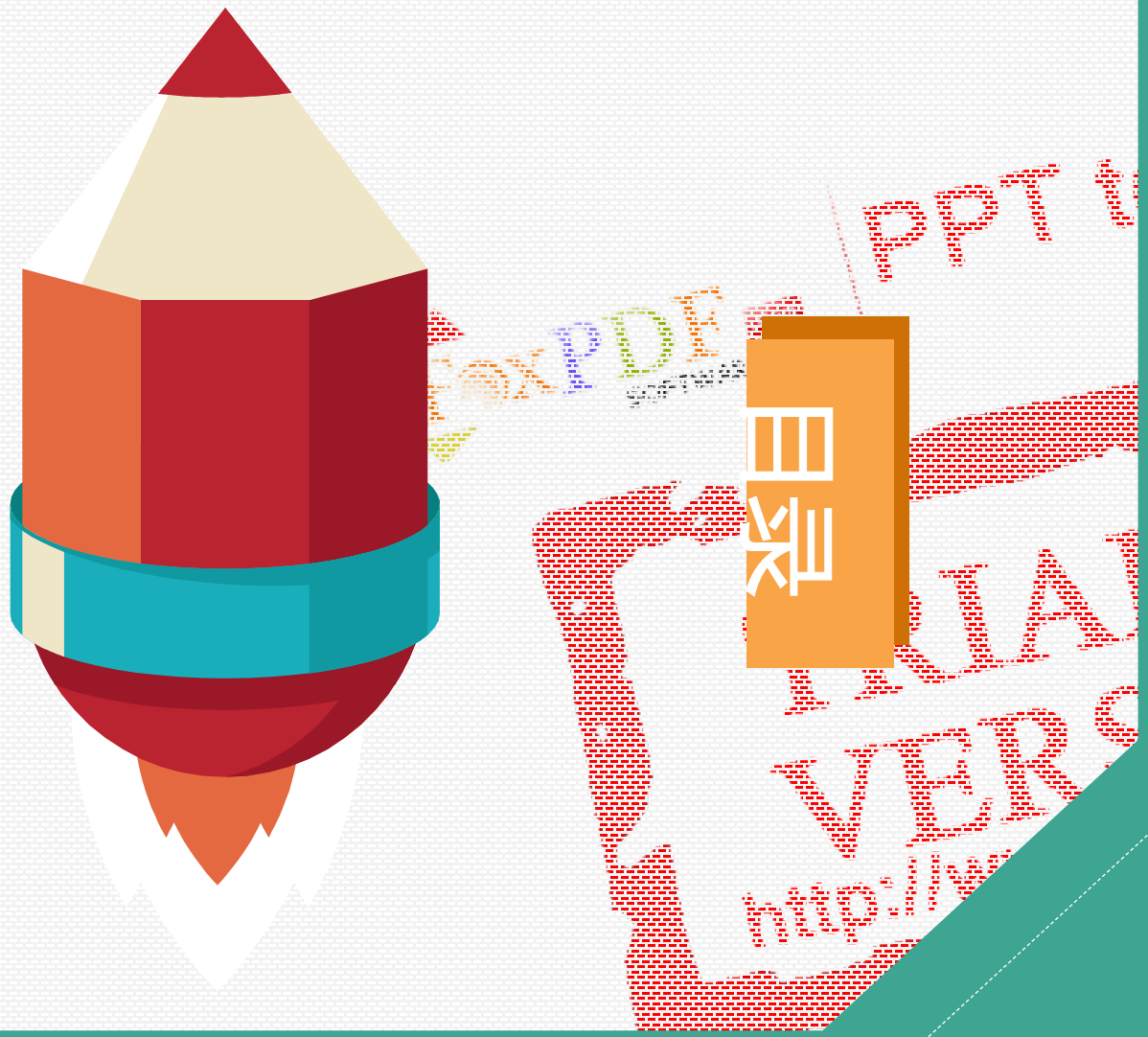

第33课：Map数据结构
主讲老师：万章



目录

- Map数据结构
- Map与其他数据结构的转换
- WeakMap数据类型及应用

Map数据结构

Map数据结构

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(ES6加入了Symbol作为属性名称)。这给它的使用带来了很大的限制。

```
const data = {};
const 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]。

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。

Map数据结构

```
const m = new Map();
const 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 Map 结构的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。读取

Map数据结构

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"
```

```
const items = [
  ['name', '张三'],
  ['title', 'Author']
];

const map = new Map();

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

Map构造函数接受数组作为参数，实际上执行的是上面的算法。

Map数据结构

```
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
```

任何具有 Iterator 接口、且每个成员都是一个双元素的数组([[a,b],[c,d]])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。

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');

map.get(1) // "bbb"
```

如果读取一个未知的键，则返回undefined。

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

Map数据结构

```
const map = new Map();

map.set(['a'], 555);
map.get(['a']) // 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。

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。

```
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
```

同理，同样的值的两个实例，在 Map 结构中被视为两个键。

简而言之：Map数据结构的键名是内存中间的某一段数据，至于这数据是啥无所谓，只有同一位置(内存地址)的同一个数据才会能当做是同一个键名

Map数据结构

如果 Map 的键是一个基础类型的值（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。

```
let map = new Map();

map.set(-0, 123);
map.get(+0) // 123

map.set(true, 1);
map.set('true', 2);
map.get(true) // 1

map.set(undefined, 3);
map.set(null, 4);
map.get(undefined) // 3

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

Map数据结构的属性和方法(和Set结构很像)

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

map.size // 2
```

(1) size 属性

size属性返回 Map 结构的成员(一个键值对算一个)总数。

```
> const m = new Map();

  m.set('edition', 6)        // 键是字符串
< ▶Map(1) {"edition" => 6}
> m.set(262, 'standard')     // 键是数值
< ▶Map(2) {"edition" => 6, 262 => "standard"}
> m.set(undefined, 'nah')    // 键是 undefined
< ▶Map(3) {"edition" => 6, 262 => "standard", undefined => "nah
  "}
```

(2) set(key, value)

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。

set方法返回的是当前的Map对象，因此可以采用链式写法。

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

Map数据结构的属性和方法(和Set结构很像)

```
const m = new Map();

const hello = function() {console.log('hello');};
m.set(hello, 'Hello ES6!') // 键是函数

m.get(hello)  // Hello ES6!
```

(3) get(key)

get方法读取key对应的键值，如果找不到key，返回undefined。

```
const 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
```

(4) has(key)

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。

Map数据结构的属性和方法(和Set结构很像)

```
const m = new Map();
m.set(undefined, 'nah');
m.has(undefined)     // true

m.delete(undefined)
m.has(undefined)       // false
```

(5) delete(key)

delete方法删除某个键，返回true。
如果删除失败，返回false。

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

map.size // 2
map.clear()
map.size // 0
```

(6) clear()

clear方法清除所有成员，没有返回值。

Map数据结构的属性和方法(和Set结构很像)

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。

- 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。

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

Map数据结构的属性和方法(和Set结构很像)

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']]
```

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

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

const map2 = new Map(
  [...map0].map(([k, v]) => [k * 2, '_' + v])
    );
// 产生 Map 结构 {2 => '_a', 4 => '_b', 6 => '_c'}
```

可以根据需求选择对值还是键名或是键值对进行数组转换

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）。

Map数据结构的属性和方法(和Set结构很像)

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);
});
```

```
const reporter = {
  report: function(key, value) {
    console.log("Key: %s, Value: %s", key, value);
  }
};

map.forEach(function(value, key, map) {
  this.report(key, value);
}, reporter);
```

forEach方法还可以接受第二个参数，用来绑定this。上面代码中，forEach方法的回调函数的this，就指向reporter。

Map数据结构

Map与其他数据结构的转换之数组切换

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

（1）Map 转为数组

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

（2）数组 转为 Map

Map与其他数据结构的转换之对象切换

```
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 }
```

(1)如果所有 Map 的键都是字符串，它可以无损地转为对象。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。

```
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}
```

(2) 对象转为Map

WeakMap数据类型及应用

WeakMap数据类型及应用

WeakMap结构与Map结构类似，也是用于生成键值对的集合。

```
// WeakMap 可以使用 set 方法添加成员
const wm1 = new WeakMap();
const key = {foo: 1};
wm1.set(key, 2);
wm1.get(key) // 2

// WeakMap 也可以接受一个数组，
// 作为构造函数的参数
const k1 = [1, 2, 3];
const k2 = [4, 5, 6];
const wm2 = new WeakMap([[k1, 'foo'], [k2, 'bar']]);
wm2.get(k2) // "bar"
```

```
const 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
map.set(null, 2)
// TypeError: Invalid value used as weak map key
```

WeakMap与Map的区别有两点。

- 首先，WeakMap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。
- 其次，WeakMap的键名所指向的对象，不计入垃圾回收机制。

WeakMap数据类型及应用

WeakMap的设计目的在于，有时我们想在某个对象上面存放一些数据，但是这会形成对于这个对象的引用。请看下面的例子

```
const e1 = document.getElementById('foo');
const e2 = document.getElementById('bar');
const arr = [
  [e1, 'foo 元素'],
  [e2, 'bar 元素'],
];
```

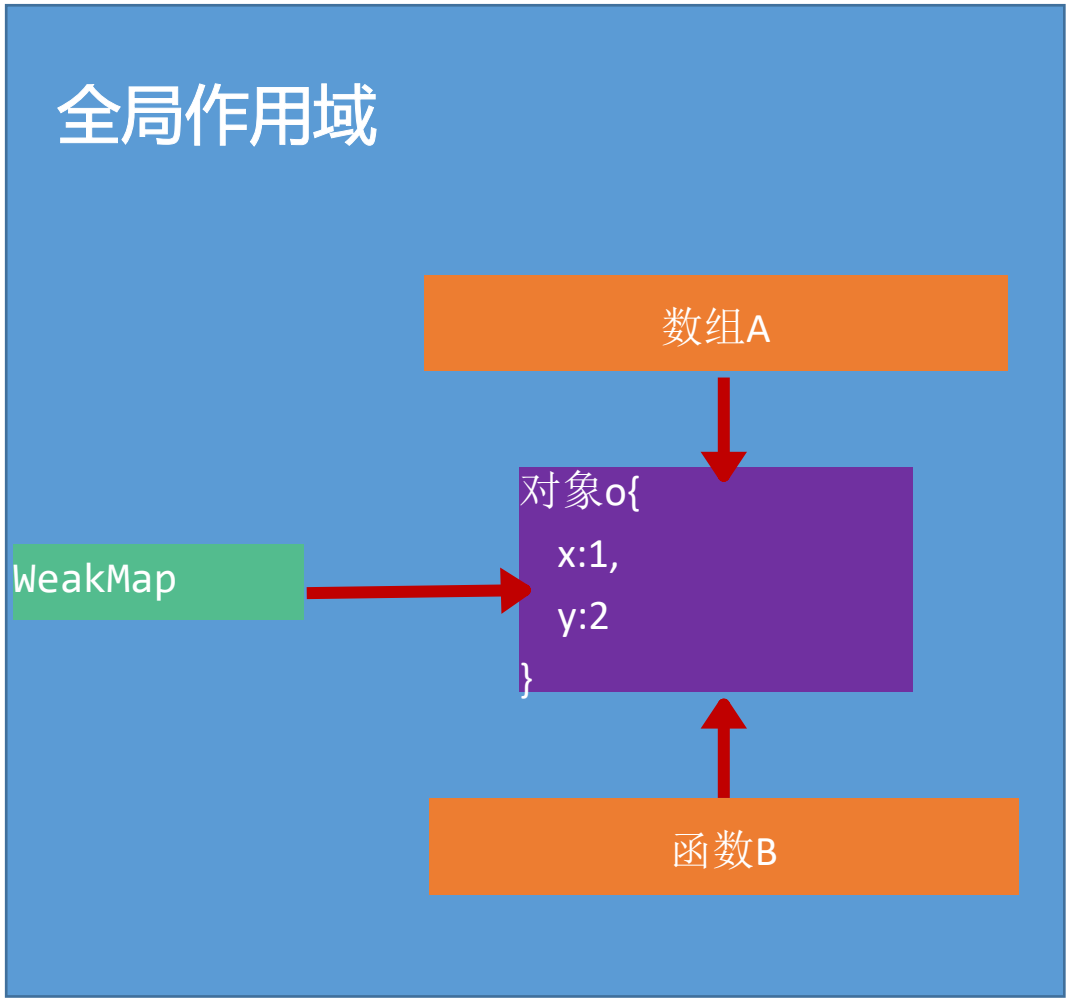
```
// 不需要 e1 和 e2 的时候
// 必须手动删除引用
arr [0] = null;
arr [1] = null;
```

上面代码中，e1和e2是两个对象，我们通过arr数组对这两个对象添加一些文字说明。这就形成了arr对e1和e2的引用。

一旦不再需要这两个对象，我们就必须手动删除这个引用，否则垃圾回收机制就不会释放e1和e2占用的内存。

WeakMap数据类型及应用

WeakMap 就是为了解决这个问题而诞生的，它的键名所引用的对象都是弱引用，即垃圾回收机制不将该引用考虑在内。因此，只要所引用的对象的其他引用都被清除，垃圾回收机制就会释放该对象所占用的内存。也就是说，一旦不再需要，WeakMap 里面的键名对象和所对应的键值对会自动消失，不用手动删除引用。



只要数组a和函数b对这个对象o的引用删除了，对象o就会被清空，不管此时WeakMap数据是否在引用这个对象o

WeakMap数据类型及应用

如果你要往对象上添加数据，又不想干扰垃圾回收机制，就可以使用 WeakMap。

一个典型应用场景是，在网页的 DOM 元素上添加数据，就可以使用WeakMap结构。当该 DOM 元素被清除，其所对应的WeakMap记录就会自动被移除。

```
const wm = new WeakMap();

const element = document.getElementById('example');

wm.set(element, 'some information');
wm.get(element) // "some information"
```

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。

WeakMap数据类型及应用

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keys()、values()和entries()方法），也没有size属性。因为没有办法列出所有键名，某个键名是否存在完全不可预测，跟垃圾回收机制是否运行相关。这一刻可以取到键名，下一刻垃圾回收机制突然运行了，这个键名就没了，为了防止出现不确定性，就统一规定不能取到键名。

- 二是无法清空，即不支持clear方法。

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

WeakMap的经典应用示例1

```
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++;
}, false);
```

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的经典应用示例2

WeakMap 的另一个用处是部署私有属性。

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

const c = new Countdown(2, () => console.log('DONE'));

c.dec()
c.dec()
// DONE
```

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。