Set 和 Map 数据结构

1. Set

1.1. 基本用法

`Set` 本身是一个构造函数，用来生成 Set 数据结构。它类似于数组，但是成员的值都是唯一的，没有重复的值。

```
const s = new Set([2, 3, 5, 4, 5, 2, 2]);
s; // Set(4) {2, 3, 5, 4}
```

Set 原始参数是 `[2, 3, 5, 4, 5, 2, 2]`，最终 s 只有 `2 3 5 4` 四个没有重复的值，后面的 `5 2 2` 因为与前面的成员重复，没有被加入到 s。证明 **Set 数据结构成员的值都是唯一的，没有重复。**

`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，如字符串、NodeList）作为参数，用来初始化。

```
// 例一，数组作为 Set 的参数
const set1 = new Set([1, 2, 3, 3]);
set1; // Set(3) {1, 2, 3}

// 例二，字符串作为 Set 的参数
const set2 = new Set("abcbcdcde");
set2; // Set(5) {'a', 'b', 'c', 'd', 'e'}

// 例三
const set3 = new Set(document.querySelectorAll('div'));
set3.size // 31，本页面 div 标签的个数
```

根据 Set 数据结构的成员是唯一的，没有重复这一特点，可以用于数组去除重复成员：

```
[...new Set(array)];
// 或者
Array.from(new Set(array));
```

也可以用于去除字符串里面的重复字符：

```
// Set 接受字符串作为参数，将其转换为数组，再转换为字符串，达到“去除字符串重复字符”的目的
[...new Set('abcbcdcde')].join(''); // 'abcde'
```

在 Set 内部，两个 `NaN` 是相等的。

```
let set = new Set();
set.add(NaN);
set.add(NaN);
set; // Set {NaN}
// 添加了两个 NaN，但是只保留了一个，由于 Set 数据结构中值是唯一的，证明 Set 内部将两个 NaN 认为是重复的
```

在 Set 内部，两个对象总是不相等的。

```
let set = new Set();
set.add({});
set.add({});
set.size; // 2
// 添加了两个空对象，成员是两个，证明将两个空对象被视为两个值。
```

1.2. Set 实例的属性

1.2.1. Set.prototype.constructor

`Set.prototype.constructor` 构造函数，默认就是 Set 函数。

```
Set.prototype.constructor === Set; // true
```

1.2.2. Set.prototype.size

`Set.prototype.size` 返回 Set 实例的成员总数。

```
let s = new Set([1, 2, 3]);
s.size; // 3
```

1.3. Set 的操作方法

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

```
let s = new Set();
s.add(1).add(2).add(2); // add 方法返回 Set 结构本身，所以可以链式操作
s.size // 2

s.has(1) // true
```

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

s.delete(2) // true
s.has(2) // false

s.delete(2) // false，删除失败，因为前面已经删除过该值了，s 中已经不存在该成员了

s.clear();
s.size; // 0
```

1.4. Set 的遍历方法

Set的遍历顺序就是插入顺序，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。

1.4.1. keys()，values()，entries()

`keys()`、`values()`、`entries()` 返回的都是遍历器对象。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']);

// keys() 和 values() 都返回 SetIterator {'red', 'green', 'blue'}
for (let item of set.keys()) { console.log(item); }
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"]
```

上例中，`entries()` 返回的遍历器，同时包括键名和键值，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
Set[Symbol.iterator] === Set.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
```

1.4.2. 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
```

上例 `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）。**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。**

1.4.3. 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']);
[...set]; // ['red', 'green', 'blue']
```

数组的 `map` 和 `filter` 方法也可以间接用于 Set 了。

```
let s1 = new Set([1, 2, 3]);
s1 = new Set([...s1].map(x => x * 2));
// s1: Set(3) {2, 4, 6}

let s2 = new Set([1, 2, 3, 4, 5]);
s2 = new Set([...s2].filter(x => (x % 2) == 0));
// s2: Set(2) {2, 4}
```

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

```
// 先构造需要求并集/交集/差集的 Set 数据结构 a 和 b
let a = new Set([1, 2, 3]);
let b = new Set([4, 3, 2]);

// 并集。将 a 和 b 转换成数组并合并起来
let union = new Set([...a, ...b]);
// union: Set {1, 2, 3, 4}
```

```
// 交集。将 a 转换成数组，并求 b 中“包含 a 的成员”的成员
let intersect1 = new Set([...a].filter(x => b.has(x)));
// intersect1: set {2, 3}
// 交集。将 b 转换成数组，并求 a 中“包含 b 的成员”的成员
let intersect2 = new Set([...b].filter(x => a.has(x)));
// intersect2: set {3, 2}

//  (a 相对于 b 的) 差集。将 a 转换成数组，并求 b 中“不包含 a 的成员”的成员
let difference1 = new Set([...a].filter(x => !b.has(x)));
// difference1: Set {1}
//  (b 相对于 a 的) 差集。将 b 转换成数组，并求 a 中“不包含 b 的成员”的成员
let difference2 = new Set([...b].filter(x => !a.has(x)));
// difference2: Set {4}
```

2. Map

2.1. 含义和基本用法

JavaScript 的对象（Object）只能用字符串当作键。Map 数据结构类似于对象，但是各种类型的值（包括对象）都可以当作键，是一种更完善的 Hash 结构实现。

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

m.set(o, 'content'); // 给 m 加成员，该成员键是对象
m.get(o); // "content" ，读取成员

m.has(o); // true，判断是否有该成员
m.delete(o); // true，删除成员
m.has(o); // false
```

Map 作为构造函数，参数是一个二维数组，该数组每一个成员数组（又包含两个成员）是表示键值对的数组。

```
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"
```

`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
```

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'); // 'bbb' 将覆盖前面的
map.get(1) // "bbb"
```

如果读取一个未知的键，则返回 `undefined`。

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

只有对同一个对象的引用，Map 结构才将其视为同一个键。

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

上例的 set 和 get 方法，表面是针对同一个键，但实际上这是两个不同的数组实例，内存地址是不一样的，因此 get 方法无法读取该键，返回 undefined。

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
```

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：

```
let map = new Map();

// 0 和 -0 是一个键
map.set(-0, 123);
map.get(+0); // 123

// 布尔值 true 和字符串 'true' 是两个不同的键
map.set(true, 1);
map.set('true', 2);
map.get(true); // 1

// undefined 和 null 是两个不同的键
map.set(undefined, 3);
map.set(null, 4);
map.get(undefined); // 3

// NaN 是一个键
map.set(NaN, 123);
map.get(NaN); // 123
```

2.2. Map 实例的属性

2.2.1. Map.prototype.constructor

`Map.prototype.constructor` 构造函数，默认就是 Map 函数。

```
Map.prototype.constructor === Map; // true
```

2.2.2. Map.prototype.size

`Map.prototype.size` 返回 Map 实例的成员总数。

```
let m = new Map(["foo", true], ["bar", false]);
m.size; // 2
```

2.3. Map 的操作方法

2.3.1. Map.prototype.set()

`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();
// set() 方法返回的是当前的 Map 对象，因此可以采用链式写法
m.set('edition', 6).set(262, 'standard').set(undefined, 'nah');
```

2.3.2. Map.prototype.get()

`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!

m.set(hello, 'Hello ES2015!'); // 更新 'hello' 键
m.get(hello);  // Hello ES2015!

m.get('world'); // undefined
```

2.3.3. Map.prototype.has()

`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).set(262, 'standard').set(undefined, 'nah');

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

2.3.4. Map.prototype.delete()

`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'); // 新增一个 undefined 键
m.has(undefined);        // true

m.delete(undefined);     // true，删除成功
m.has(undefined);        // false

m.delete(undefined);     // false，删除失败，因为上面已经删除过了，该键不存在了
```

2.3.5. Map.prototype.clear()

clear()方法清除所有成员，没有返回值。

```
let map = new Map();
map.set('foo', true).set('bar', false);
map.size // 2

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

2.4. Map 的遍历方法

Map 的遍历顺序就是插入顺序。

2.4.1. keys()，values()，entries()

```
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"
```

Map 结构的默认遍历器接口（Symbol.iterator 属性），就是 `entries()` 方法。

```
Map.prototype[Symbol.iterator] === Map.prototype.entries; // true
Map[Symbol.iterator] === Map.entries; // true
```

这意味着，**可以省略 `entries()` 方法，直接用 `for...of` 循环遍历 Map。**

```
const map = new Map([['F', 'no'], ['T',  'yes']]);
for (let [key, value] of map) {
  console.log(key, value);
}
// "F" "no"
// "T" "yes"
```

2.4.2. forEach()

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

```
const map = new Map([[1, 'one'], [2, 'two'], [3, 'three']]);
const reporter = {
  report: function(key, value) {
    console.log("Key: %s, Value: %s", key, value);
  }
};
map.forEach(function(value, key, map) {
  this.report(key, value); // this 指向 reporter
}, reporter); // forEach 第二个参数绑定 this
// Key: 1, Value: one
// Key: 2, Value: two
// Key: 3, Value: three
```

2.4.3. 遍历的应用

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`、`filter` 方法，可以实现 Map 的遍历和过滤。

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

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

2.5. 与其他数据结构的相互转换

2.5.1. Map 转为数组

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']]]
```

2.5.2. 数组 转为 Map

将数组传入 Map 构造函数，就可以转为 Map。

```
let array = [[true, 7], [{foo: 3}, ['abc']]];
let m = new Map(array);
// Map(2) {true => 7, Object {foo: 3} => ['abc']}
```

2.5.3. 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。

2.5.4. 对象转为 Map

对象转为 Map 可以通过 `Object.entries()`。

```
let obj = {"a":1, "b":2};
let map = new Map(Object.entries(obj));
// map: Map(2) {'a' => 1, 'b' => 2}
```

2.5.5. 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"]]]'
```

2.5.6. 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']}
```