shift

语法

> arr.shift()

作用

```
删除数组的第一个元素
```

参数

```
无
```

返回值

```
被删除的元素
```

实例

```
    let ary = [1,2,3];
    let result = ary.shift();
    console.log(ary);     // [2,3]
    console.log(result);  // 1
复制代码
```

unshift

语法

> arr.unshift(element1, ..., elementN)

作用

在数组的开头添加一个或多个元素

参数

要添加到数组开头的元素或多个元素

返回值

新数组的长度

实例

```
    let ary = [4, 5, 6];
    let result = ary.unshift(1, 2, 3);
    console.log(ary);    // [1, 2, 3, 4, 5, 6]
    console.log(result); // 6
复制代码
```

sort

语法

> arr.sort([compareFunction])

作用

数组的元素进行排序

参数

```
compareFunction 可选
  用来指定按某种顺序进行排列的函数。
  如果省略，则元素按照转换为的字符串的各个字符的Unicode位点进行排序。
firstEl
  第一个用于比较的元素。
secondEl
  第二个用于比较的元素。
复制代码
```

返回值

排序后的数组

实例

若 comparefn(a, b) < 0，那么a 将排到 b 前面（数字升序）

```
    let ary = [2,4,1,6,7,3,8,9,5];
    ary.sort(function(a,b){
        return a-b;
    })
    console.log(ary); // [1,2,3,4,5,6,7,8,9]
复制代码
```

若 comparefn(a, b) = 0，那么a 和 b 相对位置不变

```
    let ary = [2,4,1,6,7,3,8,9,5];
    ary.sort(function(a,b){
        return a-a;
    })
    console.log(ary); // [2,4,1,6,7,3,8,9,5]
复制代码
```

若 comparefn(a, b) > 0，那么a , b 将调换位置（数字降序）

```
    let ary = [2,4,1,6,7,3,8,9,5];
    ary.sort(function(a,b){
        return b-a;
    })
    console.log(ary); // [9,8,7,6,5,4,3,2,1]
复制代码
```

省略参数时，元素按照转换为的字符串的各个字符的Unicode位点进行排序

```
    let ary = [9,8,7,6,5,4,3,2,1,0]
    ary.sort();
    console.log(ary);  // [0,1,2,3,4,5,6,7,8,9]

    let ary = ["e","d","c","b","a"]
    ary.sort();
    console.log(ary); // ["a","b","c","d","e"]
复制代码
```

splice

语法

> array.splice(start[, deleteCount[, item1[, item2[, ...]]]])

作用

删除或替换现有元素、原地添加新的元素

参数

```
start
    指定修改的开始位置（从0计数）。如果超出了数组的长度，则从数组末尾开始添加内容；
    如果是负值，则表示从数组末位开始的第几位（从-1计数，这意味着-n是倒数第n个元素并且等价于array.length-n）；
    如果负数的绝对值大于数组的长度，则表示开始位置为第0位。
deleteCount （可选）
    整数，表示要移除的数组元素的个数。
    如果 deleteCount 大于 start 之后的元素的总数，则从 start 后面的元素都将被删除（含第 start 位）。
    如果 deleteCount 被省略了，或者它的值大于等于array.length - start
    (也就是说如果它大于或者等于start之后的所有元素的数量)，那么start之后数组的所有元素都会被删除。
    如果 deleteCount 是 0 或者负数，则不移除元素。这种情况下，至少应添加一个新元素。
item1, item2, ...（可选）
    要添加进数组的元素,从start 位置开始。如果不指定，则 splice() 将只删除数组元素。
复制代码
```

返回值

```
由被删除的元素组成的一个数组。如果只删除了一个元素，则返回只包含一个元素的数组。如果没有删除元素，则返回空数组。
```

实例

删除某个元素

```
    let ary = [1,2,3,3,4,5,6]
    ary.splice(2,1);
    console.log(ary); // [1,2,3,4,5,6]
复制代码
```

替换某个元素

```
    let ary = [1,2,3,3,5,6]
    ary.splice(3,1,4);
    console.log(ary); // [1,2,3,4,5,6]
复制代码
```

删除最后n个元素

```
    let ary = [1,2,3,3,5,6]
    ary.splice(-2);
    console.log(ary); // [1,2,3,4]
复制代码
```

reverse

语法

> arr.reverse()

作用

颠倒数组中元素的位置

参数

无

返回值

颠倒后的数组

实例

```
    let ary = [1,2,3,4,5]
    ary.reverse();
    console.log(ary); // [5,4,3,2,1]
复制代码
```

copyWith

语法

> arr.copyWithin(target[, start[, end]])

作用

复制数组的一部分到同一数组中的另一个位置

参数

```
target
    0 为基底的索引，复制序列到该位置。
    如果是负数，target 将从末尾开始计算。
    如果 target 大于等于 arr.length，将会不发生拷贝。
    如果 target 在 start 之后，复制 的序列将被修改以符合 arr.length。
start
    0 为基底的索引，开始复制元素的起始位置。
    如果是负数，start 将从末尾开始计算。
    如果 start 被忽略，copyWithin 将会从0开始复制。
end
    0 为基底的索引，开始复制元素的结束位置。
    copyWithin 将会拷贝到该位置，但不包括end这个位置的元素。
    如果是负数，end将从末尾开始计算。
    如果end被忽略，copyWithin方法将会一直复制至数组结尾（默认为 arr.length）。
复制代码
```

返回值

改变后的数组

实例

```
    [1, 2, 3, 4, 5].copyWithin(-2)         // [1, 2, 3, 1, 2]

    [1, 2, 3, 4, 5].copyWithin(0, 3)       // [4, 5, 3, 4, 5]

    [1, 2, 3, 4, 5].copyWithin(0, 3, 4)    // [4, 2, 3, 4, 5]

    [1, 2, 3, 4, 5].copyWithin(-2, -3, -1) // [1, 2, 3, 3, 4]
```
复制代码

fill

语法

> arr.fill(value[, start[, end]])

作用

用一个固定值填充一个数组中从起始索引到终止索引内的全部元素。不包括终止索引

参数

```
value
    用来填充数组元素的值。
start （可选）
    起始索引，默认值为0。
end （可选）
    终止索引，默认值为 this.length（不包含终止索引）
```
复制代码

返回值

修改后的数组

实例

```
    const ary = [1, 2, 3, 4];
    console.log(ary.fill(0, 2, 4)); // [1, 2, 0, 0]

    const ary = [1, 2, 3, 4];
    console.log(ary.fill(5, 1));    // [1, 5, 5, 5]

    const ary = [1, 2, 3, 4];
    console.log(ary.fill(6));       // [6, 6, 6, 6]
```
复制代码

省略终止索引

```
    const ary = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    console.log(ary.fill(0, 2)); // [1, 2, 0, 0, 0, 0, 0, 0, 0, 0]
复制代码
```

不改变自身值的方法

concat

语法

> var new_array = old_array.concat(value1[, value2[, ...[, valueN]]])

作用

```
合并两个或多个数组
```

参数

```
valueN可选
    将数组和/或值连接成新数组。
    如果省略了valueN参数参数，则concat会返回一个它所调用的已存在的数组的浅拷贝。
复制代码
```

返回值

```
新的 Array 实例
```

实例

合并两个数组

```
    const ary1 = ['九','九'];
    const ary2 = ['欧'];
    let ary3 = ary1.concat(ary2);
    console.log(ary3); // ['九','九','欧']
复制代码
```

合并三个数组

```
    const ary1 = [1,2,3];
    const ary2 = [4,5,6];
    const ary3 = [7,8,9];
    let ary4 = ary1.concat(ary2,ary3);
    console.log(ary4); // [1,2,3,4,5,6,7,8,9]
复制代码
```

将值连接到数组

```
    const ary1 = [1,2,3];
    const ary2 = [7,8,9];
    let ary3 = ary1.concat(4,5,6,ary2);
    console.log(ary3); // [1,2,3,4,5,6,7,8,9]
    }
复制代码
```

flat

语法

> var newArray = arr.flat([depth])

作用

将多维数组转化为一维数组

参数

指定要提取嵌套数组的结构深度，默认值为 1

返回值

一个包含将数组与字数组中所有元素的新数组

实例

```
    let ary = [1, 2, [3, [4,[5,[6,[7]]]]]];
    console.log(ary.flat());          // [1,2,3,[4,[5,[6,[7]]]]]
    console.log(ary.flat(2));         // [1,2,3,4,[5,[6,[7]]]]
    console.log(ary.flat(3));         // [1,2,3,4,5,[6,[7]]]
// 传入的参数为“Infinity”时，无论是几维数组都会变成1维数组
    console.log(ary.flat(Infinity));  // [1,2,3,4,5,6,7]
复制代码
```

flat方法会移除数组中的空项

```
    let ary = [1, 2, , 4, 5];
    console.log(ary.flat()); // [1, 2, 4, 5]
复制代码
```

flatMap

语法

> var new_array = arr.flatMap(function callback(currentValue[, index[, array]]) { // return element for new_array }[, thisArg])

作用

使用映射函数映射每个元素，然后将结果压缩成一个新数组

参数

```
callback：
    可以生成一个新数组中元素的函数，包含一下三个参数：
currentValue：
    当前正在数组中处理的元素
index(可选)：
    数组中正在处理的当前元素的索引
array：
    被调用map的数组
thisAry（可选）：
    执行callback函数值，this的值
复制代码
```

返回值

新数组

实例

```
    let ary = [1, 2, 3, 4];
    console.log(ary.flatMap(x => [x * 2]));    // [2, 4, 6, 8]
    console.log(ary.flatMap(x => [[x * 2]]));  // [[2], [4], [6], [8]]
复制代码
```

工作原理

先map()再flat()

```
    let names = ['九', '九', '欧'];
    // 步骤 1: map
    let result = names.map((name, index) => [name, index]);
    // [ ['九', 1], ['九', 2 ], ['欧', 3]]
    // 步骤 2: flat
    result.flat();
    // [ '九', 1, '九', 2 ,'欧',3]

    let names = ['九', '九', '欧'];
    let result = names.flatMap((name, index) => [name, index]);
    // [ '九', 1, '九', 2 ,'欧',3]
复制代码
```

join

语法

> arr.join([separator])

作用

将一个数组（或一个类数组对象）的所有元素连接成一个字符

参数

```
separator 可选
    指定一个字符串来分隔数组的每个元素。
    如果需要，将分隔符转换为字符串。
    如果缺省该值，数组元素用逗号（,）分隔。
    如果separator是空字符串(""),则所有元素 之间都没有任何字符。
复制代码
```

返回值

一个所有数组元素连接的字符串。如果 arr.length 为0，则返回空字符串

注意事项

如果一个元素为 undefined 或 null，它会被转换为空字符串

实例

```
    const ary = ['九', '九', '欧'];
    let ary = ary.join();       // "九,九,欧"
    let ary = ary.join(', ');   // "九, 九, 欧"
    let ary = ary.join(' + ');  // "九 + 九 + 欧"
    let ary = ary.join('');     // "九九欧"
复制代码
```

slice

语法

> arr.slice([begin[, end]])

作用

将数组中一部分元素浅复制存入新的数组对象

参数

```
begin （可选）
    提取起始处的索引（从 0 开始），从该索引开始提取原数组元素。
    如果该参数为负数，则表示从原数组中的倒数第几个元素开始提取。
    slice(-2) 表示提取原数组中的倒数第二个元素到最后一个元素（包含最后一个元素）。
    如果省略 begin，则 slice 从索引 0 开始。
    如果 begin 大于原数组的长度，则会返回空数组。
end （可选）
    提取终止处的索引（从 0 开始），在该索引处结束提取原数组元素。
    slice 会提取原数组中索引从 begin 到 end 的所有元素（包含 begin，但不包含 end）。
```

slice(1,4) 会提取原数组中从第二个元素开始一直到第四个元素的所有元素 （索引 为 1, 2, 3的元素）。
如果该参数为负数，则它表示在原数组中的倒数第几个元素结束抽取。
slice(-2,-1) 表示抽取了原数组中的倒数第二个元素到最后一个元素（不包含最后一个元素，也就是只有倒数第二个元素）。
如果 end 被省略，则 slice 会一直提取到原数组末尾。
如果 end 大于数组的长度，slice 也会一直提取到原数组末尾。
复制代码

返回值

一个含有被提取元素的新数组

实例

```
const ary = [1,2,3,4,5];
console.log(ary.slice(1,4));    // [2,3,4]
console.log(ary.slice(0,100));  // [1,2,3,4,5]
console.log(ary.slice(50,100)); // []
```
复制代码

应用

将类数组转化为数组

```
function fn() {
    return Array.prototype.slice.call(arguments);
}
console.log(fn(1, 2, 3)); // [1, 2, 3]
```
复制代码

toString

语法

> arr.toString()

作用

返回数组的字符串形式

参数

无

返回值

数组的字符串形式

实例

```
    const array = ['九', '九', '欧'];
    let str = array.toString();
    console.log(str);        // 九,九,欧
    console.log(typeof str); // string
复制代码
```

toLocaleString

语法

> arr.toLocaleString([locales[,options]]);

作用

返回一个字符串表示数组中的元素。数组中的元素将使用各自的 `toLocaleString` 方法转成字符串，这些字符串将使用一个特定语言环境的字符串（如逗号 ","）隔开

参数

```
locales 可选
    带有BCP 47语言标记的字符串或字符串数组。
options 可选
    一个可配置属性的对象，对于数字 Number.prototype.toLocaleString()
    对于日期     Date.prototype.toLocaleString().
复制代码
```

返回值

表示数组元素的字符串

indexOf

语法

> arr.indexOf(searchElement[, fromIndex])

作用

返回在数组中可以找到一个给定元素的第一个索引，如果不存在，则返回-1

参数

```
searchElement
    要查找的元素
fromIndex 可选
    开始查找的位置。
    如果该索引值大于或等于数组长度，意味着不会在数组里查找，返回-1。
    如果参数中提供的索引值是一个负值，则将其作为数组末尾的一个抵消，即-1表示从最后一个元素开始查找，-2表示从倒数第二个元素开始查找，以此类推。
    注意：如果参数中提供的索引值是一个负值，并不改变其查找顺序，查找顺序仍然是从前向后查询数组。
    如果抵消后的索引值仍小于0，则整个数组都将会被 查询。其默认值为0.
复制代码
```

返回值

```
首个被找到的元素在数组中的索引位置; 若没有找到则返回 -1
```

实例

```
    const ary = [5,6,7,8,5];
    ary.indexOf(5);     // 0
    ary.indexOf(1);     // -1
    ary.indexOf(5, 2);  // 4
    ary.indexOf(5, -2); // 4
复制代码
```

lastIndexOf

语法

> arr.lastIndexOf(searchElement[, fromIndex])

作用

```
查找元素在数组中最后一次出现时的索引，如果没有，则返回-1
```

参数

```
searchElement
    被查找的元素。
fromIndex 可选
    从此位置开始逆向查找。默认为数组的长度减 1(arr.length - 1)，即整个数组都被查 找。
    如果该值大于或等于数组的长度，则整个数组会被查找。如果为负值，将其视为从数组末尾向前的偏移。即使该值为负，数组仍然会被从后向前查找。
    如果该值为负时，其绝对值大于数组长度，则方法返回 -1，即数组不会被查找。
复制代码
```

返回值

```
数组中该元素最后一次出现的索引，如未找到返回-1
```

实例

```
    const ary = [5, 6, 7, 8, 5];
    ary.lastIndexOf(5);     // 4
    ary.lastIndexOf(1);     // -1
    ary.lastIndexOf(5, 2);  // 0
    ary.lastIndexOf(5, -2); // 0
复制代码
```

includes

语法

> arr.includes(valueToFind[, fromIndex])

作用

判断一个数组是否包含一个指定的值

参数

```
valueToFind
    需要查找的元素值。
fromIndex 可选
    从fromIndex 索引处开始查找 valueToFind。
    如果为负值，则按升序从 array.length    + fromIndex 的索引开始搜
    （即使从末尾开始往前跳 fromIndex 的绝对值个索引，   然后往后搜寻）默认为 0。
复制代码
```

返回值

布尔值

注意事项

使用 includes()比较字符串和字符时区分大小写` `includes方法与indexOf方法的功能有些类似，不同点在于includes方法可以找到NaN而indexOf不行

实例

```
    [1, 2, 3].includes(2);                 // true
    [1, 2, 3].includes(4);                 // false
    [1, 2, null].includes(null);           // true
    [1, 2, undefined].includes(undefined); // true
    [1, 2, NaN].includes(NaN);             // true
复制代码
```

toSource

语法

> array.toSource()

作用

返回一个代表该数组的源代码的字符串

参数

无

返回值

代表数组源码的字符串

实例

```
    var array = ['a', 'b', 'c'];
    console.log(array.toSource()); // ["a", "b", "c"]
复制代码
```

遍历数组的方法

forEach

语法

> arr.forEach(callback(currentValue [, index [, array]])[, thisArg])

作用

对数组的每个元素执行一次给定的函数

参数

```
callback
    为数组中每个元素执行的函数，该函数接收一至三个参数：
currentValue
    数组中正在处理的当前元素。
index 可选
    数组中正在处理的当前元素的索引。
array 可选
    forEach() 方法正在操作的数组。
thisArg 可选
    当执行回调函数 callback 时，用作 this 的值。
复制代码
```

返回值

```
undefined
```

注意事项

1、forEach() 遍历的范围在第一次调用 callback 前就会确定。调用 forEach 后添加到数组中的项不会被 callback 访问到

```
    const ary = [1,2,3,4,5];
    ary.forEach(item =>{
        if(item === 1){
            ary.push(6) // 遍历开始后数组新增的项不会被遍历
        }
        console.log(item);
    })
    // 1
    // 2
    // 3
    // 4
    // 5
```

复制代码

2、如果已经存在的值被改变，则传递给 callback 的值是 forEach() 遍历到他们那一刻的值。已删除的项不会被遍历到

```
    const ary = [1,2,3,4,5];
    ary.forEach(item =>{
        if(item === 1){
            ary.pop()//在遍历到数组第一项时就把最后一项删除了，被删除的项不会被遍历
        }
        console.log(item);
    })
    // 1
    // 2
    // 3
    // 4
```

复制代码

3、如果已访问的元素在迭代时被删除了（例如使用 shift()），之后的元素将被跳过

```
    const ary = [1,2,3,4,5];
    ary.forEach(item =>{
        if(item === 3){
            ary.shift();//遍历到第2项时，以访问的元素被删除，则下一项跳过遍历
        }
        console.log(item);
    })
    // 1
    // 2
    // 3
    // 5
```

复制代码

4、forEach() 为每个数组元素执行一次 callback 函数；与 map() 或者 reduce() 不同的是，它总是返回 undefined 值，并且不可链式调用。因为此特性，forEach一般用于链式调用的结尾。

```
    let ary = [1, 2, 3, 4, 5]
    let result = ary.forEach(item => {})
    console.log(result) // undefined
```

复制代码

5、除了抛出异常以外，没有办法中止或跳出 forEach() 循环。下面是for与forEach的对比。

```
    const ary = [1, 2, 3, 4, 5];
    for (let i = 0; i < ary.length; i++) {
        if(ary[i] === 3){
            console.log("找到了");
            break;
        }
```

```
        console.log(ary[i]);
    }
    // 1
    // 2
    // 找到了
    ary.forEach(item => {
        if(item === 3){
            console.log("找到了");
        }
        console.log(item);
    })
    // 1
    // 2
    // 找到了
    // 3
    // 4
    // 5
```
复制代码

6、forEach()虽然不能跳出整个循环，但是可以跳出当前循环，作用与循环中的continue类似

```
    const ary = [1, 2, 3, 4, 5];
    ary.forEach(item => {
        if (item === 3) {
            console.log("找到了");
            return;
        }
        console.log(item);
    })
    // 1
    // 2
    // 找到了
    // 4
    // 5
```
复制代码

7、forEach()中不可以使用break或者continue，否则会报错` `8、forEach()不对未初始化的值进行任何操作

```
    const ary = [1, 2, , 4];
    ary.forEach(item => {
        console.log(element);
    });
    // 1
    // 2
    // 4
```
复制代码

9、使用抛出异常的方式终止forEach()循环

```
    try {
        let ary = [1, 2, 3, 4, 5];
        ary.forEach(item => {
            if (item === 3) {
                throw new Error("EndIterative");
            }
            console.log(item);
        });
    } catch (e) {
        if (e.message != "EndIterative") {
            throw e;
        }
    }
    // 1
```

```
    // 2
复制代码
```

应用

```
数组扁平化
    function flatten(arr) {
        const result = [];
        arr.forEach((item) => {
            if (Array.isArray(item))
                result.push(...flatten(item));
            else
                result.push(item);
        })
        return result;
    }
复制代码
```

filter

语法

> var newArray = arr.filter(callback(element[, index[, array]])[, thisArg])

作用

使用传入的函数测试所有元素，并返回所有通过测试的元素组成的新数组

参数

```
callback
    用来测试数组的每个元素的函数。返回 true 表示该元素通过测试，保留该元素， false 则不保留。它接受以下三个参数：
element
    数组中当前正在处理的元素。
index可选
    正在处理的元素在数组中的索引。
array可选
    调用了 filter 的数组本身。
thisArg可选
    执行 callback 时，用于 this 的值。
复制代码
```

返回值

由通过测试的元素组成的新数组，如果没有元素通过测试，则返回空数组

实例

```
    var ary1 = [1, 2, 3, 4, 5];
    var ary2 = ary1.filter(value => value > 3);
    console.log(ary2); // [4,5]
复制代码
```

应用

```
1、数组对象的键值搜索
    let users = [
    { 'user': '九九', 'age': 18 },
    { 'user': '八八', 'age': 19 },
    { 'user': '七七', 'age': 20 },
    ]
    let filtered = users.filter(n => n.age===18)
    console.log(filtered)   // [{'user': '九九'}]
复制代码
2、数组去重
    let ary = [1,1,2,2,3,3,4,4,5,5]
    let result = ary.filter((item, index, arr) => arr.indexOf(item) === index)
    // indexOf只返回找到的第一个值的下标
    console.log(result); // [1,2,3,4,5]
复制代码
```

`3、去除空字符串以及null和undefined` 思路：利用了空字符串和null、undefined转化为boolean的结果为false

```
    let ary = ['A', '', 'B', null, undefined, 'C', '  ']
    let result = ary.filter((item, idx, arr) => item && item.trim())
    console.log(result) //["A", "B", "C"]
复制代码
```

map

语法

> var new_array = arr.map(function callback(currentValue[, index[, array]]) { // Return element for new_array }[, thisArg])

作用

创建一个新数组，其结果是该数组中的每个元素是调用一次提供的函数后的返回值

参数

```
callback
    生成新数组元素的函数，使用三个参数：
currentValue
    callback 数组中正在处理的当前元素。
index可选
    callback 数组中正在处理的当前元素的索引。
array可选
    map 方法调用的数组。
thisArg可选
    执行 callback 函数时值被用作this。
复制代码
```

返回值

一个由原数组每个元素执行回调函数后的结果组成的新数组

实例

```
    let numbers = [1, 4, 9];
    let result = numbers.map(Math.sqrt);

    let numbers = [1, 4, 9];
    let result = numbers.map(num => num * 2);
    console.log(result); // [2,8,18]
复制代码
```

every

语法

> arr.every(callback[, thisArg])

作用

测试一个数组内的所有元素是否都能通过某个指定函数的测试

参数

```
callback
    用来测试每个元素的函数，它可以接收三个参数：
element
    用于测试的当前值。
index可选
    用于测试的当前值的索引。
array可选
    调用 every 的当前数组。
thisArg
    执行 callback 时使用的 this 值。
复制代码
```

返回值

布尔值

实例

```
    let arr1 = [12, 5, 8, 130, 44];
    let arr2 = [12, 54, 18, 130, 44];
    arr1.every(x => x >= 10); // false
    arr2.every(x => x >= 10); // true
复制代码
```

some

语法

> arr.some(callback(element[, index[, array]])[, thisArg])

作用

测试数组中是不是至少有1个元素通过了被提供的函数测试

参数

```
callback
    用来测试每个元素的函数，接受三个参数：
element
    数组中正在处理的元素。
index 可选
    数组中正在处理的元素的索引值。
array可选
    some()被调用的数组。
thisArg可选
    执行 callback 时使用的 this 值。
复制代码
```

返回值

布尔值

实例

```
    let arr1 = [1,2,3,4,5];
    let arr2 = [1,2,3,4,5,12];
    arr1.some(x => x > 10); // false
    arr2.some(x => x > 10); // true
复制代码
```

reduce

语法

> arr.reduce(callback(accumulator, currentValue[, index[, array]])[, initialValue])

作用

对数组中的每个元素执行一个reducer函数，并将其结果汇总为单个返回值

参数

callback
    执行数组中每个值 (如果没有提供 initialValue则第一个值除外)的函数，包含四个参数：
accumulator
    累计器累计回调的返回值; 它是上一次调用回调时返回的累积值，或initialValue（见 于下方）。
currentValue
    数组中正在处理的元素。
index 可选
    数组中正在处理的当前元素的索引。 如果提供了initialValue，则起始索引号为0， 否则从索引1起始。
array可选
    调用reduce()的数组
initialValue可选
    作为第一次调用 callback函数时的第一个参数的值。 如果没有提供初始值，则将 使用数组中的第一个元素。 在没有初始值的空数组上调用 reduce 将报错。
复制代码

返回值

函数累计处理的结果

实例

```
    let arr = [1, 2, 3, 4]
    let sum = arr.reduce((acc, cur) => {
        return acc + cur;
    });
    console.log(sum); // 10
```
复制代码

应用

将二维数组转化为一维数组
```
    let arr = [[0, 1],[2, 3],[4, 5]]
    let flattened = arr.reduce(
        (acc, cur) => acc.concat(cur)
    );
    console.log(flattened); // [0,1,2,3,4,5]
```
复制代码

reduceRight

语法

> arr.reduceRight(callback(accumulator, currentValue[, index[, array]])[, initialValue])

作用

接受一个函数作为累加器（accumulator）和数组的每个值（从右到左）将其减少为单个值

参数

callback
　　一个回调函数，用于操作数组中的每个元素，它可接受四个参数：
accumulator
　　累加器：上一次调用回调函数时，回调函数返回的值。首次调用回调函数时，如果 initialValue 存在，累　加器即为 initialValue，否则须为数组中的最后一个元素（详见下方 initialValue 处相关说明）。
currentValue
　　当前元素：当前被处理的元素。
index可选
　　数组中当前被处理的元素的索引。
array可选
　　调用 reduceRight() 的数组。
initialValue可选
　　首次调用 callback 函数时，累加器 accumulator 的值。
　　如果未提供该初始值，则将使用数组中的最后一个元素，并跳过该元素。
　　如果不给出初始值，则需保证数组不为空。否则，在空数组上调用 reduce 或 reduceRight 且未提供初始值（例如 [].reduce((acc, cur, idx, arr) => {}) ）的话，会导致类型错误 TypeError: reduce of empty array with no initial value。
复制代码

返回值

执行之后的值

entires

语法

> arr.entries()

作用

返回一个新的Array Iterator对象，该对象包含数组中每个索引的键/值对

参数

无

返回值

一个新的 Array 迭代器对象

实例

```
    let arr = ["a", "b", "c"];
    let iterator = arr.entries();
    console.log(iterator.next().value); // [0, "a"]
    console.log(iterator.next().value); // [1, "b"]
    console.log(iterator.next().value); // [2, "c"]
    console.log(iterator.next().value); // undefined, 迭代器处于数组末尾时, 再迭代就会返回undefined
复制代码
```

find

语法

> arr.find(callback[, thisArg])

作用

返回数组中满足提供的测试函数的第一个元素的值。都不满足则返回 undefined

参数

```
callback
    在数组每一项上执行的函数，接收 3 个参数：
element
    当前遍历到的元素。
index可选
    当前遍历到的索引。
array可选
    数组本身。
thisArg可选
    执行回调时用作this 的对象。
复制代码
```

返回值

数组中第一个满足所提供测试函数的元素的值，都不满足则返回 undefined

实例

```
    let arr = [1, 3, 5, 7, 8, 9, 10];
    let result = arr.find(value => {
        if (value % 2 == 0) {
            return value
        }
    });
    console.log(result); // 8
复制代码
```

fineIndex

语法

> arr.findIndex(callback[, thisArg])

作用

判断数组中是否有满足测试函数的项

参数

```
callback
    针对数组中的每个元素，都会执行该回调函数，执行时会自动传入下面三个参数:
element
    当前元素。
index
    当前元素的索引。
array
    调用findIndex的数组。
thisArg
    可选。执行callback时作为this对象的值。
复制代码
```

返回值

数组中通过提供测试函数的第一个元素的索引。若都未通过则返回-1

实例

```
    let arr = [1, 3, 5, 7, 8, 9, 10];
    let result = arr.findIndex(value => {
        if (value % 2 == 0) {
            return value
        }
    });
    console.log(result); // 4
复制代码
```

keys

语法

> arr.keys()

作用

返回一个包含数组中每个索引键的Array Iterator对象

参数

无

返回值

一个新的 Array 迭代器对象

实例

```
    let arr = ["abc", "xyz"];
    let iterator = arr.keys();
    console.log(iterator.next()); // Object {value: 0, done: false}
    console.log(iterator.next()); // Object {value: 1, done: false}
    console.log(iterator.next()); // Object {value: undefined, done: true}
```

复制代码

values

语法

> arr.values()

作用

返回一个新的 Array Iterator 对象，该对象包含数组每个索引的值

参数

无

返回值

一个新的 Array 迭代器对象

实例

```
    let arr = ["abc", "def"];
    let iterator = arr.values();
    console.log(iterator.next().value); //abc
    console.log(iterator.next().value); //def
```

复制代码