



# 第32课：对象的数据结构 拓展和Set、Map数据结构

主讲老师：万章



目录

对象的数据结构拓展

Set数据结构



# 对象的数据结构拓展

## 属性的简洁表示法

ES6 允许直接写入变量和函数，作为对象的属性和方法。这样的书写更加简洁。

```
const foo = 'bar';  
const baz = {foo};  
baz // {foo: "bar"}  
  
// 等同于  
const baz = {foo: foo};
```

右侧代码表明，ES6 允许在对象之中，直接写变量。这时，属性名为变量名，属性值为变量的值。右侧是另一个例子。

```
function f(x, y) {  
  return {x, y};  
}  
  
// 等同于  
  
function f(x, y) {  
  return {x: x, y: y};  
}  
  
f(1, 2) // Object {x: 1, y: 2}
```

## 属性的简洁表示法

除了属性简写，方法也可以简写。

```
const o = {  
  method() {  
    return "Hello!";  
  }  
};  
  
// 等同于  
  
const o = {  
  method: function() {  
    return "Hello!";  
  }  
};
```

```
let birth = '2000/01/01';  
  
const Person = {  
  
  name: '张三',  
  
  //等同于birth: birth  
  birth,  
  
  // 等同于hello: function ()...  
  hello() { console.log('我的名字是', this.name); }  
  
};
```

这种写法用于函数的返回值，将会非常方便

```
function getPoint() {  
  const x = 1;  
  const y = 10;  
  return {x, y};  
}  
  
getPoint()  
// {x:1, y:10}
```

## 属性的简洁表示法

注意，**简洁写法的属性名总是字符串**，这会导致一些看上去比较奇怪的结果。

```
const obj = {  
  class () {}  
};  
  
// 等同于  
  
var obj = {  
  'class': function() {}  
};
```

```
> let obj={  
  in(){}  
}  
< undefined  
> obj  
< ▼ {in: f} ⓘ  
  ▶ in: f in()  
  ▶ __proto__: Object  
> |
```

上面代码中，class和in是字符串，所以不会因为它属于关键字，而导致语法解析报错

## 属性名表达式

```
// 方法一
obj.foo = true;

// 方法二
obj['a' + 'bc'] = 123;

var obj = {
  foo: true,
  abc: 123
};
```

JavaScript 定义对象的属性，有两种方法。

方法一是直接用标识符作为属性名

方法二是用表达式作为属性名，这时要将表达式放在方括号之内。

但是，如果使用字面量方式定义对象（使用大括号），在 ES5 中只能使用方法一（标识符）定义属性。

ES6 允许字面量定义对象时，用方法二（表达式）作为对象的属性名，即把表达式放在方括号内。

```
let propKey = 'foo';

let obj = {
  [propKey]: true,
  ['a' + 'bc']: 123
};
```

## 对象的拓展

```
let lastWord = 'last word';

const a = {
  'first word': 'hello',
  [lastWord]: 'world'
};

a['first word'] // "hello"
a[lastWord] // "world"
a['last word'] // "world"
```

可在对象字面量的定义模式里面使用某个变量作为某个属性的名称

```
let obj = {
  ['h' + 'ello']() {
    return 'hi';
  }
};

obj.hello() // hi
```

表达式还可以用于定义方法名。

```
// 报错
const foo = 'bar';
const bar = 'abc';
const baz = { [foo] };

// 正确
const foo = 'bar';
const baz = { [foo]: 'abc' };
```

注意，属性名表达式与简洁表示法，不能同时使用，会报错

注意，属性名表达式如果是一个对象，默认情况下会自动将对象转为字符串[object Object]，这一点要特别小心。

```
const keyA = {a: 1};
const keyB = {b: 2};

const myObject = {
  [keyA]: 'valueA',
  [keyB]: 'valueB'
};

myObject // Object {[object Object]: "valueB"}
```



## 对象属性的可枚举

对象的每个属性都有一个描述对象 (Descriptor) , 用来控制该属性的行为。

`Object.getOwnPropertyDescriptor`方法可以获取该属性的描述对象。

```
> let obj = {
    foo: 123
};
Object.getOwnPropertyDescriptor(obj, 'foo')
{value: 123, writable: true, enumerable: true, configurable: true}
  configurable: true
  enumerable: true
  value: 123
  writable: true
  __proto__: Object
```

描述对象的`enumerable`属性, 称为“可枚举性”, 如果该属性为`false`, 就表示某些操作会忽略当前属性。

目前, 有四个操作会忽略`enumerable`为`false`的属性。

1. `for...in`循环: 只遍历对象自身的和继承的可枚举的属性。
2. `Object.keys()`: 返回对象自身的所有可枚举的属性的键名。
3. `JSON.stringify()`: 只 串行化 对象自身的可枚举的属性。
4. `Object.assign()`: 忽略`enumerable`为`false`的属性, 只拷贝对象自身的可枚举的属性。

只有`for...in`会返回继承的属性, 其他三个方法都会忽略继承的属性, 只处理对象自身的属性

# 对象属性的遍历

ES6 一共有 5 种方法可以遍历对象的属性。

## (1) for...in

for...in 循环遍历对象自身的和继承的可枚举属性（不含 Symbol 属性）

## (2) Object.keys(obj)

Object.keys 返回一个数组，包括对象自身的（不含继承的）所有可枚举属性（不含 Symbol 属性）的键名。

## (3) Object.getOwnPropertyNames(obj)

Object.getOwnPropertyNames 返回一个数组，包含对象自身的属性（不含 Symbol 属性，但是包括不可枚举属性）的键名。

## (4) Object.getOwnPropertySymbols(obj)

Object.getOwnPropertySymbols 返回一个数组，包含对象自身的属性 Symbol 属性的键名。

## (5) Reflect.ownKeys(obj)

Reflect.ownKeys 返回一个数组，包含对象自身的所有键名，不管键名是 Symbol 或字符串，也不管是否可枚举。

```
> let obj = {
    foo: 123
}
<> undefined
> Object.keys(obj)
<> ▼ ["foo"] ⓘ
    0: "foo"
    length: 1
    ▶ __proto__: Array(0)
> Object.getOwnPropertyNames(obj)
<> ▼ ["foo"] ⓘ
    0: "foo"
    length: 1
    ▶ __proto__: Array(0)
> Reflect.ownKeys(obj)
<> ▶ ["foo"]
> Object.getOwnPropertySymbols(obj)
<> ▼ [] ⓘ
    length: 0
    ▶ __proto__: Array(0)
>
```

## 对象的原型对象属性

函数里的this关键字总是指向调用该函数所在的当前对象，ES6 又新增了另一个类似的关键字super，指向当前对象的原型对象。

```
const proto = {
  foo: 'hello'
};

const obj = {
  foo: 'world',
  find() {
    return super.foo;
  }
};

Object.setPrototypeOf(obj, proto);
obj.find() // "hello"
```

对象obj.find()方法之中，通过super.foo引用了原型对象proto的foo属性。

**注意，super关键字表示原型对象时，只能用在对象的方法之中，用在其他地方都会报错。**

## 对象的原型对象属性

```
// 报错
const obj = {
  foo: super.foo
}

// 报错
const obj = {
  foo: () => super.foo
}

// 报错
const obj = {
  foo: function () {
    return super.foo
  }
}
```

左边三种super的用法都会报错，因为对于 JavaScript 引擎来说，这里的super都没有用在对象的方法之中。

第一种写法是super用在属性里面

第二种和第三种写法是super用在一个函数里面，然后赋值给foo属性。


目前，只有**对象方法的简写法**可以让 JavaScript 引擎确认，定义的是对象的方法，所以这玩意子的使用环境苛刻，现在可以少用一些

JavaScript 引擎内部，`super.foo`等同于`Object.getPrototypeOf(this).foo`（属性）

## 对象的扩展运算符赋值

《数组的扩展》一章中，已经介绍过扩展运算符（...）。ES2018 将这个运算符引入了对象。所以这大体的用法和数组的没啥区别

对象的解构赋值用于从一个对象取值，相当于将目标对象自身的**所有可遍历的**（enumerable）、但尚未被读取的属性，分配到指定的对象上面。所有的键和它们的值，都会拷贝到新对象上面。



```
let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };
x // 1
y // 2
z // { a: 3, b: 4 }
```

解构赋值要求等号右边是一个对象，所以如果等号右边是undefined或null，就会报错，因为它们无法转为对象

```
let { ...z } = null; // 运行时错误
let { ...z } = undefined; // 运行时错误
```

```
let o1 = { a: 1 };
let o2 = { b: 2 };
o2.__proto__ = o1;
let { ...o3 } = o2;
o3 // { b: 2 }
o3.a // undefined
```

扩展运算符的解构赋值，不能复制继承自原型对象的属性

## 对象的扩展运算符复制

```
let z = { a: 3, b: 4 };  
let n = { ...z };  
n // { a: 3, b: 4 }
```

对象的扩展运算符 (...) 在赋值符号右侧是，用于取出参数对象的所有可遍历属性，拷贝到左侧的当前对象之中。

如果扩展运算符后面不是对象，则会自动将其转为对象。



我举个栗子

扩展运算符后面是整数1，会自动转为数值的包装对象Number{1}。由于该对象没有自身属性，所以返回一个空对象。  
其他的一个道理

```
> {...1 }
```

```
< ▶ {}
```

```
> {...true }
```

```
< ▶ {}
```

```
> {...undefined }
```

```
< ▶ {}
```

```
> {...null }
```

```
< ▶ {}
```



# Set数据结构

## Set数据结构

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
```

可以使用add()方法向 Set 结构加入成员

for of循环可以直接获得Set结构的内容本身



Set 结构不会添加重复的值。



## Set数据结构

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，比如NodeList）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
```

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

左侧代码也展示了一种去除数组重复成员的方法。

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

左侧的方法也可以用于，去除字符串里面的重复字符。

## Set数据结构

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}
```

```
> let set = new Set();
```

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

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

```
< 2
```

```
> set
```

```
< ▼ Set(2) {{...}, {...}} ⓘ
  size: (...)
  __proto__: Set
  [[Entries]]: Array(2)
    0: Object
    1: Object
  length: 2
```

```
>
```

```
> set.add({x:1})
```

```
< ▶ Set(3) {{...}, {...}, {...}}
```

```
> set.add({x:1})
```

```
< ▼ Set(4) {{...}, {...}, {...}, {...}} ⓘ
  size: (...)
  __proto__: Set
  [[Entries]]: Array(4)
    0: Object
    1: Object
    2:
      value: {x: 1}
    3:
      value: {x: 1}
  length: 4
```

```
>
```

Set数据结构里面的NaN只会有一个

两个对象总是不相等的。所以直接用对象字面量添加到Set结构里的数据一定是不会重复的

## Set实例的属性和方法

```
> let set = new Set();
< undefined

> set.add({x:1})
< ▶ Set(1) {{...}}

> set.__proto__
< ▼ Set {constructor: f, has: f, add: f, delete: f, clear: f, ...}
  i
  ▶ add: f add()
  ▶ clear: f clear()
  ▶ constructor: f Set()
  ▶ delete: f delete()
  ▶ entries: f entries()
  ▶ forEach: f forEach()
  ▶ has: f has()
  ▶ keys: f values()
  ▶ size: (...)
  ▶ values: f values()
  ▶ Symbol(Symbol.iterator): f values()
  Symbol(Symbol.toStringTag): "Set"
  ▶ get size: f size()
  ▶ __proto__: Object

>
```

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实例的属性和方法

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

```
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
```

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

## Set数据结构的遍历操作

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

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。

注意：如果使用`set.values`的接口进行遍历的时候，可以省去`.values`，直接`for.....of set`就阔以

```
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"]
```

## Set数据结构的遍历操作

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

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。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。

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。

## Set遍历操作的常见应用

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

去除数组的重复成员。

```
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}
```

间接调用数组方法

```
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}
```

实现并集 (Union)、交集 (Intersect) 和差集 (Difference)。



## Set遍历操作的常见应用

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；
- 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
```



## Set结构的萎缩版WeakSet

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
```

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。

## Set结构的萎缩版WeakSet

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
```

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。

## Set结构的萎缩版WeakSet

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

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

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。

WeakSet 结构有以下三个方法(与Set数据类型的方法类似)

- `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。

```
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
```

## Set结构的萎缩版WeakSet

```
ws.size // undefined
ws.forEach // undefined

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

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。