Set和Map数据结构

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。

```
var s = new Set();

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

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

其他

- 源码
- 修订历史
- 反馈意见

```
// 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（或类似数组的对象）作为参数，用来初始化。

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

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

// 例三
function divs () {
  return [...document.querySelectorAll('div')];
}

var set = new Set(divs());
set.size // 56

// 类似于
divs().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。

上面代码中，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)]
```

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 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
var properties = {
  'width': 1,
  'height': 1
};

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

// Set的写法
var 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。

```
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]
```

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()`

`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。

```
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];
// ['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`方法也可以用于Set了。

```
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

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。

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。这个特点意味着，无法引用

WeakSet的成员，因此WeakSet是不可遍历的。

```
var 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，结果报错，因为WeakSet只能放置对象。

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

```
var 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。

```
var a = [[1,2], [3,4]];
var ws = new WeakSet(a);
```

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。

```
var b = [3, 4];
var 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。

```
var ws = new WeakSet();
var obj = {};
var 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

Map结构的目的和基本用法

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。

```
var data = {};
var 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。

```
var m = new Map();
var 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`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。

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。

```
var 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。

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

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

下面的例子中，字符串`true`和布尔值`true`是两个不同的键。

```
var m = new Map([
  [true, 'foo'],
  ['true', 'bar']
]);

m.get(true) // 'foo'
m.get('true') // 'bar'
```

如果对同一个键多次赋值，后面的值将覆盖前面的值。

```
let 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。

```
var 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。

```
var map = new Map();

var k1 = ['a'];
var 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，包括`0`和`-0`。另外，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(NaN, 123);
map.get(NaN) // 123

map.set(-0, 123);
map.get(+0) // 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。

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

map.size // 2
```

（2）set(key, value)

`set`方法设置`key`所对应的键值，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。

```
var 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Map本身，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`。

```
var m = new Map();

var 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Map数据结构中。

```
var 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。

```
var 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。

下面是使用实例。

```
let 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);
}

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

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，比较快速的方法是结合使用扩展运算符（`...`）。

```
let 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）。

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

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

let 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`。

```
var 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（...）。

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

（2）数组转为Map

将数组转入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，它可以转为对象。

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

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

（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})
// [ [ '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。这往往是数组转为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`结构基本类似，唯一的区别是它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，而且键名所指向的对象，不计入垃圾回收机制。

```
var 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
```

上面代码中，如果将`1`和`Symbol`作为WeakMap的键名，都会报错。

`WeakMap`的设计目的在于，键名是对象的弱引用（垃圾回收机制不将该引用考虑在内），所以其所对应的对象可能会被自动回收。当对象被回收后，`WeakMap`自动移除对应的键值对。典型应用是，一个对应DOM元素的`WeakMap`结构，当某个DOM元素被清除，其所对应的`WeakMap`记录就会自动被移除。基本上，`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。

下面是`WeakMap`结构的一个例子，可以看到用法上与`Map`几乎一样。

```
var wm = new WeakMap();
var element = document.querySelector(".element");

wm.set(element, "Original");
wm.get(element) // "Original"

element.parentNode.removeChild(element);
element = null;
wm.get(element) // undefined
```

上面代码中，变量`wm`是一个`WeakMap`实例，我们将一个`DOM`节点`element`作为键名，然后销毁这个节点，`element`对应的键就自动消失了，再引用这个键名就返回`undefined`。

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），也没有`size`属性；二是无法清空，即不支持`clear`方法。这与`WeakMap`的键不被计入引用、被垃圾回收机制忽略有关。因

此，`WeakMap`只有四个方法可用：`get()`、`set()`、`has()`、`delete()`。

```
var wm = new WeakMap();

wm.size
// undefined

wm.forEach
// undefined
```

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。

```
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++;
  myWeakmap.set(myElement, logoData);
}, 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。

```
let _counter = new WeakMap();
let _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)();
    }
  }
}

let 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

上一章 下一章