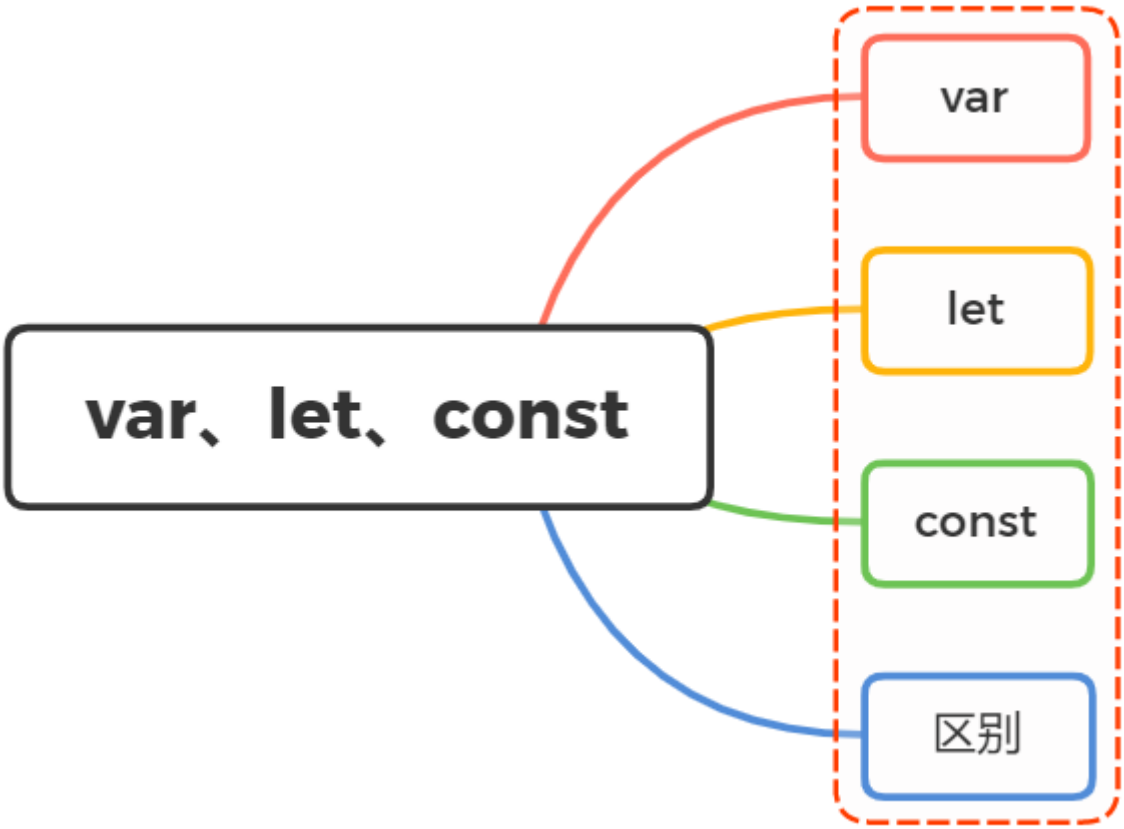
ES6面试真题（10题）

1. 说说var、let、const之间的区别



1.1. var

在ES5中，顶层对象的属性和全局变量是等价的，用 `var` 声明的变量既是全局变量，也是顶层变量

注意：顶层对象，在浏览器环境指的是 `window` 对象，在 `Node` 指的是 `global` 对象

```
var a = 10;
console.log(window.a) // 10
```

使用 `var` 声明的变量存在变量提升的情况

```
console.log(a) // undefined
var a = 20
```

在编译阶段，编译器会将其变成以下执行

```
var a
console.log(a)
a = 20
```

使用 `var` ，我们能够对一个变量进行多次声明，后面声明的变量会覆盖前面的变量声明

```
var a = 20
var a = 30
console.log(a) // 30
```

在函数中使用使用 `var` 声明变量时候，该变量是局部的

```
var a = 20
function change(){
    var a = 30
}
change()
console.log(a) // 20
```

而如果在函数内不使用 `var` ，该变量是全局的

```
var a = 20
function change(){
   a = 30
}
change()
console.log(a) // 30
```

1.2. let

`let` 是 `ES6` 新增的命令，用来声明变量

用法类似于 `var` ，但是所声明的变量，只在 `let` 命令所在的代码块内有效

```
{
    let a = 20
}
console.log(a) // ReferenceError: a is not defined.
```

不存在变量提升

```
console.log(a) // 报错ReferenceError
let a = 2
```

这表示在声明它之前，变量 `a` 是不存在的，这时如果用到它，就会抛出一个错误

只要块级作用域内存在 `let` 命令，这个区域就不再受外部影响

```
var a = 123
if (true) {
    a = 'abc' // ReferenceError
    let a;
}
```

使用 `let` 声明变量前，该变量都不可用，也就是大家常说的“暂时性死区”

最后， `let` 不允许在相同作用域中重复声明

```
let a = 20
let a = 30
// Uncaught SyntaxError: Identifier 'a' has already been declared
```

注意的是相同作用域，下面这种情况是不会报错的

```
let a = 20
{
    let a = 30
}
```

因此，我们不能在函数内部重新声明参数

```
function func(arg) {
  let arg;
}
func()
// Uncaught SyntaxError: Identifier 'arg' has already been declared
```

1.3. const

`const` 声明一个只读的常量，一旦声明，常量的值就不能改变

```
const a = 1
a = 3
// TypeError: Assignment to constant variable.
```

这意味着， `const` 一旦声明变量，就必须立即初始化，不能留到以后赋值

```
const a;
// SyntaxError: Missing initializer in const declaration
```

如果之前用 `var` 或 `let` 声明过变量，再用 `const` 声明同样会报错

```
var a = 20
let b = 20
const a = 30
const b = 30
// 都会报错
```

`const` 实际上保证的并不是变量的值不得改动，而是变量指向的那个内存地址所保存的数据不得改动

对于简单类型的数据，值就保存在变量指向的那个内存地址，因此等同于常量

对于复杂类型的数据，变量指向的内存地址，保存的只是一个指向实际数据的指针， `const` 只能保证这个指针是固定的，并不能确保改变量的结构不变

```
const foo = {};
// 为 foo 添加一个属性，可以成功
foo.prop = 123;
foo.prop // 123
// 将 foo 指向另一个对象，就会报错
foo = {}; // TypeError: "foo" is read-only
```

其它情况， `const` 与 `let` 一致

1.4. 区别

`var` 、 `let` 、 `const` 三者区别可以围绕下面五点展开：

- 变量提升
- 暂时性死区
- 块级作用域
- 重复声明
- 修改声明的变量
- 使用

1.4.1. 变量提升

`var` 声明的变量存在变量提升，即变量可以在声明之前调用，值为 `undefined`

`let` 和 `const` 不存在变量提升，即它们所声明的变量一定要在声明后使用，否则报错

```
// var
console.log(a)  // undefined
var a = 10
// let
console.log(b)  // Cannot access 'b' before initialization
let b = 10
// const
console.log(c)  // Cannot access 'c' before initialization
const c = 10
```

1.4.2. 暂时性死区

`var` 不存在暂时性死区

`let` 和 `const` 存在暂时性死区，只有等到声明变量的那一行代码出现，才可以获取和使用该变量

```
// var
console.log(a)  // undefined
var a = 10
// let
console.log(b)  // Cannot access 'b' before initialization
let b = 10
// const
console.log(c)  // Cannot access 'c' before initialization
const c = 10
```

1.4.3. 块级作用域

`var` 不存在块级作用域

`let` 和 `const` 存在块级作用域

```
// var
{
    var a = 20
}
console.log(a)  // 20
// let
{
    let b = 20
}
console.log(b)  // Uncaught ReferenceError: b is not defined
// const
{
    const c = 20
}
console.log(c)  // Uncaught ReferenceError: c is not defined
```

1.4.4. 重复声明

`var` 允许重复声明变量

`let` 和 `const` 在同一作用域不允许重复声明变量

```
// var
var a = 10
var a = 20 // 20
// let
let b = 10
let b = 20 // Identifier 'b' has already been declared
// const
const c = 10
const c = 20 // Identifier 'c' has already been declared
```

1.4.5. 修改声明的变量

`var` 和 `let` 可以

`const` 声明一个只读的常量。一旦声明，常量的值就不能改变

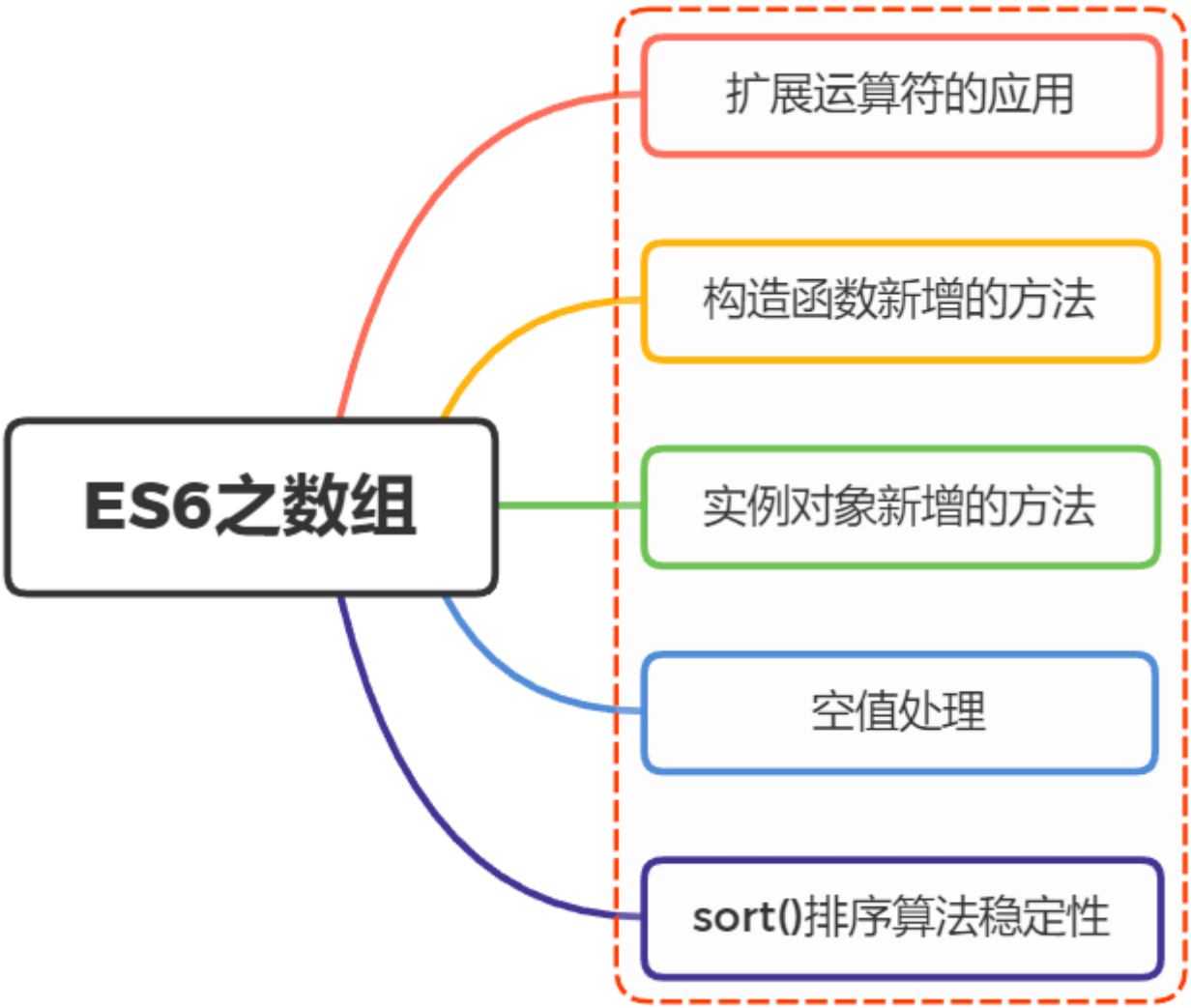
```
// var
var a = 10
a = 20
```

```
4 console.log(a)  // 20
5 //let
6 let b = 10
7 b = 20
8 console.log(b)  // 20
9 // const
10 const c = 10
11 c = 20
12 console.log(c) // Uncaught TypeError: Assignment to constant variable
```

1.4.6. 使用

能用 `const` 的情况尽量使用 `const` ，其他情况下大多数使用 `let` ，避免使用 `var`

2. ES6中数组新增了哪些扩展?



2.1. 扩展运算符的应用

ES6通过扩展元素符 `...` ，好比 `rest` 参数的逆运算，将一个数组转为用逗号分隔的参数序列

```
console.log(...[1, 2, 3])
// 1 2 3
console.log(1, ...[2, 3, 4], 5)
// 1 2 3 4 5
[...document.querySelectorAll('div')]
// [<div>, <div>, <div>]
```

主要用于函数调用的时候，将一个数组变为参数序列

```
function push(array, ...items) {
  array.push(...items);
}
function add(x, y) {
  return x + y;
}
const numbers = [4, 38];
add(...numbers) // 42
```

可以将某些数据结构转为数组

```
[...document.querySelectorAll('div')]
```

能够更简单实现数组复制

```
const a1 = [1, 2];
const [...a2] = a1;
// [1,2]
```

数组的合并也更为简洁了

```
const arr1 = ['a', 'b'];
const arr2 = ['c'];
const arr3 = ['d', 'e'];
[...arr1, ...arr2, ...arr3]
// [ 'a', 'b', 'c', 'd', 'e' ]
```

注意：通过扩展运算符实现的是浅拷贝，修改了引用指向的值，会同步反映到新数组

下面看个例子就清楚多了

```
const arr1 = ['a', 'b',[1,2]];
const arr2 = ['c'];
const arr3  = [...arr1,...arr2]
arr[1][0] = 9999 // 修改arr1里面数组成员值
console.log(arr[3]) // 影响到arr3,['a','b',[9999,2],'c']
```

扩展运算符可以与解构赋值结合起来，用于生成数组

```
const [first, ...rest] = [1, 2, 3, 4, 5];
first // 1
rest  // [2, 3, 4, 5]
const [first, ...rest] = [];
first // undefined
rest  // []
const [first, ...rest] = ["foo"];
first  // "foo"
rest   // []
```

如果将扩展运算符用于数组赋值，只能放在参数的最后一位，否则会报错

```
const [...butLast, last] = [1, 2, 3, 4, 5];
// 报错
const [first, ...middle, last] = [1, 2, 3, 4, 5];
// 报错
```

可以将字符串转为真正的数组

```
[...'hello']
// [ "h", "e", "l", "l", "o" ]
```

定义了遍历器（Iterator）接口的对象，都可以用扩展运算符转为真正的数组

```
let nodeList = document.querySelectorAll('div');
let array = [...nodeList];
let map = new Map([
  [1, 'one'],
```

```
  [2, 'two'],
  [3, 'three'],
]);
let arr = [...map.keys()]; // [1, 2, 3]
```

如果对没有 Iterator 接口的对象，使用扩展运算符，将会报错

```
const obj = {a: 1, b: 2};
let arr = [...obj]; // TypeError: Cannot spread non-iterable object
```

2.2. 构造函数新增的方法

关于构造函数，数组新增的方法有如下：

- Array.from()
- Array.of()

2.2.1. Array.from()

将两类对象转为真正的数组：类似数组的对象和可遍历（iterable）的对象（包括 ES6 新增的数据结构 Set 和 Map）

```
let arrayLike = {
    '0': 'a',
    '1': 'b',
    '2': 'c',
    length: 3
};
let arr2 = Array.from(arrayLike); // ['a', 'b', 'c']
```

还可以接受第二个参数，用来对每个元素进行处理，将处理后的值放入返回的数组

```
Array.from([1, 2, 3], (x) => x * x)
// [1, 4, 9]
```

2.2.2. Array.of()

用于将一组值，转换为数组

```
Array.of(3, 11, 8) // [3,11,8]
```

没有参数的时候，返回一个空数组

当参数只有一个的时候，实际上是指定数组的长度

参数个数不少于 2 个时，`Array()` 才会返回由参数组成的新数组

```
Array() // []
Array(3) // [, , ,]
Array(3, 11, 8) // [3, 11, 8]
```

2.2.3. 实例对象新增的方法

关于数组实例对象新增的方法有如下：

- copyWithin()
- find()、findIndex()
- fill()
- entries()，keys()，values()
- includes()
- flat()，flatMap()

2.2.4. copyWithin()

将指定位置的成员复制到其他位置（会覆盖原有成员），然后返回当前数组

参数如下：

- target（必需）：从该位置开始替换数据。如果为负值，表示倒数。
- start（可选）：从该位置开始读取数据，默认为 0。如果为负值，表示从末尾开始计算。
- end（可选）：到该位置前停止读取数据，默认等于数组长度。如果为负值，表示从末尾开始计算。

```
[1, 2, 3, 4, 5].copyWithin(0, 3) // 将从 3 号位直到数组结束的成员（4 和 5），复制到从 0 号位开始的位置，结果覆盖了原来的 1 和 2
// [4, 5, 3, 4, 5]
```

2.2.5. find()、findIndex()

`find()` 用于找出第一个符合条件的数组成员

参数是一个回调函数，接受三个参数依次为当前的值、当前的位置和原数组

```
[1, 5, 10, 15].find(function(value, index, arr) {
  return value > 9;
}) // 10
```

`findIndex` 返回第一个符合条件的数组成员的位置，如果所有成员都不符合条件，则返回 `-1`

```
[1, 5, 10, 15].findIndex(function(value, index, arr) {
  return value > 9;
}) // 2
```

这两个方法都可以接受第二个参数，用来绑定回调函数的 `this` 对象。

```
function f(v){
  return v > this.age;
}
let person = {name: 'John', age: 20};
[10, 12, 26, 15].find(f, person);    // 26
```

2.2.6. fill()

使用给定值，填充一个数组

```
['a', 'b', 'c'].fill(7)
// [7, 7, 7]
new Array(3).fill(7)
// [7, 7, 7]
```

还可以接受第二个和第三个参数，用于指定填充的起始位置和结束位置

```
['a', 'b', 'c'].fill(7, 1, 2)
// ['a', 7, 'c']
```

注意，如果填充的类型为对象，则是浅拷贝

2.2.7. entries()，keys()，values()

`keys()` 是对键名的遍历、 `values()` 是对键值的遍历， `entries()` 是对键值对的遍历

```
or (let index of ['a', 'b'].keys()) {
  console.log(index);
}
// 0
// 1
for (let elem of ['a', 'b'].values()) {
  console.log(elem);
}
// 'a'
// 'b'
for (let [index, elem] of ['a', 'b'].entries()) {
  console.log(index, elem);
}
// 0 "a"
```

2.2.8. includes()

用于判断数组是否包含给定的值

```
[1, 2, 3].includes(2)     // true
[1, 2, 3].includes(4)     // false
[1, 2, NaN].includes(NaN) // true
```

方法的第二个参数表示搜索的起始位置，默认为 `0`

参数为负数则表示倒数的位置

```
[1, 2, 3].includes(3, 3);  // false
[1, 2, 3].includes(3, -1); // true
```

2.2.9. flat()，flatMap()

将数组扁平化处理，返回一个新数组，对原数据没有影响

```
[1, 2, [3, 4]].flat()
// [1, 2, 3, 4]
```

`flat()` 默认只会“拉平”一层，如果想要“拉平”多层的嵌套数组，可以将 `flat()` 方法的参数写成一个整数，表示想要拉平的层数，默认为1

```
[1, 2, [3, [4, 5]]].flat()
// [1, 2, 3, [4, 5]]
[1, 2, [3, [4, 5]]].flat(2)
// [1, 2, 3, 4, 5]
```

`flatMap()` 方法对原数组的每个成员执行一个函数相当于执行 `Array.prototype.map()` ，然后对返回值组成的数组执行 `flat()` 方法。该方法返回一个新数组，不改变原数组

```
// 相当于 [[2, 4], [3, 6], [4, 8]].flat()
[2, 3, 4].flatMap((x) => [x, x * 2])
// [2, 4, 3, 6, 4, 8]
```

`flatMap()` 方法还可以有第二个参数，用来绑定遍历函数里面的 `this`

2.2.10. 数组的空位

数组的空位指，数组的某一个位置没有任何值

ES6 则是明确将空位转为 `undefined` ，包括 `Array.from` 、扩展运算符、 `copyWithin()` 、 `fill()` 、 `entries()` 、 `keys()` 、 `values()` 、 `find()` 和 `findIndex()`

建议大家在日常书写中，避免出现空位

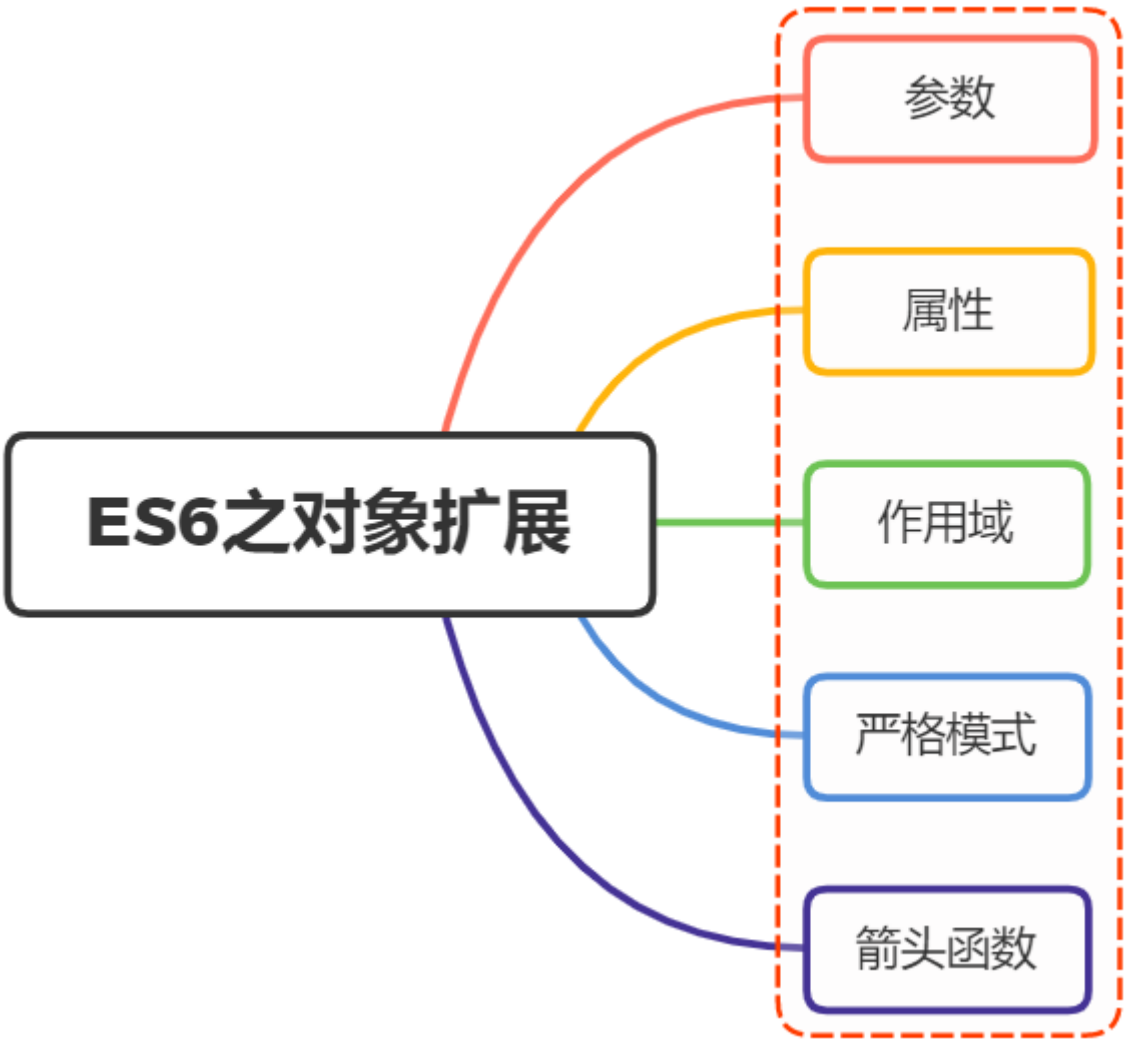
2.2.11. 排序稳定性

将 `sort()` 默认设置为稳定的排序算法

```
const arr = [
  'peach',
  'straw',
  'apple',
  'spork'
];
const stableSorting = (s1, s2) => {
  if (s1[0] < s2[0]) return -1;
  return 1;
};
arr.sort(stableSorting)
// ["apple", "peach", "straw", "spork"]
```

排序结果中，`straw`在`spork`的前面，跟原始顺序一致

3. 函数新增了哪些扩展?



3.1. 参数

`ES6`允许为函数的参数设置默认值

```
1 function log(x, y = 'World') {
2   console.log(x, y);
3 }
4 console.log('Hello') // Hello World
5 console.log('Hello', 'China') // Hello China
6 console.log('Hello', '') // Hello
```

函数的形参是默认声明的，不能使用`let`或`const`再次声明

```
function foo(x = 5) {
    let x = 1; // error
    const x = 2; // error
}
```

参数默认值可以与解构赋值的默认值结合起来使用

```
function foo({x, y = 5}) {
  console.log(x, y);
}
foo({}) // undefined 5
foo({x: 1}) // 1 5
foo({x: 1, y: 2}) // 1 2
foo() // TypeError: Cannot read property 'x' of undefined
```

上面的 `foo` 函数，当参数为对象的时候才能进行解构，如果没有提供参数的时候，变量 `x` 和 `y` 就不会生成，从而报错，这里设置默认值避免

```
function foo({x, y = 5} = {}) {
  console.log(x, y);
}
foo() // undefined 5
```

参数默认值应该是函数的尾参数，如果不是非尾部的参数设置默认值，实际上这个参数是没发省略的

```
function f(x = 1, y) {
  return [x, y];
}
f() // [1, undefined]
f(2) // [2, undefined]
f(, 1) // 报错
f(undefined, 1) // [1, 1]
```

3.2. 属性

3.2.1. 函数的length属性

`length` 将返回没有指定默认值的参数个数

```
```

```
(function (a) {}).length // 1
(function (a = 5) {}).length // 0
(function (a, b, c = 5) {}).length // 2
```

`rest` 参数也不会计入 `length` 属性

```
(function(...args) {}).length // 0
```

如果设置了默认值的参数不是尾参数，那么 `length` 属性也不再计入后面的参数了

```
(function (a = 0, b, c) {}).length // 0
(function (a, b = 1, c) {}).length // 1
```

3.2.2. name属性

返回该函数的函数名

```
var f = function () {};
// ES5
f.name // ""
// ES6
f.name // "f"
```

如果将一个具名函数赋值给一个变量，则 `name` 属性都返回这个具名函数原本的名字

```
const bar = function baz() {};
bar.name // "baz"
```

`Function` 构造函数返回的函数实例， `name` 属性的值为 `anonymous`

```
(new Function).name // "anonymous"
```

`bind` 返回的函数， `name` 属性值会加上 `bound` 前缀

```
function foo() {};
```

```
foo.bind({}).name // "bound foo"
(function(){}).bind({}).name // "bound "
```

3.3. 作用域

一旦设置了参数的默认值，函数进行声明初始化时，参数会形成一个单独的作用域

等到初始化结束，这个作用域就会消失。这种语法行为，在不设置参数默认值时，是不会出现的

下面例子中，`y=x` 会形成一个单独作用域，`x` 没有被定义，所以指向全局变量 `x`

```
let x = 1;
function f(y = x) {
  // 等同于 let y = x

  let x = 2;
  console.log(y);
}
f() // 1
```

3.4. 严格模式

只要函数参数使用了默认值、解构赋值、或者扩展运算符，那么函数内部就不能显式设定为严格模式，否则会报错

```
// 报错
function doSomething(a, b = a) {
  'use strict';
  // code
}
// 报错
const doSomething = function ({a, b}) {
  'use strict';
  // code
};
// 报错
const doSomething = (...a) => {
  'use strict';
  // code
};
const obj = {
  // 报错
  doSomething({a, b}) {
    'use strict';
```

```
    // code
  }
};
```

3.5. 箭头函数

使用“箭头”（`=>`）定义函数

```
var f = v => v;
// 等同于
var f = function (v) {
  return v;
};
```

如果箭头函数不需要参数或需要多个参数，就使用一个圆括号代表参数部分

```
var f = () => 5;
// 等同于
var f = function () { return 5 };
var sum = (num1, num2) => num1 + num2;
// 等同于
var sum = function(num1, num2) {
  return num1 + num2;
};
```

如果箭头函数的代码块部分多于一条语句，就要使用大括号将它们括起来，并且使用`return`语句返回

```
var sum = (num1, num2) => { return num1 + num2; }
```

如果返回对象，需要加括号将对象包裹

```
let getTempItem = id => ({ id: id, name: "Temp" });
```

注意点：

- 函数体内的`this`对象，就是定义时所在的对象，而不是使用时所在的对象

- 不可以当作构造函数，也就是说，不可以使用 `new` 命令，否则会抛出一个错误
- 不可以使用 `arguments` 对象，该对象在函数体内不存在。如果要用，可以用 `rest` 参数代替
- 不可以使用 `yield` 命令，因此箭头函数不能用作 Generator 函数

4. 对象新增了哪些扩展?

4.1. 属性的简写

ES6中，当对象键名与对应值名相等的时候，可以进行简写

```
const baz = {foo:foo}
// 等同于
const baz = {foo}
```

方法也能够进行简写

```
const o = {
  method() {
    return "Hello!";
  }
};
// 等同于
const o = {
  method: function() {
    return "Hello!";
  }
}
```

在函数内作为返回值，也会变得方便很多

```
function getPoint() {
  const x = 1;
  const y = 10;
  return {x, y};
}
getPoint()
// {x:1, y:10}
```

注意：简写的对象方法不能用作构造函数，否则会报错

```
const obj = {
  f() {
    this.foo = 'bar';
  }
};
new obj.f() // 报错
```

4.2. 属性名表达式

ES6 允许字面量定义对象时，将表达式放在括号内

```
let lastWord = 'last word';
const a = {
  'first word': 'hello',
```

```
  [lastWord]: 'world'
};
a['first word'] // "hello"
a[lastWord] // "world"
a['last word'] // "world"
```

表达式还可以用于定义方法名

```
let obj = {
  ['h' + 'ello']() {
    return 'hi';
  }
};
obj.hello() // hi
```

注意，属性名表达式与简洁表示法，不能同时使用，会报错

```
// 报错
const foo = 'bar';
const bar = 'abc';
const baz = { [foo] };
// 正确
const foo = 'bar';
const baz = { [foo]: 'abc'};
```

注意，属性名表达式如果是一个对象，默认情况下会自动将对象转为字符串`[object Object]`

```
const keyA = {a: 1};
const keyB = {b: 2};
const myObject = {
  [keyA]: 'valueA',
  [keyB]: 'valueB'
};
myObject // Object {[object Object]: "valueB"}
```

4.3. super关键字

`this`关键字总是指向函数所在的当前对象，ES6 又新增了另一个类似的关键字`super`，指向当前对象的原型对象

```
const proto = {
  foo: 'hello'
};
const obj = {
  foo: 'world',
  find() {
    return super.foo;
  }
};
Object.setPrototypeOf(obj, proto); // 为obj设置原型对象
obj.find() // "hello"
```

4.4. 扩展运算符的应用

在解构赋值中，未被读取的可遍历的属性，分配到指定的对象上面

```
let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };
x // 1
y // 2
z // { a: 3, b: 4 }
```

注意：解构赋值必须是最后一个参数，否则会报错

解构赋值是浅拷贝

```
let obj = { a: { b: 1 } };
let { ...x } = obj;
obj.a.b = 2; // 修改obj里面a属性中键值
x.a.b // 2，影响到了结构出来x的值
```

对象的扩展运算符等同于使用 `Object.assign()` 方法

4.5. 属性的遍历

ES6 一共有 5 种方法可以遍历对象的属性。

- for...in：循环遍历对象自身的和继承的可枚举属性（不含 Symbol 属性）
- Object.keys(obj)：返回一个数组，包括对象自身的（不含继承的）所有可枚举属性（不含 Symbol 属性）的键名
- Object.getOwnPropertyNames(obj)：回一个数组，包含对象自身的所有属性（不含 Symbol 属性，但是包括不可枚举属性）的键名

- Object.getOwnPropertySymbols(obj)：返回一个数组，包含对象自身的所有 Symbol 属性的键名
- Reflect.ownKeys(obj)：返回一个数组，包含对象自身的（不含继承的）所有键名，不管键名是 Symbol 或字符串，也不管是否可枚举

上述遍历，都遵守同样的属性遍历的次序规则：

- 首先遍历所有数值键，按照数值升序排列
- 其次遍历所有字符串键，按照加入时间升序排列
- 最后遍历所有 Symbol 键，按照加入时间升序排

```
Reflect.ownKeys({ [Symbol()]:0, b:0, 10:0, 2:0, a:0 })
// ['2', '10', 'b', 'a', Symbol()]
```

4.6. 对象新增的方法

关于对象新增的方法，分别有以下：

- Object.is()
- Object.assign()
- Object.getOwnPropertyDescriptors()
- Object.setPrototypeOf()，Object.getPrototypeOf()
- Object.keys()，Object.values()，Object.entries()
- Object.fromEntries()

4.6.1. Object.is()

严格判断两个值是否相等，与严格比较运算符（===）的行为基本一致，不同之处只有两个：一是 `+0` 不等于 `-0` ，二是 `NaN` 等于自身

```
+0 === -0 //true
NaN === NaN // false
Object.is(+0, -0) // false
Object.is(NaN, NaN) // true
```

4.6.2. Object.assign()

`Object.assign()` 方法用于对象的合并，将源对象 `source` 的所有可枚举属性，复制到目标对象 `target`

`Object.assign()` 方法的第一个参数是目标对象，后面的参数都是源对象

```
const target = { a: 1, b: 1 };
const source1 = { b: 2, c: 2 };
const source2 = { c: 3 };
Object.assign(target, source1, source2);
target // {a:1, b:2, c:3}
```

注意： `Object.assign()` 方法是浅拷贝，遇到同名属性会进行替换

4.6.3. Object.getOwnPropertyDescriptors()

返回指定对象所有自身属性（非继承属性）的描述对象

```
const obj = {
  foo: 123,
  get bar() { return 'abc' }
};
Object.getOwnPropertyDescriptors(obj)
// { foo:
//    { value: 123,
//      writable: true,
//      enumerable: true,
//      configurable: true },
//   bar:
//    { get: [Function: get bar],
//      set: undefined,
//      enumerable: true,
//      configurable: true } }
```

4.6.4. Object.setPrototypeOf()

`Object.setPrototypeOf` 方法用来设置一个对象的原型对象

```
Object.setPrototypeOf(object, prototype)
// 用法
const o = Object.setPrototypeOf({}, null);
```

4.6.5. Object.getPrototypeOf()

用于读取一个对象的原型对象

```
Object.getPrototypeOf(obj);
```

4.6.6. Object.keys()

返回自身的（不含继承的）所有可遍历（enumerable）属性的键名的数组

```
var obj = { foo: 'bar', baz: 42 };
Object.keys(obj)
// ["foo", "baz"]
```

4.6.7. Object.values()

返回自身的（不含继承的）所有可遍历（enumerable）属性的键对应值的数组

```
const obj = { foo: 'bar', baz: 42 };
Object.values(obj)
// ["bar", 42]
```

4.6.8. Object.entries()

返回一个对象自身的（不含继承的）所有可遍历（enumerable）属性的键值对的数组

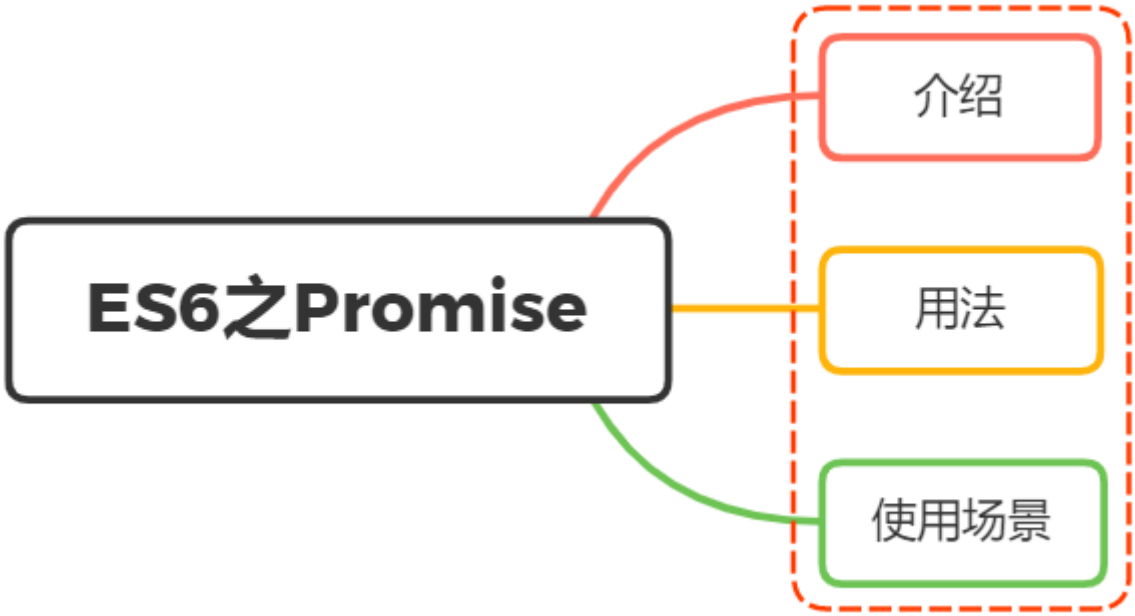
```
const obj = { foo: 'bar', baz: 42 };
Object.entries(obj)
// [ ["foo", "bar"], ["baz", 42] ]
```

4.6.9. Object.fromEntries()

用于将一个键值对数组转为对象

```
Object.fromEntries([
  ['foo', 'bar'],
  ['baz', 42]
])
// { foo: "bar", baz: 42 }
```

5. 你是怎么理解ES6中 Promise的？使用场景？



5.1. 介绍

`Promise`，译为承诺，是异步编程的一种解决方案，比传统的解决方案（回调函数）更加合理和更加强大

在以往我们如果处理多层异步操作，我们往往会像下面那样编写我们的代码

```
doSomething(function(result) {
  doSomethingElse(result, function(newResult) {
    doThirdThing(newResult, function(finalResult) {
      console.log('得到最终结果: ' + finalResult);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

阅读上面代码，是不是很难受，上述形成了经典的回调地狱

现在通过 `Promise` 的改写上面的代码

```
doSomething().then(function(result) {
  return doSomethingElse(result);
})
.then(function(newResult) {
  return doThirdThing(newResult);
})
.then(function(finalResult) {
  console.log('得到最终结果: ' + finalResult);
})
.catch(failureCallback);
```

瞬间感受到 `promise` 解决异步操作的优点：

- 链式操作减低了编码难度
- 代码可读性明显增强

下面我们正式来认识 `promise` ：

5.1.1. 状态

`promise` 对象仅有三种状态

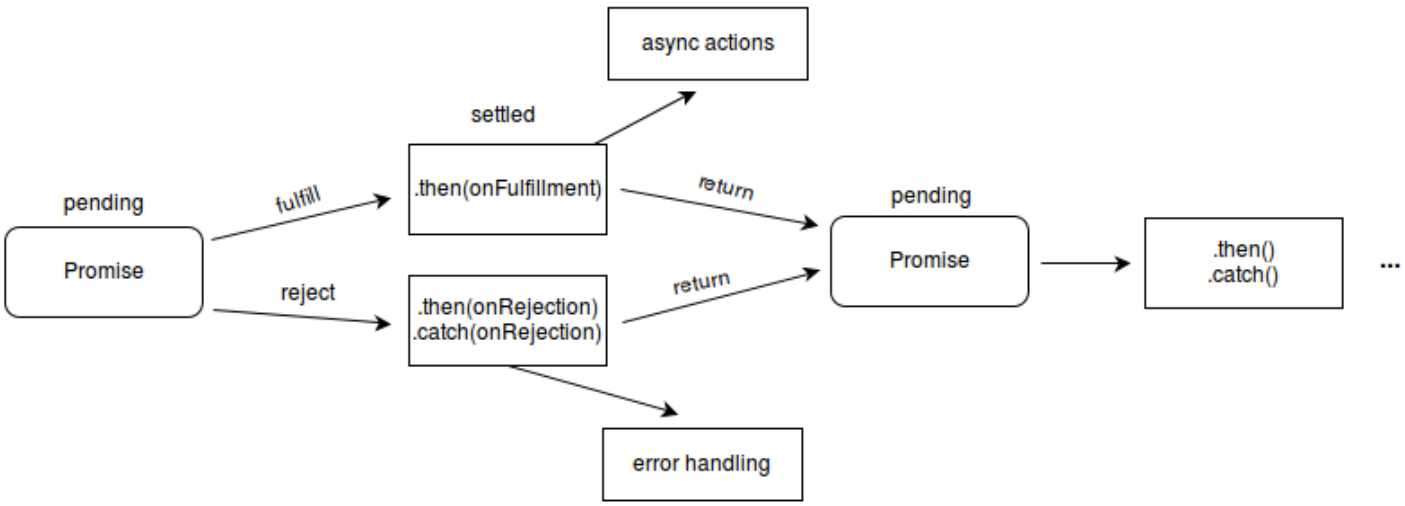
- `pending` （进行中）
- `fulfilled` （已成功）
- `rejected` （已失败）

5.1.2. 特点

- 对象的状态不受外界影响，只有异步操作的结果，可以决定当前是哪一种状态
- 一旦状态改变（从 `pending` 变为 `fulfilled` 和从 `pending` 变为 `rejected` ），就不会再变，任何时候都可以得到这个结果

5.1.3. 流程

认真阅读下图，我们能够轻松了解 `promise` 整个流程



5.2. 用法

`Promise` 对象是一个构造函数，用来生成 `Promise` 实例

```
1 const promise = new Promise(function(resolve, reject) {});
```

`Promise` 构造函数接受一个函数作为参数，该函数的两个参数分别是 `resolve` 和 `reject`

- `resolve` 函数的作用是，将 `Promise` 对象的状态从“未完成”变为“成功”
- `reject` 函数的作用是，将 `Promise` 对象的状态从“未完成”变为“失败”

5.2.1. 实例方法

`Promise` 构建出来的实例存在以下方法：

- then()
- catch()
- finally()

5.2.1.1. then()

`then` 是实例状态发生改变时的回调函数，第一个参数是 `resolved` 状态的回调函数，第二个参数是 `rejected` 状态的回调函数

`then` 方法返回的是一个新的 `Promise` 实例，也就是 `promise` 能链式书写的原因

```
getJSON("/posts.json").then(function(json) {
  return json.post;
}).then(function(post) {
  // ...
});
```

5.2.1.2. catch

`catch()` 方法是 `.then(null, rejection)` 或 `.then(undefined, rejection)` 的别名，用于指定发生错误时的回调函数

```
getJSON('/posts.json').then(function(posts) {
  // ...
}).catch(function(error) {
  // 处理 getJSON 和 前一个回调函数运行时发生的错误
  console.log('发生错误! ', error);
});
```

`Promise` 对象的错误具有“冒泡”性质，会一直向后传递，直到被捕获为止

```
getJSON('/post/1.json').then(function(post) {
  return getJSON(post.commentURL);
}).then(function(comments) {
```

```
  // some code
}).catch(function(error) {
  // 处理前面三个Promise产生的错误
});
```

一般来说，使用 `catch` 方法代替 `then()` 第二个参数

`Promise` 对象抛出的错误不会传递到外层代码，即不会有任何反应

```
const someAsyncThing = function() {
  return new Promise(function(resolve, reject) {
    // 下面一行会报错，因为x没有声明
    resolve(x + 2);
  });
};
```

浏览器运行到这一行，会打印出错误提示 `ReferenceError: x is not defined` ，但是不会退出进程

`catch()` 方法之中，还能再抛出错误，通过后面 `catch` 方法捕获到

5.2.1.3. finally()

`finally()` 方法用于指定不管 Promise 对象最后状态如何，都会执行的操作

```
promise
.then(result => {···})
.catch(error => {···})
.finally(() => {···});
```

5.2.2. 构造函数方法

`Promise` 构造函数存在以下方法：

- all()
- race()
- allSettled()
- resolve()
- reject()
- try()

5.2.3. all()

`Promise.all()` 方法用于将多个 `Promise` 实例，包装成一个新的 `Promise` 实例

```
const p = Promise.all([p1, p2, p3]);
```

接受一个数组（迭代对象）作为参数，数组成员都应为 `Promise` 实例

实例 `p` 的状态由 `p1` 、 `p2` 、 `p3` 决定，分为两种：

- 只有 `p1` 、 `p2` 、 `p3` 的状态都变成 `fulfilled` ， `p` 的状态才会变成 `fulfilled` ，此时 `p1` 、 `p2` 、 `p3` 的返回值组成一个数组，传递给 `p` 的回调函数
- 只要 `p1` 、 `p2` 、 `p3` 之中有一个被 `rejected` ， `p` 的状态就变成 `rejected` ，此时第一个被 `reject` 的实例的返回值，会传递给 `p` 的回调函数

注意，如果作为参数的 `Promise` 实例，自己定义了 `catch` 方法，那么它一旦被 `rejected` ，并不会触发 `Promise.all()` 的 `catch` 方法

```
const p1 = new Promise((resolve, reject) => {
  resolve('hello');
})
.then(result => result)
.catch(e => e);
const p2 = new Promise((resolve, reject) => {
  throw new Error('报错了');
})
.then(result => result)
.catch(e => e);
Promise.all([p1, p2])
.then(result => console.log(result))
.catch(e => console.log(e));
// ["hello", Error: 报错了]
```

如果 `p2` 没有自己的 `catch` 方法，就会调用 `Promise.all()` 的 `catch` 方法

```
const p1 = new Promise((resolve, reject) => {
  resolve('hello');
})
.then(result => result);
const p2 = new Promise((resolve, reject) => {
  throw new Error('报错了');
})
```

```
.then(result => result);
Promise.all([p1, p2])
.then(result => console.log(result))
.catch(e => console.log(e));
// Error: 报错了
```

5.2.4. race()

`Promise.race()` 方法同样是将多个 Promise 实例，包装成一个新的 Promise 实例

```
const p = Promise.race([p1, p2, p3]);
```

只要 `p1` 、 `p2` 、 `p3` 之中有一个实例率先改变状态， `p` 的状态就跟着改变

率先改变的 Promise 实例的返回值则传递给 `p` 的回调函数

```
const p = Promise.race([
  fetch('/resource-that-may-take-a-while'),
  new Promise(function (resolve, reject) {
    setTimeout(() => reject(new Error('request timeout')), 5000)
  })
]);
p
.then(console.log)
.catch(console.error);
```

5.2.5. allSettled()

`Promise.allSettled()` 方法接受一组 Promise 实例作为参数，包装成一个新的 Promise 实例

只有等到所有这些参数实例都返回结果，不管是 `fulfilled` 还是 `rejected` ，包装实例才会结束

```
const promises = [
  fetch('/api-1'),
  fetch('/api-2'),
  fetch('/api-3'),
];
await Promise.allSettled(promises);
removeLoadingIndicator();
```

5.2.5.1. resolve()

将现有对象转为 `Promise` 对象

```
Promise.resolve('foo')
// 等价于
new Promise(resolve => resolve('foo'))
```

参数可以分成四种情况，分别如下：

- 参数是一个 Promise 实例， `promise.resolve` 将不做任何修改、原封不动地返回这个实例
- 参数是一个 `thenable` 对象， `promise.resolve` 会将这个对象转为 `Promise` 对象，然后就立即执行 `thenable` 对象的 `then()` 方法
- 参数不是具有 `then()` 方法的对象，或根本就不是对象， `Promise.resolve()` 会返回一个新的 Promise 对象，状态为 `resolved`
- 没有参数时，直接返回一个 `resolved` 状态的 Promise 对象

5.2.5.2. reject()

`Promise.reject(reason)` 方法也会返回一个新的 Promise 实例，该实例的状态为 `rejected`

```
const p = Promise.reject('出错了');
// 等同于
const p = new Promise((resolve, reject) => reject('出错了'))
p.then(null, function (s) {
  console.log(s)
});
// 出错了
```

`Promise.reject()` 方法的参数，会原封不动地变成后续方法的参数

```
Promise.reject('出错了')
.catch(e => {
  console.log(e === '出错了')
})
// true
```

5.3. 使用场景

将图片的加载写成一个 `Promise` ，一旦加载完成， `Promise` 的状态就发生变化

```
const preloadImage = function (path) {
  return new Promise(function (resolve, reject) {
    const image = new Image();
    image.onload  = resolve;
    image.onerror = reject;
    image.src = path;
  });
};
```

通过链式操作，将多个渲染数据分别给个 `then` ，让其各司其职。或当下个异步请求依赖上个请求结果的时候，我们也能够通过链式操作友好解决问题

```
// 各司其职
getInfo().then(res=>{
    let { bannerList } = res
    //渲染轮播图
    console.log(bannerList)
    return res
}).then(res=>{

    let { storeList } = res
    //渲染店铺列表
    console.log(storeList)
    return res
}).then(res=>{
    let { categoryList } = res
    console.log(categoryList)
    //渲染分类列表
    return res
})
```

通过 `all()` 实现多个请求合并在一起，汇总所有请求结果，只需设置一个 `loading` 即可

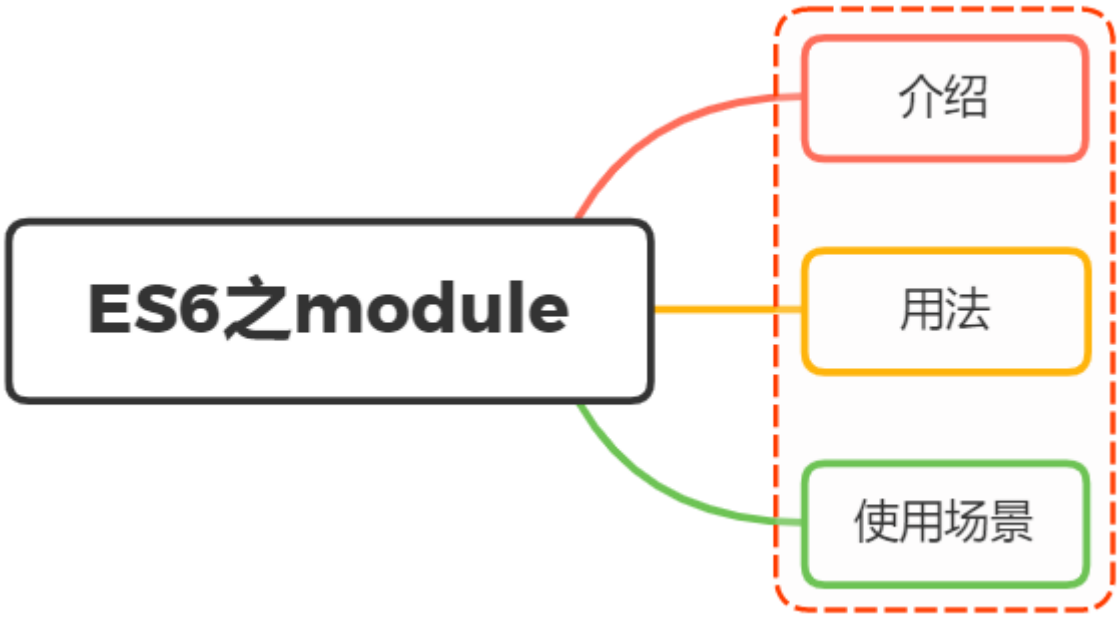
```
function initLoad(){
    // loading.show() //加载loading
    Promise.all([getBannerList(),getStoreList(),getCategoryList()]).then(res=>{
        console.log(res)
        loading.hide() //关闭loading
    }).catch(err=>{
        console.log(err)
        loading.hide()//关闭loading
    })
}
```

```
//数据初始化initLoad()
```

通过 `race` 可以设置图片请求超时

```
//请求某个图片资源
function requestImg(){
    var p = new Promise(function(resolve, reject){
        var img = new Image();
        img.onload = function(){
           resolve(img);
        }
        //img.src = "https://b-gold-cdn.xitu.io/v3/static/img/logo.a7995ad.svg"; 正确的
        img.src = "https://b-gold-cdn.xitu.io/v3/static/img/logo.a7995ad.svg1";
    });
    return p;
}
//延时函数，用于给请求计时
function timeout(){
    var p = new Promise(function(resolve, reject){
        setTimeout(function(){
            reject('图片请求超时');
        }, 5000);
    });
    return p;
}
Promise
.race([requestImg(), timeout()])
.then(function(results){
    console.log(results);
})
.catch(function(reason){
    console.log(reason);
});
```

6. 你是怎么理解ES6中Module的？使用场景？



6.1. 介绍

模块，（Module），是能够单独命名并独立地完成一定功能的程序语句的**集合（即程序代码和数据结构的集合体）**。

两个基本的特征：外部特征和内部特征

- 外部特征是指模块跟外部环境联系的接口（即其他模块或程序调用该模块的方式，包括有输入输出参数、引用的全局变量）和模块的功能
- 内部特征是指模块的内部环境具有的特点（即该模块的局部数据和程序代码）

6.1.1. 为什么需要模块化

- 代码抽象
- 代码封装
- 代码复用
- 依赖管理

如果没有模块化，我们代码会怎样？

- 变量和方法不容易维护，容易污染全局作用域
- 加载资源的方式通过script标签从上到下。
- 依赖的环境主观逻辑偏重，代码较多就会比较复杂。
- 大型项目资源难以维护，特别是多人合作的情况下，资源的引入会让人奔溃

因此，需要一种将 `JavaScript` 程序模块化的机制，如

- CommonJs (典型代表：node.js早期)
- AMD (典型代表：require.js)

- CMD (典型代表：sea.js)

6.1.2. AMD

`Asynchronous ModuleDefinition`（AMD），异步模块定义，采用异步方式加载模块。所有依赖模块的语句，都定义在一个回调函数中，等到模块加载完成之后，这个回调函数才会运行

代表库为 `require.js`

```
/** main.js 入口文件/主模块 **/
// 首先用config()指定各模块路径和引用名
require.config({
  baseUrl: "js/lib",
  paths: {
    "jquery": "jquery.min",  //实际路径为js/lib/jquery.min.js
    "underscore": "underscore.min",
  }
});
// 执行基本操作
require(["jquery","underscore"],function($,_){
  // some code here
});
```

6.1.3. CommonJs

`CommonJS` 是一套 `Javascript` 模块规范，用于服务端

```
// a.js
module.exports={ foo , bar}
// b.js
const { foo,bar } = require('./a.js')
```

其有如下特点：

- 所有代码都运行在模块作用域，不会污染全局作用域
- 模块是同步加载的，即只有加载完成，才能执行后面的操作
- 模块在首次执行后就会缓存，再次加载只返回缓存结果，如果想要再次执行，可清除缓存
- `require` 返回的值是被输出的值的拷贝，模块内部的变化也不会影响这个值

既然存在了 `AMD` 以及 `CommonJs` 机制， `ES6` 的 `Module` 又有什么不一样?

ES6 在语言标准的层面上，实现了 `Module` ，即模块功能，完全可以取代 `CommonJS` 和 `AMD` 规范，成为浏览器和服务器通用的模块解决方案

`CommonJS` 和 `AMD` 模块，都只能在运行时确定这些东西。比如， `CommonJS` 模块就是对象，输入时必须查找对象属性

```
// CommonJS模块
let { stat, exists, readfile } = require('fs');
// 等同于
let _fs = require('fs');
let stat = _fs.stat;
let exists = _fs.exists;
let readfile = _fs.readfile;
```

`ES6` 设计思想是尽量的静态化，使得编译时就能确定模块的依赖关系，以及输入和输出的变量

```
// ES6模块
import { stat, exists, readFile } from 'fs';
```

上述代码，只加载3个方法，其他方法不加载，即 `ES6` 可以在编译时就完成模块加载

由于编译加载，使得静态分析成为可能。包括现在流行的 `typeScript` 也是依靠静态分析实现功能

6.2. 二、使用

`ES6` 模块内部自动采用了严格模式，这里就不展开严格模式的限制，毕竟这是 `ES5` 之前就已经规定好

模块功能主要由两个命令构成：

- `export` ：用于规定模块的对外接口
- `import` ：用于输入其他模块提供的功能

6.2.1. export

一个模块就是一个独立的文件，该文件内部的所有变量，外部无法获取。如果你希望外部能够读取模块内部的某个变量，就必须使用 `export` 关键字输出该变量

```
// profile.js
export var firstName = 'Michael';
export var lastName = 'Jackson';
export var year = 1958;
或
// 建议使用下面写法，这样能瞬间确定输出了哪些变量
var firstName = 'Michael';
var lastName = 'Jackson';
```

```
var year = 1958;
export { firstName, lastName, year };
```

输出函数或类

```
export function multiply(x, y) {
  return x * y;
};
```

通过 `as` 可以进行输出变量的重命名

```
function v1() { ... }
function v2() { ... }
export {
  v1 as streamV1,
  v2 as streamV2,
  v2 as streamLatestVersion
};
```

6.2.2. import

使用 `export` 命令定义了模块的对外接口以后，其他 JS 文件就可以通过 `import` 命令加载这个模块

```
// main.js
import { firstName, lastName, year } from './profile.js';
function setName(element) {
  element.textContent = firstName + ' ' + lastName;
}
```

同样如果想要输入变量起别名，通过 `as` 关键字

```
import { lastName as surname } from './profile.js';
```

当加载整个模块的时候，需要用到星号 `*`

```
// circle.js
export function area(radius) {
```

```
  return Math.PI * radius * radius;
}
export function circumference(radius) {
  return 2 * Math.PI * radius;
}
// main.js
import * as circle from './circle';
console.log(circle)   // {area:area,circumference:circumference}
```

输入的变量都是只读的，不允许修改，但是如果是对象，允许修改属性

```
import {a} from './xxx.js'
a.foo = 'hello'; // 合法操作
a = {}; // Syntax Error : 'a' is read-only;
```

不过建议即使能修改，但我们不建议。因为修改之后，我们很难差错

`import` 后面我们常接着 `from` 关键字， `from` 指定模块文件的位置，可以是相对路径，也可以是绝对路径

```
import { a } from './a';
```

如果只有一个模块名，需要有配置文件，告诉引擎模块的位置

```
import { myMethod } from 'util';
```

在编译阶段， `import` 会提升到整个模块的头部，首先执行

```
foo();
import { foo } from 'my_module';
```

多次重复执行同样的导入，只会执行一次

```
import 'lodash';
import 'lodash';
```

上面的情况，大家都能看到用户在导入模块的时候，需要知道加载的变量名和函数，否则无法加载

如果不需要知道变量名或函数就完成加载，就要用到 `export default` 命令，为模块指定默认输出

```
// export-default.js
export default function () {
    console.log('foo');
}
```

加载该模块的时候， `import` 命令可以为该函数指定任意名字

```
// import-default.js
import customName from './export-default';
customName(); // 'foo'
```

6.2.3. 动态加载

允许您仅在需要时动态加载模块，而不必预先加载所有模块，这存在明显的性能优势

这个新功能允许您将 `import()` 作为函数调用，将其作为参数传递给模块的路径。 它返回一个 `promise` ，它用一个模块对象来实现，让你可以访问该对象的导出

```
import('/modules/myModule.mjs')
  .then((module) => {
    // Do something with the module.
  });
```

6.2.4. 复合写法

如果在一个模块之中，先输入后输出同一个模块， `import` 语句可以与 `export` 语句写在一起

```
export { foo, bar } from 'my_module';
// 可以简单理解为
import { foo, bar } from 'my_module';
export { foo, bar };
```

同理能够搭配 `as` 、 `*` 搭配使用

6.3. 使用场景

如今，`ES6`模块化已经深入我们日常项目开发中，像`vue`、`react`项目搭建项目，组件化开发处处可见，其也是依赖模块化实现

`vue`组件

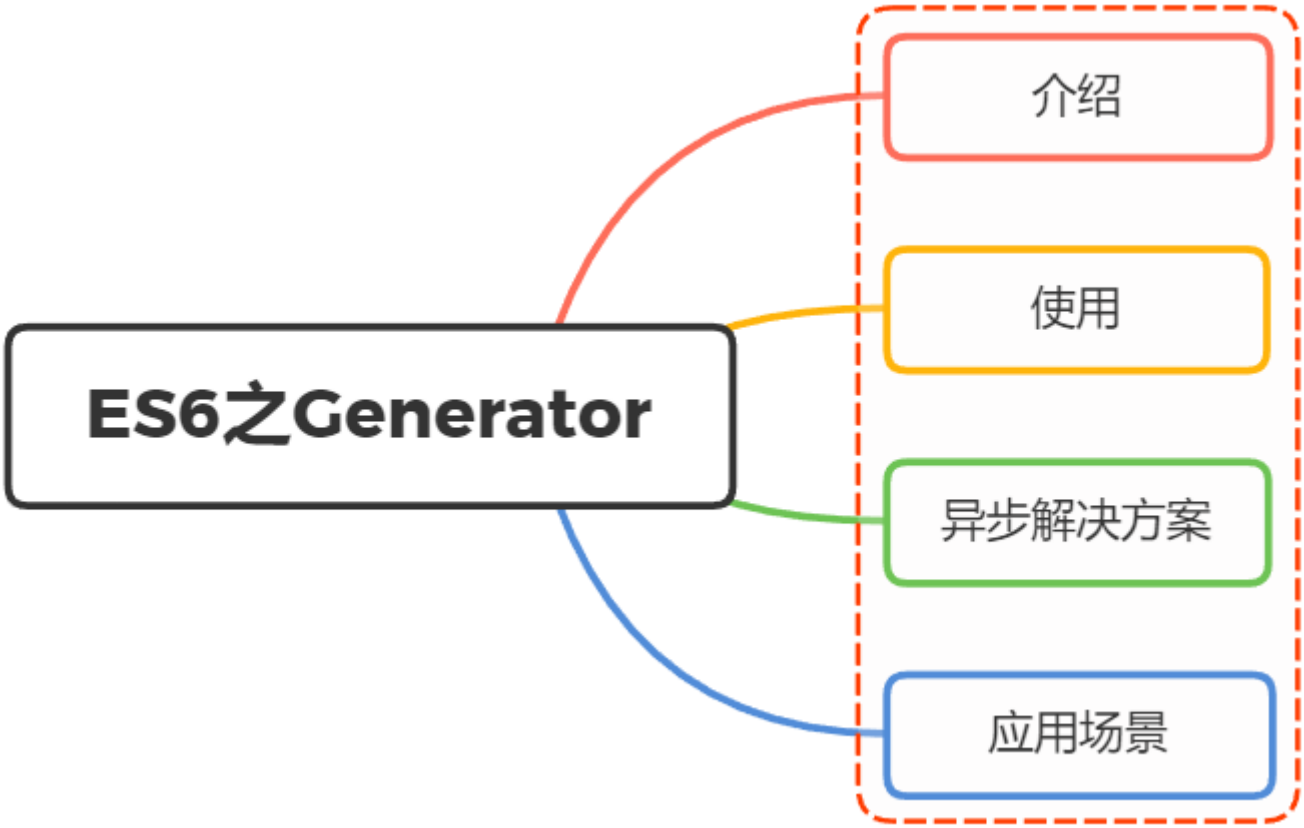
```
<template>
  <div class="App">
      组件化开发 ---- 模块化
  </div>
</template>
<script>
export default {
  name: 'HelloWorld',
  props: {
    msg: String
  }
}
</script>
```

`react`组件

```
function App() {
  return (
    <div className="App">
                组件化开发 ---- 模块化
    </div>
  );
}
export default App;
```

包括完成一些复杂应用的时候，我们也可以拆分成各个模块

7. 你是怎么理解ES6中 Generator的？使用场景？



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

```
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

```
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

```
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`

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

```
8 // { 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

```
1  function* foo(x) {
2    var y = 2 * (yield (x + 1));
3    var z = yield (y / 3);
4    return (x + y + z);
5  }
6  var a = foo(5);
7  a.next() // Object{value:6, done:false}
8  a.next() // Object{value:NaN, done:false}
9  a.next() // Object{value:NaN, done:true}
10 var b = foo(5);
11 b.next() // { value:6, done:false }
12 b.next(12) // { value:8, done:false }
13 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

```
1  function* foo() {
2    yield 1;
3    yield 2;
4    yield 3;
5    yield 4;
6    yield 5;
7    return 6;
8  }
9  for (let v of foo()) {
10   console.log(v);
11 }
12 // 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

```
1  function* objectEntries(obj) {
2    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

```
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

```
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

```
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

```
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 `Interator` 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

```
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

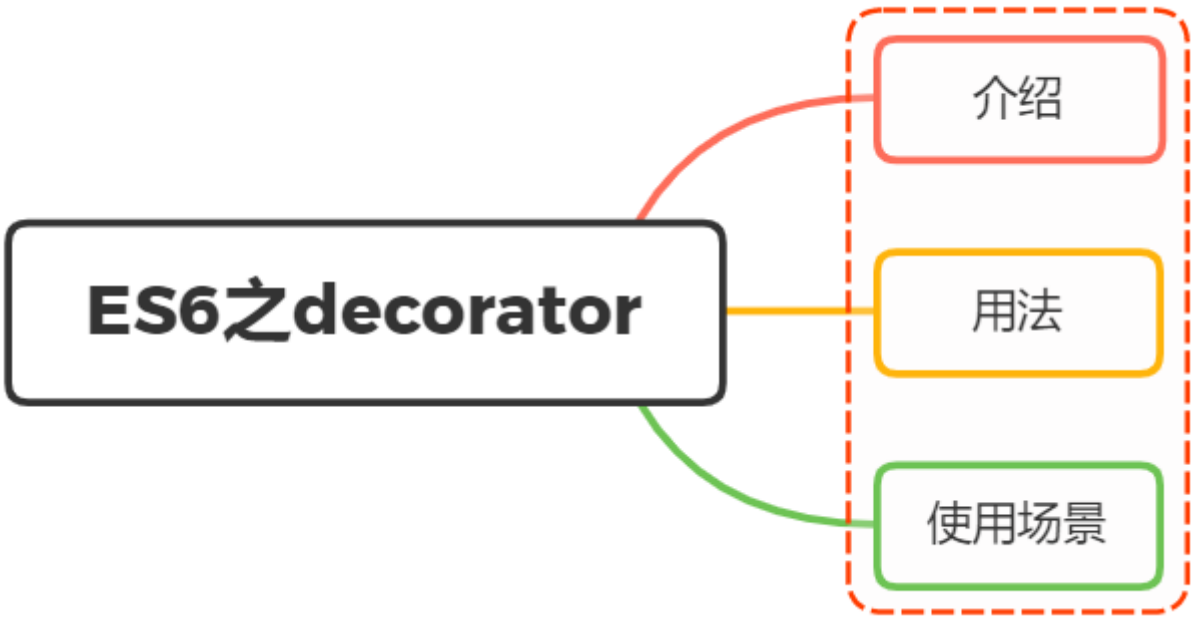
```
import { call, put, takeEvery, takeLatest } from 'redux-saga/effects'
import Api from '...'
function* fetchUser(action) {
   try {
      const user = yield call(Api.fetchUser, action.payload.userId);
      yield put({type: "USER_FETCH_SUCCEEDED", user: user});
   } catch (e) {
      yield put({type: "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

```
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

```
class soldier{
}
```

定义一个得到 AK 装备的函数，即装饰器

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

使用该装饰器对士兵进行增强

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

这时候士兵就有武器了

```
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

```
@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

```
@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

```
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

```
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

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

相当于以下调用

```
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

```
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`@dec(1)`先进入，但是内层装饰`@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

```
var counter = 0;
var add = function () {
  counter++;
};
@add
function foo() {
}
```

编译阶段，变成下面

```
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

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

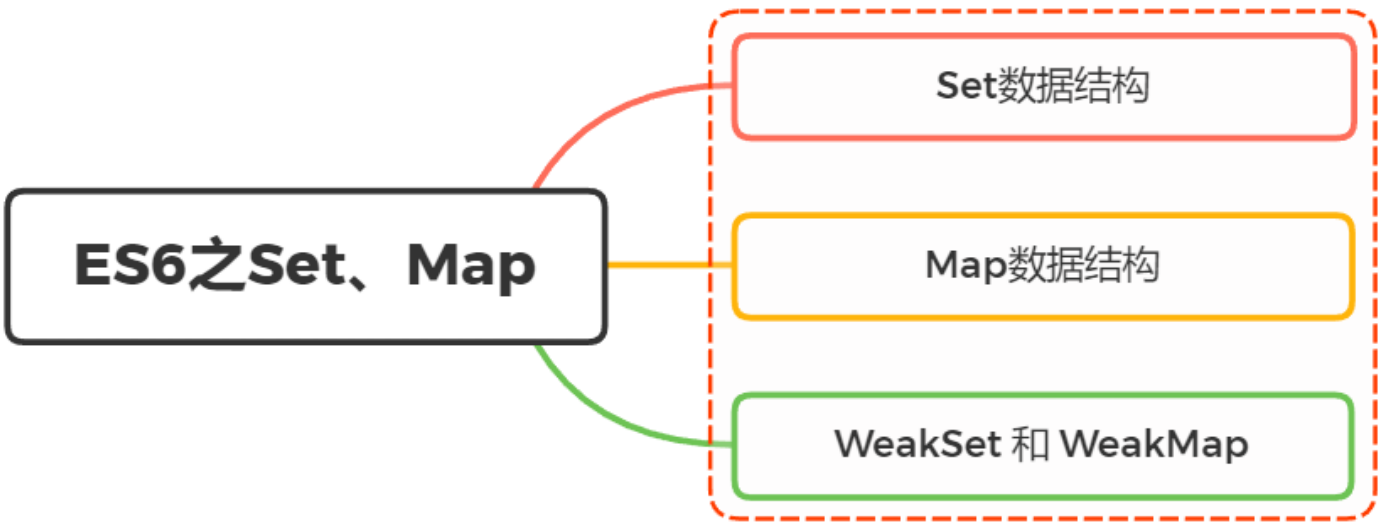
通过装饰器就变得简洁多了

```
@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

```
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

```
1 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

```
1 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

```
1 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

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

9.1.5. clear()

清除所有成员，没有返回值

```
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

```
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`

```
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

```
// 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

```
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

```
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。

```
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

```
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`

```
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

```
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`

```
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

```
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

```
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

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

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

```
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

```
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

```
// 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

```
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

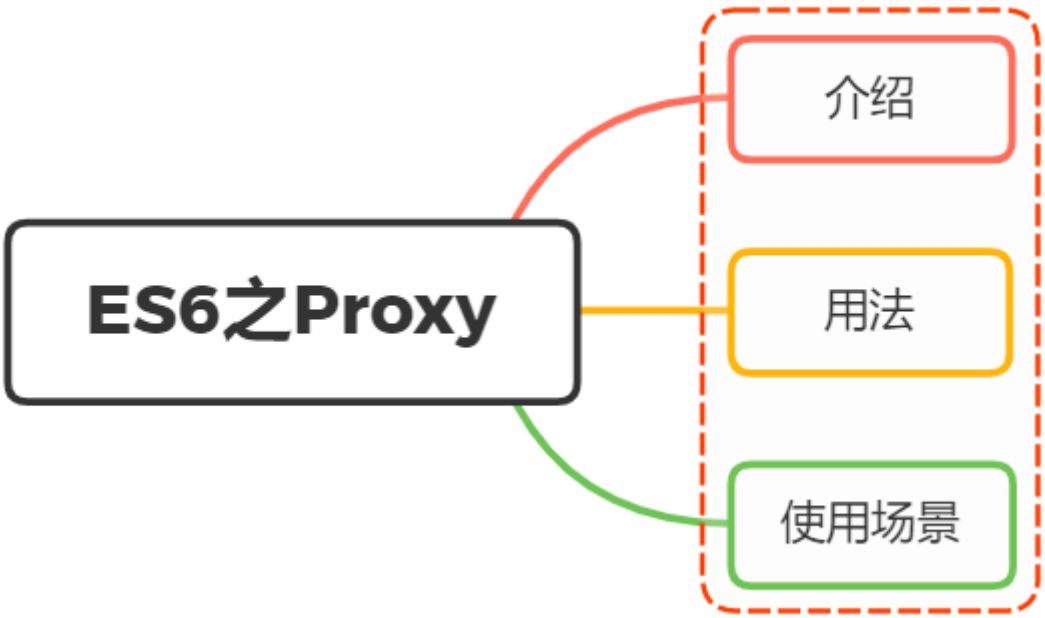
```
const wm = new WeakMap();
const element = document.getElementById('example');
wm.set(element, 'some information');
wm.get(element) // "some information"
```

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

```
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

```
#!/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

```
1 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

```
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

```
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

```
const target = Object.defineProperties({}, {
  foo: {
    value: 123,
    writable: false,
    configurable: false
  },
});
const handler = {
  get(target, propKey) {
    return 'abc';
  }
};
const proxy = new Proxy(target, handler);
proxy.foo
// TypeError: Invariant check failed
```

10.2.5. set()

`set` 方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

假定 `Person` 对象有一个 `age` 属性，该属性应该是一个不大于 200 的整数，那么可以使用 `Proxy` 保证 `age` 的属性值符合要求

```
let validator = {
  set: function(obj, prop, value) {
    if (prop === 'age') {
      if (!Number.isInteger(value)) {
        throw new TypeError('The age is not an integer');
      }
      if (value > 200) {
        throw new RangeError('The age seems invalid');
      }
    }
    // 对于满足条件的 age 属性以及其他属性，直接保存
    obj[prop] = value;
  }
};
let person = new Proxy({}, validator);
person.age = 100;
person.age // 100
person.age = 'young' // 报错
person.age = 300 // 报错
```

如果目标对象自身的某个属性，不可写且不可配置，那么 `set` 方法将不起作用

```
const obj = {};
Object.defineProperty(obj, 'foo', {
  value: 'bar',
  writable: false,
});
const handler = {
  set: function(obj, prop, value, receiver) {
    obj[prop] = 'baz';
  }
};
const proxy = new Proxy(obj, handler);
proxy.foo = 'baz';
proxy.foo // "bar"
```

注意，严格模式下， `set` 代理如果没有返回 `true` ，就会报错

```
'use strict';
const handler = {
  set: function(obj, prop, value, receiver) {
    obj[prop] = receiver;
```

```
    // 无论有没有下面这一行，都会报错
    return false;
  }
};
const proxy = new Proxy({}, handler);
proxy.foo = 'bar';
// TypeError: 'set' on proxy: trap returned falsish for property 'foo'
```

10.2.6. deleteProperty()

`deleteProperty` 方法用于拦截 `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 handler = {
  deleteProperty (target, key) {
    invariant(key, 'delete');
    Reflect.deleteProperty(target,key)
    return true;
  }
};
function invariant (key, action) {
  if (key[0] === '_') {
    throw new Error(
无法删除私有属性
);
  }
}
var target = { _prop: 'foo' };
var proxy = new Proxy(target, handler);
delete proxy._prop
// Error: 无法删除私有属性
```

注意，目标对象自身的不可配置（configurable）的属性，不能被 `deleteProperty` 方法删除，否则报错

10.2.7. 取消代理

```
Proxy.revocable(target, handler);
```

10.3. 使用场景

`Proxy` 其功能非常类似于设计模式中的代理模式，常用功能如下：

- 拦截和监视外部对对象的访问
- 降低函数或类的复杂度
- 在复杂操作前对操作进行校验或对所需资源进行管理

使用 `Proxy` 保障数据类型的准确性

```
let numericDataStore = { count: 0, amount: 1234, total: 14 };
numericDataStore = new Proxy(numericDataStore, {
    set(target, key, value, proxy) {
        if (typeof value !== 'number') {
            throw Error("属性只能是number类型");
        }
        return Reflect.set(target, key, value, proxy);
    }
});
numericDataStore.count = "foo"
// Error: 属性只能是number类型
numericDataStore.count = 333
// 赋值成功
```

声明了一个私有的 `apiKey`，便于 `api` 这个对象内部的方法调用，但不希望从外部也能够访问 `api._apiKey`

```
let api = {
    _apiKey: '123abc456def',
    getUsers: function(){ },
    getUser: function(userId){ },
    setUser: function(userId, config){ }
};
const RESTRICTED = ['_apiKey'];
api = new Proxy(api, {
    get(target, key, proxy) {
        if(RESTRICTED.indexOf(key) > -1) {
            throw Error(
${key} 不可访问.
);
        } return Reflect.get(target, key, proxy);
    },
    set(target, key, value, proxy) {
        if(RESTRICTED.indexOf(key) > -1) {
            throw Error(
${key} 不可修改
```

```
);
        } return Reflect.get(target, key, value, proxy);
    }
});
console.log(api._apiKey)
api._apiKey = '987654321'
// 上述都抛出错误
```

还能通过使用 `Proxy` 实现观察者模式

观察者模式（Observer mode）指的是函数自动观察数据对象，一旦对象有变化，函数就会自动执行

`observable` 函数返回一个原始对象的 `Proxy` 代理，拦截赋值操作，触发充当观察者的各个函数

```
const queuedObservers = new Set();
const observe = fn => queuedObservers.add(fn);
const observable = obj => new Proxy(obj, {set});
function set(target, key, value, receiver) {
  const result = Reflect.set(target, key, value, receiver);
  queuedObservers.forEach(observer => observer());
  return result;
}
```

观察者函数都放进 `Set` 集合，当修改 `obj` 的值，在会 `set` 函数中拦截，自动执行 `Set` 所有的观察者