05-第五章 ECMAScript6 Set Map Proxy

ECMAscript 6

一、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。

```
const s = new Set();
        console.log(s);//{}
```

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。

```
        const arr = [1,2,3,2,4,4];
        var r = new Set(arr);
        console.log(r)//{1, 2, 3, 4}
        var r = [...new Set(arr)];
        console.log( r);//[1, 2, 3, 4]

```

```
var res = new Set(document.getElementsByTagName('li'));
        var r = [...res];
        console.log( res );//{li {}, li {}, li {}, li {}};
        console.log( r );//[li, li, li, li]
        console.log( r instanceof Array );//true
```

```
1. 值的唯一性
```

```
1. const arr = new Set([1,6,2,4,5,4]);
2.          console.log( arr );//Set {1, 2, 3, 4, 5}
```

数组去重

```
1.          const arr = [1,2,3,2,4,4];
2.          console.log( [...new Set(arr)]);//[1, 2, 3, 4]
```

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`”
精确相等运算符认为`NaN`不等于自身

```
1.      let s = new Set(['5',5]);
2.          console.log( s );//{"5", 5}
```

```
1.  let s = new Set([8,8,NaN,NaN]);
2.          console.log(s);//{8, NaN}
```

二、Set对象的属性和方法

Set对象的属性：
`.size` —返回Set实例成员总数

```
1. var s = new Set();
2.          console.log(s.size);//0
3.          s.add(1);
4.          console.log(s.size);//1
```

Set对象的方法：

`add(`value`)` 添加某个值，返回结构本身
`delete(`value`)` 删除某个值，返回布尔值，表示删除是否成功
`has(`value`)` 是否为Set成员,返回布尔值
`clear()` 清除所有成员

```
var s = new Set(['a','b']);
        s.add(5).add(6).add(7);
        console.log( s.has(5));//true
         console.log( s );//{"a", "b", 5, 6, 7}
        s.delete(5);
        console.log( s.has(5));//false
        console.log( s );//{"a", "b",  6, 7}
        s.clear();
        console.log(s) //{}

```

三、set 遍历

Set结构的实例的四个遍历方法

> `keys()` –返回键 名 的遍历器
> `values()`–返回键 值 的遍历器
> `entries()` –返回 键值对 的遍历器
> `forEach()` –使用回调函数遍历每个成员

(1) keys() values() entries()

```
let  set = new Set(['green','red']);
    for (key of set.keys()) {
        console.log( key );// green red
    }
    for (key of set.values()) {
        console.log( key );// green red
    }

    for (key of set.entries()) {
        console.log( key );
        // ['green','red']
        // ['green','red']
    }


```

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

---

(2) forEach()

```
set.forEach((value,index)=>{
    console.log(value,index)
    // ['green','red']
    // ['green','red']
})
```

---

(3) 遍历的应用

拓展运算符（`...`）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']);

    console.log( [...set] );//["red", "green", "blue"]
```

四。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

```
var weak = new WeakSet();
    console.log( weak); //WeakSet {}
    weak.add(['index','age']);
     console.log( weak);//{["index", "age"]}
    var weak = new WeakSet([['index','age']]);
    console.log( weak);
    weak.add({'ppp':111});
    console.log( weak);
```

其他报错

```
var weak = new WeakSet(['index','age']);//TypeError
    var weak = new WeakSet(1);//TypeError
    var weak = new WeakSet( Symbol() );//TypeError

    var weak = new WeakSet( {index:888,age:999} );//TypeError
    var weak = new WeakSet('hello');//TypeError
```

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。）

WeakSet 结构有以下三个方法

> `add(` value `)`：向 WeakSet 实例添加一个新成员。
> `delete(` value `)`：清除 WeakSet 实例的指定成员。
> `has(` value `)`：返回一个布尔值，表示某个值是否在 WeakSet 实例之中。

```
var weak = new WeakSet();
    var obj = {index:888};
    var arr = ['age'];
    weak.add(obj);
    console.log( weak.has(obj) );//true
    weak.delete(obj)
     console.log( weak.has(obj) );//false

```

五。Map()

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。

> S6 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,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构造函数可以接受二维数组作为参数，

```
1.      let map =  new Map([['a','b'],[1,2]]);
2.     console.log( map);//{"a" => "b", 1 => 2}
```

```
1.      let map = new Map();
2.      const o= {say:'Hello World'};
3.      map.set(o,'content');
4.      var r = map.get(o);
5.      console.log( r );//content
6.      console.log( map );//Map {Object {say: "Hello World"} => "con
    tent"}
```

实际上执行的是下面的算法。

```
1.  let map =  new Map();
2.     var arr = [
3.      ['a','b'],
4.      [1,2]
5.     ];
6.     arr.forEach(([key,val])=>{
7.      map.set(key,val);
8.     })
9.     console.log(map);//{"a" => "b", 1 => 2}
```

任何具有 Iterator 接口、且每个成员都是一个双元素的数组的数据结构（详见《Iterator》一章）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

```
1.  var set = new Set([
2.      ['age',20],
3.      ['name','二狗']
4.     ]);
5.
6.      let map =  new Map(set);
7.      console.log( map.get('age') );//20
8.
9.      let map2 = new Map(map);
10.     console.log(map2);//{"age" => 20, "name" => "二狗"}
```

相同的对象，在 Map 结构中被视为两个键。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

```
const map = new Map();
   const k1= ['a'];
   const k2= ['b'];
   map.set(k1,1).set(k2,1);
   console.log( map.get(k1),map.get(k2) );
```

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，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

六。Map() 实例的属性和方法

`size` –Map结构成员个数
`set(` key,value `)` –方法设置键名key对应的键值为value
`get(` key `)` –方法读取key对应的键值,返回的是当前的Map对象，因此可以采用链式写法
`has(` key `)` 返回一个布尔值，表示某个键是否在当前 Map 对象之中
`delete(` key `)` –删除某个键，返回true。如果删除失败，返回false
`clear()` 方法 清除所有成员

```
 var map = new Map([
        ['a',1],
        ['b',2]
    ]);

   console.log(map.size);//2
   map.set('age',20).set('name','二狗');
   console.log( map.get('name'));//二狗
   console.log( map.has('age'))//true
   map.delete('age');
   console.log( map.has('age'))//false
   map.clear();
   console.log( map );//Map {}
```

七。Map() 遍历方法

Map 遍历方法：

`keys()`–键名的遍历器
`values()`–键值的遍历器
`entries()`–返回键值对遍历器
`forEach()`–遍历Map的所有成员

```
var map = new Map([
        ['a',1],
        ['b',2],
    ]);

    for (key of map.keys()) {
        console.log( key );// a b
    }
    for (key of map.values()) {
        console.log( key );// 1 2
    }
    for (key of map.entries()) {
        console.log( key );
        //["a", 1]
        //["b", 2]
    }

    map.forEach((value,index,self)=>{
         console.log( value );
         console.log( index );
         console.log( self );
         // 1 a {"a" => 1, "b" => 2}
         // 2 b {"a" => 1, "b" => 2}

    })
```

扩展运算符（`...`）使Map 结构转为数组结构

```
 var map = new Map([
        ['a',1],
        ['b',2],
    ]);

    console.log( [...map]);//[['a',1]['b',2]]
```

八、WeakMap()

WeakMap结构与Map结构类似，也是用于生成键值对的集合。
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。

```
var a = ['a'];
    var b = ['b'];
    var weakM = new WeakMap([[a,1],[b,2]]);
    console.log(weakM);//{["a"] => 1, ["b"] => 2}
```

```
var weakM = new WeakMap();
    var key = {foo:1};
    weakM.set(key,'world');
    console.log(weakM);//{Object {foo: 1} => "world"}
```

```
var aLi = document.getElementsByTagName('li');
    var arr =[
        [aLi[0],'元素1'],
        [aLi[1],'元素2'],
    ]
    var wmap = new WeakMap(arr);
    console.log(wmap);//{li {} => "元素2", li {} => "元素1"}
```

九。WeakMap方法

WeakMap 与 Map 在 API 上的区别主要是两个：

WeakMap只有四个方法可用：`get()`、`set()`、`has()`、delete()
一 WeakMap没有size属性，也没有遍历方法

```
let wmap = new Map();
    var obj = {};
    var val = 888;
    wmap.set(obj,val);
    console.log( wmap.has(obj) );//true
    console.log( wmap.get(obj) );//888
   wmap.delete(obj);
    console.log( wmap.get(obj) );//undefined
```

proxy

一、

> Proxy 用于修改某些操作的默认行为，等同于在语言层面做出修改，所以属于一种“元编程”（meta programming），Proxy 可以理解成，在目标对象之前架设一层“拦截”，外界对该对象的访问，都必须先通过这层拦截，因此提供了一种机制，可以对外界的访问进行过滤和改写。Proxy 这个词的原意是代理，用在这里表示由它来“代理”某些操作，可以译为“代理器”

ES6 原生提供 Proxy 构造函数，用来生成 Proxy 实例

> 语法
> `var proxy= new Proxy(target,handler);`
> target 是被拦截目标，
> handler是用于定义拦截操作

如果handler没有设置任何拦截，那就等同于直接通向原对象。

```
var target = {};
        var handler = {};
        var proxy = new Proxy(target,handler);
        proxy.a = '666';
        console.log( target.a );//666
```

设置拦截

```
var obj = new Proxy({age:888},{
            get:function(target,key,receive){
                console.log(target);//Object {age: 888}
                console.log(key);//index
                console.log(receive);//Proxy {age: 888}
            }
        })
        console.log(obj);////Proxy {age: 888}
        //obj.index;
```

`target` — 就是被拦截目标
`key` —指的属性
`receive` –是Proxy实例对象

```
var obj = new Proxy({age:888},{
            get:function(target,key,receive){
                console.log(target);
                console.log(key);
                console.log(receive);
            },
            set:function(target,key,value,receive){
                console.log(target);//Object {age: 888}
                console.log(key);//index
                console.log(value);//55
                console.log(receive);//Proxy {age: 888}
            }
        })
        obj.age = 55;
```

`target` — 就是被拦截目标
`key` —指的属性
`value` —要设置的值
`receive` –是Proxy实例对象

Proxy实例的方法

Proxy 支持的拦截操作：13种

`get(` target, propKey, receiver `)` ：拦截对象属性的读取，比如proxy.foo和proxy['foo']。
`set(` target, propKey, value, receiver `)` ：拦截对象属性的设置，比如proxy.foo = v或proxy['foo'] = v，返回一个布尔值。
`has(` target, propKey `)` ：拦截propKey in proxy的操作，返回一个布尔值。
`deleteProperty(` target, propKey `)` ：拦截delete proxy[propKey]的操作，返回一个布尔值。
`ownKeys(` target `)` ：拦截Object.getOwnPropertyNames(proxy)、Object.getOwnPropertySymbols(proxy)、Object.keys(proxy)，返回一个数组。该方法返回目标对象所有自身的属性的属性名，而Object.keys()的返回结果仅包括目标对象自身的可遍历属性。
`getOwnPropertyDescriptor(` target, propKey `)` ：拦截Object.getOwnPropertyDescriptor(proxy, propKey)，返回属性的描述对象。
`defineProperty(` target, propKey, propDesc `)` ：拦截Object.defineProperty(proxy, propKey, propDesc）、Object.defineProperties(proxy, propDescs)，返回一个布尔值。
`preventExtensions(` target `)` ：拦截Object.preventExtensions(proxy)，返回一个布尔值。
`getPrototypeOf(` target `)` ：拦截Object.getPrototypeOf(proxy)，返回一个对象。
`isExtensible(` target `)` ：拦截Object.isExtensible(proxy)，返回一个布尔值。
`setPrototypeOf(` target, proto `)` ：拦截Object.setPrototypeOf(proxy, proto)，返回一个布尔值。如果目标对象是函数，那么还有两种额外操作可以拦截。
`apply(` target, object, args `)` ：拦截 Proxy 实例作为函数调用的操作，比如proxy(...args)、proxy.call(object, ...args)、proxy.apply(...)。
`construct(` target, args `)` ：拦截 Proxy 实例作为构造函数调用的操作，比如new proxy(...args)。

Proxy实例的方法详细介绍

(1) `get()`

方法用于拦截某个属性的读取操作，可以接受三个参数，依次为目标对象、属性名和 proxy 实例本身（即this关键字指向的那个对象），其中最后一个参数可选

```
var target = {};
        var proxy = new Proxy(target,{
            get:function(target,property,prox){}

        });
```

(2) `set()`

> 法用来拦截某个属性的赋值操作，可以接受四个参数，依次为目标对象、属性名、属性值和 Proxy 实例本身，其中最后一个参数可选

```
var target = {};
        var proxy = new Proxy(target,{
            set:function(target,property,value,prox){}

        });
```

(3) `apply()`

> apply方法拦截 `函数的调用` 、 `call` 和 `apply` 操作。
>
> apply方法可以接受三个参数，分别是 `目标对象` 、 `目标对象的上下文对象` （this）和 `目标对象的参数数组` 。

```
        var target = function(){};
        var proxy = new Proxy(target,{
            apply:function(target,ctx,args){
                console.log(target);//function (){}
                console.log(ctx);//undefined
                console.log(args);//["ok"]


            }
        });

        proxy('ok');
```

(4) has()

> has方法用来拦截HasProperty操作，即判断对象是否具有某个属性时，这个方法会生效。典型的操作就是in运算符。

下面的例子使用has方法隐藏某些属性，不被in运算符发现

```
var target = {};
        var proxy = new Proxy(target,{
            set:function(target,ctx,args){}

        });
```

(5) construct()

> 方法用于拦截new命令，下面是拦截对象的写法。

```
 var target = function(){};
        var proxy = new Proxy(target,{
            construct:function(target,args,newTarget){
                console.log(target);//{age: 20}
                console.log(args);//age
                console.log(newTarget);//anonymous()
                return newTarget;
            }
        });
        new proxy('ok');
```

construct方法的`必须返回一个对象`，否则会报错。

```
var target = function(){};
        var proxy = new Proxy(target,{
            construct:function(target,args,newTarget){
                console.log(target);//{age: 20}
                console.log(args);//age
                console.log(newTarget);//anonymous()

            }
        });
        new proxy('ok');//'construct' on proxy: trap returned no
    n-object ('undefined')
```

(6) deleteProperty()

> 用于拦截delete操作，如果这个方法抛出错误或者返回false，当前属性就无法被delete命令删除

```
var target = function(){};
        var proxy = new Proxy(target,{
            construct:function(target,args,descripor){
            }
        });
```

(7) defineProperty()

> 拦截了Object.defineProperty操作

```
var target = function(){};
        var proxy = new Proxy(target,{
            construct:function(target,args,descripor){
            }
        });
proxy.foo
```

(8) `getOwnPropertyDescriptor()`

> 拦截Object.getOwnPropertyDescriptor()，返回一个属性描述对象或者undefined

```
var target = function(){};
        var proxy = new Proxy(target,{
            getOwnPropertyDescriptor:function(target,key){
            }
        });
        Object.getOwnPropertyDescriptor(proxy, 'wat')
```

---

(9) `getPrototypeOf()`

> getPrototypeOf方法主要用来拦截获取对象原型。具体来说，拦截下面这些操作
>
> Object.prototype.`__proto__`
> Object.prototype.`isPrototypeOf()`
> Object.`getPrototypeOf()`
> Reflect.`getPrototypeOf()`
> `instanceof`

```
var proto = {};
var p = new Proxy({}, {
  getPrototypeOf(target) {
    return proto;
  }
});
Object.getPrototypeOf(p) === proto // true
```

---

(10) `isExtensible()`

> 拦截Object.isExtensible操作

```
var target = function(){};
        var proxy = new Proxy(target,{
           isExtensible:function(target){
            }
        });
        Obejct.isExtensible(proxy);
```

(11) ownkeys()

> 方法用来拦截对象自身属性的读取操作

```
Object.getOwnPropertyNames()
Object.getOwnPropertySymbols()
Object.keys()
```

(12) preventExtensions()

> 这个方法有一个限制，只有目标对象不可扩展时（即Object.isExtensible(proxy)为false），proxy.preventExtensions才能返回true，否则会报错。

```
var p = new Proxy({}, {
  preventExtensions: function(target) {
    return true;
  }
});

Object.preventExtensions(p) // 报错
```

(13) setPrototypeOf()

> 方法主要用来拦截Object.setPrototypeOf方法

注意，该方法只能返回布尔值，否则会被自动转为布尔值。另外，如果目标对象不可扩展（extensible），setPrototypeOf方法不得改变目标对象的原型。

三 Proxy.revocable()

> Proxy.revocable方法返回一个可取消的 Proxy 实例

```
let target = {};
let handler = {};

let {proxy, revoke} = Proxy.revocable(target, handler);

proxy.foo = 123;
proxy.foo // 123

revoke();
proxy.foo // TypeError: Revoked
```

Proxy.revocable方法返回一个对象，该对象的proxy属性是Proxy实例，revoke属性是一个函数，可以取消Proxy实例。上面代码中，当执行revoke函数之后，再访问Proxy实例，就会抛出一个错误。

Proxy.revocable的一个使用场景是，目标对象不允许直接访问，必须通过代理访问，一旦访问结束，就收回代理权，不允许再次访问

> 总结

一`set()`
二`weakSet()`
三`Map()`
四`weakMap()`
五`Proxy`

> `get(` target, propKey, receiver `)`：拦截对象属性的读取，比如proxy.foo和proxy[‘foo’]。
>
> `set(` target, propKey, value, receiver `)`：拦截对象属性的设置，比如proxy.foo = v或proxy[‘foo’] = v，返回一个布尔值。
> `has(` target, propKey `)`：拦截propKey in proxy的操作，返回一个布尔值。

`deleteProperty(` target, propKey `)`：拦截delete proxy[propKey]的操作，返回一个布尔值。
`ownKeys(` target `)`：拦截Object.getOwnPropertyNames(proxy)、Object.getOwnPropertySymbols(proxy)、Object.keys(proxy)，返回一个数组。该方法返回目标对象所有自身的属性的属性名，而Object.keys()的返回结果仅包括目标对象自身的可遍历属性。

`getOwnPropertyDescriptor(` target, propKey `)`：拦截Object.getOwnPropertyDescriptor(proxy, propKey)，返回属性的描述对象。
`defineProperty(` target, propKey, propDesc `)`：拦截Object.defineProperty(proxy, propKey, propDesc）、Object.defineProperties(proxy, propDescs)，返回一个布尔值。
`preventExtensions(` target `)`：拦截Object.preventExtensions(proxy)，返回一个布尔值。
`getPrototypeOf(` target `)`：拦截Object.getPrototypeOf(proxy)，返回一个对象。

`setPrototypeOf(` target, proto `)`：拦截Object.setPrototypeOf(proxy, proto)，返回一个布尔值。如果目标对象是函数，那么还有两种额外操作可以拦截。
`apply(` target, object, args `)`：拦截 Proxy 实例作为函数调用的操作，比如proxy(...args)、proxy.call(object, ...args)、proxy.apply(...)。
`construct(` target, args `)`：拦截 Proxy 实例作为构造函数调用的操作，比如new proxy(...args)。
`isExtensible(` target `)`：拦截Object.isExtensible(proxy)，返回一个布尔值。