

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.ECMAScript 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.正则的扩展
- 6.数值的扩展
- 7.函数的扩展
- 8.数组的扩展
- 9.对象的扩展
- 10.对象的新增方法
- 11.Symbol
- 12.Set 和 Map 数据结构
- 13.Proxy
- 14.Reflect
- 15.Promise 对象
- 16.Iterator 和 for...of 循环
- 17.Generator 函数的语法
- 18.Generator 函数的异步应用
- 19.async 函数
- 20.Class 的基本语法
- 21.Class 的继承
- 22.Decorator
- 23.Module 的语法
- 24.Module 的加载实现
- 25.编程风格
- 26.读懂规格
- 27.ArrayBuffer
- 28.最新提案
- 29.参考链接

其他

- 源码
- 修订历史
- 反馈意见

Set 和 Map 数据结构

- 1.Set
- 2.WeakSet
- 3.Map
- 4.WeakMap

1. Set

基本用法

ES6 提供了新的数据结构 **Set**。它类似于数组，但是成员的值都是唯一的，没有重复的值。

Set 本身是一个构造函数，用来生成 **Set** 数据结构。

```
const s = new Set();

[2, 3, 5, 4, 5, 2, 2].forEach(x => s.add(x));

for (let i of s) {
  console.log(i);
}
// 2 3 5 4
```

上面代码通过 **add()** 方法向 **Set** 结构加入成员，结果表明 **Set** 结构不会添加重复的值。

Set 函数可以接受一个数组（或者具有 **iterable** 接口的其他数据结构）作为参数，用来初始化。

```
// 例一
const set = new Set([1, 2, 3, 4, 4]);
[...set]
// [1, 2, 3, 4]

// 例二
const items = new Set([1, 2, 3, 4, 5, 5, 5, 5]);
items.size // 5

// 例三
const set = new Set(document.querySelectorAll('div'));
set.size // 56

// 类似于
const set = new Set();
document
  .querySelectorAll('div')
  .forEach(div => set.add(div));
set.size // 56
```

上面代码中，例一和例二都是 **Set** 函数接受数组作为参数，例三是接受类似数组的对象作为参数。

上面代码也展示了一种去除数组重复成员的方法。

```
// 去除数组的重复成员
[...new Set(array)]
```

上面的方法也可以用于，去除字符串里面的重复字符。

```
[...new Set('ababbc')].join('')
// "abc"
```

向 **Set** 加入值的时候，不会发生类型转换，所以 **5** 和 **"5"** 是两个不同的值。**Set** 内部判断两个值是否不同，使用的算法叫做“**Same-value-zero equality**”，它类似于精确相等运算符（**===**），主要的区别是 **NaN** 等于自身，而精确相等运算符认为 **NaN** 不等于自身。

```
let set = new Set();
let a = NaN;
let b = NaN;
set.add(a);
set.add(b);
set // Set {NaN}
```

上面代码向 **Set** 实例添加了两个 **NaN**，但是只能加入一个。这表明，在 **Set** 内部，两个 **NaN** 是相等。

另外，两个对象总是不相等的。

```
let set = new Set();

set.add({});
set.size // 1

set.add({});
set.size // 2
```

上面代码表示，由于两个空对象不相等，所以它们被视为两个值。

Set 实例的属性和方法

Set 结构的实例有以下属性。

- **Set.prototype.constructor**：构造函数，默认就是 **Set** 函数。
- **Set.prototype.size**：返回 **Set** 实例的成员总数。

Set 实例的方法分为两大类：操作方法（用于操作数据）和遍历方法（用于遍历成员）。下面先介绍四个操作方法。

- **add(value)**：添加某个值，返回 **Set** 结构本身。
- **delete(value)**：删除某个值，返回一个布尔值，表示删除是否成功。
- **has(value)**：返回一个布尔值，表示该值是否为 **Set** 的成员。
- **clear()**：清除所有成员，没有返回值。

上面这些属性和方法的实例如下。

```
s.add(1).add(2).add(2);
// 注意2被加入了两次

s.size // 2

s.has(1) // true
s.has(2) // true
s.has(3) // false

s.delete(2);
s.has(2) // false
```

下面是一个对比，看看在判断是否包括一个键上面，**Object** 结构和 **Set** 结构的写法不同。

```
// 对象的写法
const properties = {
  'width': 1,
  'height': 1
```

```
};

if (properties[someName]) {
  // do something
}

// Set的写法
const properties = new Set();

properties.add('width');
properties.add('height');

if (properties.has(someName)) {
  // do something
}
```

`Array.from` 方法可以将 `Set` 结构转为数组。

```
const items = new Set([1, 2, 3, 4, 5]);
const array = Array.from(items);
```

这就提供了去除数组重复成员的另一种方法。

```
function dedupe(array) {
  return Array.from(new Set(array));
}

dedupe([1, 1, 2, 3]) // [1, 2, 3]
```

遍历操作

`Set` 结构的实例有四个遍历方法，可以用于遍历成员。

- `keys()`：返回键名的遍历器
- `values()`：返回键值的遍历器
- `entries()`：返回键值对的遍历器
- `forEach()`：使用回调函数遍历每个成员

需要特别指出的是，`Set` 的遍历顺序就是插入顺序。这个特性有时非常有用，比如使用 `Set` 保存一个回调函数列表，调用时就能保证按照添加顺序调用。

(1) `keys()`，`values()`，`entries()`

`keys` 方法、`values` 方法、`entries` 方法返回的都是遍历器对象（详见《Iterator 对象》一章）。由于 `Set` 结构没有键名，只有键值（或者说键名和键值是同一个值），所以 `keys` 方法和 `values` 方法的行为完全一致。

```
let set = new Set(['red', 'green', 'blue']);

for (let item of set.keys()) {
  console.log(item);
}
// red
// green
// blue

for (let item of set.values()) {
  console.log(item);
```

```

    }
    // red
    // green
    // blue

    for (let item of set.entries()) {
        console.log(item);
    }
    // ["red", "red"]
    // ["green", "green"]
    // ["blue", "blue"]

```

上面代码中，`entries` 方法返回的遍历器，同时包括键名和键值，所以每次输出一个数组，它的两个成员完全相等。

`Set` 结构的实例默认可遍历，它的默认遍历器生成函数就是它的 `values` 方法。

```

Set.prototype[Symbol.iterator] === Set.prototype.values
// true

```

这意味着，可以省略 `values` 方法，直接用 `for...of` 循环遍历 `Set`。

```

let set = new Set(['red', 'green', 'blue']);

for (let x of set) {
    console.log(x);
}
// red
// green
// blue

```

(2) `forEach()`

`Set` 结构的实例与数组一样，也拥有 `forEach` 方法，用于对每个成员执行某种操作，没有返回值。

```

let set = new Set([1, 4, 9]);
set.forEach((value, key) => console.log(key + ' : ' + value))
// 1 : 1
// 4 : 4
// 9 : 9

```

上面代码说明，`forEach` 方法的参数就是一个处理函数。该函数的参数与数组的 `forEach` 一致，依次为键值、键名、集合本身（上例省略了该参数）。这里需要注意，`Set` 结构的键名就是键值（两者是同一个值），因此第一个参数与第二个参数的值永远都是一样的。

另外，`forEach` 方法还可以有第二个参数，表示绑定处理函数内部的 `this` 对象。

(3) 遍历的应用

扩展运算符（`...`）内部使用 `for...of` 循环，所以也可以用于 `Set` 结构。

```

let set = new Set(['red', 'green', 'blue']);
let arr = [...set];
// ['red', 'green', 'blue']

```

扩展运算符和 `Set` 结构相结合，就可以去除数组的重复成员。

```

let arr = [3, 5, 2, 2, 5, 5];
let unique = [...new Set(arr)];
// [3, 5, 2]

```

而且，数组的 `map` 和 `filter` 方法也可以间接用于 `Set` 了

[上一章](#)

[下一章](#)

```
let set = new Set([1, 2, 3]);
set = new Set([...set].map(x => x * 2));
// 返回Set结构: {2, 4, 6}

let set = new Set([1, 2, 3, 4, 5]);
set = new Set([...set].filter(x => (x % 2) == 0));
// 返回Set结构: {2, 4}
```

因此使用 **Set** 可以很容易地实现并集（Union）、交集（Intersect）和差集（Difference）。

```
let a = new Set([1, 2, 3]);
let b = new Set([4, 3, 2]);

// 并集
let union = new Set([...a, ...b]);
// Set {1, 2, 3, 4}

// 交集
let intersect = new Set([...a].filter(x => b.has(x)));
// set {2, 3}

// 差集
let difference = new Set([...a].filter(x => !b.has(x)));
// Set {1}
```

如果想在遍历操作中，同步改变原来的 **Set** 结构，目前没有直接的方法，但有两种变通方法。一种是利用原 **Set** 结构映射出一个新的结构，然后赋值给原来的 **Set** 结构；另一种是利用 **Array.from** 方法。

```
// 方法一
let set = new Set([1, 2, 3]);
set = new Set([...set].map(val => val * 2));
// set的值是2, 4, 6

// 方法二
let set = new Set([1, 2, 3]);
set = new Set(Array.from(set, val => val * 2));
// set的值是2, 4, 6
```

上面代码提供了两种方法，直接在遍历操作中改变原来的 **Set** 结构。

2. WeakSet

含义

WeakSet 结构与 **Set** 类似，也是不重复的值的集合。但是，它与 **Set** 有两个区别。

首先，**WeakSet** 的成员只能是对象，而不能是其他类型的值。

```
const ws = new WeakSet();
ws.add(1)
// TypeError: Invalid value used in weak set
ws.add(Symbol())
// TypeError: invalid value used in weak set
```

上面代码试图向 **WeakSet** 添加一个数值和 **Symbol** 值，[上一章](#) [母](#) [下一章](#)：只能放置对象。

其次，**WeakSet** 中的对象都是弱引用，即垃圾回收机制不考虑 **WeakSet** 对该对象的引用，也就是说，如果其他对象都不再引用该对象，那么垃圾回收机制会自动回收该对象所占用的内存，不考虑该对象还存在于 **WeakSet** 之中。

这是因为垃圾回收机制依赖引用计数，如果一个值的引用次数不为 **0**，垃圾回收机制就不会释放这块内存。结束使用该值之后，有时会忘记取消引用，导致内存无法释放，进而可能会引发内存泄漏。**WeakSet** 里面的引用，都不计入垃圾回收机制，所以就不存在这个问题。因此，**WeakSet** 适合临时存放一组对象，以及存放跟对象绑定的信息。只要这些对象在外部消失，它在 **WeakSet** 里面的引用就会自动消失。

由于上面这个特点，**WeakSet** 的成员是不适合引用的，因为它会随时消失。另外，由于 **WeakSet** 内部有多少个成员，取决于垃圾回收机制有没有运行，运行前后很可能成员个数是不一样的，而垃圾回收机制何时运行是不可预测的，因此 ES6 规定 **WeakSet** 不可遍历。

这些特点同样适用于本章后面要介绍的 **WeakMap** 结构。

语法

WeakSet 是一个构造函数，可以使用 **new** 命令，创建 **WeakSet** 数据结构。

```
const ws = new WeakSet();
```

作为构造函数，**WeakSet** 可以接受一个数组或类似数组的对象作为参数。（实际上，任何具有 **Iterable** 接口的对象，都可以作为 **WeakSet** 的参数。）该数组的所有成员，都会自动成为 **WeakSet** 实例对象的成员。

```
const a = [[1, 2], [3, 4]];
const ws = new WeakSet(a);
// WeakSet {[1, 2], [3, 4]}
```

上面代码中，**a** 是一个数组，它有两个成员，也都是数组。将 **a** 作为 **WeakSet** 构造函数的参数，**a** 的成员会自动成为 **WeakSet** 的成员。

注意，是 **a** 数组的成员成为 **WeakSet** 的成员，而不是 **a** 数组本身。这意味着，数组的成员只能是对象。

```
const b = [3, 4];
const ws = new WeakSet(b);
// Uncaught TypeError: Invalid value used in weak set(...)
```

上面代码中，数组 **b** 的成员不是对象，加入 **WeakSet** 就会报错。

WeakSet 结构有以下三个方法。

- **WeakSet.prototype.add(value)**: 向 **WeakSet** 实例添加一个新成员。
- **WeakSet.prototype.delete(value)**: 清除 **WeakSet** 实例的指定成员。
- **WeakSet.prototype.has(value)**: 返回一个布尔值，表示某个值是否在 **WeakSet** 实例之中。

下面是一个例子。

```
const ws = new WeakSet();
const obj = {};
const foo = {};

ws.add(window);
ws.add(obj);

ws.has(window); // true
ws.has(foo);    // false
```

```
ws.delete(window);  
ws.has(window);    // false
```

WeakSet 没有 `size` 属性，没有办法遍历它的成员。

```
ws.size // undefined  
ws.forEach // undefined  
  
ws.forEach(function(item){ console.log('WeakSet has ' + item)})  
// TypeError: undefined is not a function
```

上面代码试图获取 `size` 和 `forEach` 属性，结果都不能成功。

WeakSet 不能遍历，是因为成员都是弱引用，随时可能消失，遍历机制无法保证成员的存在，很可能刚刚遍历结束，成员就取不到了。WeakSet 的一个用处，是储存 DOM 节点，而不用担心这些节点从文档移除时，会引发内存泄漏。

下面是 WeakSet 的另一个例子。

```
const foos = new WeakSet()  
class Foo {  
  constructor() {  
    foos.add(this)  
  }  
  method () {  
    if (!foos.has(this)) {  
      throw new TypeError('Foo.prototype.method 只能在Foo的实例上调用！');  
    }  
  }  
}
```

上面代码保证了 `Foo` 的实例方法，只能在 `Foo` 的实例上调用。这里使用 WeakSet 的好处是，`foos` 对实例的引用，不会被计入内存回收机制，所以删除实例的时候，不用考虑 `foos`，也不会出现内存泄漏。

3. Map

含义和基本用法

JavaScript 的对象（Object），本质上是键值对的集合（Hash 结构），但是传统上只能用字符串当作键。这给它的使用带来了很大的限制。

```
const data = {};  
const element = document.getElementById('myDiv');  
  
data[element] = 'metadata';  
data['[object HTMLDivElement]'] // "metadata"
```

上面代码原意是将一个 DOM 节点作为对象 `data` 的键，但是由于对象只接受字符串作为键名，所以 `element` 被自动转为字符串 `[object HTMLDivElement]`。

为了解决这个问题，ES6 提供了 Map 数据结构。它类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。也就是说，Object 结构提供了“字符串—值”的对应，Map 结构提供了“值—值”的对应，是一种更完善的 Hash 结构实现。如果你需要“键值对”的数据结构，Map 比 Object 更合适。


```
const m = new Map();
const o = {p: 'Hello World'};

m.set(o, 'content')
m.get(o) // "content"

m.has(o) // true
m.delete(o) // true
m.has(o) // false
```

上面代码使用 **Map** 结构的 **set** 方法，将对象 **o** 当作 **m** 的一个键，然后又使用 **get** 方法读取这个键，接着使用 **delete** 方法删除了这个键。

上面的例子展示了如何向 **Map** 添加成员。作为构造函数，**Map** 也可以接受一个数组作为参数。该数组的成员是一个个表示键值对的数组。

```
const map = new Map([
  ['name', '张三'],
  ['title', 'Author']
]);

map.size // 2
map.has('name') // true
map.get('name') // "张三"
map.has('title') // true
map.get('title') // "Author"
```

上面代码在新建 **Map** 实例时，就指定了两个键 **name** 和 **title**。

Map 构造函数接受数组作为参数，实际上执行的是下面的算法。

```
const items = [
  ['name', '张三'],
  ['title', 'Author']
];

const map = new Map();

items.forEach(
  ([key, value]) => map.set(key, value)
);
```

事实上，不仅仅是数组，任何具有 **Iterator** 接口、且每个成员都是一个双元素的数组的数据结构（详见《**Iterator**》一章）都可以当作 **Map** 构造函数的参数。这就是说，**Set** 和 **Map** 都可以用来生成新的 **Map**。

```
const set = new Set([
  ['foo', 1],
  ['bar', 2]
]);
const m1 = new Map(set);
m1.get('foo') // 1

const m2 = new Map([['baz', 3]]);
const m3 = new Map(m2);
m3.get('baz') // 3
```

上面代码中，我们分别使用 **Set** 对象和 **Map** 对象，当作 **Map** 构造函数的参数，结果都生成了新的 **Map** 对象。

如果对同一个键多次赋值，后面的值将覆盖前面的值。

```
const map = new Map();

map
```

```
.set(1, 'aaa')
.set(1, 'bbb');

map.get(1) // "bbb"
```

上面代码对键 **1** 连续赋值两次，后一次的值覆盖前一次的值。

如果读取一个未知的键，则返回 **undefined**。

```
new Map().get('asfddfsasadf')
// undefined
```

注意，只有对同一个对象的引用，**Map** 结构才将其视为同一个键。这一点要非常小心。

```
const map = new Map();

map.set(['a'], 555);
map.get(['a']) // undefined
```

上面代码的 **set** 和 **get** 方法，表面是针对同一个键，但实际上这是两个值，内存地址是不一样的，因此 **get** 方法无法读取该键，返回 **undefined**。

同理，同样的值的两个实例，在 **Map** 结构中被视为两个键。

```
const map = new Map();

const k1 = ['a'];
const k2 = ['a'];

map
.set(k1, 111)
.set(k2, 222);

map.get(k1) // 111
map.get(k2) // 222
```

上面代码中，变量 **k1** 和 **k2** 的值是一样的，但是它们在 **Map** 结构中被视为两个键。

由上可知，**Map** 的键实际上是跟内存地址绑定的，只要内存地址不一样，就视为两个键。这就解决了同名属性碰撞（**clash**）的问题，我们扩展别人的库的时候，如果使用对象作为键名，就不用担心自己的属性与原作者的属性同名。

如果 **Map** 的键是一个简单类型的值（数字、字符串、布尔值），则只要两个值严格相等，**Map** 将其视为一个键，比如 **0** 和 **-0** 就是一个键，布尔值 **true** 和字符串 **true** 则是两个不同的键。另外，**undefined** 和 **null** 也是两个不同的键。虽然 **NaN** 不严格相等于自身，但 **Map** 将其视为同一个键。

```
let map = new Map();

map.set(-0, 123);
map.get(+0) // 123

map.set(true, 1);
map.set('true', 2);
map.get(true) // 1

map.set(undefined, 3);
map.set(null, 4);
map.get(undefined) // 3

map.set(NaN, 123);
map.get(NaN) // 123
```

实例的属性和操作方法

Map 结构的实例有以下属性和操作方法。

(1) size 属性

size 属性返回 Map 结构的成员总数。

```
const map = new Map();
map.set('foo', true);
map.set('bar', false);

map.size // 2
```

(2) set(key, value)

set 方法设置键名 key 对应的键值为 value，然后返回整个 Map 结构。如果 key 已经有值，则键值会被更新，否则就新生成该键。

```
const m = new Map();

m.set('edition', 6)      // 键是字符串
m.set(262, 'standard')    // 键是数值
m.set(undefined, 'nah')   // 键是 undefined
```

set 方法返回的是当前的 Map 对象，因此可以采用链式写法。

```
let map = new Map()
  .set(1, 'a')
  .set(2, 'b')
  .set(3, 'c');
```

(3) get(key)

get 方法读取 key 对应的键值，如果找不到 key，返回 undefined。

```
const m = new Map();

const hello = function() {console.log('hello');};
m.set(hello, 'Hello ES6!') // 键是函数

m.get(hello) // Hello ES6!
```

(4) has(key)

has 方法返回一个布尔值，表示某个键是否在当前 Map 对象之中。

```
const m = new Map();

m.set('edition', 6);
m.set(262, 'standard');
m.set(undefined, 'nah');

m.has('edition')    // true
m.has('years')      // false
m.has(262)          // true
m.has(undefined)    // true
```

(5) delete(key)

`delete` 方法删除某个键，返回 `true`。如果删除失败，返回 `false`。

```
const m = new Map();
m.set(undefined, 'nah');
m.has(undefined)    // true

m.delete(undefined)
m.has(undefined)    // false
```

(6) `clear()`

`clear` 方法清除所有成员，没有返回值。

```
let map = new Map();
map.set('foo', true);
map.set('bar', false);

map.size // 2
map.clear()
map.size // 0
```

遍历方法

`Map` 结构原生提供三个遍历器生成函数和一个遍历方法。

- `keys()`：返回键名的遍历器。
- `values()`：返回键值的遍历器。
- `entries()`：返回所有成员的遍历器。
- `forEach()`：遍历 `Map` 的所有成员。

需要特别注意的是，`Map` 的遍历顺序就是插入顺序。

```
const map = new Map([
  ['F', 'no'],
  ['T', 'yes'],
]);

for (let key of map.keys()) {
  console.log(key);
}
// "F"
// "T"

for (let value of map.values()) {
  console.log(value);
}
// "no"
// "yes"

for (let item of map.entries()) {
  console.log(item[0], item[1]);
}
// "F" "no"
// "T" "yes"

// 或者
for (let [key, value] of map.entries()) {
  console.log(key, value);
}
```

```
// "F" "no"
// "T" "yes"

// 等同于使用map.entries()
for (let [key, value] of map) {
  console.log(key, value);
}
// "F" "no"
// "T" "yes"
```

上面代码最后的那个例子，表示 Map 结构的默认遍历器接口（`Symbol.iterator` 属性），就是 `entries` 方法。

```
map[Symbol.iterator] === map.entries
// true
```

Map 结构转为数组结构，比较快速的方法是使用扩展运算符（`...`）。

```
const map = new Map([
  [1, 'one'],
  [2, 'two'],
  [3, 'three'],
]);

[...map.keys()]
// [1, 2, 3]

[...map.values()]
// ['one', 'two', 'three']

[...map.entries()]
// [[1, 'one'], [2, 'two'], [3, 'three']]

[...map]
// [[1, 'one'], [2, 'two'], [3, 'three']]
```

结合数组的 `map` 方法、`filter` 方法，可以实现 Map 的遍历和过滤（Map 本身没有 `map` 和 `filter` 方法）。

```
const map0 = new Map()
  .set(1, 'a')
  .set(2, 'b')
  .set(3, 'c');

const map1 = new Map(
  [...map0].filter(([k, v]) => k < 3)
);
// 产生 Map 结构 {1 => 'a', 2 => 'b'}

const map2 = new Map(
  [...map0].map(([k, v]) => [k * 2, '_' + v])
);
// 产生 Map 结构 {2 => '_a', 4 => '_b', 6 => '_c'}
```

此外，Map 还有一个 `forEach` 方法，与数组的 `forEach` 方法类似，也可以实现遍历。

```
map.forEach(function(value, key, map) {
  console.log("Key: %s, Value: %s", key, value);
});
```

`forEach` 方法还可以接受第二个参数，用来绑定 `this` 。

```
const reporter = {
  report: function(key, value) {
```

```
        console.log("Key: %s, Value: %s", key, value);
    }
};

map.forEach(function(value, key, map) {
    this.report(key, value);
}, reporter);
```

上面代码中，`forEach` 方法的回调函数的 `this`，就指向 `reporter`。

与其他数据结构的互相转换

（1）Map 转为数组

前面已经提过，Map 转为数组最方便的方法，就是使用扩展运算符（`...`）。

```
const myMap = new Map()
  .set(true, 7)
  .set({foo: 3}, ['abc']);
[...myMap]
// [ [ true, 7 ], [ { foo: 3 }, [ 'abc' ] ] ]
```

（2）数组 转为 Map

将数组传入 Map 构造函数，就可以转为 Map。

```
new Map([
  [true, 7],
  [{foo: 3}, ['abc']]
])
// Map {
//   true => 7,
//   Object {foo: 3} => ['abc']
// }
```

（3）Map 转为对象

如果所有 Map 的键都是字符串，它可以无损地转为对象。

```
function strMapToObj(strMap) {
  let obj = Object.create(null);
  for (let [k,v] of strMap) {
    obj[k] = v;
  }
  return obj;
}

const myMap = new Map()
  .set('yes', true)
  .set('no', false);
strMapToObj(myMap)
// { yes: true, no: false }
```

如果有非字符串的键名，那么这个键名会被转成字符串，再作为对象的键名。

（4）对象转为 Map

```
function objToStrMap(obj) {
  let strMap = new Map();
```

```

    for (let k of Object.keys(obj)) {
      strMap.set(k, obj[k]);
    }
    return strMap;
  }

  objToStrMap({yes: true, no: false})
  // Map {"yes" => true, "no" => false}

```

（5）Map 转为 JSON

Map 转为 JSON 要区分两种情况。一种情况是，Map 的键名都是字符串，这时可以选择转为对象 JSON。

```

function strMapToJson(strMap) {
  return JSON.stringify(strMapToObj(strMap));
}

let myMap = new Map().set('yes', true).set('no', false);
strMapToJson(myMap)
// '{"yes":true,"no":false}'

```

另一种情况是，Map 的键名有非字符串，这时可以选择转为数组 JSON。

```

function mapToArrayJson(map) {
  return JSON.stringify([...map]);
}

let myMap = new Map().set(true, 7).set({foo: 3}, ['abc']);
mapToArrayJson(myMap)
// '[[true,7],[{"foo":3},["abc"]]]'

```

（6）JSON 转为 Map

JSON 转为 Map，正常情况下，所有键名都是字符串。

```

function jsonToStrMap(jsonStr) {
  return objToStrMap(JSON.parse(jsonStr));
}

jsonToStrMap('{"yes": true, "no": false}')
// Map {"yes" => true, "no" => false}

```

但是，有一种特殊情况，整个 JSON 就是一个数组，且每个数组成员本身，又是一个有两个成员的数组。这时，它可以一一对应地转为 Map。这往往是 Map 转为数组 JSON 的逆操作。

```

function jsonToMap(jsonStr) {
  return new Map(JSON.parse(jsonStr));
}

jsonToMap('[[true,7],[{"foo":3},["abc"]]]')
// Map {true => 7, Object {foo: 3} => ['abc']}

```

4. WeakMap

WeakMap 结构与 **Map** 结构类似，也是用于生成键值对的集合。

```
// WeakMap 可以使用 set 方法添加成员
const wm1 = new WeakMap();
const key = {foo: 1};
wm1.set(key, 2);
wm1.get(key) // 2

// WeakMap 也可以接受一个数组，
// 作为构造函数的参数
const k1 = [1, 2, 3];
const k2 = [4, 5, 6];
const wm2 = new WeakMap([[k1, 'foo'], [k2, 'bar']]);
wm2.get(k2) // "bar"
```

WeakMap 与 **Map** 的区别有两点。

首先，**WeakMap** 只接受对象作为键名（**null** 除外），不接受其他类型的值作为键名。

```
const map = new WeakMap();
map.set(1, 2)
// TypeError: 1 is not an object!
map.set(Symbol(), 2)
// TypeError: Invalid value used as weak map key
map.set(null, 2)
// TypeError: Invalid value used as weak map key
```

上面代码中，如果将数值 **1** 和 **Symbol** 值作为 **WeakMap** 的键名，都会报错。

其次，**WeakMap** 的键名所指向的对象，不计入垃圾回收机制。

WeakMap 的设计目的在于，有时我们想在某个对象上面存放一些数据，但是这会形成对于这个对象的引用。请看下面的例子。

```
const e1 = document.getElementById('foo');
const e2 = document.getElementById('bar');
const arr = [
  [e1, 'foo 元素'],
  [e2, 'bar 元素'],
];
```

上面代码中，**e1** 和 **e2** 是两个对象，我们通过 **arr** 数组对这两个对象添加一些文字说明。这就形成了 **arr** 对 **e1** 和 **e2** 的引用。

一旦不再需要这两个对象，我们就必须手动删除这个引用，否则垃圾回收机制就不会释放 **e1** 和 **e2** 占用的内存。

```
// 不需要 e1 和 e2 的时候
// 必须手动删除引用
arr[0] = null;
arr[1] = null;
```

上面这样的写法显然很不方便。一旦忘了写，就会造成内存泄露。

WeakMap 就是为了解决这个问题而诞生的，它的键名所引用的对象都是弱引用，即垃圾回收机制不将该引用考虑在内。因此，只要所引用的对象的其他引用都被清除，垃圾回收机制就会释放该对象所占用的内存。也就是说，一旦不再需要，**WeakMap** 里面的键名对象和所对应的键值对会自动消失，不用手动删除引用。

基本上，如果你要往对象上添加数据，又不想干扰垃圾回收机制，就可以使用 **WeakMap**。一个典型应用场景是，在网页的 **DOM** 元素上添加数据，就可以使用 **WeakMap** 结构。当该 **DOM** 元素被清除，其所对应的 **WeakMap** 记录就会自动被移除。

```
const wm = new WeakMap();
```



```
const element = document.getElementById('example');
```

```
wm.set(element, 'some information');  
wm.get(element) // "some information"
```

上面代码中，先新建一个 **Weakmap** 实例。然后，将一个 **DOM** 节点作为键名存入该实例，并将一些附加信息作为键值，一起存放在 **WeakMap** 里面。这时，**WeakMap** 里面对 **element** 的引用就是弱引用，不会被计入垃圾回收机制。

也就是说，上面的 **DOM** 节点对象的引用计数是 **1**，而不是 **2**。这时，一旦消除对该节点的引用，它占用的内存就会被垃圾回收机制释放。**Weakmap** 保存的这个键值对，也会自动消失。

总之，**WeakMap** 的专用场合就是，它的键所对应的对象，可能会在将来消失。**WeakMap** 结构有助于防止内存泄漏。

注意，**WeakMap** 弱引用的只是键名，而不是键值。键值依然是正常引用。

```
const wm = new WeakMap();  
let key = {};  
let obj = {foo: 1};  
  
wm.set(key, obj);  
obj = null;  
wm.get(key)  
// Object {foo: 1}
```

上面代码中，键值 **obj** 是正常引用。所以，即使在 **WeakMap** 外部消除了 **obj** 的引用，**WeakMap** 内部的引用依然存在。

WeakMap 的语法

WeakMap 与 **Map** 在 **API** 上的区别主要是两个，一是没有遍历操作（即没有 **keys()**、**values()** 和 **entries()** 方法），也没有 **size** 属性。因为没有办法列出所有键名，某个键名是否存在完全不可预测，跟垃圾回收机制是否运行相关。这一时刻可以取到键名，下一刻垃圾回收机制突然运行了，这个键名就没了，为了防止出现不确定性，就统一规定不能取到键名。二是无法清空，即不支持 **clear** 方法。因此，**WeakMap** 只有四个方法可用：**get()**、**set()**、**has()**、**delete()**。

```
const wm = new WeakMap();  
  
// size、forEach、clear 方法都不存在  
wm.size // undefined  
wm.forEach // undefined  
wm.clear // undefined
```

WeakMap 的示例

WeakMap 的例子很难演示，因为无法观察它里面的引用会自动消失。此时，其他引用都解除了，已经没有引用指向 **WeakMap** 的键名了，导致无法证实那个键名是不是存在。

贺师俊老师提示，如果引用所指向的值占用特别多的内存，就可以通过 **Node** 的 **process.memoryUsage** 方法看出来。根据这个思路，网友 **vtxf** 补充了下面的例子。

首先，打开 **Node** 命令行。

```
$ node --expose-gc
```

上面代码中，`--expose-gc` 参数表示允许手动执行垃圾回收机制。

然后，执行下面的代码。

```
// 手动执行一次垃圾回收，保证获取的内存使用状态准确
> global.gc();
undefined

// 查看内存占用的初始状态，heapUsed 为 4M 左右
> process.memoryUsage();
{ rss: 21106688,
  heapTotal: 7376896,
  heapUsed: 4153936,
  external: 9059 }

> let wm = new WeakMap();
undefined

// 新建一个变量 key，指向一个 5*1024*1024 的数组
> let key = new Array(5 * 1024 * 1024);
undefined

// 设置 WeakMap 实例的键名，也指向 key 数组
// 这时，key 数组实际被引用了两次，
// 变量 key 引用一次，WeakMap 的键名引用了第二次
// 但是，WeakMap 是弱引用，对于引擎来说，引用计数还是1
> wm.set(key, 1);
WeakMap {}

> global.gc();
undefined

// 这时内存占用 heapUsed 增加到 45M 了
> process.memoryUsage();
{ rss: 67538944,
  heapTotal: 7376896,
  heapUsed: 45782816,
  external: 8945 }

// 清除变量 key 对数组的引用，
// 但没有手动清除 WeakMap 实例的键名对数组的引用
> key = null;
null

// 再次执行垃圾回收
> global.gc();
undefined

// 内存占用 heapUsed 变回 4M 左右，
// 可以看到 WeakMap 的键名引用没有阻止 gc 对内存的回收
> process.memoryUsage();
{ rss: 20639744,
  heapTotal: 8425472,
  heapUsed: 3979792,
  external: 8956 }
```

上面代码中，只要外部的引用消失，WeakMap 内部的引用，就会自动被垃圾回收清除。由此可见，有了 WeakMap 的帮助，解决内存泄漏就会简单很多。

WeakMap 的用途

前文说过，WeakMap 应用的典型场合就是 DOM 节点作为键名。下面是一个例子。

[上一章](#)

[下一章](#)

```
let myElement = document.getElementById('logo');
let myWeakmap = new WeakMap();

myWeakmap.set(myElement, {timesClicked: 0});

myElement.addEventListener('click', function() {
  let logoData = myWeakmap.get(myElement);
  logoData.timesClicked++;
}, false);
```

上面代码中，`myElement` 是一个 DOM 节点，每当发生 `click` 事件，就更新一下状态。我们将这个状态作为键值放在 `WeakMap` 里，对应的键名就是 `myElement`。一旦这个 DOM 节点删除，该状态就会自动消失，不存在内存泄漏风险。

`WeakMap` 的另一个用处是部署私有属性。

```
const _counter = new WeakMap();
const _action = new WeakMap();

class Countdown {
  constructor(counter, action) {
    _counter.set(this, counter);
    _action.set(this, action);
  }
  dec() {
    let counter = _counter.get(this);
    if (counter < 1) return;
    counter--;
    _counter.set(this, counter);
    if (counter === 0) {
      _action.get(this)();
    }
  }
}

const c = new Countdown(2, () => console.log('DONE'));

c.dec()
c.dec()
// DONE
```

上面代码中，`Countdown` 类的两个内部属性 `_counter` 和 `_action`，是实例的弱引用，所以如果删除实例，它们也就随之消失，不会造成内存泄漏。

留言

91 Comments

ECMAScript 6 入门

1 Login ▾

Recommend 9

Tweet

Share

Sort by Best ▾



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name



slogear • 2 years ago

上一章

下一章



“WeakSet不能遍历，是因为成员都是弱引用，随时可能消失，遍历机制无法保证成员的存在，很可能刚刚遍历结束，成员就取不到了。WeakSet的一个用处，是储存DOM节点，而不用担心这些节点从文档移除时，会引发内存泄漏。”

WeakSet的一个用处，是储存DOM节点，而不用担心这些节点从文档移除时，会引发内存泄漏。这句话不太明白。如果使用Set存储DOM节点，当这些节点从文档移除时，为何会引起内存泄漏。请大神解释

84 ^ | v • Reply • Share ›



庄浩铭 → slogeor • 2 years ago

“引起内存泄漏”并不代表说一定会内存泄漏，只是说增加了内存泄漏的风险。如果使用Set存储DOM节点,由于Set对成员的引用是计入“引用计数”的，所以从文档中移除节点后，垃圾回收器还是没有办法回收节点的内存，只有你显式地在Set中删除节点后节点的内存才会被回收。

^ | v • Reply • Share ›



anjianshi • 3 years ago

WeakMap 那里有个错别字：“WeakMap与Map在API上的区别主要是两个，一是没有遍历操作（即没有key()、values()和entries()方法）”，key() 应该是 keys()

16 ^ | v • Reply • Share ›



Lionheart Zhang • 5 years ago

Map结构还有一个forEach方法，与数组的forEach方法类似，也可以实现遍历。
这行下面的代码第一行 多了个)

3 ^ | v • Reply • Share ›



ruanyf Mod → Lionheart Zhang • 5 years ago

谢谢指出，改成方括号了。

^ | v • Reply • Share ›



lee telen → ruanyf • 3 years ago

没改啊？

^ | v • Reply • Share ›



Yann • 5 years ago

搞出这么多类型的类来，开始往面向对象模式上靠了，不喜欢，js是面向函数的语言,搞出这么胶水层，看上去是方便了，可本质的东西确模糊了

2 ^ | v • Reply • Share ›



helm sun → Yann • 4 years ago

我不赞成你这种说法，语言只是工具，它的本质应该是提高开发效率，减少开发错误。

45 ^ | v • Reply • Share ›



第十一维 • a year ago

“注意，WeakMap 弱引用的只是键名，而不是键值。键值依然是正常引用。”
请问如果键值是强引用，在键名被回收之后，键值不能造成内存泄漏吗

1 ^ | v • Reply • Share ›



王建新 → 第十一维 • 8 months ago

在键名被回收后，其实，WeakMan也就不持有这个键值的引用了（等同于从WeakMap中delete这个键），至于什么时候回收，那就^{上一章}回收^{下一章}的事情了



hanzhijue • 2 years ago

Set的遍历操作，小标题（1）`keys()`，`values()`，`entries()` 的文本：

`key`方法、`value`方法、`entries`方法返回的都是遍历器对象（详见《`Iterator`对象》一章）。由于`Set`结构没有键名，只有键值（或者说键名和键值是同一个值），所以`key`方法和`value`方法的行为完全一致。

貌似文本内容中应该是 `keys`方法，`values`方法

1 ^ | v • Reply • Share ›



ruanyf Mod ➔ hanzhijue • 2 years ago

谢谢指出，已经改正。

^ | v • Reply • Share ›



Stuart Zhang • 3 years ago

用`WeakMap`实现私有属性，好有创造性！

1 ^ | v • Reply • Share ›



Gabe Yuan • 4 months ago

一种更简洁的对象转`map`写法：

```
function objToStrMap(obj) {  
  return new Map(Object.entries(obj));  
}
```

^ | v • Reply • Share ›



Parker Zhu • a year ago

“而且，数组的`map`和`filter`方法也可以用于 `Set` 了。”

这里的例子写错了，实际上例子里还是写的在数组上的`map`和`filter`调用，并且现在`set`也还不支持`map`和`filter`

^ | v • Reply • Share ›



ruanyf Mod ➔ Parker Zhu • a year ago

谢谢指出，我加上“间接用于”，可能更准确一些。

1 ^ | v • Reply • Share ›



ASCII • a year ago

```
const items = [  
  ['name', '张三'],  
  ['title', 'Author']  
];
```

```
const map = new Map();
```

```
items.forEach(  
  ([key, value]) => map.set(key, value)  
);
```

//以上输出

//张三 name

//Author title

//forEach格式

```
//forEach格式
//forEach(value,index,array)
//应改为
//([value,key]) => map.set(key, value)
```

^ | v • Reply • Share ›



StevenLeung • a year ago

其实我这里想问一下，垃圾机制有两种方式，一种是标记清除，另一种是您所说的引用计数，因为引用计数存在循环引用的问题，所以很多浏览器已经不使用了，那么想问一下您，文中关于WeakSet和WeakMap的"不参与垃圾回收机制"，为什么使用的是“引用计数”而不是“标记清除”呢？

^ | v • Reply • Share ›



ruanyf Mod ➔ StevenLeung • a year ago

这里的“引用计数”，就是泛指引擎内部的无用内存的判断机制。因为它比较容易解释和理解，就用它来做例子。

^ | v • Reply • Share ›



StevenLeung ➔ ruanyf • a year ago

额。。可是现在都不用引用计数了，，而且，引用计数跟标记清除，是不一样的两个机制，应该不能“泛指”。

^ | v • Reply • Share ›



ruanyf Mod ➔ StevenLeung • a year ago

确实，只说引用计数不够完整，这里应该补充一下循环计数的解决方案。

^ | v • Reply • Share ›



我就看吓不说话 • a year ago

「...则只要两个值严格相等，Map 将其视为一个键，包括0和-0，布尔值true和字符串true则是两个不同的键...」0 和 -0 是相同的键吧，这句话很容易理解成是不同的键

^ | v • Reply • Share ›



ruanyf Mod ➔ 我就看吓不说话 • a year ago

谢谢指出，已经修改了文字。

^ | v • Reply • Share ›



梁川 • a year ago

“

```
const listener = new WeakMap();
```

```
listener.set(element1, handler1);
```

```
listener.set(element2, handler2);
```

```
element1.addEventListener('click', listener.get(element1), false);
```

```
element2.addEventListener('click', listener.get(element2), false);
```

上面代码中，监听函数放在 WeakMap 里面。一旦 DOM 对象消失，跟它绑定的监听函数也会自动消失。

”

这个地方有点想当然了。

```
.addEventListener('click', listener.get(element1), false);
```

这一行应该还是会把 handler 的强引用，[上一章](#) [5.2](#) [下一章](#) 添加到 eventListener 列表里。

如果这个 `eventListener` 列表会跟着 `Dom` 消失，那么外围多一个 `Map` 是画蛇添足。反之，则这个 `Map` 并不会促进这个 `handler` 被回收。

^ | v • Reply • Share ›



ruanyf Mod ➔ 梁川 • a year ago

说得有道理，我把这个例子删除了。

^ | v • Reply • Share ›



Zoe Zhang • a year ago

图中红框的地方是不是写错了？我不是很确定，如果没错请忽视这条评论



^ | v • Reply • Share ›



ruanyf Mod ➔ Zoe Zhang • a year ago

谢谢指出，确实是 `WeakSet`，我改一下。

^ | v • Reply • Share ›



huahua • 2 years ago

事实上，不仅仅是数组，任何具有 `Iterator` 接口的数据结构（详见《`Iterator`》一章）都可以当作 `Map` 构造函数的参数。

求解惑，若是我要传一串字符串呢？

```
let map = new Map('qqq')
```

```
> TypeError: iterable for Map should have array-like objects
```

^ | v • Reply • Share ›



ruanyf Mod ➔ huahua • 2 years ago

这个是我没写清楚。

`Map` 的参数必须是具有 `Iterator` 接口，每个成员都是一个双元素数组（比如 `[['foo', 1], ['bar', 2]]`）的数据结构。

谢谢指出，已经改过来了。

^ | v • Reply • Share ›



longboy • 2 years ago

文中的“`Set` 函数可以接受一个数组（或类似数组的对象）作为参数，用来初始化。”然后我就试了一下：

```
var obj = {0:'a',1:'b',2:'c',length:3};
```

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

```
new Set(obj)//undefined is not a function
```

```
new Set(arr)//{"a", "b", "c"}
```

这样会报错，按照 `javascript` 的官方文档，`obj` 应该是属于类数组了，为什么还会报错呢？

^ | v • Reply • Share ›



ruanyf Mod ➔ longboy • 2 years ago

[上一章](#)

[下一章](#)

这是我写错了，必须是具有 `Iterator` 接口的类数组才可以，比如 `Set` 的 `constructor` 参数，已经改过来了。

这是我与错了，必须具有 `Iterator` 接口的类数组才可以，比如函数的 `arguments` 对象。已经改过来了。

^ | v • Reply • Share ›



常修 • 2 years ago

`map`和`localStorage`很像。或者`stor2.js`

^ | v • Reply • Share ›



常修 • 2 years ago

还没看完，我感觉跟`python`越来越靠近了。

^ | v • Reply • Share ›



Martin Wang ➔ 常修 • 2 years ago

是的，我也有这种感觉。`ES6`的`js`越来越像`python`了。

^ | v • Reply • Share ›



LisiurDay ➔ Martin Wang • 2 years ago

只是增加了两种数据结构而已，怎么能说和`python`越来越像了。其他语言比如`C++`，`ruby`，`java`等都有包或者默认支持这些数据结构。只能说`js`越来越完善了。。

1 ^ | v • Reply • Share ›



陆德靖 • 2 years ago

"

^ | v • Reply • Share ›



Nicole Wu • 2 years ago

```
var s = new Set();
```

```
[2, 3, 5, 4, 5, 2, 2].map(x => s.add(x));
```

```
for (let i of s) {  
  console.log(i);  
}
```

```
// 2 3 5 4
```

这里用`map`不太合适吧，应该用`forEach`

^ | v • Reply • Share ›



ruanyf Mod ➔ Nicole Wu • 2 years ago

谢谢指出，已经改正。

^ | v • Reply • Share ›



庄浩铭 • 2 years ago



我觉得例子三并没有展示出接受类数组对象做参数，它还是接受了数组对象做参数了吧。

^ | v • Reply • Share ›



ruanyf Mod ➔ 庄浩铭 • 2 years ago

`document.querySelectorAll <https: developer.mozilla.org="" en-us="" docs="" web="" api="" document="" queryselectorall="">` 方法返回一个`NodeList`对象，这个对象不是数组。但是，Chrome会返回数组。

<https://developer.mozilla.o...>

^ | v • Reply • Share ›



田淮仁 • 2 years ago

阮老师写文章确实是业界良心，这篇介绍`map`,`set`的article确实很赞。但,里面有个点,关于`set`来实现交并差的.确实有点简单,能否提高点难度,让其可用性多一点呢?鄙人写了一个方法,不知道行不。。

...

```
function union(){
let params = Array.from(arguments);
return params.reduce((pre,cur)=>{
return new Set([...pre,...cur]);
})
}
// 测试
console.log(union([123,123],[456,456],[345,345]));

function intersect(){
let params = Array.from(arguments);
return params.reduce((pre,cur)=>{
return new Set([...cur].filter(val=>'has' in pre?pre.has(val):pre.includes(val)));
})
}
```

see more

^ | v • Reply • Share ›



ruanyf Mod ➔ 田淮仁 • 2 years ago

不错！

^ | v • Reply • Share ›



heartwalker • 3 years ago

Map的forEach处代码:

上一章

下一章

```
map.forEach(function(value, key, map)) {  
  console.log("Key: %s, Value: %s", key, value);  
};
```

小括号位置有问题,应为

```
map.forEach(function(value, key, map){  
  console.log("Key: %s, Value: %s", key, value);  
});
```

对象转化为Map处结果代码:

```
objToStrMap({yes: true, no: false})
```

```
// [ [ 'yes', true ], [ 'no', false ] ]
```

应该是Map {"yes" => true, "no" => false}

^ | v • Reply • Share ›



ruanyf Mod ➔ heartwalker • 3 years ago

谢谢指出, 已经改正。

^ | v • Reply • Share ›



heartwalker • 3 years ago

Map结构的基本目的和用法开头代码

```
var data = {};  
var element = document.getElementById("myDiv");  
data[element] = metadata;  
data["Object HTMLDivElement"] // metadata
```

: metadata加上引号比较合适, [Object HTMLDivElement]中的O应该为小写
[object HTMLDivElement]

^ | v • Reply • Share ›



ruanyf Mod ➔ heartwalker • 3 years ago

谢谢指出, 已经改正。

^ | v • Reply • Share ›



heartwalker • 3 years ago

Set实现并集, 交集和差集,如

```
let union = new Set([...a, ...b]);  
// [1, 2, 3, 4]  
union 返回的应该是Set {1,2,3,4}
```

^ | v • Reply • Share ›



ruanyf Mod ➔ heartwalker • 3 years ago

谢谢指出, 已经改正。

^ | v • Reply • Share ›



Mansum Sun • 3 years ago

介绍Map的正文的第一句说Object只能拿字符串作key, 肯定不是啊, 也能用Symbol作为key

^ | v • Reply • Share ›



ruanyf Mod ➔ Mansum Sun • 3 year

上一章

下一章

Object 只能拿字符串作key 是ES6新增的, 数值不能作key, Symbol 只能作key 不能作value

Symbol作为键名是ES6新增的。数值不能作为键名，JavaScript默认将其转为字符串。

^ | v • Reply • Share ›

Load more comments

ALSO ON ECMASCRIPT 6 入门

函数的扩展

188 comments • 5 years ago

Doujun Zhang —

Module

98 comments • 4 years ago

Don — @ruanyfruanyf

Generator 函数：异步操作

30 comments • 2 years ago



ruanyf — 理解正确。Promise 和 Thunk 本质都是对回调函数的包装，只是 API 更友好了。2018-05-19 0:31

...

编程风格

51 comments • 4 years ago

Tom Bian — mark

 Subscribe  Add Disqus to your siteAdd DisqusAdd  Disqus' Privacy PolicyPrivacy PolicyPrivacy Policy