Array 对象

构造函数

`Array`是 JavaScript 的原生对象，同时也是一个构造函数，可以用它生成新的数组。

```javascript
var arr = new Array(2);
arr.length // 2
arr // [ empty x 2 ]
```

上面代码中，`Array`构造函数的参数`2`，表示生成一个两个成员的数组，每个位置都是空值。

如果没有使用`new`，运行结果也是一样的。

```javascript
var arr = new Array(2);
// 等同于
var arr = Array(2);
```

`Array`构造函数有一个很大的缺陷，就是不同的参数，会导致它的行为不一致。

```javascript
// 无参数时，返回一个空数组
new Array() // []

// 单个正整数参数，表示返回的新数组的长度
new Array(1) // [ empty ]
new Array(2) // [ empty x 2 ]

// 非正整数的数值作为参数，会报错
new Array(3.2) // RangeError: Invalid array length
new Array(-3) // RangeError: Invalid array length

// 单个非数值（比如字符串、布尔值、对象等）作为参数，
// 则该参数是返回的新数组的成员
new Array('abc') // ['abc']
new Array([1]) // [Array[1]]

// 多参数时，所有参数都是返回的新数组的成员
new Array(1, 2) // [1, 2]
new Array('a', 'b', 'c') // ['a', 'b', 'c']
```

可以看到，`Array`作为构造函数，行为很不一致。因此，不建议使用它生成新数组，直接使用数组字面量是更好的做法。

```javascript
// bad
var arr = new Array(1, 2);

// good
var arr = [1, 2];
```

注意，如果参数是一个正整数，返回数组的成员都是空位。虽然读取的时候返回`undefined`，但实际上该位置没有任何值。虽然可以取到`length`属性，但是取不到键名。

```javascript
var a = new Array(3);
var b = [undefined, undefined, undefined];

a.length // 3
b.length // 3

a[0] // undefined
b[0] // undefined

0 in a // false
0 in b // true
```

上面代码中，`a`是一个长度为3的空数组，`b`是一个三个成员都是`undefined`的数组。读取键值的时候，`a`和`b`都返回`undefined`，但是`a`的键位都是空的，`b`的键位是有值的。

静态方法

Array.isArray()

`Array.isArray`方法返回一个布尔值，表示参数是否为数组。它可以弥补`typeof`运算符的不足。

```javascript
var arr = [1, 2, 3];

typeof arr // "object"
Array.isArray(arr) // true
```

上面代码中，`typeof`运算符只能显示数组的类型是`Object`，而`Array.isArray`方法可以识别数组。

实例方法

valueOf()，toString()

`valueOf`方法是一个所有对象都拥有的方法，表示对该对象求值。不同对象的`valueOf`方法不尽一致，数组的`valueOf`方法返回数组本身。

```javascript
var arr = [1, 2, 3];
arr.valueOf() // [1, 2, 3]
```

`toString`方法也是对象的通用方法，数组的`toString`方法返回数组的字符串形式。

```javascript
var arr = [1, 2, 3];
arr.toString() // "1,2,3"

var arr = [1, 2, 3, [4, 5, 6]];
arr.toString() // "1,2,3,4,5,6"
```

push()，pop()

`push`方法用于在数组的末端添加一个或多个元素，并返回添加新元素后的数组长度。注意，该方法会改变原数组。

```javascript
var arr = [];

arr.push(1) // 1
arr.push('a') // 2
arr.push(true, {}) // 4
arr // [1, 'a', true, {}]
```

上面代码使用`push`方法，往数组中添加了四个成员。

`pop`方法用于删除数组的最后一个元素，并返回该元素。注意，该方法会改变原数组。

```javascript
var arr = ['a', 'b', 'c'];

arr.pop() // 'c'
arr // ['a', 'b']
```

对空数组使用`pop`方法，不会报错，而是返回`undefined`。

```javascript
[].pop() // undefined
```

`push`和`pop`结合使用，就构成了“后进先出”的栈结构（stack）。

```javascript
var arr = [];
arr.push(1, 2);
arr.push(3);
arr.pop();
arr // [1, 2]
```

上面代码中，`3`是最后进入数组的，但是最早离开数组。

shift()，unshift()

`shift()`方法用于删除数组的第一个元素，并返回该元素。注意，该方法会改变原数组。

```javascript
var a = ['a', 'b', 'c'];

a.shift() // 'a'
a // ['b', 'c']
```

上面代码中，使用`shift()`方法以后，原数组就变了。

`shift()`方法可以遍历并清空一个数组。

```javascript
var list = [1, 2, 3, 4];
var item;

while (item = list.shift()) {
  console.log(item);
}

list // []
```

上面代码通过`list.shift()`方法每次取出一个元素，从而遍历数组。它的前提是数组元素不能是`0`或任何布尔值等于`false`的元素，因此这样的遍历不是很可靠。

`push()`和`shift()`结合使用，就构成了“先进先出”的队列结构（queue）。

`unshift()`方法用于在数组的第一个位置添加元素，并返回添加新元素后的数组长度。注意，该方法会改变原数组。

```javascript
var a = ['a', 'b', 'c'];

a.unshift('x'); // 4
a // ['x', 'a', 'b', 'c']
```

```

`unshift()`方法可以接受多个参数，这些参数都会添加到目标数组头部。

```javascript
var arr = [ 'c', 'd' ];
arr.unshift('a', 'b') // 4
arr // [ 'a', 'b', 'c', 'd' ]
```

join()

`join()`方法以指定参数作为分隔符，将所有数组成员连接为一个字符串返回。如果不提供参数，默认用逗号分隔。

```javascript
var a = [1, 2, 3, 4];

a.join(' ') // '1 2 3 4'
a.join(' | ') // "1 | 2 | 3 | 4"
a.join() // "1,2,3,4"
```

如果数组成员是`undefined`或`null`或空位，会被转成空字符串。

```javascript
[undefined, null].join('#')
// '#'

['a',, 'b'].join('-')
// 'a--b'
```

通过`call`方法，这个方法也可以用于字符串或类似数组的对象。

```javascript
Array.prototype.join.call('hello', '-')
// "h-e-l-l-o"

var obj = { 0: 'a', 1: 'b', length: 2 };
Array.prototype.join.call(obj, '-')
// 'a-b'
```

concat()

`concat`方法用于多个数组的合并。它将新数组的成员，添加到原数组成员的后部，然后返回一个新数组，原数组不变。

```javascript
['hello'].concat(['world'])
```
```

```
// ["hello", "world"]

['hello'].concat(['world'], ['!'])
// ["hello", "world", "!"]

[].concat({a: 1}, {b: 2})
// [{ a: 1 }, { b: 2 }]

[2].concat({a: 1})
// [2, {a: 1}]
```

除了数组作为参数，`concat`也接受其他类型的值作为参数，添加到目标数组尾部。

```javascript
[1, 2, 3].concat(4, 5, 6)
// [1, 2, 3, 4, 5, 6]
```

如果数组成员包括对象，`concat`方法返回当前数组的一个浅拷贝。所谓“浅拷贝”，指的是新数组拷贝的是对象的引用。

```javascript
var obj = { a: 1 };
var oldArray = [obj];

var newArray = oldArray.concat();

obj.a = 2;
newArray[0].a // 2
```

上面代码中，原数组包含一个对象，`concat`方法生成的新数组包含这个对象的引用。所以，改变原对象以后，新数组跟着改变。

reverse()

`reverse`方法用于颠倒排列数组元素，返回改变后的数组。注意，该方法将改变原数组。

```javascript
var a = ['a', 'b', 'c'];

a.reverse() // ["c", "b", "a"]
a // ["c", "b", "a"]
```

slice()

`slice`方法用于提取目标数组的一部分，返回一个新数组，原数组不变。

```javascript
```

arr.slice(start, end);
```

它的第一个参数为起始位置（从0开始），第二个参数为终止位置（但该位置的元素本身不包括在内）。如果省略第二个参数，则一直返回到原数组的最后一个成员。

```javascript
var a = ['a', 'b', 'c'];

a.slice(0) // ["a", "b", "c"]
a.slice(1) // ["b", "c"]
a.slice(1, 2) // ["b"]
a.slice(2, 6) // ["c"]
a.slice() // ["a", "b", "c"]
```

上面代码中，最后一个例子`slice`没有参数，实际上等于返回一个原数组的拷贝。

如果`slice`方法的参数是负数，则表示倒数计算的位置。

```javascript
var a = ['a', 'b', 'c'];
a.slice(-2) // ["b", "c"]
a.slice(-2, -1) // ["b"]
```

上面代码中，`-2`表示倒数计算的第二个位置，`-1`表示倒数计算的第一个位置。

如果第一个参数大于等于数组长度，或者第二个参数小于第一个参数，则返回空数组。

```javascript
var a = ['a', 'b', 'c'];
a.slice(4) // []
a.slice(2, 1) // []
```

`slice`方法的一个重要应用，是将类似数组的对象转为真正的数组。

```javascript
Array.prototype.slice.call({ 0: 'a', 1: 'b', length: 2 })
// ['a', 'b']

Array.prototype.slice.call(document.querySelectorAll("div"));
Array.prototype.slice.call(arguments);
```

上面代码的参数都不是数组，但是通过`call`方法，在它们上面调用`slice`方法，就可以把它们转为真正的数组。

splice()
```

`splice`方法用于删除原数组的一部分成员，并可以在删除的位置添加新的数组成员，返回值是被删除的元素。注意，该方法会改变原数组。

```javascript
arr.splice(start, count, addElement1, addElement2, ...);
```

`splice`的第一个参数是删除的起始位置（从0开始），第二个参数是被删除的元素个数。如果后面还有更多的参数，则表示这些就是要被插入数组的新元素。

```javascript
var a = ['a', 'b', 'c', 'd', 'e', 'f'];
a.splice(4, 2) // ["e", "f"]
a // ["a", "b", "c", "d"]
```

上面代码从原数组4号位置，删除了两个数组成员。

```javascript
var a = ['a', 'b', 'c', 'd', 'e', 'f'];
a.splice(4, 2, 1, 2) // ["e", "f"]
a // ["a", "b", "c", "d", 1, 2]
```

上面代码除了删除成员，还插入了两个新成员。

起始位置如果是负数，就表示从倒数位置开始删除。

```javascript
var a = ['a', 'b', 'c', 'd', 'e', 'f'];
a.splice(-4, 2) // ["c", "d"]
```

上面代码表示，从倒数第四个位置`c`开始删除两个成员。

如果只是单纯地插入元素，`splice`方法的第二个参数可以设为`0`。

```javascript
var a = [1, 1, 1];

a.splice(1, 0, 2) // []
a // [1, 2, 1, 1]
```

如果只提供第一个参数，等同于将原数组在指定位置拆分成两个数组。

```javascript
var a = [1, 2, 3, 4];
a.splice(2) // [3, 4]
```

a // [1, 2]
```

### sort()

`sort`方法对数组成员进行排序，默认是按照字典顺序排序。排序后，原数组将被改变。

```javascript
['d', 'c', 'b', 'a'].sort()
// ['a', 'b', 'c', 'd']

[4, 3, 2, 1].sort()
// [1, 2, 3, 4]

[11, 101].sort()
// [101, 11]

[10111, 1101, 111].sort()
// [10111, 1101, 111]
```

上面代码的最后两个例子，需要特殊注意。`sort`方法不是按照大小排序，而是按照字典顺序。也就是说，数值会被先转成字符串，再按照字典顺序进行比较，所以`101`排在`11`的前面。

如果想让`sort`方法按照自定义方式排序，可以传入一个函数作为参数。

```javascript
[10111, 1101, 111].sort(function (a, b) {
  return a - b;
})
// [111, 1101, 10111]
```

上面代码中，`sort`的参数函数本身接受两个参数，表示进行比较的两个数组成员。如果该函数的返回值大于`0`，表示第一个成员排在第二个成员后面；其他情况下，都是第一个元素排在第二个元素前面。

```javascript
[
  { name: "张三", age: 30 },
  { name: "李四", age: 24 },
  { name: "王五", age: 28  }
].sort(function (o1, o2) {
  return o1.age - o2.age;
})
// [
//   { name: "李四", age: 24 },
//   { name: "王五", age: 28  },
//   { name: "张三", age: 30 }
```
```

//]
```

### map()

`map`方法将数组的所有成员依次传入参数函数，然后把每一次的执行结果组成一个新数组返回。

```javascript
var numbers = [1, 2, 3];

numbers.map(function (n) {
  return n + 1;
});
// [2, 3, 4]

numbers
// [1, 2, 3]
```

上面代码中，`numbers`数组的所有成员依次执行参数函数，运行结果组成一个新数组返回，原数组没有变化。

`map`方法接受一个函数作为参数。该函数调用时，`map`方法向它传入三个参数：当前成员、当前位置和数组本身。

```javascript
[1, 2, 3].map(function(elem, index, arr) {
  return elem * index;
});
// [0, 2, 6]
```

上面代码中，`map`方法的回调函数有三个参数，`elem`为当前成员的值，`index`为当前成员的位置，`arr`为原数组（`[1, 2, 3]`）。

`map`方法还可以接受第二个参数，用来绑定回调函数内部的`this`变量（详见《this 变量》一章）。

```javascript
var arr = ['a', 'b', 'c'];

[1, 2].map(function (e) {
  return this[e];
}, arr)
// ['b', 'c']
```

上面代码通过`map`方法的第二个参数，将回调函数内部的`this`对象，指向`arr`数组。

如果数组有空位，`map`方法的回调函数在这个位置不会执行，会跳过数组的空位。
```

```javascript
var f = function (n) { return 'a' };

[1, undefined, 2].map(f) // ["a", "a", "a"]
[1, null, 2].map(f) // ["a", "a", "a"]
[1, , 2].map(f) // ["a", , "a"]
```

上面代码中，`map`方法不会跳过`undefined`和`null`，但是会跳过空位。

forEach()

`forEach`方法与`map`方法很相似，也是对数组的所有成员依次执行参数函数。但是，`forEach`方法不返回值，只用来操作数据。这就是说，如果数组遍历的目的是为了得到返回值，那么使用`map`方法，否则使用`forEach`方法。

`forEach`的用法与`map`方法一致，参数是一个函数，该函数同样接受三个参数：当前值、当前位置、整个数组。

```javascript
function log(element, index, array) {
  console.log('[' + index + '] = ' + element);
}

[2, 5, 9].forEach(log);
// [0] = 2
// [1] = 5
// [2] = 9
```

上面代码中，`forEach`遍历数组不是为了得到返回值，而是为了在屏幕输出内容，所以不必使用`map`方法。

`forEach`方法也可以接受第二个参数，绑定参数函数的`this`变量。

```javascript
var out = [];

[1, 2, 3].forEach(function(elem) {
  this.push(elem * elem);
}, out);

out // [1, 4, 9]
```

上面代码中，空数组`out`是`forEach`方法的第二个参数，结果，回调函数内部的`this`关键字就指向`out`。

注意，`forEach`方法无法中断执行，总是会将所有成员遍历完。如果希望符合某种条件时，就中断遍历，要使用`for`循环。

```javascript
var arr = [1, 2, 3];

for (var i = 0; i < arr.length; i++) {
  if (arr[i] === 2) break;
  console.log(arr[i]);
}
// 1
```

上面代码中，执行到数组的第二个成员时，就会中断执行。`forEach`方法做不到这一点。

`forEach`方法也会跳过数组的空位。

```javascript
var log = function (n) {
  console.log(n + 1);
};

[1, undefined, 2].forEach(log)
// 2
// NaN
// 3

[1, null, 2].forEach(log)
// 2
// 1
// 3

[1, , 2].forEach(log)
// 2
// 3
```

上面代码中，`forEach`方法不会跳过`undefined`和`null`，但会跳过空位。

filter()

`filter`方法用于过滤数组成员，满足条件的成员组成一个新数组返回。

它的参数是一个函数，所有数组成员依次执行该函数，返回结果为`true`的成员组成一个新数组返回。该方法不会改变原数组。

```javascript
[1, 2, 3, 4, 5].filter(function (elem) {
  return (elem > 3);
})
// [4, 5]
```

```

上面代码将大于`3`的数组成员，作为一个新数组返回。

```javascript
var arr = [0, 1, 'a', false];

arr.filter(Boolean)
// [1, "a"]
```

上面代码中，`filter`方法返回数组`arr`里面所有布尔值为`true`的成员。

`filter`方法的参数函数可以接受三个参数：当前成员，当前位置和整个数组。

```javascript
[1, 2, 3, 4, 5].filter(function (elem, index, arr) {
  return index % 2 === 0;
});
// [1, 3, 5]
```

上面代码返回偶数位置的成员组成的新数组。

`filter`方法还可以接受第二个参数，用来绑定参数函数内部的`this`变量。

```javascript
var obj = { MAX: 3 };
var myFilter = function (item) {
  if (item > this.MAX) return true;
};

var arr = [2, 8, 3, 4, 1, 3, 2, 9];
arr.filter(myFilter, obj) // [8, 4, 9]
```

上面代码中，过滤器`myFilter`内部有`this`变量，它可以被`filter`方法的第二个参数`obj`绑定，返回大于`3`的成员。

some()，every()

这两个方法类似“断言”（assert），返回一个布尔值，表示判断数组成员是否符合某种条件。

它们接受一个函数作为参数，所有数组成员依次执行该函数。该函数接受三个参数：当前成员、当前位置和整个数组，然后返回一个布尔值。

`some`方法是只要一个成员的返回值是`true`，则整个`some`方法的返回值就是`true`，否则返回`false`。
```

```javascript
var arr = [1, 2, 3, 4, 5];
arr.some(function (elem, index, arr) {
  return elem >= 3;
});
// true
```

上面代码中，如果数组`arr`有一个成员大于等于3，`some`方法就返回`true`。

`every`方法是所有成员的返回值都是`true`，整个`every`方法才返回`true`，否则返回`false`。

```javascript
var arr = [1, 2, 3, 4, 5];
arr.every(function (elem, index, arr) {
  return elem >= 3;
});
// false
```

上面代码中，数组`arr`并非所有成员大于等于`3`，所以返回`false`。

注意，对于空数组，`some`方法返回`false`，`every`方法返回`true`，回调函数都不会执行。

```javascript
function isEven(x) { return x % 2 === 0 }

[].some(isEven) // false
[].every(isEven) // true
```

`some`和`every`方法还可以接受第二个参数，用来绑定参数函数内部的`this`变量。

reduce()，reduceRight()

`reduce`方法和`reduceRight`方法依次处理数组的每个成员，最终累计为一个值。它们的差别是，`reduce`是从左到右处理（从第一个成员到最后一个成员），`reduceRight`则是从右到左（从最后一个成员到第一个成员），其他完全一样。

```javascript
[1, 2, 3, 4, 5].reduce(function (a, b) {
  console.log(a, b);
  return a + b;
})
// 1 2
// 3 3
// 6 4
// 10 5
//最后结果：15
```

上面代码中，`reduce`方法求出数组所有成员的和。第一次执行，`a`是数组的第一个成员`1`，`b`是数组的第二个成员`2`。第二次执行，`a`为上一轮的返回值`3`，`b`为第三个成员`3`。第三次执行，`a`为上一轮的返回值`6`，`b`为第四个成员`4`。第四次执行，`a`为上一轮返回值`10`，`b`为第五个成员`5`。至此所有成员遍历完成，整个方法的返回值就是最后一轮的返回值`15`。

`reduce`方法和`reduceRight`方法的第一个参数都是一个函数。该函数接受以下四个参数。

1. 累积变量，默认为数组的第一个成员
2. 当前变量，默认为数组的第二个成员
3. 当前位置（从0开始）
4. 原数组

这四个参数之中，只有前两个是必须的，后两个则是可选的。

如果要对累积变量指定初值，可以把它放在`reduce`方法和`reduceRight`方法的第二个参数。

```javascript
[1, 2, 3, 4, 5].reduce(function (a, b) {
  return a + b;
}, 10);
// 25
```

上面代码指定参数`a`的初值为10，所以数组从10开始累加，最终结果为25。注意，这时`b`是从数组的第一个成员开始遍历。

上面的第二个参数相当于设定了默认值，处理空数组时尤其有用。

```javascript
function add(prev, cur) {
  return prev + cur;
}

[].reduce(add)
// TypeError: Reduce of empty array with no initial value
[].reduce(add, 1)
// 1
```

上面代码中，由于空数组取不到初始值，`reduce`方法会报错。这时，加上第二个参数，就能保证总是会返回一个值。

下面是一个`reduceRight`方法的例子。

```javascript
function subtract(prev, cur) {
  return prev - cur;
```

```
}

[3, 2, 1].reduce(subtract) // 0
[3, 2, 1].reduceRight(subtract) // -4
```

上面代码中，`reduce`方法相当于`3`减去`2`再减去`1`，`reduceRight`方法相当于`1`减去`2`再减去`3`。

由于这两个方法会遍历数组，所以实际上还可以用来做一些遍历相关的操作。比如，找出字符长度最长的数组成员。

```javascript
function findLongest(entries) {
  return entries.reduce(function (longest, entry) {
    return entry.length > longest.length ? entry : longest;
  }, '');
}

findLongest(['aaa', 'bb', 'c']) // "aaa"
```

上面代码中，`reduce`的参数函数会将字符长度较长的那个数组成员，作为累积值。这导致遍历所有成员之后，累积值就是字符长度最长的那个成员。

indexOf()，lastIndexOf()

`indexOf`方法返回给定元素在数组中第一次出现的位置，如果没有出现则返回`-1`。

```javascript
var a = ['a', 'b', 'c'];

a.indexOf('b') // 1
a.indexOf('y') // -1
```

`indexOf`方法还可以接受第二个参数，表示搜索的开始位置。

```javascript
['a', 'b', 'c'].indexOf('a', 1) // -1
```

上面代码从1号位置开始搜索字符`a`，结果为`-1`，表示没有搜索到。

`lastIndexOf`方法返回给定元素在数组中最后一次出现的位置，如果没有出现则返回`-1`。

```javascript
var a = [2, 5, 9, 2];
a.lastIndexOf(2) // 3
a.lastIndexOf(7) // -1
```

```

注意，这两个方法不能用来搜索`NaN`的位置，即它们无法确定数组成员是否包含`NaN`。

```javascript
[NaN].indexOf(NaN) // -1
[NaN].lastIndexOf(NaN) // -1
```

这是因为这两个方法内部，使用严格相等运算符（`===`）进行比较，而`NaN`是唯一一个不等于自身的值。

链式使用

上面这些数组方法之中，有不少返回的还是数组，所以可以链式使用。

```javascript
var users = [
  {name: 'tom', email: 'tom@example.com'},
  {name: 'peter', email: 'peter@example.com'}
];

users
.map(function (user) {
  return user.email;
})
.filter(function (email) {
  return /^t/.test(email);
})
.forEach(function (email) {
  console.log(email);
});
// "tom@example.com"
```

上面代码中，先产生一个所有 Email 地址组成的数组，然后再过滤出以`t`开头的 Email 地址，最后将它打印出来。

参考链接

- Nicolas Bevacqua, [Fun with JavaScript Native Array Functions](http://flippinawesome.org/2013/11/25/fun-with-javascript-native-array-functions/)
```