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。

```
x => x * x;
```

相当于：

```
function (x) {
    return x * x;
}
```

```
const set = new Set();

[1,2,3,45].forEach(x =>set.add(x));
for(let i of set){
  console.log(i);
}
```

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。

在 Set 内部，两个`NaN`是相等的。

另外，两个对象总是不相等的。

一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。

`Array.from`方法可以将Set结构转为数组。

```
var items = new Set([1, 2, 3, 4, 5]);
var 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]
```

二set遍历

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。

```
var ite = new Set([1,3,3,45,5]);
for(let item of ite.entries()){
  console.log(item);
}
```

使用entries遍历的时候取到的是键值对数组，set的键值是一样的

```
(2) [1, 1]
(2) [3, 3]
(2) [45, 45]
(2) [5, 5]
```

Set结构的实例的`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, 2, 3]);
set.forEach((value, key) => console.log(value * 2) )
// 2
// 4
// 6
```

上面代码说明，forEach方法的参数就是一个处理函数。该函数的参数依次为键值、键名、集合本身（上例省略了该参数）。另外，forEach方法还可以有第二个参数，表示绑定的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];   和 Array.from(new 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]
```

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。

三：weakSet

只能放置对象的 set 集合

WeakSet 适合临时存放一组对象，以及存放跟对象绑定的信息。

ES6 规定 WeakSet 不可遍历。

四：Map

1：向map中添加成员

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

m.set(o, 'content')
```

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"
```

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。

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。

map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。

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

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

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

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）。

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']}
```