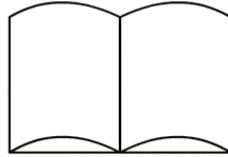


# ES6 系列之模拟实现一个 Set 数据结构

你的导师 架构师日记 2019-07-18



阅读本文约需要5分钟

大家好，我是你们的导师，我每天都会给大家分享一下干货内容（当然了，周末也要允许老师休息一下哈）。上次给大家分享了ES6的迭代器与 for of，今天给大家分享一下模拟实现一个 Set 数据结构。

## 初始化

Set 本身是一个构造函数，用来生成 Set 数据结构。

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

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。

```
let set = new Set([1, 2, 3, 4, 4]);  
console.log(set); // Set(4) {1, 2, 3, 4}  
  
set = new Set(document.querySelectorAll('div'));  
console.log(set.size); // 66  
  
set = new Set(new Set([1, 2, 3, 4]));  
console.log(set.size); // 4
```

## 属性和方法

操作方法有：

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

举个例子:

```
let set = new Set();
console.log(set.add(1).add(2)); // Set [ 1, 2 ]

console.log(set.delete(2)); // true
console.log(set.has(2)); // false

console.log(set.clear()); // undefined
console.log(set.has(1)); // false
```

之所以每个操作都 `console` 一下, 就是为了让大家注意每个操作的返回值。遍历方法有:

1. `keys()`: 返回键名的遍历器
2. `values()`: 返回键值的遍历器
3. `entries()`: 返回键值对的遍历器
4. `forEach()`: 使用回调函数遍历每个成员, 无返回值

注意 `keys()`、`values()`、`entries()` 返回的是遍历器。

```
let set = new Set(['a', 'b', 'c']);
console.log(set.keys()); // SetIterator {"a", "b", "c"}
console.log([...set.keys()]); // ["a", "b", "c"]
```

```
let set = new Set(['a', 'b', 'c']);
console.log(set.values()); // SetIterator {"a", "b", "c"}
console.log([...set.values()]); // ["a", "b", "c"]
```

```
let set = new Set(['a', 'b', 'c']);
console.log(set.entries()); // SetIterator {"a", "b", "c"}
console.log([...set.entries()]); // [{"a", "a"}, {"b", "b"}, {"c", "c"}]
```

```
let set = new Set([1, 2, 3]);
set.forEach((value, key) => console.log(key + ': ' + value));
// 1: 1
// 2: 2
// 3: 3
```

属性：

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

### 模拟实现第一版

如果要模拟实现一个简单的 Set 数据结构，实现 add、delete、has、clear、forEach 方法，还是很容易写出来的，这里直接给出代码：

```
(function(global) {

  function Set(data) {
    this._values = [];
    this.size = 0;

    data && data.forEach(function(item) {
      this.add(item);
    }, this);
  }

  Set.prototype['add'] = function(value) {
    if (this._values.indexOf(value) == -1) {
      this._values.push(value);
      ++this.size;
    }
    return this;
  }

  Set.prototype['has'] = function(value) {
    return (this._values.indexOf(value) !== -1);
  }

  Set.prototype['delete'] = function(value) {
    var idx = this._values.indexOf(value);
    if (idx == -1) return false;
    this._values.splice(idx, 1);
    --this.size;
    return true;
  }

  Set.prototype['clear'] = function(value) {
    this._values = [];
    this.size = 0;
  }

  Set.prototype['forEach'] = function(callbackFn, thisArg) {
    thisArg = thisArg || global;
    for (var i = 0; i < this._values.length; i++) {
      callbackFn.call(thisArg, this._values[i], this._values[i], this);
    }
  }

  Set.length = 0;

  global.Set = Set;

})(this)
```

我们可以写段测试代码：

```
let set = new Set([1, 2, 3, 4, 4]);
console.log(set.size); // 4

set.delete(1);
console.log(set.has(1)); // false

set.clear();
console.log(set.size); // 0

set = new Set([1, 2, 3, 4, 4]);
set.forEach((value, key, set) => {
    console.log(value, key, set.size)
});
// 1 1 4
// 2 2 4
// 3 3 4
// 4 4 4
```

## 模拟实现第二版

在第一版中，我们使用 `indexOf` 来判断添加的元素是否重复，本质上，还是使用 `===` 来进行比较，对于 `NaN` 而言，因为：

```
console.log([NaN].indexOf(NaN)); // -1
```

模拟实现的 Set 其实可以添加多个 `NaN` 而不会去重，然而对于真正的 Set 数据结构：

```
let set = new Set();
set.add(NaN);
set.add(NaN);
console.log(set.size); // 1
```

所以我们需要对 `NaN` 这个值进行单独的处理。处理的方式是当判断添加的值是 `NaN` 时，将其替换为一个独一无二的值，比如说一个很难重复的字符串类似于 `@@NaNValue`，当然了，说到独一无二的值，我们也可以直接使用 `Symbol`，代码如下：

```

(function(global) {

    var NaNSymbol = Symbol('NaN');

    var encodeVal = function(value) {
        return value !== value ? NaNSymbol : value;
    }

    var decodeVal = function(value) {
        return (value === NaNSymbol) ? NaN : value;
    }

    function Set(data) {
        this._values = [];
        this.size = 0;

        data && data.forEach(function(item) {
            this.add(item);
        }, this);
    }

    Set.prototype['add'] = function(value) {
        value = encodeVal(value);
        if (this._values.indexOf(value) == -1) {
            this._values.push(value);
            ++this.size;
        }
        return this;
    }

    Set.prototype['has'] = function(value) {
        return (this._values.indexOf(encodeVal(value)) !== -1);
    }

```

```

    Set.prototype['delete'] = function(value) {
        var idx = this._values.indexOf(encodeVal(value));
        if (idx == -1) return false;
        this._values.splice(idx, 1);
        --this.size;
        return true;
    }

    Set.prototype['clear'] = function(value) {
        ...
    }

    Set.prototype['forEach'] = function(callbackFn, thisArg) {
        ...
    }

    Set.length = 0;

    global.Set = Set;

})(this)

```

写段测试用例：

```
let set = new Set([1, 2, 3]);

set.add(NaN);
console.log(set.size); // 3

set.add(NaN);
console.log(set.size); // 3
```

### 模拟实现第三版

在模拟实现 Set 时，最麻烦的莫过于迭代器的实现和处理，比如初始化以及执行 keys()、values()、entries() 方法时都会返回迭代器：

```
let set = new Set([1, 2, 3]);

console.log([...set]); // [1, 2, 3]
console.log(set.keys()); // SetIterator {1, 2, 3}
console.log([...set.keys()]); // [1, 2, 3]
console.log([...set.values()]); // [1, 2, 3]
console.log([...set.entries()]); // [[1, 1], [2, 2], [3, 3]]
```

而且 Set 也支持初始化的时候传入迭代器：

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

当初始化传入一个迭代器的时候，我们可以根据我们在上一篇中模拟实现的 forOf 函数，遍历传入的迭代器的 Symbol.iterator 接口，然后依次执行 add 方法。

而当执行 keys() 方法时，我们可以返回一个对象，然后为其部署 Symbol.iterator 接口，实现的代码，也是最终的代码如下：

```
(function(global) {

  var NaNSymbol = Symbol('NaN');

  var encodeVal = function(value) {
    return value !== value ? NaNSymbol : value;
  }

  var decodeVal = function(value) {
    return (value === NaNSymbol) ? NaN : value;
  }

  var makeIterator = function(array, iterator) {
    var nextIndex = 0;

    // new Set(new Set()) 会调用这里
    var obj = {
      next: function() {
        return nextIndex < array.length ? { value: iterator(array[nextIndex++]), done: false } : { done: true };
      }
    };

    // [...set.keys()] 会调用这里
    obj[Symbol.iterator] = function() {
      return obj;
    };

    return obj;
  }

})(global);
```



```
function forOf(obj, cb) {
  let iterable, result;

  if (typeof obj[Symbol.iterator] !== "function") throw new TypeError(obj + " is not iterable");
  if (typeof cb !== "function") throw new TypeError('cb must be callable');

  iterable = obj[Symbol.iterator]();

  result = iterable.next();
  while (!result.done) {
    cb(result.value);
    result = iterable.next();
  }
}

function Set(data) {
  this._values = [];
  this.size = 0;

  forOf(data, (item) => {
    this.add(item);
  })
}

Set.prototype['add'] = function(value) {
  value = encodeVal(value);
  if (this._values.indexOf(value) == -1) {
    this._values.push(value);
    ++this.size;
  }
  return this;
}

Set.prototype['has'] = function(value) {
  return (this._values.indexOf(encodeVal(value)) !== -1);
}
```

```

Set.prototype['delete'] = function(value) {
  var idx = this._values.indexOf(encodeVal(value));
  if (idx == -1) return false;
  this._values.splice(idx, 1);
  --this.size;
  return true;
}

Set.prototype['clear'] = function(value) {
  this._values = [];
  this.size = 0;
}

Set.prototype['forEach'] = function(callbackFn, thisArg) {
  thisArg = thisArg || global;
  for (var i = 0; i < this._values.length; i++) {
    callbackFn.call(thisArg, this._values[i], this._values[i], this);
  }
}

Set.prototype['values'] = Set.prototype['keys'] = function() {
  return makeIterator(this._values, function(value) { return decodeVal(value); });
}

Set.prototype['entries'] = function() {
  return makeIterator(this._values, function(value) { return [decodeVal(value), decodeVal(value)]; });
}

Set.prototype[Symbol.iterator] = function(){
  return this.values();
}

```

```

Set.prototype['forEach'] = function(callbackFn, thisArg) {
  thisArg = thisArg || global;
  var iterator = this.entries();

  forOf(iterator, (item) => {
    callbackFn.call(thisArg, item[1], item[0], this);
  })
}

Set.length = 0;

global.Set = Set;

})(this)

```

写段测试代码：

```

let set = new Set(new Set([1, 2, 3]));
console.log(set.size); // 3

console.log([...set.keys()]); // [1, 2, 3]
console.log([...set.values()]); // [1, 2, 3]
console.log([...set.entries()]); // [1, 2, 3]

```

由上我们也可以发现，每当我们进行一版的修改时，只是写了新的测试代码，但是代码改写后，对于之前的测试代码是否还能生效呢？是否不小心改了什么导致以前的测试代码没有通过呢？

为了解决这个问题，针对模拟实现 Set 这样一个简单的场景，我们可以引入 QUnit 用于编写测试用例，我们新建一个 HTML 文件：

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>Set 的模拟实现</title>
  <link rel="stylesheet" href="qunit-2.4.0.css">
</head>

<body>
  <div id="qunit"></div>
  <div id="qunit-fixture"></div>
  <script src="qunit-2.4.0.js"></script>
  <script src="polyfill-set.js"></script>
  <script src="test.js"></script>
</body>

</html>
```

编写测试用例，因为语法比较简单，我们就直接看编写的一些例子：

```
QUnit.test("unique value", function(assert) {
  const set = new Set([1, 2, 3, 4, 4]);
  assert.deepEqual([...set], [1, 2, 3, 4], "Passed!");
});

QUnit.test("unique value", function(assert) {
  const set = new Set(new Set([1, 2, 3, 4, 4]));
  assert.deepEqual([...set], [1, 2, 3, 4], "Passed!");
});

QUnit.test("NaN", function(assert) {
  const items = new Set([NaN, NaN]);
  assert.ok(items.size == 1, "Passed!");
});

QUnit.test("Object", function(assert) {
  const items = new Set([{}, {}]);
  assert.ok(items.size == 2, "Passed!");
});

QUnit.test("set.keys", function(assert) {
  let set = new Set(['red', 'green', 'blue']);
  assert.deepEqual([...set.keys()], ["red", "green", "blue"], "Passed!");
});

QUnit.test("set.forEach", function(assert) {
  let temp = [];
  let set = new Set([1, 2, 3]);
  set.forEach((value, key) => temp.push(value * 2) )

  assert.deepEqual(temp, [2, 4, 6], "Passed!");
});
```

用浏览器预览 HTML 页面，效果如下图：

# Set 的模拟实现

☐ Hide passed tests
 ☐ Check for Globals
 ☐ No try-catch

Module:  ▼

Filter:

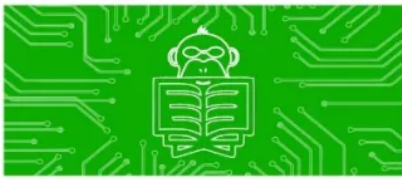
**QUnit 2.4.0; Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_13\_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/66.0.3359.181 Safari/537.36**

17 tests completed in 18 milliseconds, with 0 failed, 0 skipped, and 0 todo.  
17 assertions of 17 passed, 0 failed.

1. <b>unique value (1)</b> <small>Rerun</small>	2 ms
<div> <div></div> <div>1. Passed!</div> <div>@ 2 ms</div> </div>	
<b>Source:</b> at	
file:///Users/kevin/Desktop/articles/JavaScript/ES6%E7%B3%BB%E5%88%97/demos/qunit/test	
2. <b>unique value (1)</b> <small>Rerun</small>	1 ms
3. <b>size (1)</b> <small>Rerun</small>	0 ms
4. <b>NaN (1)</b> <small>Rerun</small>	1 ms
5. <b>Object (1)</b> <small>Rerun</small>	0 ms
6. <b>add function (1)</b> <small>Rerun</small>	3 ms
7. <b>has function (1)</b> <small>Rerun</small>	0 ms
8. <b>delete function (1)</b> <small>Rerun</small>	1 ms
9. <b>clear function (1)</b> <small>Rerun</small>	0 ms
10. <b>Array from (1)</b> <small>Rerun</small>	0 ms
11. <b>set.keys (1)</b> <small>Rerun</small>	0 ms
12. <b>set.values (1)</b> <small>Rerun</small>	0 ms
13. <b>set.entries (1)</b> <small>Rerun</small>	0 ms
14. <b>set.forEach (1)</b> <small>Rerun</small>	0 ms
15. <b>并集测试 (1)</b> <small>Rerun</small>	0 ms
16. <b>交集测试 (1)</b> <small>Rerun</small>	0 ms
17. <b>差集测试 (1)</b> <small>Rerun</small>	1 ms

怎么样，还不错吧？

今天就分享到这，今日留言话题：**Set数据结构你们学会了吗？**一起来说说吧，对于有价值的留言，我们都会一一回复的。如果觉得对你有一丢丢帮助，请点右下角【**在看**】，让更多人看到该文章。



▼ 长按二维码关注




打个赌，关注了我，  
你工资最少**涨30%**

**不仅每天干货分享**  
**每周 1, 3, 5 还送书**

别看了，快关注吧！

**优秀的程序员都关注了！**

 程序员手册大全