

ES6~ES12（二）

王红元 coderwhy



let/const基本使用

- 在ES5中我们声明变量都是使用的var关键字，从ES6开始新增了两个关键字可以声明变量：let、const
 - let、const在其他编程语言中都是有的，所以也并不是新鲜的关键字；
 - 但是let、const确实确实给JavaScript带来一些不一样的东西；
- let关键字：
 - 从直观的角度来说，let和var是没有太大的区别的，都是用于声明一个变量
- const关键字：
 - const关键字是constant的单词的缩写，表示常量、衡量的意思；
 - 它表示保存的数据一旦被赋值，就不能被修改；
 - 但是如果赋值的是引用类型，那么可以通过引用找到对应的对象，修改对象的内容；
- 注意：另外let、const不允许重复声明变量；

let/const作用域提升

■ let、const和var的另一个重要区别是作用域提升：

- 我们知道var声明的变量是会进行作用域提升的；
- 但是如果使用let声明的变量，在声明之前访问会报错；

```
console.log(foo) // ReferenceError: Cannot access 'foo' before initialization  
  
let foo = "foo"
```

■ 那么是不是意味着foo变量只有在代码执行阶段才会创建的呢？

- 事实上并不是这样的，我们可以看一下ECMA262对let和const的描述；
- 这些变量会被创建在包含他们的词法环境被实例化时，但是是不可以访问它们的，直到词法绑定被求值；

let and const declarations define variables that are scoped to the running execution context's LexicalEnvironment. The variables are created when their containing Lexical Environment is instantiated but may not be accessed in any way until the variable's LexicalBinding is evaluated.

A variable defined by a LexicalBinding with an Initializer is assigned the value of its Initializer's AssignmentExpression when the LexicalBinding is evaluated, not when the variable is created. If a LexicalBinding in a let declaration does not have an Initializer the variable is assigned the value undefined when the LexicalBinding is evaluated.

let/const有没有作用域提升呢？

- 从上面我们可以看出，在执行上下文的词法环境创建出来的时候，变量事实上已经被创建了，只是这个变量是不能被访问的。
 - 那么变量已经有了，但是不能被访问，是不是一种作用域的提升呢？
- 事实上维基百科并没有对作用域提升有严格的解释，那么我们自己从字面量上理解；
 - **作用域提升**：在声明变量的作用域中，如果这个变量可以在声明之前被访问，那么我们可以称之为作用域提升；
 - 在这里，它虽然被创建出来了，但是不能被访问，我认为不能称之为作用域提升；
- 所以我的观点是let、const没有进行作用域提升，但是会在解析阶段被创建出来。

Window对象添加属性

■ 我们知道，在全局通过var来声明一个变量，事实上会在window上添加一个属性：

□ 但是let、const是不会给window上添加任何属性的。

■ 那么我们可能会想这个变量是保存在哪里呢？

■ 我们先回顾一下最新的ECMA标准中对执行上下文的描述

Every execution context has associated with it a variable object. Variables and functions declared in the source text are added as properties of the variable object. For function code, parameters are added as properties of the variable object.

每一个执行上下文会被关联到一个变量环境（variable object, VO），在源代码中的变量和函数声明会被作为属性添加到VO中。

对于函数来说，参数也会被添加到VO中。

Every execution context has an associated VariableEnvironment. Variables and functions declared in ECMAScript code evaluated in an execution context are added as bindings in that VariableEnvironment's Environment Record. For function code, parameters are also added as bindings to that Environment Record.

每一个执行上下文会关联到一个变量环境（VariableEnvironment）中，在执行代码中变量和函数的声明会作为环境记录（Environment Record）添加到变量环境中。

对于函数来说，参数也会被作为环境记录添加到变量环境中。

变量被保存到VariableMap中

- 也就是说我们声明的变量和环境记录是被添加到变量环境中的：
 - 但是标准有没有规定这个对象是window对象或者其他对象呢？
 - 其实并没有，那么JS引擎在解析的时候，其实会有自己的实现；
 - 比如v8中其实是通过VariableMap的一个hashmap来实现它们的存储的。
 - 那么window对象呢？而window对象是早期的GO对象，在最新的实现中其实是浏览器添加的全局对象，并且一直保持了window和var之间值的相等性；

```
// A hash map to support fast variable declaration and lookup.  
class VariableMap : public ZoneHashMap {  
public:  
    explicit VariableMap(Zone* zone);  
    VariableMap(const VariableMap& other, Zone* zone);  
  
    VariableMap(VariableMap&& other) V8_NOEXCEPT : ZoneHashMap(std::move(other)) {  
    }  
}
```

var的块级作用域

- 在我们前面的学习中，JavaScript只会形成两个作用域：全局作用域和函数作用域。



- ES5中放到一个代码中定义的变量，外面是可以访问的：

```
// var 没有块级作用域
{
  // 编写语句
  var foo = "foo"
}

console.log(foo) // foo 可以访问到
```

let/const的块级作用域

- 在ES6中新增了块级作用域，并且通过let、const、function、class声明的标识符是具备块级作用域的限制的：

```
{  
  let foo = "foo"  
  function bar() {  
    console.log("bar")  
  }  
  class Person {}  
}  
  
console.log(foo) // ReferenceError: foo is not defined  
bar() // 可以访问  
var p = new Person() // ReferenceError: foo is not defined
```

- 但是我们会发现函数拥有块级作用域，但是外面依然是可以访问的：
 - 这是因为引擎会对函数的声明进行特殊的处理，允许像var那样进行提升；

块级作用域的应用

- 我来看一个实际的案例：获取多个按钮监听点击

```
<button>按钮1</button>
<button>按钮2</button>
<button>按钮3</button>
<button>按钮4</button>
```

- 使用let或者const来实现：

```
var btns = document.getElementsByTagName("button")
for (let i = 0; i < btns.length; i++) {
  btns[i].onclick = function() {
    console.log("第" + i + "个按钮被点击")
  }
}
```

暂时性死区

■ 在ES6中，我们还有一个概念称之为暂时性死区：

- 它表达的意思是在一个代码中，使用let、const声明的变量，在声明之前，变量都是不可以访问的；
- 我们将这种现象称之为 temporal dead zone (暂时性死区，TDZ) ；

```
var foo = "foo"

if (true) {
  console.log(foo) // ReferenceError: Cannot access 'foo' before initialization

  let foo = "bar"
}
```



var、let、const的选择

■ 那么在开发中，我们到底应该选择使用哪一种方式来定义我们的变量呢？

■ 对于var的使用：

- 我们需要明白一个事实，var所表现出来的特殊性：比如作用域提升、window全局对象、没有块级作用域等都是一些历史遗留问题；
- 其实是JavaScript在设计之初的一种语言缺陷；
- 当然目前市场上也在利用这种缺陷出一系列的面试题，来考察大家对JavaScript语言本身以及底层的理解；
- 但是在实际工作中，我们可以使用最新的规范来编写，也就是不再使用var来定义变量了；

■ 对于let、const：

- 对于let和const来说，是目前开发中推荐使用的；
- 我们会优先推荐使用const，这样可以保证数据的安全性不会被随意的篡改；
- 只有当我们明确知道一个变量后续会需要被重新赋值时，这个时候再使用let；
- 这种在很多其他语言里面也都是一种约定俗成的规范，尽量我们也遵守这种规范；

字符串模板基本使用

- 在ES6之前，如果我们想要将字符串和一些动态的变量（标识符）拼接到一起，是非常麻烦和丑陋的（ugly）。
- ES6允许我们使用字符串模板来嵌入JS的变量或者表达式来进行拼接：
 - 首先，我们会使用 `` 符号来编写字符串，称之为模板字符串；
 - 其次，在模板字符串中，我们可以通过 **`${expression}`** 来嵌入动态的内容；

```
const name = "why"
const age = 18
const height = 1.88

console.log(`my name is ${name}, age is ${age}, height is ${height}`)
console.log(`我是成年人吗? ${age >= 18 ? '是' : '否'}`)

function foo() {
  return "function is foo"
}

console.log(`my function is ${foo()}`)
```

标签模板字符串使用

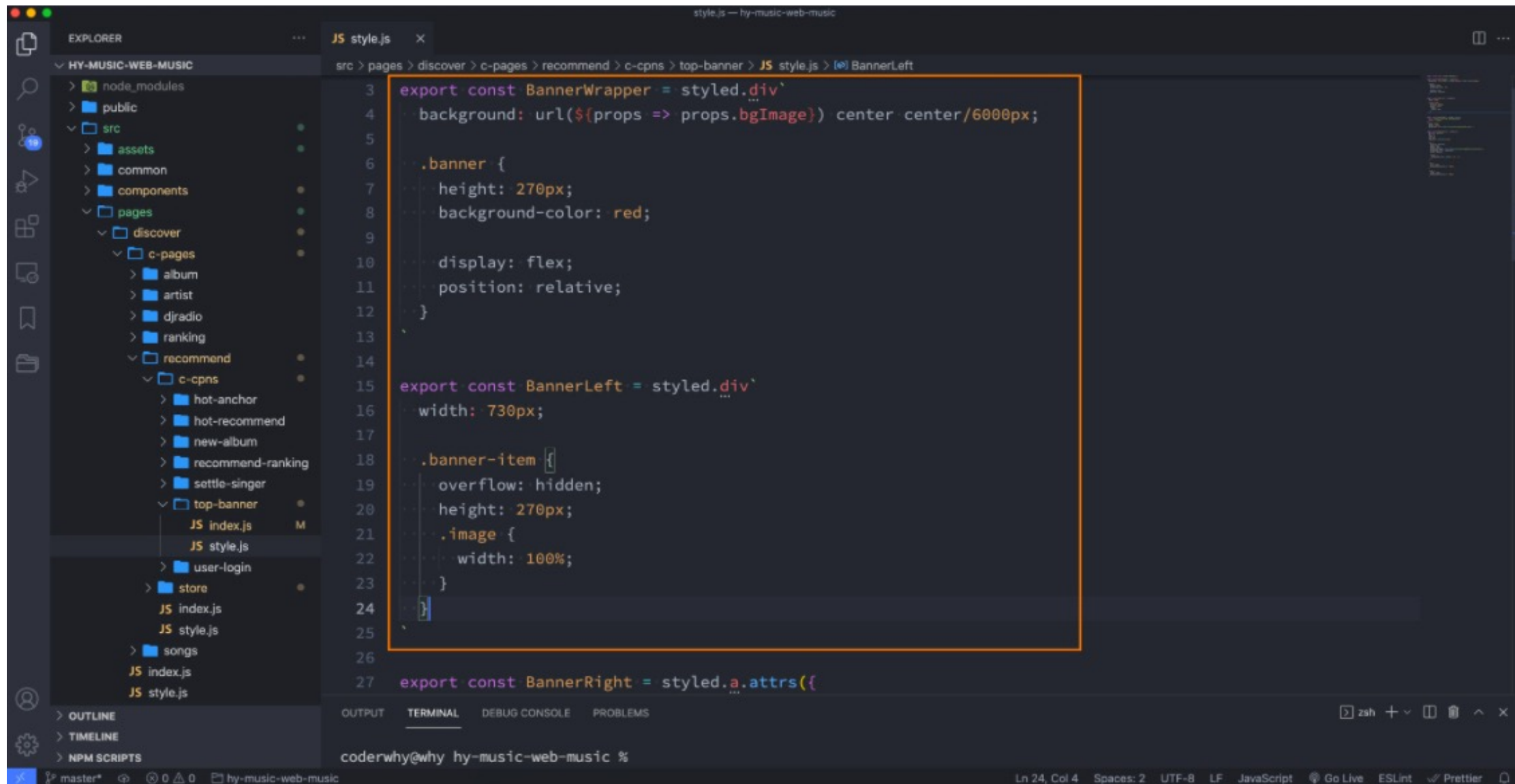
- 模板字符串还有另外一种用法：标签模板字符串（Tagged Template Literals）。
- 我们一起来看一个普通的JavaScript的函数：

```
function foo(...args) {  
  console.log(args)  
}  
  
// ['Hello World']  
foo("Hello World")
```

- 如果我们使用标签模板字符串，并且在调用的时候插入其他的变量：
 - 模板字符串被拆分了；
 - 第一个元素是数组，是被模板字符串拆分的字符串组合；
 - 后面的元素是一个个模板字符串传入的内容；

```
const name = "why"  
const age = 18  
// [['Hello ', 'World ', ''], 'why', 18]  
foo`Hello ${name} World ${age}`
```

React的styled-components库



The screenshot shows a VS Code editor with a project named 'HY-MUSIC-WEB-MUSIC'. The Explorer panel on the left shows the project structure, including folders like 'node_modules', 'public', 'src', 'assets', 'common', 'components', 'pages', 'discover', 'c-pages', 'album', 'artist', 'djradio', 'ranking', 'recommend', 'c-cpns', 'hot-anchor', 'hot-recommend', 'new-album', 'recommend-ranking', 'settle-singer', 'top-banner', 'user-login', 'store', 'songs', and 'index.js'. The main editor displays the file 'style.js' in the path 'src > pages > discover > c-pages > recommend > c-cpns > top-banner > JS style.js'. The code in the editor is as follows:

```
3 export const BannerWrapper = styled.div`
4   background: url(${props => props.bgImage}) center center/6000px;
5
6   .banner {
7     height: 270px;
8     background-color: red;
9
10    display: flex;
11    position: relative;
12  }
13
14
15 export const BannerLeft = styled.div`
16   width: 730px;
17
18   .banner-item {
19     overflow: hidden;
20     height: 270px;
21     .image {
22       width: 100%;
23     }
24   }
25
26
27 export const BannerRight = styled.a.attrs({
```

The bottom status bar shows the file path 'Ln 24, Col 4', encoding 'UTF-8', language 'JavaScript', and various extensions like 'Go Live', 'ESLint', and 'Prettier'.

函数的默认参数

■ 在ES6之前，我们编写的函数参数是没有默认值的，所以我们在编写函数时，如果有下面的需求：

- 传入了参数，那么使用传入的参数；
- 没有传入参数，那么使用一个默认值；

■ 而在ES6中，我们允许给函数一个默认值：

```
function foo(x = 20, y = 30) {  
  console.log(x, y)  
}
```

```
foo(50, 100) // 50 100  
foo() // 20 30
```

```
function foo() {  
  var x =  
    arguments.length > 0 && arguments[0] !== undefined ? arguments[0] : 20;  
  var y =  
    arguments.length > 1 && arguments[1] !== undefined ? arguments[1] : 30;  
  console.log(x, y);  
}
```

函数默认值的补充

- 默认值也可以和解构一起来使用：

```
// 写法一:  
function foo({name, age} = {name: "why", age: 18}) {  
  console.log(name, age)  
}  
  
// 写法二:  
function foo({name = "why", age = 18} = {}) {  
  console.log(name, age)  
}
```

- 另外参数的默认值我们通常会将其放到最后（在很多语言中，如果不放到最后其实会报错的）：
 - 但是JavaScript允许不将其放到最后，但是意味着还是会按照顺序来匹配；
- 另外默认值会改变函数的length的个数，默认值以及后面的参数都不计算在length之内了。

函数的剩余参数

■ ES6中引用了rest parameter，可以将不定数量的参数放入到一个数组中：

□ 如果最后一个参数是 ... 为前缀的，那么它会将剩余的参数放到该参数中，并且作为一个数组；

```
function foo(m, n, ...args) {  
  console.log(m, n)  
  console.log(args)  
}
```

■ 那么剩余参数和arguments有什么区别呢？

□ 剩余参数只包含那些没有对应形参的实参，而 arguments 对象包含了传给函数的所有实参；

□ arguments对象不是一个真正的数组，而rest参数是一个真正的数组，可以进行数组的所有操作；

□ arguments是早期的ECMAScript中为了方便去获取所有的参数提供的一个数据结构，而rest参数是ES6中提供并且希望以此来替代arguments的；

■ 剩余参数必须放到最后一个位置，否则会报错。

函数箭头函数的补充

■ 在前面我们已经学习了箭头函数的用法，这里进行一些补充：

□ 箭头函数是没有显式原型的，所以不能作为构造函数，使用new来创建对象；

```
var foo = () => {  
  console.log("foo")  
}  
  
console.log(foo.prototype) // undefined  
  
// TypeError: foo is not a constructor  
var f = new foo()
```



展开语法

■ 展开语法(Spread syntax) :

- 可以在函数调用/数组构造时, 将数组表达式或者string在语法层面展开;
- 还可以在构造字面量对象时, 将对象表达式按key-value的方式展开;

■ 展开语法的场景 :

- 在函数调用时使用;
- 在数组构造时使用;
- 在构建对象字面量时, 也可以使用展开运算符, 这个是在ES2018 (ES9) 中添加的新特性;

■ 注意: 展开运算符其实是一种浅拷贝;

数值的表示

- 在ES6中规范了二进制和八进制的写法：

```
const num1 = 100
// b -> binary
const num2 = 0b100
// octonary
const num3 = 0o100
// hexadecimal
const num4 = 0x100
```

- 另外在ES2021新增特性：数字过长时，可以使用_作为连接符

```
// ES2021新增特性
const num5 = 100_000_000
```

Symbol的基本使用

- Symbol是什么呢？Symbol是ES6中新增的一个基本数据类型，翻译为符号。
- 那么为什么需要Symbol呢？
 - 在ES6之前，对象的属性名都是字符串形式，那么很容易造成属性名的冲突；
 - 比如原来有一个对象，我们希望在其中添加一个新的属性和值，但是我们在不确定它原来内部有什么内容的情况下，很容易造成冲突，从而覆盖掉它内部的某个属性；
 - 比如我们前面在讲apply、call、bind实现时，我们有给其中添加一个fn属性，那么如果它内部原来已经有了fn属性了呢？
 - 比如开发中我们使用混入，那么混入中出现了同名的属性，必然有一个会被覆盖掉；
- Symbol就是为了解决上面的问题，用来生成一个独一无二的值。
 - Symbol值是通过Symbol函数来生成的，生成后可以作为属性名；
 - 也就是在ES6中，对象的属性名可以使用字符串，也可以使用Symbol值；
- Symbol即使多次创建值，它们也是不同的：Symbol函数执行后每次创建出来的值都是独一无二的；
- 我们也可以在创建Symbol值的时候传入一个描述description：这个是ES2019（ES10）新增的特性；

Symbol作为属性名

- 我们通常会使用Symbol在对象中表示唯一的属性名：

```
const s1 = Symbol("abc")
const s2 = Symbol("cba")

const obj = {}

// 1. 写法一：属性名赋值
obj[s1] = "abc"
obj[s2] = "cba"

// 2. 写法二：Object.defineProperty
Object.defineProperty(obj, s1, {
  enumerable: true,
  configurable: true,
  writable: true,
  value: "abc"
})

// 3. 写法三：定义字面量是直接使用
const info = {
  [s1]: "abc",
  [s2]: "cba"
}
```

```
console.log(Object.getOwnPropertySymbols(info))

const symbolKeys = Object.getOwnPropertySymbols(info)
for (const key of symbolKeys) {
  console.log(info[key])
}
```

相同值的Symbol

■ 前面我们讲Symbol的目的是为了创建一个独一无二的值，那么如果我们现在就是想创建相同的Symbol应该怎么做呢？

- 我们可以使用Symbol.for方法来做到这一点；
- 并且我们可以通过Symbol.keyFor方法来获取对应的key；

```
const s1 = Symbol.for("abc")
const s2 = Symbol.for("abc")

console.log(s1 === s2) // true
const key = Symbol.keyFor(s1)
console.log(key) // abc
const s3 = Symbol.for(key)
console.log(s2 === s3) // true
```

Set的基本使用

- 在ES6之前，我们存储数据的结构主要有两种：数组、对象。
 - 在ES6中新增了另外两种数据结构：Set、Map，以及它们的另外形式WeakSet、WeakMap。
- Set是一个新增的数据结构，可以用来保存数据，类似于数组，但是和数组的区别是**元素不能重复**。
 - 创建Set我们需要通过**Set构造函数**（暂时没有字面量创建的方式）：
- 我们可以发现Set中存放的元素是不会重复的，那么Set有一个非常常用的功能就是给数组去重。

```
const set1 = new Set()
set1.add(10)
set1.add(14)
set1.add(16)
console.log(set1) // Set(3) {10, 14, 16}

const set2 = new Set([11, 15, 18, 11])
console.log(set2) // Set(3) {11, 15, 18}
```

```
const arr = [10, 20, 10, 44, 78, 44]
const set = new Set(arr)
const newArray1 = [...set]
const newArray2 = Array.from(set)
console.log(newArray1, newArray2)
```




Set的常见方法

■ Set常见的属性：

- size：返回Set中元素的个数；

■ Set常用的方法：

- add(value)：添加某个元素，返回Set对象本身；

- delete(value)：从set中删除和这个值相等的元素，返回boolean类型；

- has(value)：判断set中是否存在某个元素，返回boolean类型；

- clear()：清空set中所有的元素，没有返回值；

- forEach(callback, [, thisArg])：通过forEach遍历set；

■ 另外Set是支持for of的遍历的。



WeakSet使用

■ 和Set类似的另外一个数据结构称之为WeakSet，也是内部元素不能重复的数据结构。

■ 那么和Set有什么区别呢？

□ 区别一：WeakSet中只能存放对象类型，不能存放基本数据类型；

□ 区别二：WeakSet对元素的引用是弱引用，如果没有其他引用对某个对象进行引用，那么GC可以对该对象进行回收；

```
const wset = new WeakSet()

// TypeError: Invalid value used in weak set
wset.add(10)
```

■ WeakSet常见的方法：

□ add(value)：添加某个元素，返回WeakSet对象本身；

□ delete(value)：从WeakSet中删除和这个值相等的元素，返回boolean类型；

□ has(value)：判断WeakSet中是否存在某个元素，返回boolean类型；

WeakSet的应用

■ 注意：WeakSet不能遍历

- 因为WeakSet只是对对象的弱引用，如果我们遍历获取到其中的元素，那么有可能造成对象不能正常的销毁。
- 所以存储到WeakSet中的对象是没办法获取的；

■ 那么这个东西有什么用呢？

- 事实上这个问题并不好回答，我们来使用一个Stack Overflow上的答案；

```
const pwset = new WeakSet()
class Person {
  constructor() {
    pwset.add(this)
  }
  running() {
    if(!pwset.has(this)) throw new Error("不能通过其他对象调用running方法")
    console.log("running", this)
  }
}
```

Map的基本使用

- 另外一个新增的数据结构是Map，用于存储映射关系。
- 但是我们可能会想，在之前我们可以使用对象来存储映射关系，他们有什么区别呢？
 - 事实上我们对象存储映射关系只能用字符串（ES6新增了Symbol）作为属性名（key）；
 - 某些情况下我们可能希望通过其他类型作为key，比如对象，这个时候会自动将对象转成字符串来作为key；
- 那么我们就可以使用Map：

```
const obj1 = { name: "why" }  
const obj2 = { age: 18 }  
  
const map = new Map()  
map.set(obj1, "abc")  
map.set(obj2, "cba")  
console.log(map.get(obj1))  
console.log(map.get(obj2))
```

```
const map = new Map([  
  [obj1, "abc"],  
  [obj2, "cba"],  
  [obj1, "nba"]  
)  
console.log(map.get(obj1)) // nba  
console.log(map.get(obj2)) // cba
```



Map的常用方法

■ Map常见的属性：

- size：返回Map中元素的个数；

■ Map常见的方法：

- set(key, value)：在Map中添加key、value，并且返回整个Map对象；
- get(key)：根据key获取Map中的value；
- has(key)：判断是否包括某一个key，返回Boolean类型；
- delete(key)：根据key删除一个键值对，返回Boolean类型；
- clear()：清空所有的元素；
- forEach(callback, [, thisArg])：通过forEach遍历Map；

■ Map也可以通过for of进行遍历。

WeakMap的使用

- 和Map类型的另外一个数据结构称之为WeakMap，也是以键值对的形式存在的。
- 那么和Map有什么区别呢？
 - 区别一：WeakMap的key只能使用对象，不接受其他的类型作为key；
 - 区别二：WeakMap的key对对象想的引用是弱引用，如果没有其他引用引用这个对象，那么GC可以回收该对象；

```
const weakMap = new WeakMap()  
// Invalid value used as weak map key  
weakMap.set(1, "abc")  
// Invalid value used as weak map key  
weakMap.set("aaa", "cba")
```

- WeakMap常见的方法有四个：
 - set(key, value)：在Map中添加key、value，并且返回整个Map对象；
 - get(key)：根据key获取Map中的value；
 - has(key)：判断是否包括某一个key，返回Boolean类型；
 - delete(key)：根据key删除一个键值对，返回Boolean类型；

WeakMap的应用

■ 注意：WeakMap也是不能遍历的

□ 因为没有forEach方法，也不支持通过for of的方式进行遍历；

■ 那么我们的WeakMap有什么作用呢？

```
// WeakMap({key(对象): value}): key是一个对象, 弱引用
const targetMap = new WeakMap();
function getDep(target, key) {
  // 1. 根据对象(target) 取出对应的Map对象
  let depsMap = targetMap.get(target);
  if (!depsMap) {
    depsMap = new Map();
    targetMap.set(target, depsMap);
  }

  // 2. 取出具体的dep对象
  let dep = depsMap.get(key);
  if (!dep) {
    dep = new Dep();
    depsMap.set(key, dep);
  }
  return dep;
}
```