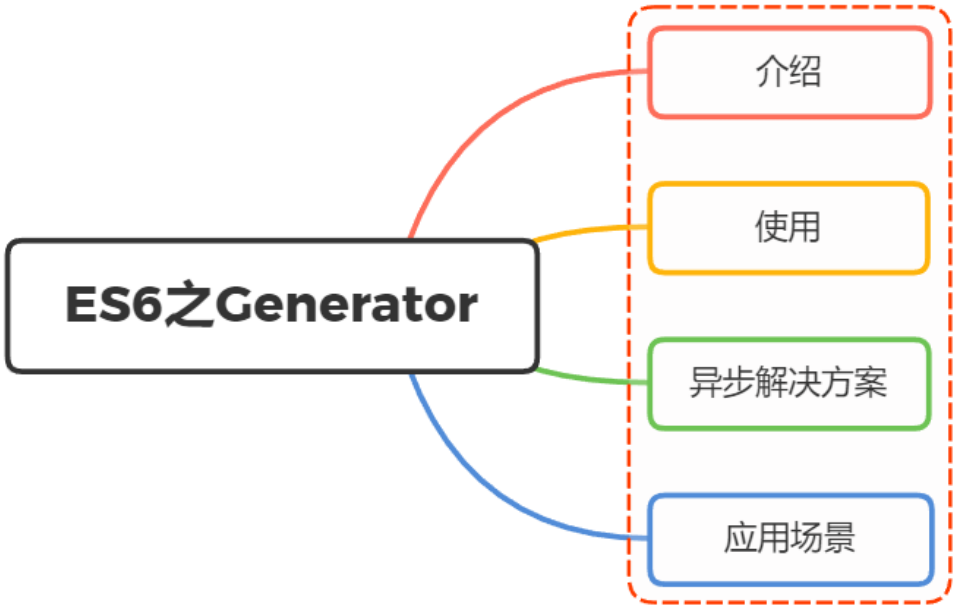


7.1. 介绍

Generator 函数是 ES6 提供的一种异步编程解决方案，语法行为与传统函数完全不同

回顾下上文提到的解决异步的手段：

- 回调函数
- promise

那么，上文我们提到 `promsie` 已经是一种比较流行的解决异步方案，那么为什么还出现 `Generator`？甚至 `async/await` 呢?

该问题我们留在后面再进行分析，下面先认识下 `Generator`

7.1.1. Generator函数

执行 `Generator` 函数会返回一个遍历器对象，可以依次遍历 `Generator` 函数内部的每一个状态

形式上， `Generator` 函数是一个普通函数，但是有两个特征：

- `function` 关键字与函数名之间有一个星号
- 函数体内部使用 `yield` 表达式，定义不同的内部状态

```JavaScript
function* helloWorldGenerator() {
  yield 'hello';
  yield 'world';
  return 'ending';
}
```

7.2. 使用

`Generator` 函数会返回一个遍历器对象，即具有 `Symbol.iterator` 属性，并且返回给自己

```JavaScript
function* gen(){
  // some code
}

var g = gen();

g[Symbol.iterator]() === g
// true
```

通过 `yield` 关键字可以暂停 `generator` 函数返回的遍历器对象的状态

```JavaScript
function* helloWorldGenerator() {
  yield 'hello';
  yield 'world';
  return 'ending';
}
var hw = helloWorldGenerator();
```

上述存在三个状态： `hello` 、 `world` 、 `return`

通过 `next` 方法才会遍历到下一个内部状态，其运行逻辑如下：

- 遇到 `yield` 表达式，就暂停执行后面的操作，并将紧跟在 `yield` 后面的那个表达式的值，作为返回的对象的 `value` 属性值。
- 下一次调用 `next` 方法时，再继续往下执行，直到遇到下一个 `yield` 表达式
- 如果没有再遇到新的 `yield` 表达式，就一直运行到函数结束，直到 `return` 语句为止，并将 `return` 语句后面的表达式的值，作为返回的对象的 `value` 属性值。

- 如果该函数没有 `return` 语句，则返回的对象的 `value` 属性值为 `undefined`

JavaScript 复制代码

```
hw.next()
// { value: 'hello', done: false }

hw.next()
// { value: 'world', done: false }

hw.next()
// { value: 'ending', done: true }

hw.next()
// { value: undefined, done: true }
```

`done` 用来判断是否存在下个状态， `value` 对应状态值

`yield` 表达式本身没有返回值，或者说总是返回 `undefined`

通过调用 `next` 方法可以带一个参数，该参数就会被当作上一个 `yield` 表达式的返回值

JavaScript 复制代码

```
function* foo(x) {
  var y = 2 * (yield (x + 1));
  var z = yield (y / 3);
  return (x + y + z);
}

var a = foo(5);
a.next() // Object{value:6, done:false}
a.next() // Object{value:NaN, done:false}
a.next() // Object{value:NaN, done:true}

var b = foo(5);
b.next() // { value:6, done:false }
b.next(12) // { value:8, done:false }
b.next(13) // { value:42, done:true }
```

正因为 `Generator` 函数返回 `Iterator` 对象，因此我们还可以通过 `for...of` 进行遍历

```javascript
function* foo() {
  yield 1;
  yield 2;
  yield 3;
  yield 4;
  yield 5;
  return 6;
}

for (let v of foo()) {
  console.log(v);
}
// 1 2 3 4 5
```

原生对象没有遍历接口，通过 `Generator` 函数为它加上这个接口，就能使用 `for...of` 进行遍历了

```javascript
function* objectEntries(obj) {
  let propKeys = Reflect.ownKeys(obj);

  for (let propKey of propKeys) {
    yield [propKey, obj[propKey]];
  }
}

let jane = { first: 'Jane', last: 'Doe' };

for (let [key, value] of objectEntries(jane)) {
  console.log(`${key}: ${value}`);
}
// first: Jane
// last: Doe
```

7.3. 异步解决方案

回顾之前展开异步解决的方案：

- 回调函数
- Promise 对象
- generator 函数

- async/await

这里通过文件读取案例，将几种解决异步的方案进行一个比较：

7.3.1. 回调函数

所谓回调函数，就是把任务的第二段单独写在一个函数里面，等到重新执行这个任务的时候，再调用这个函数

JavaScript | 复制代码

```
fs.readFile('/etc/fstab', function (err, data) {
  if (err) throw err;
  console.log(data);
  fs.readFile('/etc/shells', function (err, data) {
    if (err) throw err;
    console.log(data);
  });
});
```

`readFile` 函数的第三个参数，就是回调函数，等到操作系统返回了 `/etc/passwd` 这个文件以后，回调函数才会执行

7.3.2. Promise

`Promise` 就是为了解决回调地狱而产生的，将回调函数的嵌套，改成链式调用

```javascript
const fs = require('fs');

const readFile = function (fileName) {
  return new Promise(function (resolve, reject) {
    fs.readFile(fileName, function(error, data) {
      if (error) return reject(error);
      resolve(data);
    });
  });
};


readFile('/etc/fstab').then(data =>{
    console.log(data)
    return readFile('/etc/shells')
}).then(data => {
    console.log(data)
})
```

这种链式操作形式，使异步任务的两段执行更清楚了，但是也存在了很明显的问题，代码变得冗杂了，语义化并不强

7.3.3. generator

`yield` 表达式可以暂停函数执行， `next` 方法用于恢复函数执行，这使得 `Generator` 函数非常适合将异步任务同步化

```javascript
const gen = function* () {
  const f1 = yield readFile('/etc/fstab');
  const f2 = yield readFile('/etc/shells');
  console.log(f1.toString());
  console.log(f2.toString());
};
```

7.3.4. async/await

将上面 `Generator` 函数改成 `async/await` 形式，更为简洁，语义化更强了

```javascript
const asyncReadFile = async function () {
  const f1 = await readFile('/etc/fstab');
  const f2 = await readFile('/etc/shells');
  console.log(f1.toString());
  console.log(f2.toString());
};
```

7.3.5. 区别

通过上述代码进行分析，将 `promise` 、 `Generator` 、 `async/await` 进行比较：

- `promise` 和 `async/await` 是专门用于处理异步操作的
- `Generator` 并不是为异步而设计出来的，它还有其他功能（对象迭代、控制输出、部署 `Iterator` 接口...）
- `promise` 编写代码相比 `Generator` 、 `async` 更为复杂化，且可读性也稍差
- `Generator` 、 `async` 需要与 `promise` 对象搭配处理异步情况
- `async` 实质是 `Generator` 的语法糖，相当于会自动执行 `Generator` 函数
- `async` 使用上更为简洁，将异步代码以同步的形式进行编写，是处理异步编程的最终方案

7.4. 使用场景

`Generator` 是异步解决的一种方案，最大特点则是将异步操作同步化表达出来

```javascript
function* loadUI() {
  showLoadingScreen();
  yield loadUIDataAsynchronously();
  hideLoadingScreen();
}
var loader = loadUI();
// 加载UI
loader.next()

// 卸载UI
loader.next()
```

包括 `redux-saga` 中间件也充分利用了 `Generator` 特性

JavaScript | 复制代码

```
import { call, put, takeEvery, takeLatest } from 'redux-saga/effects'
import Api from '...'

function* fetchUser(action) {
   try {
      const user = yield call(Api.fetchUser, action.payload.userId);
      yield put({type: "USER_FETCH_SUCCEEDED", user: user});
   } catch (e) {
      yield put({type: "USER_FETCH_FAILED", message: e.message});
   }
}

function* mySaga() {
  yield takeEvery("USER_FETCH_REQUESTED", fetchUser);
}

function* mySaga() {
  yield takeLatest("USER_FETCH_REQUESTED", fetchUser);
}

export default mySaga;
```

还能利用 `Generator` 函数，在对象上实现 `Iterator` 接口

JavaScript | 复制代码

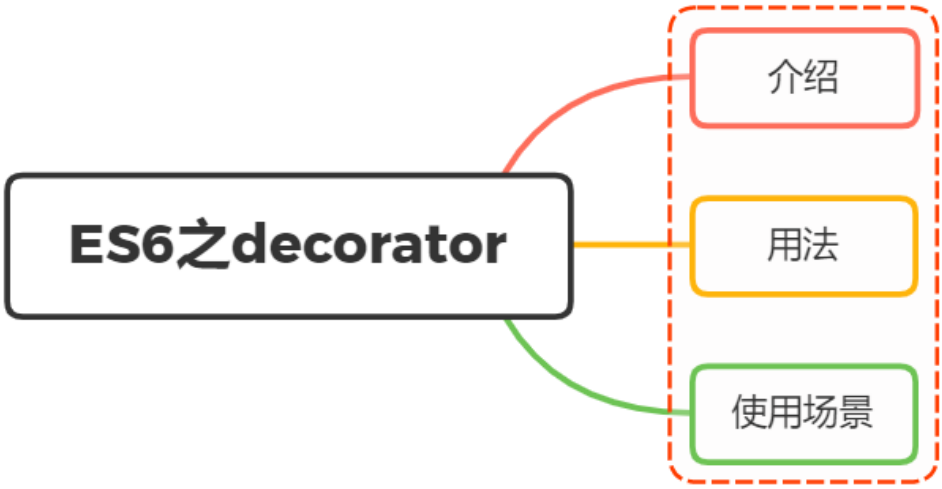
```
function* iterEntries(obj) {
  let keys = Object.keys(obj);
  for (let i=0; i < keys.length; i++) {
    let key = keys[i];
    yield [key, obj[key]];
  }
}

let myObj = { foo: 3, bar: 7 };

for (let [key, value] of iterEntries(myObj)) {
  console.log(key, value);
}

// foo 3
// bar 7
```

8. 你是怎么理解ES6中 Decorator 的？使用场景？



8.1. 介绍

Decorator，即装饰器，从名字上很容易让我们联想到装饰者模式

简单来讲，装饰者模式就是一种在不改变原类和使用继承的情况下，动态地扩展对象功能的设计理论。

`ES6` 中 `Decorator` 功能亦如此，其本质也不是什么高大上的结构，就是一个普通的函数，用于扩展类属性和类方法

这里定义一个士兵，这时候他什么装备都没有

JavaScript 复制代码

```
class soldier{
}
```

定义一个得到 AK 装备的函数，即装饰器

JavaScript 复制代码

```
function strong(target){
    target.AK = true
}
```

使用该装饰器对士兵进行增强

```JavaScript
@strong
class soldier{
}
```

这时候士兵就有武器了

```JavaScript
soldier.AK // true
```

上述代码虽然简单，但也能够清晰看到了使用 `Decorator` 两大优点：

- 代码可读性变强了，装饰器命名相当于一个注释
- 在不改变原有代码情况下，对原来功能进行扩展

8.2. 用法

`Docorator` 修饰对象为下面两种：

- 类的装饰
- 类属性的装饰

8.2.1. 类的装饰

当对类本身进行装饰的时候，能够接受一个参数，即类本身

将装饰器行为进行分解，大家能够有个更深入的了解

```JavaScript
@decorator
class A {}

// 等同于

class A {}
A = decorator(A) || A;
```

下面 `@testable` 就是一个装饰器， `target` 就是传入的类，即 `MyTestableClass` ，实现了为类添加静态属性

```javascript
@testable
class MyTestableClass {
  // ...
}

function testable(target) {
  target.isTestable = true;
}

MyTestableClass.isTestable // true
```

如果想要传递参数，可以在装饰器外层再封装一层函数

```javascript
function testable(isTestable) {
  return function(target) {
    target.isTestable = isTestable;
  }
}

@testable(true)
class MyTestableClass {}
MyTestableClass.isTestable // true

@testable(false)
class MyClass {}
MyClass.isTestable // false
```

8.2.2. 类属性的装饰

当对类属性进行装饰的时候，能够接受三个参数：

- 类的原型对象
- 需要装饰的属性名
- 装饰属性名的描述对象

首先定义一个 `readonly` 装饰器

```JavaScript
function readonly(target, name, descriptor){
  descriptor.writable = false; // 将可写属性设为false
  return descriptor;
}
```

使用 `readonly` 装饰类的 `name` 方法

```JavaScript
class Person {
  @readonly
  name() { return `${this.first} ${this.last}` }
}
```

相当于以下调用

```JavaScript
readonly(Person.prototype, 'name', descriptor);
```

如果一个方法有多个装饰器，就像洋葱一样，先从外到内进入，再由内到外执行

```JavaScript
function dec(id){
    console.log('evaluated', id);
    return (target, property, descriptor) =>console.log('executed', id);
}

class Example {
    @dec(1)
    @dec(2)
    method(){}
}
// evaluated 1
// evaluated 2
// executed 2
// executed 1
```

外层装饰器 `@dec(1)` 先进入，但是内层装饰器 `@dec(2)` 先执行

8.2.3. 注意

装饰器不能用于修饰函数，因为函数存在变量声明情况

```javascript
var counter = 0;

var add = function () {
  counter++;
};

@add
function foo() {
}
```

编译阶段，变成下面

```javascript
var counter;
var add;

@add
function foo() {
}

counter = 0;

add = function () {
  counter++;
};
```

意图是执行后 `counter` 等于 1，但是实际上结果是 `counter` 等于 0

8.3. 使用场景

基于 `Decorator` 强大的作用，我们能够完成各种场景的需求，下面简单列举几种：

使用 `react-redux` 的时候，如果写成下面这种形式，既不雅观也很麻烦

```javascript
class MyReactComponent extends React.Component {}

export default connect(mapStateToProps, mapDispatchToProps)(MyReactComponent);
```

通过装饰器就变得简洁多了

```JavaScript
@connect(mapStateToProps, mapDispatchToProps)
export default class MyReactComponent extends React.Component {}
```

将 `mixins` ，也可以写成装饰器，让使用更为简洁了

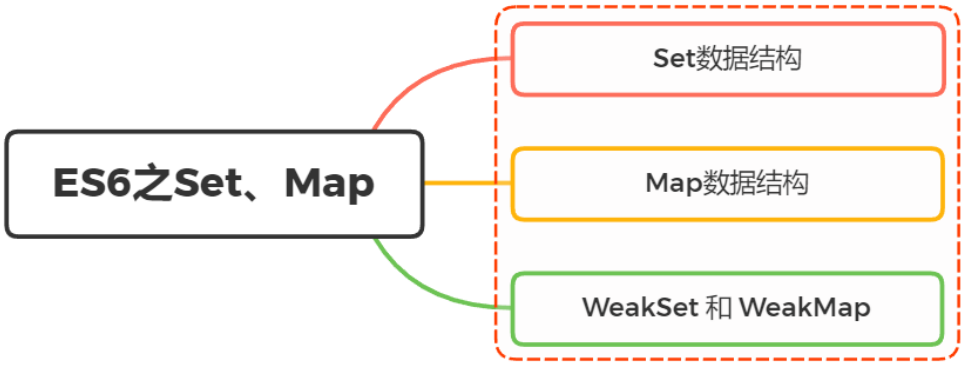
```JavaScript
function mixins(...list) {
  return function (target) {
    Object.assign(target.prototype, ...list);
  };
}

// 使用
const Foo = {
  foo() { console.log('foo') }
};

@mixins(Foo)
class MyClass {}

let obj = new MyClass();
obj.foo() // "foo"
```

9. 你是怎么理解ES6新增Set、Map两种数据结构的?



如果要用一句来描述，我们可以说

`Set` 是一种叫做集合的数据结构， `Map` 是一种叫做字典的数据结构

什么是集合？什么又是字典？

- 集合
  是由一堆无序的、相关联的，且不重复的内存结构【数学中称为元素】组成的组合
- 字典
  是一些元素的集合。每个元素有一个称作key 的域，不同元素的key 各不相同

区别?

- 共同点：集合、字典都可以存储不重复的值
- 不同点：集合是以[值，值]的形式存储元素，字典是以[键，值]的形式存储

9.1. Set

`Set` 是 `es6` 新增的数据结构，类似于数组，但是成员的值都是唯一的，没有重复的值，我们一般称为集合

`Set` 本身是一个构造函数，用来生成 Set 数据结构

JavaScript | 复制代码

```
const s = new Set();
```

9.1.1. 增删改查

`Set` 的实例关于增删改查的方法：

- add()
- delete()
- has()
- clear()

9.1.2. add()

添加某个值，返回 `Set` 结构本身

当添加实例中已经存在的元素， `set` 不会进行处理添加

JavaScript | 复制代码

```
s.add(1).add(2).add(2); // 2只被添加了一次
```

9.1.3. delete()

删除某个值，返回一个布尔值，表示删除是否成功

```JavaScript
s.delete(1)
```

9.1.4. has()

返回一个布尔值，判断该值是否为 `Set` 的成员

```JavaScript
s.has(2)
```

9.1.5. clear()

清除所有成员，没有返回值

```JavaScript
s.clear()
```

9.1.6. 遍历

`Set` 实例遍历的方法有如下：

关于遍历的方法，有如下：

- 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

`Set` 的遍历顺序就是插入顺序

`keys` 方法、 `values` 方法、 `entries` 方法返回的都是遍历器对象

```javascript
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"]
```

`forEach()` 用于对每个成员执行某种操作，没有返回值，键值、键名都相等，同样的 `forEach` 方法有第二个参数，用于绑定处理函数的 `this`

```javascript
let set = new Set([1, 4, 9]);
set.forEach((value, key) => console.log(key + ' : ' + value))
// 1 : 1
// 4 : 4
// 9 : 9
```

扩展运算符和 `Set` 结构相结合实现数组或字符串去重

```JavaScript
// 数组
let arr = [3, 5, 2, 2, 5, 5];
let unique = [...new Set(arr)]; // [3, 5, 2]

// 字符串
let str = "352255";
let unique = [...new Set(str)].join(""); // "352"
```

实现并集、交集、和差集

```JavaScript
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}

//  (a 相对于 b 的) 差集
let difference = new Set([...a].filter(x => !b.has(x)));
// Set {1}
```

9.2. Map

`Map` 类型是键值对的有序列表，而键和值都可以是任意类型

`Map` 本身是一个构造函数，用来生成 `Map` 数据结构

```JavaScript
const m = new Map()
```

9.2.1. 增删改查

`Map` 结构的实例针对增删改查有以下属性和操作方法：

- size 属性

- set()
- get()
- has()
- delete()
- clear()

9.2.2. size

`size` 属性返回 Map 结构的成员总数。

JavaScript | 复制代码

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

map.size // 2
```

9.2.3. set()

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

同时返回的是当前 `Map` 对象，可采用链式写法

JavaScript | 复制代码

```
const 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
m.set(1, 'a').set(2, 'b').set(3, 'c') // 链式操作
```

9.2.4. get()

`get` 方法读取 `key` 对应的键值，如果找不到 `key` ，返回 `undefined`

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

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

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

9.2.5. has()

`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

```javascript
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
```

9.2.6. delete()

`delete` 方法删除某个键，返回 `true` 。如果删除失败，返回 `false`

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

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

9.2.7. clear()

`clear` 方法清除所有成员，没有返回值

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

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

9.2.8. 遍历

`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

遍历顺序就是插入顺序

JavaScript | 复制代码

```javascript
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.forEach(function(value, key, map) {
  console.log("Key: %s, Value: %s", key, value);
});
```

9.3. WeakSet 和 WeakMap

9.3.1. WeakSet

创建 `WeakSet` 实例

```JavaScript
const ws = new WeakSet();
```

`WeakSet` 可以接受一个具有 `Iterable` 接口的对象作为参数

```JavaScript
const a = [[1, 2], [3, 4]];
const ws = new WeakSet(a);
// WeakSet {[1, 2], [3, 4]}
```

在 `API` 中 `WeakSet` 与 `Set` 有两个区别：

- 没有遍历操作的 `API`
- 没有 `size` 属性

`WeakSet` 只能成员只能是引用类型，而不能是其他类型的值

```JavaScript
let ws=new WeakSet();

// 成员不是引用类型
let weakSet=new WeakSet([2,3]);
console.log(weakSet) // 报错

// 成员为引用类型
let obj1={name:1}
let obj2={name:1}
let ws=new WeakSet([obj1,obj2]);
console.log(ws) //WeakSet {{…}, {…}}
```

`WeakSet` 里面的引用只要在外部消失，它在 `WeakSet` 里面的引用就会自动消失

9.3.2. WeakMap

`WeakMap` 结构与 `Map` 结构类似，也是用于生成键值对的集合

在 `API` 中 `WeakMap` 与 `Map` 有两个区别：

- 没有遍历操作的 `API`

- 没有 `clear` 清空方法

```JavaScript
// 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"
```

`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

```JavaScript
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` 的键名所指向的对象，一旦不再需要，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

举个场景例子：

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

```JavaScript
const wm = new WeakMap();

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

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

注意：`WeakMap` 弱引用的只是键名，而不是键值。键值依然是正常引用

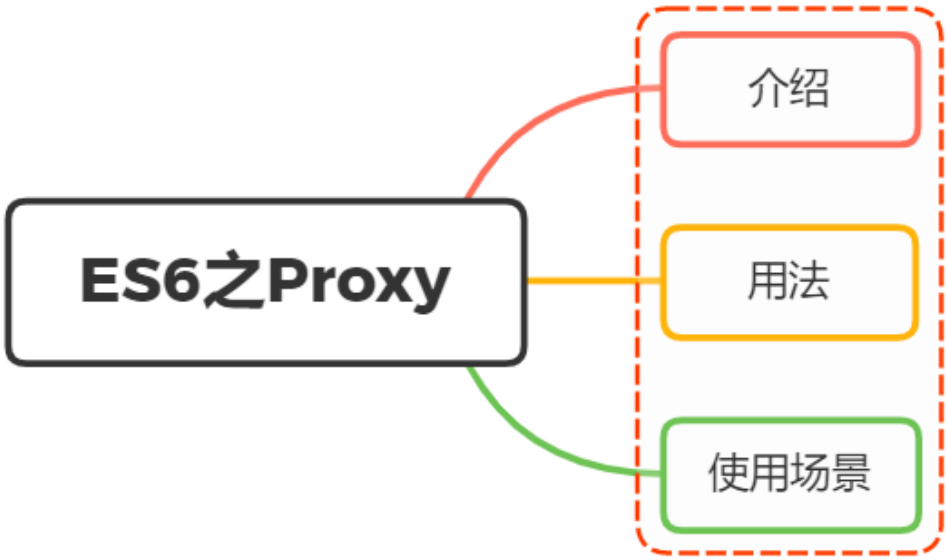
下面代码中，键值 `obj` 会在 `WeakMap` 产生新的引用，当你修改 `obj` 不会影响到内部

JavaScript | 复制代码

```
const wm = new WeakMap();
let key = {};
let obj = {foo: 1};

wm.set(key, obj);
obj = null;
wm.get(key)
// Object {foo: 1}
```

10. 你是怎么理解ES6中Proxy的？使用场景？



10.1. 介绍

定义： 用于定义基本操作的自定义行为

本质： 修改的是程序默认形为，就形同于在编程语言层面上做修改，属于元编程 `(meta programming)`

元编程（Metaprogramming，又译超编程，是指某类计算机程序的编写，这类计算机程序编写或者操纵其它程序（或者自身）作为它们的数据，或者在运行时完成部分本应在编译时完成的工作

一段代码来理解

```bash
#!/bin/bash
# metaprogram
echo '#!/bin/bash' >program
for ((I=1; I<=1024; I++)) do
    echo "echo $I" >>program
done
chmod +x program
```

这段程序每执行一次能帮我们生成一个名为 `program` 的文件，文件内容为1024行 `echo` ，如果我们手动来写1024行代码，效率显然低效

- 元编程优点：与手工编写全部代码相比，程序员可以获得更高的工作效率，或者给与程序更大的灵活度去处理新的情形而无需重新编译

`Proxy` 亦是如此，用于创建一个对象的代理，从而实现基本操作的拦截和自定义（如属性查找、赋值、枚举、函数调用等）

10.2. 用法

`Proxy` 为 构造函数，用来生成 `Proxy` 实例

```javascript
var proxy = new Proxy(target, handler)
```

10.2.1. 参数

`target` 表示所要拦截的目标对象（任何类型的对象，包括原生数组，函数，甚至另一个代理））

`handler` 通常以函数作为属性的对象，各属性中的函数分别定义了在执行各种操作时代理 `p` 的行为

10.2.2. handler解析

关于 `handler` 拦截属性，有如下：

- get(target,propKey,receiver)：拦截对象属性的读取
- set(target,propKey,value,receiver)：拦截对象属性的设置
- 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.keys(proxy)` 、 `for...in` 等循环，返回一个数组
- 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)` ，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
- apply(target, object, args)：拦截 Proxy 实例作为函数调用的操作
- construct(target, args)：拦截 Proxy 实例作为构造函数调用的操作

10.2.3. Reflect

若需要在 `Proxy` 内部调用对象的默认行为，建议使用 `Reflect` ，其是 `ES6` 中操作对象而提供的新 `API`

基本特点：

- 只要 `Proxy` 对象具有的代理方法， `Reflect` 对象全部具有，以静态方法的形式存在
- 修改某些 `Object` 方法的返回结果，让其变得更合理（定义不存在属性行为的时候不报错而是返回 `false` ）
- 让 `Object` 操作都变成函数行为

下面我们介绍 `proxy` 几种用法：

10.2.4. get()

`get` 接受三个参数，依次为目标对象、属性名和 `proxy` 实例本身，最后一个参数可选

JavaScript 复制代码

```
var person = {
  name: "张三"
};

var proxy = new Proxy(person, {
  get: function(target, propKey) {
    return Reflect.get(target,propKey)
  }
});

proxy.name // "张三"
```

`get` 能够对数组增删改查进行拦截，下面是试下你数组读取负数的索引

JavaScript 复制代码

```
function createArray(...elements) {
  let handler = {
    get(target, propKey, receiver) {
      let index = Number(propKey);
      if (index < 0) {
        propKey = String(target.length + index);
      }
      return Reflect.get(target, propKey, receiver);
    }
  };

  let target = [];
  target.push(...elements);
  return new Proxy(target, handler);
}

let arr = createArray('a', 'b', 'c');
arr[-1] // c
```

注意：如果一个属性不可配置（configurable）且不可写（writable），则 Proxy 不能修改该属性，否则会报错