# 数组的扩展

1. **[扩展运算符](http://es6.ruanyifeng.com/" \l "docs/array#扩展运算符)**
2. **[Array.from()](http://es6.ruanyifeng.com/" \l "docs/array#Array.from())**
3. **[Array.of()](http://es6.ruanyifeng.com/" \l "docs/array#Array.of())**
4. **[数组实例的 copyWithin()](http://es6.ruanyifeng.com/" \l "docs/array#数组实例的 copyWithin())**
5. **[数组实例的 find() 和 findIndex()](http://es6.ruanyifeng.com/" \l "docs/array#数组实例的 find() 和 findIndex())**
6. **[数组实例的fill()](http://es6.ruanyifeng.com/" \l "docs/array#数组实例的fill())**
7. **[数组实例的 entries()，keys() 和 values()](http://es6.ruanyifeng.com/" \l "docs/array#数组实例的 entries()，keys() 和 values())**
8. **[数组实例的 includes()](http://es6.ruanyifeng.com/" \l "docs/array#数组实例的 includes())**
9. **[数组的空位](http://es6.ruanyifeng.com/" \l "docs/array#数组的空位)**

## 扩展运算符

### 含义

扩展运算符（spread）是三个点（...）。它好比 rest 参数的逆运算，将一个数组转为用逗号分隔的参数序列。

console.log(...[1, 2, 3])

// 1 2 3

console.log(1, ...[2, 3, 4], 5)

// 1 2 3 4 5[...document.querySelectorAll('div')]

// [<div>, <div>, <div>]

该运算符主要用于函数调用。

function push(array, ...items) {

array.push(...items);}

function add(x, y) {

return x + y;}

const numbers = [4, 38];add(...numbers) // 42

上面代码中，array.push(...items)和add(...numbers)这两行，都是函数的调用，它们的都使用了扩展运算符。该运算符将一个数组，变为参数序列。

扩展运算符与正常的函数参数可以结合使用，非常灵活。

function f(v, w, x, y, z) { }

const args = [0, 1];f(-1, ...args, 2, ...[3]);

扩展运算符后面还可以放置表达式。

const arr = [

...(x > 0 ? ['a'] : []),

'b',];

如果扩展运算符后面是一个空数组，则不产生任何效果。

[...[], 1]

// [1]

### 替代数组的 apply 方法

由于扩展运算符可以展开数组，所以不再需要apply方法，将数组转为函数的参数了。

// ES5 的写法function f(x, y, z) {

// ...}var args = [0, 1, 2];

f.apply(null, args);

// ES6的写法function f(x, y, z) {

// ...}let args = [0, 1, 2];f(...args);

下面是扩展运算符取代apply方法的一个实际的例子，应用Math.max方法，简化求出一个数组最大元素的写法。

// ES5 的写法Math.max.apply(null, [14, 3, 77])

// ES6 的写法Math.max(...[14, 3, 77])

// 等同于Math.max(14, 3, 77);

上面代码中，由于 JavaScript 不提供求数组最大元素的函数，所以只能套用Math.max函数，将数组转为一个参数序列，然后求最大值。有了扩展运算符以后，就可以直接用Math.max了。

另一个例子是通过push函数，将一个数组添加到另一个数组的尾部。

// ES5的 写法var arr1 = [0, 1, 2];var arr2 = [3, 4, 5];

Array.prototype.push.apply(arr1, arr2);

// ES6 的写法let arr1 = [0, 1, 2];let arr2 = [3, 4, 5];

arr1.push(...arr2);

上面代码的 ES5 写法中，push方法的参数不能是数组，所以只好通过apply方法变通使用push方法。有了扩展运算符，就可以直接将数组传入push方法。

下面是另外一个例子。

// ES5new (Date.bind.apply(Date, [null, 2015, 1, 1]))

// ES6new Date(...[2015, 1, 1]);

### 扩展运算符的应用

****（1）复制数组****

数组是复合的数据类型，直接复制的话，只是复制了指向底层数据结构的指针，而不是克隆一个全新的数组。

const a1 = [1, 2];

const a2 = a1;

a2[0] = 2;

a1 // [2, 2]

上面代码中，a2并不是a1的克隆，而是指向同一份数据的另一个指针。修改a2，会直接导致a1的变化。

ES5 只能用变通方法来复制数组。

const a1 = [1, 2];

const a2 = a1.concat();

a2[0] = 2;

a1 // [1, 2]

上面代码中，a1会返回原数组的克隆，再修改a2就不会对a1产生影响。

扩展运算符提供了复制数组的简便写法。

const a1 = [1, 2];

// 写法一const a2 = [...a1];

// 写法二const [...a2] = a1;

上面的两种写法，a2都是a1的克隆。

****（2）合并数组****

扩展运算符提供了数组合并的新写法。

// ES5[1, 2].concat(more)

// ES6[1, 2, ...more]

var arr1 = ['a', 'b'];var arr2 = ['c'];var arr3 = ['d', 'e'];

// ES5的合并数组arr1.concat(arr2, arr3);

// [ 'a', 'b', 'c', 'd', 'e' ]

// ES6的合并数组[...arr1, ...arr2, ...arr3]

// [ 'a', 'b', 'c', 'd', 'e' ]

****（3）与解构赋值结合****

扩展运算符可以与解构赋值结合起来，用于生成数组。

// ES5a = list[0], rest = list.slice(1)

// ES6[a, ...rest] = list

下面是另外一些例子。

const [first, ...rest] = [1, 2, 3, 4, 5];

first // 1rest // [2, 3, 4, 5]

const [first, ...rest] = [];

first // undefinedrest // []

const [first, ...rest] = ["foo"];

first // "foo"rest // []

如果将扩展运算符用于数组赋值，只能放在参数的最后一位，否则会报错。

const [...butLast, last] = [1, 2, 3, 4, 5];

// 报错

const [first, ...middle, last] = [1, 2, 3, 4, 5];

// 报错

****（4）字符串****

扩展运算符还可以将字符串转为真正的数组。

[...'hello']

// [ "h", "e", "l", "l", "o" ]

上面的写法，有一个重要的好处，那就是能够正确识别四个字节的 Unicode 字符。

'x\uD83D\uDE80y'.length // 4[...'x\uD83D\uDE80y'].length // 3

上面代码的第一种写法，JavaScript 会将四个字节的 Unicode 字符，识别为2个字符，采用扩展运算符就没有这个问题。因此，正确返回字符串长度的函数，可以像下面这样写。

function length(str) {

return [...str].length;}

length('x\uD83D\uDE80y') // 3

凡是涉及到操作四个字节的 Unicode 字符的函数，都有这个问题。因此，最好都用扩展运算符改写。

let str = 'x\uD83D\uDE80y';

str.split('').reverse().join('')

// 'y\uDE80\uD83Dx'[...str].reverse().join('')

// 'y\uD83D\uDE80x'

上面代码中，如果不用扩展运算符，字符串的reverse操作就不正确。

****（5）实现了 Iterator 接口的对象****

任何 Iterator 接口的对象（参阅 Iterator 一章），都可以用扩展运算符转为真正的数组。

let nodeList = document.querySelectorAll('div');let array = [...nodeList];

上面代码中，querySelectorAll方法返回的是一个nodeList对象。它不是数组，而是一个类似数组的对象。这时，扩展运算符可以将其转为真正的数组，原因就在于NodeList对象实现了 Iterator 。

对于那些没有部署 Iterator 接口的类似数组的对象，扩展运算符就无法将其转为真正的数组。

let arrayLike = {

'0': 'a',

'1': 'b',

'2': 'c',

length: 3};

// TypeError: Cannot spread non-iterable object.let arr = [...arrayLike];

上面代码中，arrayLike是一个类似数组的对象，但是没有部署 Iterator 接口，扩展运算符就会报错。这时，可以改为使用Array.from方法将arrayLike转为真正的数组。

****（6）Map 和 Set 结构，Generator 函数****

扩展运算符内部调用的是数据结构的 Iterator 接口，因此只要具有 Iterator 接口的对象，都可以使用扩展运算符，比如 Map 结构。

let map = new Map([

[1, 'one'],

[2, 'two'],

[3, 'three'],]);

let arr = [...map.keys()]; // [1, 2, 3]

Generator 函数运行后，返回一个遍历器对象，因此也可以使用扩展运算符。

const go = function\*(){

yield 1;

yield 2;

yield 3;};

[...go()] // [1, 2, 3]

上面代码中，变量go是一个 Generator 函数，执行后返回的是一个遍历器对象，对这个遍历器对象执行扩展运算符，就会将内部遍历得到的值，转为一个数组。

如果对没有 Iterator 接口的对象，使用扩展运算符，将会报错。

const obj = {a: 1, b: 2};let arr = [...obj]; // TypeError: Cannot spread non-iterable object

## Array.from()

Array.from方法用于将两类对象转为真正的数组：类似数组的对象（array-like object）和可遍历（iterable）的对象（包括ES6新增的数据结构Set和Map）。

下面是一个类似数组的对象，Array.from将它转为真正的数组。

let arrayLike = {

'0': 'a',

'1': 'b',

'2': 'c',

length: 3};

// ES5的写法var arr1 = [].slice.call(arrayLike); // ['a', 'b', 'c']

// ES6的写法let arr2 = Array.from(arrayLike); // ['a', 'b', 'c']

实际应用中，常见的类似数组的对象是DOM操作返回的NodeList集合，以及函数内部的arguments对象。Array.from都可以将它们转为真正的数组。

// NodeList对象let ps = document.querySelectorAll('p');

Array.from(ps).forEach(function (p) {

console.log(p);});

// arguments对象function foo() {

var args = Array.from(arguments);

// ...}

上面代码中，querySelectorAll方法返回的是一个类似数组的对象，可以将这个对象转为真正的数组，再使用forEach方法。

只要是部署了Iterator接口的数据结构，Array.from都能将其转为数组。

Array.from('hello')

// ['h', 'e', 'l', 'l', 'o']let namesSet = new Set(['a', 'b'])

Array.from(namesSet) // ['a', 'b']

上面代码中，字符串和Set结构都具有Iterator接口，因此可以被Array.from转为真正的数组。

如果参数是一个真正的数组，Array.from会返回一个一模一样的新数组。

Array.from([1, 2, 3])

// [1, 2, 3]

值得提醒的是，扩展运算符（...）也可以将某些数据结构转为数组。

// arguments对象function foo() {

const args = [...arguments];}

// NodeList对象[...document.querySelectorAll('div')]

扩展运算符背后调用的是遍历器接口（Symbol.iterator），如果一个对象没有部署这个接口，就无法转换。Array.from方法还支持类似数组的对象。所谓类似数组的对象，本质特征只有一点，即必须有length属性。因此，任何有length属性的对象，都可以通过Array.from方法转为数组，而此时扩展运算符就无法转换。

Array.from({ length: 3 });

// [ undefined, undefined, undefined ]

上面代码中，Array.from返回了一个具有三个成员的数组，每个位置的值都是undefined。扩展运算符转换不了这个对象。

对于还没有部署该方法的浏览器，可以用Array.prototype.slice方法替代。

const toArray = (() =>

Array.from ? Array.from : obj => [].slice.call(obj))();

Array.from还可以接受第二个参数，作用类似于数组的map方法，用来对每个元素进行处理，将处理后的值放入返回的数组。

Array.from(arrayLike, x => x \* x);

// 等同于Array.from(arrayLike).map(x => x \* x);

Array.from([1, 2, 3], (x) => x \* x)

// [1, 4, 9]

下面的例子是取出一组DOM节点的文本内容。

let spans = document.querySelectorAll('span.name');

// map()let names1 = Array.prototype.map.call(spans, s => s.textContent);

// Array.from()let names2 = Array.from(spans, s => s.textContent)

下面的例子将数组中布尔值为false的成员转为0。

Array.from([1, , 2, , 3], (n) => n || 0)

// [1, 0, 2, 0, 3]

另一个例子是返回各种数据的类型。

function typesOf () {

return Array.from(arguments, value => typeof value)}typesOf(null, [], NaN)

// ['object', 'object', 'number']

如果map函数里面用到了this关键字，还可以传入Array.from的第三个参数，用来绑定this。

Array.from()可以将各种值转为真正的数组，并且还提供map功能。这实际上意味着，只要有一个原始的数据结构，你就可以先对它的值进行处理，然后转成规范的数组结构，进而就可以使用数量众多的数组方法。

Array.from({ length: 2 }, () => 'jack')

// ['jack', 'jack']

上面代码中，Array.from的第一个参数指定了第二个参数运行的次数。这种特性可以让该方法的用法变得非常灵活。

Array.from()的另一个应用是，将字符串转为数组，然后返回字符串的长度。因为它能正确处理各种Unicode字符，可以避免JavaScript将大于\uFFFF的Unicode字符，算作两个字符的bug。

function countSymbols(string) {

return Array.from(string).length;}

## Array.of()

Array.of方法用于将一组值，转换为数组。

Array.of(3, 11, 8) // [3,11,8]Array.of(3) // [3]Array.of(3).length // 1

这个方法的主要目的，是弥补数组构造函数Array()的不足。因为参数个数的不同，会导致Array()的行为有差异。

Array() // []Array(3) // [, , ,]Array(3, 11, 8) // [3, 11, 8]

上面代码中，Array方法没有参数、一个参数、三个参数时，返回结果都不一样。只有当参数个数不少于2个时，Array()才会返回由参数组成的新数组。参数个数只有一个时，实际上是指定数组的长度。

Array.of基本上可以用来替代Array()或new Array()，并且不存在由于参数不同而导致的重载。它的行为非常统一。

Array.of() // []Array.of(undefined) // [undefined]Array.of(1) // [1]Array.of(1, 2) // [1, 2]

Array.of总是返回参数值组成的数组。如果没有参数，就返回一个空数组。

Array.of方法可以用下面的代码模拟实现。

function ArrayOf(){

return [].slice.call(arguments);}

## 数组实例的 copyWithin()

数组实例的copyWithin方法，在当前数组内部，将指定位置的成员复制到其他位置（会覆盖原有成员），然后返回当前数组。也就是说，使用这个方法，会修改当前数组。

Array.prototype.copyWithin(target, start = 0, end = this.length)

它接受三个参数。

* target（必需）：从该位置开始替换数据。
* start（可选）：从该位置开始读取数据，默认为0。如果为负值，表示倒数。
* end（可选）：到该位置前停止读取数据，默认等于数组长度。如果为负值，表示倒数。

这三个参数都应该是数值，如果不是，会自动转为数值。

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

// [4, 5, 3, 4, 5]

上面代码表示将从3号位直到数组结束的成员（4和5），复制到从0号位开始的位置，结果覆盖了原来的1和2。

下面是更多例子。

// 将3号位复制到0号位[1, 2, 3, 4, 5].copyWithin(0, 3, 4)

// [4, 2, 3, 4, 5]

// -2相当于3号位，-1相当于4号位[1, 2, 3, 4, 5].copyWithin(0, -2, -1)

// [4, 2, 3, 4, 5]

// 将3号位复制到0号位[].copyWithin.call({length: 5, 3: 1}, 0, 3)

// {0: 1, 3: 1, length: 5}

// 将2号位到数组结束，复制到0号位let i32a = new Int32Array([1, 2, 3, 4, 5]);

i32a.copyWithin(0, 2);

// Int32Array [3, 4, 5, 4, 5]

// 对于没有部署 TypedArray 的 copyWithin 方法的平台// 需要采用下面的写法[].copyWithin.call(new Int32Array([1, 2, 3, 4, 5]), 0, 3, 4);

// Int32Array [4, 2, 3, 4, 5]

## 数组实例的 find() 和 findIndex()

数组实例的find方法，用于找出第一个符合条件的数组成员。它的参数是一个回调函数，所有数组成员依次执行该回调函数，直到找出第一个返回值为true的成员，然后返回该成员。如果没有符合条件的成员，则返回undefined。

[1, 4, -5, 10].find((n) => n < 0)

// -5

上面代码找出数组中第一个小于0的成员。

[1, 5, 10, 15].find(function(value, index, arr) {

return value > 9;}) // 10

上面代码中，find方法的回调函数可以接受三个参数，依次为当前的值、当前的位置和原数组。

数组实例的findIndex方法的用法与find方法非常类似，返回第一个符合条件的数组成员的位置，如果所有成员都不符合条件，则返回-1。

[1, 5, 10, 15].findIndex(function(value, index, arr) {

return value > 9;}) // 2

这两个方法都可以接受第二个参数，用来绑定回调函数的this对象。

另外，这两个方法都可以发现NaN，弥补了数组的IndexOf方法的不足。

[NaN].indexOf(NaN)

// -1[NaN].findIndex(y => Object.is(NaN, y))

// 0

上面代码中，indexOf方法无法识别数组的NaN成员，但是findIndex方法可以借助Object.is方法做到。

## 数组实例的fill()

fill方法使用给定值，填充一个数组。

['a', 'b', 'c'].fill(7)

// [7, 7, 7]new Array(3).fill(7)

// [7, 7, 7]

上面代码表明，fill方法用于空数组的初始化非常方便。数组中已有的元素，会被全部抹去。

fill方法还可以接受第二个和第三个参数，用于指定填充的起始位置和结束位置。

['a', 'b', 'c'].fill(7, 1, 2)

// ['a', 7, 'c']

上面代码表示，fill方法从1号位开始，向原数组填充7，到2号位之前结束。

## 数组实例的 entries()，keys() 和 values()

ES6 提供三个新的方法——entries()，keys()和values()——用于遍历数组。它们都返回一个遍历器对象（详见《Iterator》一章），可以用for...of循环进行遍历，唯一的区别是keys()是对键名的遍历、values()是对键值的遍历，entries()是对键值对的遍历。

for (let index of ['a', 'b'].keys()) {

console.log(index);}

// 0// 1for (let elem of ['a', 'b'].values()) {

console.log(elem);}

// 'a'// 'b'for (let [index, elem] of ['a', 'b'].entries()) {

console.log(index, elem);}

// 0 "a"// 1 "b"

如果不使用for...of循环，可以手动调用遍历器对象的next方法，进行遍历。

let letter = ['a', 'b', 'c'];let entries = letter.entries();

console.log(entries.next().value); // [0, 'a']console.log(entries.next().value); // [1, 'b']console.log(entries.next().value); // [2, 'c']

## 数组实例的 includes()

Array.prototype.includes方法返回一个布尔值，表示某个数组是否包含给定的值，与字符串的includes方法类似。ES2016 引入了该方法。

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

该方法的第二个参数表示搜索的起始位置，默认为0。如果第二个参数为负数，则表示倒数的位置，如果这时它大于数组长度（比如第二个参数为-4，但数组长度为3），则会重置为从0开始。

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

没有该方法之前，我们通常使用数组的indexOf方法，检查是否包含某个值。

if (arr.indexOf(el) !== -1) {

// ...}

indexOf方法有两个缺点，一是不够语义化，它的含义是找到参数值的第一个出现位置，所以要去比较是否不等于-1，表达起来不够直观。二是，它内部使用严格相等运算符（===）进行判断，这会导致对NaN的误判。

[NaN].indexOf(NaN)

// -1

includes使用的是不一样的判断算法，就没有这个问题。

[NaN].includes(NaN)

// true

下面代码用来检查当前环境是否支持该方法，如果不支持，部署一个简易的替代版本。

const contains = (() =>

Array.prototype.includes

? (arr, value) => arr.includes(value)

: (arr, value) => arr.some(el => el === value))();contains(['foo', 'bar'], 'baz'); // => false

另外，Map 和 Set 数据结构有一个has方法，需要注意与includes区分。

* Map 结构的has方法，是用来查找键名的，比如Map.prototype.has(key)、WeakMap.prototype.has(key)、Reflect.has(target, propertyKey)。
* Set 结构的has方法，是用来查找值的，比如Set.prototype.has(value)、WeakSet.prototype.has(value)。

## 数组的空位

数组的空位指，数组的某一个位置没有任何值。比如，Array构造函数返回的数组都是空位。

Array(3) // [, , ,]

上面代码中，Array(3)返回一个具有3个空位的数组。

注意，空位不是undefined，一个位置的值等于undefined，依然是有值的。空位是没有任何值，in运算符可以说明这一点。

0 in [undefined, undefined, undefined] // true0 in [, , ,] // false

上面代码说明，第一个数组的0号位置是有值的，第二个数组的0号位置没有值。

ES5 对空位的处理，已经很不一致了，大多数情况下会忽略空位。

* forEach(), filter(), every() 和some()都会跳过空位。
* map()会跳过空位，但会保留这个值
* join()和toString()会将空位视为undefined，而undefined和null会被处理成空字符串。

// forEach方法[,'a'].forEach((x,i) => console.log(i)); // 1

// filter方法['a',,'b'].filter(x => true) // ['a','b']

// every方法[,'a'].every(x => x==='a') // true

// some方法[,'a'].some(x => x !== 'a') // false

// map方法[,'a'].map(x => 1) // [,1]

// join方法[,'a',undefined,null].join('#') // "#a##"

// toString方法[,'a',undefined,null].toString() // ",a,,"

ES6 则是明确将空位转为undefined。

Array.from方法会将数组的空位，转为undefined，也就是说，这个方法不会忽略空位。

Array.from(['a',,'b'])

// [ "a", undefined, "b" ]

扩展运算符（...）也会将空位转为undefined。

[...['a',,'b']]

// [ "a", undefined, "b" ]

copyWithin()会连空位一起拷贝。

[,'a','b',,].copyWithin(2,0) // [,"a",,"a"]

fill()会将空位视为正常的数组位置。

new Array(3).fill('a') // ["a","a","a"]

for...of循环也会遍历空位。

let arr = [, ,];for (let i of arr) {

console.log(1);}

// 1// 1

上面代码中，数组arr有两个空位，for...of并没有忽略它们。如果改成map方法遍历，空位是会跳过的。

entries()、keys()、values()、find()和findIndex()会将空位处理成undefined。

// entries()[...[,'a'].entries()] // [[0,undefined], [1,"a"]]

// keys()[...[,'a'].keys()] // [0,1]

// values()[...[,'a'].values()] // [undefined,"a"]

// find()[,'a'].find(x => true) // undefined

// findIndex()[,'a'].findIndex(x => true) // 0

由于空位的处理规则非常不统一，所以建议避免出现空位。