

ECMAScript 6 入门

作者：阮一峰

授权：署名-非商用许可证



目录

- 0.前言
- 1.ECMA Script 6简介
- 2.let 和 const 命令
- 3.变量的解构赋值
- 4.字符串的扩展
- 5.字符串的新增方法
- 6.正则的扩展
- 7.数值的扩展
- 8.函数的扩展
- 9.数组的扩展
- 10.对象的扩展
- 11.对象的新增方法
- 12.Symbol
- 13.Set 和 Map 数据结构
- 14.Proxy
- 15.Reflect
- 16.Promise 对象
- 17.Iterator 和 for...of 循环
- 18.Generator 函数的语法
- 19.Generator 函数的异步应用
- 20.async 函数
- 21.Class 的基本语法
- 22.Class 的继承
- 23.Module 的语法
- 24.Module 的加载实现
- 25.编程风格
- 26.读懂规格
- 27.异步遍历器
- 28.ArrayBuffer
- 29.最新提案
- 30.Decorator
- 31.参考链接

其他

- 源码
- 修订历史
- 反馈意见

数组的扩展

- 1.扩展运算符
- 2.Array.from()
- 3.Array.of()

- 4.数组实例的 `copyWithin()`
- 5.数组实例的 `find()` 和 `findIndex()`
- 6.数组实例的 `fill()`
- 7.数组实例的 `entries()`, `keys()` 和 `values()`
- 8.数组实例的 `includes()`
- 9.数组实例的 `flat()`, `flatMap()`
- 10.数组的空位
- 11.`Array.prototype.sort()` 的排序稳定性

《ES6 实战教程》 深入学习一线大厂必备 ES6 技能。VIP 教程限时免费领取。 [⇐ 立即查看](#)

1. 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]
```

注意，只有函数调用时，扩展运算符才可以放在圆括号中，否则会报错。

```
(...[1, 2])
// Uncaught SyntaxError: Unexpected number

console.log(...[1, 2])
// Uncaught SyntaxError: Unexpected number

console.log(...[1, 2])
// 1 2
```

上面三种情况，扩展运算符都放在圆括号里面，但是前两种情况会报错，因为扩展运算符所在的括号不是函数调用。

替代函数的 **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。

```
// ES5
new (Date.bind.apply(Date, [null, 2015, 1, 1]))
// ES6
new 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。

```
const arr1 = ['a', 'b'];
const arr2 = ['c'];
const 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' ]
```

不过，这两种方法都是浅拷贝，使用的时候需要注意。

```
const a1 = [{ foo: 1 }];
const a2 = [{ bar: 2 }];

const a3 = a1.concat(a2);
const a4 = [...a1, ...a2];

a3[0] === a1[0] // true
a4[0] === a1[0] // true
```

上面代码中，`a3` 和 `a4` 是用两种不同方法合并而成的新数组，但是它们的成员都是对原数组成员的引用，这就是浅拷贝。如果修改了引用指向的值，会同步反映到新数组。

(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。

```
// ES5
a = 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 // 1
rest  // [2, 3, 4, 5]

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

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

任何定义了遍历器（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`。

```
Number.prototype[Symbol.iterator] = function*() {  
  let i = 0;  
  let num = this.valueOf();  
  while (i < num) {  
    yield i++;  
  }  
}  
  
console.log([...5]) // [0, 1, 2, 3, 4]
```

上面代码中，先定义了 `Number` 对象的遍历器接口，扩展运算符将 `5` 自动转成 `Number` 实例以后，就会调用这个接口，就会返回自定义的结果。

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
```

2. 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).filter(p => {
  return p.textContent.length > 100;
});

// 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 `filter` 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;
}
```

3. 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);
}
```

4. 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。如果为负值，表示倒数。
- `start`（可选）：从该位置开始读取数据，默认为 0。如果为负值，表示从末尾开始计算。
- `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，表示从末尾开始计算。

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]
```

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。

```
function f(v) {
  return v > this.age;
}
let person = {name: 'John', age: 20};
[10, 12, 26, 15].find(f, person);    // 26
```

上面的代码中，`find` 函数接收了第二个参数 `person` 对象，回调函数中的 `this` 对象指向 `person` 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。

6. 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。

注意，如果填充的类型为对象，那么被赋值的是同一个内存地址的对象，而不是深拷贝对象。

```
let arr = new Array(3).fill({name: "Mike"});
arr[0].name = "Ben";
arr
// [{name: "Ben"}, {name: "Ben"}, {name: "Ben"}]

let arr = new Array(3).fill([]);
arr[0].push(5);
arr
// [[5], [5], [5]]
```

7. 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
// 1

for (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']
```

8. 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` 的判断失败。

```
[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)` 。

9. 数组实例的 `flat()`，`flatMap()`

数组的成员有时还是数组，`Array.prototype.flat()` 用于将嵌套的数组“拉平”，变成一维的数组。该方法返回一个新数组，对原数据没有影响。

```
[1, 2, [3, 4]].flat()
// [1, 2, 3, 4]
```

上面代码中，原数组的成员里面有一个数组，`flat()` 方法将子数组的成员取出来，添加在原来的位置。

`flat()` 默认只会“拉平”一层，如果想要“拉平”多层的嵌套数组，可以将 `flat()` 方法的参数写成一个整数，表示想要拉平的层数，默认为 1。

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

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

上面代码中，`flat()` 的参数为 2，表示要“拉平”两层的嵌套数组。

如果不管有多少层嵌套，都要转成一维数组，可以用 `Infinity` 关键字作为参数。

```
[1, [2, [3]]].flat(Infinity)
// [1, 2, 3]
```

如果原数组有空位，`flat()` 方法会跳过空位。

```
[1, 2, , 4, 5].flat()  
// [1, 2, 4, 5]
```

`flatMap()` 方法对原数组的每个成员执行一个函数（相当于执行 `Array.prototype.map()` ），然后对返回值组成的数组执行 `flat()` 方法。该方法返回一个新数组，不改变原数组。

```
// 相当于 [[2, 4], [3, 6], [4, 8]].flat()  
[2, 3, 4].flatMap((x) => [x, x * 2])  
// [2, 4, 3, 6, 4, 8]
```

`flatMap()` 只能展开一层数组。

```
// 相当于 [[[2]], [[4]], [[6]], [[8]]].flat()  
[1, 2, 3, 4].flatMap(x => [[x * 2]])  
// [[2], [4], [6], [8]]
```

上面代码中，遍历函数返回的是一个双层的数组，但是默认只能展开一层，因此 `flatMap()` 返回的还是一个嵌套数组。

`flatMap()` 方法的参数是一个遍历函数，该函数可以接受三个参数，分别是当前数组成员、当前数组成员的位置（从零开始）、原数组。

```
arr.flatMap(function callback(currentValue[, index[, array]]) {  
  // ...  
}[, thisArg])
```

`flatMap()` 方法还可以有第二个参数，用来绑定遍历函数里面的 `this` 。

10. 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] // true  
0 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()`，`reduce()`，`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

[上一章](#)

[下一章](#)

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

```
// every方法
```

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

```
// reduce方法
```

```
[1,,2].reduce((x,y) => x+y) // 3
```

```
// 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。

11. Array.prototype.sort() 的排序稳定性

排序稳定性（stable sorting）是排序算法的重要属性，指的是排序关键字相同的项目，排序前后的顺序不变。

```
const arr = [
  'peach',
  'straw',
  'apple',
  'spork'
];

const stableSorting = (s1, s2) => {
  if (s1[0] < s2[0]) return -1;
  return 1;
};

arr.sort(stableSorting)
// ["apple", "peach", "straw", "spork"]
```

上面代码对数组 `arr` 按照首字母进行排序。排序结果中，`straw` 在 `spork` 的前面，跟原始顺序一致，所以排序算法 `stableSorting` 是稳定排序。

```
const unstableSorting = (s1, s2) => {
  if (s1[0] <= s2[0]) return -1;
  return 1;
};

arr.sort(unstableSorting)
// ["apple", "peach", "spork", "straw"]
```

上面代码中，排序结果是 `spork` 在 `straw` 前面，跟原始顺序相反，所以排序算法 `unstableSorting` 是不稳定的。

常见的排序算法之中，插入排序、合并排序、冒泡排序等都是稳定的，堆排序、快速排序等是不稳定的。不稳定排序的主要缺点是，多重排序时可能会产生问题。假设有一个姓和名的列表，要求按照“姓氏为主要关键字，名字为次要关键字”进行排序。开发者可能会先按名字排序，再按姓氏进行排序。如果排序算法是稳定的，这样就可以达到“先姓氏，后名字”的排序效果。如果是不稳定的，就不行。

早先的 ECMAScript 没有规定，`Array.prototype.sort()` 的默认排序算法是否稳定，留给浏览器自己决定，这导致某些实现是不稳定的。**ES2019** 明确规定，`Array.prototype.sort()` 的默认排序算法必须稳定。这个规定已经做到了，现在 JavaScript 各个主要实现的默认排序算法都是稳定的。

