集合引用类型

Map

ES6新增特性
与ES6之前通过Object属性为Key相比，带来了真正的Key/Value
Map可以用任何类型作为Key，object只能用数值、字符串或符号做key

基本API

new Map() 构造函数创建空映射
可以给Map构造函数传参可迭代对象，需要包含Key/Value数组

```
// 使用嵌套数组初始化映射
const m1 = new Map([
 ["key1", "val1"],
 ["key2", "val2"],
 ["key3", "val3"]
]);
alert(m1.size); // 3
// 使用自定义迭代器初始化映射
const m2 = new Map({
 [Symbol.iterator]: function*() {
 yield ["key1", "val1"];
 yield ["key2", "val2"];
 yield ["key3", "val3"];
 }
});
alert(m2.size); // 3
// 映射期待的键/值对，无论是否提供
const m3 = new Map([[]]);
alert(m3.has(undefined)); // true
alert(m3.get(undefined)); // undefined
```

- set(key,value) 插入键值 返回整个map
- get(key) 获取值，没有返回undefined
- has(key) 返回boolean
- delete(key) 删除 返回true/false 表示删除成功失败
- clear() 清空所有键值对，没有返回值

Map使用严格对象相等来检查键的匹配性

```
const m = new Map();
const functionKey = function() {};
const symbolKey = Symbol();
const objectKey = new Object();
m.set(functionKey, "functionValue");
m.set(symbolKey, "symbolValue");
```

```
m.set(objectKey, "objectValue");
alert(m.get(functionKey)); // functionValue
alert(m.get(symbolKey)); // symbolValue
alert(m.get(objectKey)); // objectValue
// SameValueZero 比较意味着独立实例不冲突
alert(m.get(function() {})); // undefined
```

在映射中用作键和值的对象及其他“集合”类型，在自己的内容或属性被修改时仍然保持不变

```
const m = new Map();
const objKey = {},
 objVal = {},
 arrKey = [],
 arrVal = [];
m.set(objKey, objVal);
m.set(arrKey, arrVal);
objKey.foo = "foo";
objVal.bar = "bar";
arrKey.push("foo");
arrVal.push("bar");
console.log(m.get(objKey)); // {bar: "bar"}
console.log(m.get(arrKey)); // ["bar"]
```

意想不到的冲突

```
const m = new Map();
const a = 0/"", // NaN
 b = 0/"", // NaN
 pz = +0,
 nz = -0;
alert(a === b); // false
alert(pz === nz); // true
m.set(a, "foo");
m.set(pz, "bar");
alert(m.get(b)); // foo
alert(m.get(nz)); // bar
```

顺序与迭代

Map与Object主要差异，Map维护插入顺序

```
const m = new Map([
 ["key1", "val1"],
 ["key2", "val2"],
 ["key3", "val3"]
]);
alert(m.entries === m[Symbol.iterator]); // true
for (let pair of m.entries()) {
 alert(pair);
}
// [key1,val1]
// [key2,val2]
// [key3,val3]
for (let pair of m[Symbol.iterator]()) {
```

```
 alert(pair);
}
// [key1,val1]
// [key2,val2]
// [key3,val3]


const m = new Map([
 ["key1", "val1"],
 ["key2", "val2"],
 ["key3", "val3"]
]);
console.log([...m]); // [[key1,val1],[key2,val2],[key3,val3]]
```

Map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 m = new Map([
 ["key1", "val1"],
 ["key2", "val2"],
 ["key3", "val3"]
]);
m.forEach((val, key) => alert(`${key} -> ${val}`));
// key1 -> val1
// key2 -> val2
// key3 -> val3

const m = new Map([
 ["key1", "val1"],
 ["key2", "val2"],
 ["key3", "val3"]
]);
for (let key of m.keys()) {
 alert(key);
}
// key1
// key2
// key3
for (let key of m.values()) {
 alert(key);
}
// value1
// value2
// value3
```

使用Object还是Map

1. 内存
   定固定大小的内存，Map 大约可以比 Object 多存储 50%的键/值对
2. 插入性能
   Map稍快

3. 查找速度
   大型 差异不大
   小型 Object快
   连续 属性Object快
4. 删除性能
   Object性能很差(垃圾回收)，一般设置成null/undefined
   大量删除使用Map

WeakMap

ES6新增
WeakMap是Map"兄弟类型",API是Map的子集
Weak指代垃圾回收程序对待它的方式

与Map相比区别：

- 只接受对象（Object 或者继承自 Object 的类型）作为key(null除外)
- WeakMap键名所指的对象，不计入垃圾回收机制
  悄悄地拿着，垃圾回收机制不知道，所以照样回收
  一旦被回收，key/value对自动被移除
- 因为随时可能被回收，所以不可迭代，也不需要clear()

使用WeakMap

1. 私有变量

```
const wm = new WeakMap();
class User {
 constructor(id) {
 this.idProperty = Symbol('id');
 this.setId(id);
 }
 setPrivate(property, value) {
 const privateMembers = wm.get(this) || {};
 privateMembers[property] = value;
 wm.set(this, privateMembers);
 }
 getPrivate(property) {
 return wm.get(this)[property];
 }
 setId(id) {
 this.setPrivate(this.idProperty, id);
 }
 getId() {
 return this.getPrivate(this.idProperty);
 }
}
const user = new User(123);
alert(user.getId()); // 123
user.setId(456);
alert(user.getId()); // 456
// 并不是真正私有的
alert(wm.get(user)[user.idProperty]); // 456
```

使用闭包

```
const User = (() => {
 const wm = new WeakMap();
 class User {
 constructor(id) {
 this.idProperty = Symbol('id');
 }
 setPrivate(property, value) {
 const privateMembers = wm.get(this) || {};
 privateMembers[property] = value;
 wm.set(this, privateMembers);
 }
 getPrivate(property) {
 return wm.get(this)[property];
 }
 setId(id) {
 this.setPrivate(this.idProperty, id);
 }
 getId(id) {
 return this.getPrivate(this.idProperty);
 }
 }
 return User;
})();
const user = new User(123);
alert(user.getId()); // 123
user.setId(456);
alert(user.getId()); // 456
```

2. DOM节点元数据
   因为 WeakMap 实例不会妨碍垃圾回收，所以非常适合保存关联元数据

```
const m = new Map();
const loginButton = document.querySelector('#login');
// 给这个节点关联一些元数据
m.set(loginButton, {disabled: true});

const wm = new WeakMap();
// 给这个节点关联一些元数据
wm.set(loginButton, {disabled: true});
```

假设loginButton的DOM节点被删掉了map的内存无法释放，WeakMap则自动释放