### let 和const命令

**let**声明的变量只在let命令所在的代码块内有效。

Var命令会发生变量提升的现象，即变量可以在声明之前使用，值为undefined。

// var 的情况

console.log(foo); // 输出undefined

var foo = 2;

var 没有块区域，仅有全局或函数区域，可以越过块区域（if for等）

let声明的变量不会发生变量提升现象，在声明之前调用变量，直接报错

// let 的情况

console.log(bar); // 报错ReferenceError

let bar = 2;

暂时性死区：

只要块级作用域内存在let命令，它所声明的变量就“绑定”（binding）这个区域，不再受外部的影响。

var tamp = 123;

if (true) {

tamp = 'abc'; // ReferenceError

let tmp;

}

上面代码中，存在全局变量tmp，但是块级作用域内let又声明了一个局部变量tmp，导致后者绑定这个块级作用域，所以在let声明变量前，对tmp赋值会报错。

总之，在代码块内，使用let命令声明变量之前，该变量都是不可用的。这在语法上，称为“暂时性死区”（temporal dead zone，简称 TDZ）。

if (true) {

// TDZ开始

tmp = 'abc'; // ReferenceError

console.log(tmp); // ReferenceError

let tmp; // TDZ结束

console.log(tmp); // undefined

tmp = 123;

console.log(tmp); // 123

}

在let命令声明变量tmp之前，都属于变量tmp的“死区”

**const**: 声明一个只读常量；一旦声明，常量值不可变。

const的作用于跟let相同，只在声明所在的块级作用域内有效。

本质：

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不得改动。对于简单类型的数据（数值、字符串、布尔值），值就保存在变量指向的那个内存地址，因此等同于常量。但对于复合类型的数据（主要是对象和数组），变量指向的内存地址，保存的只是一个指针，const只能保证这个指针是固定的，至于它指向的数据结构是不是可变的，就完全不能控制了。

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

### 数组、对象、字符串的解构赋值

数组的解构赋值：let [a, b, c] = [1, 2, 3];

默认值：let [foo = true] = [];

foo // true

ES6 内部使用严格相等运算符（===），判断一个位置是否有值。所以，只有当一个数组成员严格等于undefined，默认值才会生效。

|  |
| --- |
| let [x = 1] = [undefined];  x // 1  let [x = 1] = [null];  x // null |

对象的解构赋值：

第一种：变量名必须和属性名相同

let { bar, foo } = { foo: "aaa", bar: "bbb" };

foo // "aaa" 这里的foo 变量名必须和属性名相同

bar // "bbb"

第二种：变量名和属性名不同

let { foo: foo, bar: bar } = { foo: "aaa", bar: "bbb" };

也就是说，对象的解构赋值的内部机制，是先找到同名属性，然后再赋给对应的变量。真正被赋值的是后者，而不是前者。

let { foo: baz } = { foo: "aaa", bar: "bbb" };

baz // "aaa"

foo // error: foo is not defined

foo是匹配的模式，baz才是变量。真正被赋值的是变量baz，而不是模式foo。

即：冒号前面的是匹配模式，冒号后面的才是要被赋值的变量。

综合例子：

|  |
| --- |
| let obj = {  p: [  'Hello',  { y: 'World' }  ]  };  let { p, p: [x, { y }] } = obj; 前面的p是属性的同名便量，后面的p只是匹配模式  x // "Hello"  y // "World"  p // ["Hello", {y: "World"}] |

数组的本质是特殊的对象，可以对数组进行对象属性的解构：

|  |
| --- |
| let arr = [1, 2, 3];  let {0 : first, [arr.length - 1] : last} = arr;  first // 1  last // 3 |

对象的解构赋值，可以很方便地将现有对象的方法，赋值到某个变量。

let { log, sin, cos } = Math;

字符串的解构赋值：

字符串被转换成了一个类似数组的对象

const [a, b, c, d, e] = 'hello';

a // "h"

b // "e"

c // "l"

d // "l"

e // "o"

let {length : len} = 'hello'; 类似数组的对象都有一个length属性

len // 5

数值和布尔值的解构赋值：

如果等号右边是数值和布尔值，则会先转为对象。

let {toString: s} = 123;

s === Number.prototype.toString // true

用途：

1. 交换变量的值

[x,y] = [y,x];

1. 函数只能返回一个值，如果要返回多个值，只能将它们放在数组或对象里返回。有了解构赋值，取出这些值就非常方便。

|  |
| --- |
| // 返回一个数组  function example() {  return [1, 2, 3];  }  let [a, b, c] = example();  // 返回一个对象  function example() {  return {  foo: 1,  bar: 2  };  }  let { foo, bar } = example(); |

1. 函数参数的定义

|  |
| --- |
| // 参数是一组有次序的值  function f([x, y, z]) { ... }  f([1, 2, 3]);  // 参数是一组无次序的值  function f({x, y, z}) { ... }  f({z: 3, y: 2, x: 1}); |

1. 提取JSON数据

|  |
| --- |
| let jsonData = {  id: 42,  status: "OK",  data: [867, 5309]  };  let { id, status, data: number } = jsonData; |

### 字符串扩展

JavaScript 内部，字符以 UTF-16 的格式储存，每个字符固定为2个字节。对于那些需要4个字节储存的字符（Unicode 码点大于0xFFFF的字符），JavaScript 会认为它们是两个字符。

字符串遍历：

let text = String.fromCodePoint(0x20BB7);

//text只有一个字符，但是for 循环会当成两个字符。

for (let i = 0; i < text.length; i++) {

console.log(text[i]);

}

// " "

// " "

//for …of循环可以正确识别出一个字符。

for (let i of text) {

console.log(i);

}

// "𠮷"

‘hello’.indexOf(‘e’) 返回字符在字符串中的位置，找不到则返回-1；

includes(str,start)：返回布尔值，表示是否找到了参数字符串。

startsWith(str,start)：返回布尔值，表示参数字符串是否在原字符串的头部。

endsWith(str,start)：返回布尔值，表示参数字符串是否在原字符串的尾部。

start参数：开始搜索的位置。

endsWith()的start参数，针对前n个字符串（不包含第n个字符串）

s.repeat(n) 将字符串s重复n次

s. padStart(n,str) 字符串头部补全，n为补全后的长度，str为用来补全的字串

s. padEnd(n,str) 字符串尾部补全

matchAll() 返回一个正则表达式在当前字符串的所有匹配

模板字符串：

$('#result').append(**`**

There are <b>${basket.count}</b> items

in your basket, <em>${basket.onSale}</em>

are on sale!

**`**);

使用反引号``, ${}里可以放js变量。不用像以前那样用+号拼字符串。所有的空格和缩进都会保留在输出中。

array.map(function(value,index,[arr]){

//value为array里每次循环的值，index为索引，arr为当前元素属于的数组对象

})

### 正则

match()、replace()、search()和split()

先行断言：x只有在y前面才匹配，必须写成/x(?=y)/

先行否定断言：x只有不在y前面才匹配，必须写成/x(?!y)/

后行断言：x只有在y后面才匹配，必须写成/(?<=y)x/

后行否定断言：x只有不在y后面才匹配，必须写成/(?<!y)x/

以上括号中的匹配结果，不计入返回结果中

### 数值

Number.isFinite(value) 检查一个数值是否为有限的（finite），即不是Infinity

如果参数类型不是数值，Number.isFinite一律返回false

Number.isNaN()用来检查一个值是否为NaN

它们与传统的全局方法isFinite()和isNaN()的区别在于，传统方法先调用Number()将非数值的值转为数值，再进行判断，而这两个新方法只对数值有效，Number.isFinite()对于非数值一律返回false, Number.isNaN()只有对于NaN才返回true，非NaN一律返回false。

Number.parseInt() 跟parseInt()完全一致，为了逐步减少全局性方法

Number.parseFloat() 同上

Number.isInteger() 判断数值是否是整数

Math.trunc() 去除一个数的小数部分，返回整数部分 (会把参数转为数值，对于空值和无法截取整数的值，返回NaN)

Math.sigin() 判断一个数到底是正数、负数还是零

返回：

参数为正数，返回+1；

参数为负数，返回-1；

参数为 0，返回0；

参数为-0，返回-0;

其他值，返回NaN。

指数运算符： （\*\*）

2 \*\* 2 //4

2 \*\* 3 //8

指数赋值运算符 （\*\*=）

a \*\*= 2 // a = a\*a

### 数组

* 1. 扩展运算符：...

将一个数组转为用逗号分隔的参数序列。

该运算符主要用于函数调用。

用法1：

function add(x, y) {

return x + y;

}

const numbers = [4, 38];

add(...numbers) // 42

用法2：

// ES6 的写法

let arr1 = [0, 1, 2];

let arr2 = [3, 4, 5];

arr1.push(...arr2); //push后可以跟多个参数

复制数组：

const a1 = [1, 2];

const a2 = a1; //这种赋值，复制了指向底层数据结构的指针，而不是克隆一个全新的数组。

a2[0] = 2;

a1 // [2, 2]

使用扩展运算符复制数组：

const a1 = [1, 2];

const a2 = [...a1];

合并数组：

var arr1 = ['a', 'b'];

var arr2 = ['c'];

[...arr1, ...arr2]

将字符串转为数组：

[...'hello']

// [ "h", "e", "l", "l", "o" ]

* 1. Array.from()

Array.from()将：类似数组的对象（array-like object）和可遍历（iterable）的对象（包括 ES6 新增的数据结构 Set 和 Map）转为真正的数组；

实际应用中，常见的类似数组的对象是 DOM 操作返回的 NodeList 集合，以及函数内部的arguments对象。

注意：任何有length属性的对象，都可以通过Array.from方法转为数组

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

Array.from(arraylike, x => x\*2) //接受第二个参数，类似于数组的map方法，用来对每个元素处理，将处理后的值放入返回的数组中。

应用：取出DOM节点的文本内容

|  |
| --- |
| let spans = document.querySelectorAll('span.name');  // Array.from()  let names2 = Array.from(spans, s => s.textContent) |

* 1. Array.of()

将一组值，转换为数组

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

该方法主要目的，是弥补数组构造函数Array()的不足，因为参数个数不同，会导致Array()行为有差异；

|  |
| --- |
| Array() // [] 无参数返回空数组  Array(3) // [, , ,] 一个参数，表示数组长度  Array(3, 11, 8) // [3, 11, 8] 两个及以上参数，才返回由他们组成的数组 |

但Array.of() 完全返回传入的参数组成的数组；

* 1. copyWithin()

Array.prototype.copyWithin(target, start = 0, end = this.length)

三个参数：

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，表示倒数。

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，表示倒数。不包含end这个值。

[1, 2, 3, 4, 5,6].copyWithin(0, 2, 5) // [3, 4, 5, 4, 5, 6]

* 1. find()和findIndex()

返回第一个返回值为True的成员，若没有符合条件的成员，返回undefined。

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

find的回调函数，接受三个参数，依次是: 当前的值、当前的位置和原数组。

findIndex() 返回第一个符合条件的成员的位置，若都不符合条件，返回-1

find(fn,obj) 两个方法都接受第二个参数，一个对象，指定了匿名函数里this指向的对象。

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

* 1. fill()

用给定的值，填充数组。

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

参数依次是：要填充的值，从哪个位置开始，到哪个位置结束（不包含）

* 1. keys() values() entries()

都用于遍历数组，都返回一个遍历器对象，可以用for… of循环遍历。

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

|  |
| --- |
| 遍历迭代器时，获取的是数组结构  for (let [index, elem] of ['a', 'b'].entries()) {  console.log(index, elem);  }  // 0 "a"  // 1 "b"  for (let elem of ['a', 'b'].entries()) {  console.log(elem);  }  // [0, "a"]  // [1, "b"] |

* 1. includes(val,start)

数组是否包含给定的值 返回true或false

[1, 2, 3].includes(2) //true

接受第二个参数，表示起始搜索位置。

此方法之前一般用indexOf方法，检查数组是否包含某个值

|  |
| --- |
| if (arr.indexOf(el) !== -1) {  // ...  } |

注意：

1. Map 结构的has方法，是用来查找键名的
2. Set 结构的has方法，是用来查找值的

延伸：

in操作符

语法：prop in object

var trees = new Array("redwood", "bay", "cedar", "oak", "maple");

0 in trees // 返回true

3 in trees // 返回true

6 in trees // 返回false

"bay" in trees // 返回false (必须使用索引号,而不是数组元素的值)

// 自定义对象

var mycar = {make: "Honda", model: "Accord", year: 1998};

"make" in mycar // 返回true

"model" in mycar // 返回true

In 右侧是数组时，左侧必须是索引，

右侧是对象时，左侧是对象的属性值。

* 1. map

arr.map(x => x\*2) //返回处理后的新数组

* 1. filter

arr.filter( x => x/2===0 ) //若结果为true，则返回到新数组中，false过滤掉

* 1. reduce() 接收一个函数作为累加器，数组中的每个值（从左到右）开始缩减，最终计算为一个值。

array.reduce(function(total, currentValue, currentIndex, arr), initialValue)

|  |  |
| --- | --- |
| **参数** | **描述** |
| *function(total,currentValue, index,arr)* | 必需。用于执行每个数组元素的函数。 函数参数:   |  |  | | --- | --- | | **参数** | **描述** | | *total* | 必需。*初始值*, 或者计算结束后的返回值。 | | *currentValue* | 必需。当前元素 | | *currentIndex* | 可选。当前元素的索引 | | *arr* | 可选。当前元素所属的数组对象。 | |
| *initialValue* | 可选。传递给函数的初始值 |

|  |
| --- |
| var numbers = [65, 44, 12, 4];  function getSum(total, num,index,arr) {  console.log(total,num,index,arr);  return total + num;  }  function myFunction(item) {  console.log(numbers.reduce(getSum));  }  myFunction(numbers); //125 |

### 函数

* 1. 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（context）。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。

|  |
| --- |
| var x = 1;  function f(x, y = x) {  console.log(y);  }  f(2) // 2  上面代码中，参数y的默认值等于变量x。调用函数f时，参数形成一个单独的作用域。在这个作用域里面，默认值变量x指向第一个参数x，而不是全局变量x，所以输出是2。 |

* 1. Rest参数

用于获取函数多余的参数

|  |
| --- |
| function add(...values) { //此处的变量是个数组  let sum = 0;  for (var val of values) {  sum += val;  }  return sum;  }  add(2, 5, 3) // 10  function push(array, ...items) {  items.forEach(function(item) {  array.push(item);  console.log(item);  });  }  var a = [];  push(a, 1, 2, 3) |

注意：rest参数后不能有其他参数，否则报错

* 1. => 定义函数

|  |
| --- |
| var sum = (num1, num2) => num1 + num2;  // 等同于  var sum = function(num1, num2) {  return num1 + num2;  }; |

若代码块部分多余一条语句，就要用大括号括起来，并使用return返回。

由于大括号被解释为代码块，所以如果箭头函数直接返回一个对象，必须在对象外面加上括号，否则会报错。

|  |
| --- |
| // 报错  let getTempItem = id => { id: id, name: "Temp" };  // 不报错  let getTempItem = id => ({ id: id, name: "Temp" }); |

注意：箭头函数体内的this对象，是定义时所在的对象，不是使用时所在的对象。

### 对象

* 1. 属性的简洁表达

可以直接写入变量和函数，作为对象的属性和方法。

var foo = 'bar';

var obj = {foo}; // {foo: "bar"}

等同于：var obj = {foo:”bar”}

方法简写：

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

例子：

|  |
| --- |
| let birth = '2000/01/01';  const Person = {  name: '张三',  //等同于birth: birth  birth,  // 等同于hello: function ()...  hello() { console.log('我的名字是', this.name); }  }; |

* 1. Object.assign 对象合并

Object.assign(target, source1, source2);

参数：目标对象（target）、源对象（source）

若源对象和目标对象有同名属性，后面的会覆盖前面的。

注意：1、Object.assign 使用的是浅拷贝，是对源对象的引用。

2、同名属性，直接替换，而不是添加或合并。

|  |
| --- |
| const target = { a: { b: 'c', d: 'e' } }  const source = { a: { b: 'hello' } }  Object.assign(target, source)  // { a: { b: 'hello' } } |

* 1. 属性的遍历

1. for …in

循环遍历对象自身的和继承的可枚举属性。（循环的是属性）

2、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；

* 1. 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 } } |

* 1. Object.setPrototypeOf()

设置对象的prototype对象，返回第一个参数对象本身

Object.setPrototypeOf(object, prototype)

Object.getPrototypeOf() 获取对象的原型对象。

* 1. super关键字

super关键字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);  obj.find() // "hello" |

super只能用在对象的方法当中，且只有对象方法的简写法可以让 JavaScript 引擎确认，定义的是对象的方法。

|  |
| --- |
| // 报错  const obj = {  foo: super.foo  }  // 报错  const obj = {  foo: () => super.foo  }  // 报错  const obj = {  foo: function () {  return super.foo  }  } |

* 1. Object.keys() 返回对象属性的键名的数组；

Object.values() 返回对象属性的键值的数组；

Object.entries() 返回对象属性的键值对数组；

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

* 1. …对象的扩展运算符

对象的扩展运算符（...）用于取出参数对象的所有可遍历属性，拷贝到当前对象之中。

let z = { a: 3, b: 4 };

let n = { ...z };

n // { a: 3, b: 4 }

合并对象：

let ab = { ...a, ...b };

// 等同于

let ab = Object.assign({}, a, b);

### 易混淆

* 1. for( variable in object){…} 遍历可枚举属性,包括继承的属性。

variable:

在每次迭代时，将不同的属性名分配给变量。

object:

被迭代枚举其属性的对象。

它以任意序迭代一个对象的属性,所以不适宜遍历数组，它可能不按次序访问元素，遍历数组最好用 for循环，forEach()，或for…of

结论： 在循环对象属性的时候，使用for...in,

在遍历数组的时候的时候使用for...of

* 1. obj.hasOwnProperty(prop) 判断prop属性是否是obj自有属性，非继承属性。
  2. for (variable of iterable) {…} 迭代值

在可迭代对象（包括 Array，Map，Set，String，TypedArray，arguments 对象等等）上创建一个迭代循环

参数：

variable：

在每次迭代中，将不同属性的值分配给变量。

Iterable：

被迭代枚举其属性的对象。

迭代Array：

|  |
| --- |
| let iterable = [10, 20, 30];  for (let value of iterable) {  value += 1;  console.log(value);  }  // 10  // 20  // 30 |

迭代String：

|  |
| --- |
| let iterable = "boo";  for (let value of iterable) {  console.log(value);  }  // "b"  // "o"  // "o" |

迭代Map：

|  |
| --- |
| let iterable = new Map([["a", 1], ["b", 2], ["c", 3]]);  for (let entry of iterable) {  console.log(entry);  }  // ["a", 1]  // ["b", 2]  // ["c", 3] |

迭代Set：

|  |
| --- |
| let iterable = new Set([1, 1, 2, 2, 3, 3]);  for (let value of iterable) {  console.log(value);  }  // 1  // 2  // 3 |

关闭迭代器：

用break, continue, throw 或return终止

### Symbol

### Set

Set类似于数组，但成员的值都是唯一的。

|  |
| --- |
| const s = new Set(); //Set本身就是一个构造函数，用来生成set数据结构。  const set = new Set([1, 2, 3, 4, 4]); //Set可接受数组参数（或者具有 iterable 接口的其他数据结构），用来初始化 |

去除数组重复项的一个方法：

[...new Set(arr)];

Array.from(new Set(arr))

Set里，NaN等于自身，即NaN等于NaN，不会重复添加NaN

对象总是不相等的：

|  |
| --- |
| let set = new Set();  set.add({});  set.size // 1  set.add({});  set.size // 2 |

Set属性：

size: 返回成员总数

set方法：

add(value):添加某个值，返回set结构本身

delete(value):删除某个值，返回true or false

has(value):返回true or false，表示该值是否为set成员

clear(value):清除所有成员，无返回值；

判断是否包含一个键：

|  |
| --- |
| // 对象的写法  const properties = {  'width': 1,  'height': 1  };  if (properties[someName]) {  // do something  }  // Set的写法  const properties = new Set();  properties.add('width');  if (properties.has(someName)) {  // do something  } |

* 1. set遍历

了解：

keys() 返回键名的遍历器

values() 返回键值的遍历器

entries() 包括键名和键值的遍历器，每次输出一个数组。

Set结构没有键名，所以键名和键值相同。

所以遍历时候，键名并不是索引，而是与键值相同

|  |
| --- |
| var arr = new Set(['a','b','c']);  arr.forEach(function(value,index){  console.log(value,index);  })  //a a  //b b  //c c  for(var item of arr){  console.log(item)  }  //a  //b  //c |

扩展运算符（...）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']);  let arr = [...set];  // ['red', 'green', 'blue'] |

|  |
| --- |
| let a = new Set([1, 2, 3]);  let b = new Set([4, 3, 2]);  合并set：  let union = new Set([...a, ...b]);  交集：  let intersect = new Set([...a].filter(x => b.has(x)));  差集：  let intersect = new Set([...a].filter(x => !b.has(x))); |

### Map

Map结构提供了“值-值”的对应。因为Object对象只能用字符串作为键名，Map允许可以使用各种类型的值作为键。

* 1. new Map()

new Map(array) 可接受数组作为参数

|  |
| --- |
| const items = [  ['name', '张三'],  ['title', 'Author']  ];  new Map(items) |

new Map(set) 接受set结构作为参数

|  |
| --- |
| const set = new Set([  ['foo', 1],  ['bar', 2]  ]);  const m1 = new Map(set); |

new Map(map) 接受map结构作为参数

|  |
| --- |
| const m2 = new Map([['baz', 3]]);  const m3 = new Map(m2); |

总之，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都可以当作Map构造函数的参数。

map 结构键名不能重复，后面的会覆盖前面的。

且只有对同一个对象的引用，Map 结构才将其视为同一个键

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

如果 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，

|  |
| --- |
| var map = new Map();  var a = ['a'];  map.set(['a'], 555);  map.set(['a'], 333);  // {Array(1) => 555, Array(1) => 333} //此处两个[‘a’]指向两个不同的内存地址，所以不会覆盖。  var b = ['b'];  map.set(b,555);  map.set(b,333);  //  {Array(1) => 333} //此处b是对同一个内存的引用，会覆盖  map.set(1,555)  map.set(1,333)  //{1=>333} //简单类型的键名，会覆盖 |

* 1. 属性和方法：

size : Map结构成员总数

m.set(key,value) 返回当前的Map对象，so可以用链式写法；

m.get(key) 返回key对应的值

m.has(key) 某个键在不在map中，返回布尔值

m.delete(key) 删除某个键，返回true或false

m.clear() 清除所有成员，无返回值

* 1. 遍历

m.keys() 返回所有键名的遍历器

m.values() 返回所有键值的遍历器

m.entries() 返回所有成员的遍历器

|  |
| --- |
| for (let [key, value] of map.entries()) {  console.log(key, value);  }  或：  for (let item of map.entries()) {  console.log(item[0], item[1]);  } |

或者不用entries，

直接：

|  |
| --- |
| for (let [key, value] of map) {  console.log(key, value);  } |

map转数组：

[…map]

之后可以使用数组的map()和filter()方法，Map结构并没有这两个方法。

forEach() ： 遍历

|  |
| --- |
| var obj = {name:'yang'};  map.forEach(function(value , key , map){  console.log(this.name+' : '+value)  } , obj)  接受第二个参数，一个对象，回调函数里this对应的对象 |

* 1. 几种数组遍历比较

for循环：

|  |
| --- |
| for (var index = 0; index < myArray.length; index++) {  console.log(myArray[index]);  } |

书写比较麻烦

forEach:

|  |
| --- |
| myArray.forEach(function (value) {  console.log(value);  }); |

数组内置的forEach()循环，不能使用break或return跳出循环

for…in：

|  |
| --- |
| for (var index in myArray) {  console.log(myArray[index]);  } |

for…in缺点：

数组的键名是数字，但是for…in是以字符串为键名“0”，“1”；

不仅数字键名，还会遍历其他手动添加的键，包括原型链上的键；

某些情况，for…in会以任意顺序遍历键名

所以，for…in主要为了对象设计的，不适用于遍历数组；

for…of:

不同于forEach,可以与break、continue和return配合使用