### 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 |

* 1. splice

通过索引删除数组元素并添加元素，且会改变原数组。

返回：被删除的元素组成的数组

array.splice(start [ , deleteCount [,item…]])

start:从start索引位置开始删除元素

deleteCount:删除元素数量

item:在删除元素的位置添加的元素

* 1. slice

arr.slice(start,end) 从索引start处截取至end处，（不含end）

start: 可选 截取的开始位置，不选则从0开始

end: 可选 截取的结束位置，不含该位置的元素

返回：返回截取元素组成的新数组。

slice不改变原数组，只是浅拷贝

### 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。不会拷贝继承的属性

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' } } |

注：浅拷贝和深拷贝的区别：浅拷贝只是拷贝对象独有的属性，不含继承的属性，而深拷贝是也拷贝继承的属性！

* 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 |

### 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);  }); |

注意：

数组内置的forEach()循环，不能使用break或return跳出循环

\_.each ：

可以使用return;跳过本次循环，相当于continue;

使用return false;跳出循环，相当于break；

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

### Promise

#### 含义

Promise 就是一个容器，里面保存着某个未来才会结束的事件的结果。

特点：

1）：对象的状态不受外界影响。Promise对象代表一个异步操作，有三种状态：pending（进行中）、fulfilled（已成功）和rejected（已失败）；只有异步操作的结果，可以决定当前是哪种状态，其他任何操作都无法改变这个状态。

2）：状态一旦改变，就不会再变，任何时候都能得到这个结果。

Promise对象的状态改变，只有两种可能：从pending变为fulfilled和从pending变为rejected。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果，这时就称为 resolved（已定型）；

3）：无法取消Promise，一旦新建它就会立即执行，无法中途取消。

#### 基础用法

Promise对象是一个构造函数，用来生成Promise实例。

|  |
| --- |
| const promise = new Promise(function(resolve, reject) {  // ... some code  if (/\* 异步操作成功 \*/){  resolve(value);  } else {  reject(error);  }  }); |

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。它们是两个函数，由 JavaScript 引擎提供，不用自己部署。

resolve函数的作用是，将Promise对象的状态从“未完成”变为“成功”，在异步操作成功时调用，并将异步操作的结果，作为参数传递出去；

reject函数的作用是，将Promise对象的状态从“未完成”变为“失败”， 在异步操作失败时调用，并将异步操作报出的错误，作为参数传递出去。

Promise实例生成以后，可以用then方法分别指定resolved状态和rejected状态的回调函数。

|  |
| --- |
| promise.then(function(value) {  // success  }, function(error) {  // failure  }); |

then方法可以接受两个回调函数作为参数。第一个回调函数是Promise对象的状态变为resolved时调用，第二个回调函数是Promise对象的状态变为rejected时调用。其中，第二个函数是可选的，不一定要提供。两个函数都接受Promise对象传出的值作为参数。

Promise新建后就立即执行：

|  |
| --- |
| let promise = new Promise(function(resolve, reject) {  console.log('Promise');  resolve();  });  promise.then(function() {  console.log('resolved.');  });  console.log('Hi!');  // Promise  // Hi!  // resolved |

resolve的参数除了正常值之外还可以是另一个Promise:

|  |
| --- |
| const p1 = new Promise(function (resolve, reject) {  setTimeout(() => reject(new Error('fail')), 3000)  })  const p2 = new Promise(function (resolve, reject) {  setTimeout(() => resolve(p1), 1000)  })  p2  .then(result => console.log(result))  .catch(error => console.log(error))  // Error: fail |

上面代码中，p1是一个 Promise，3 秒之后变为rejected。p2的状态在 1 秒之后改变，resolve方法返回的是p1。由于p2返回的是另一个 Promise，导致p2自己的状态无效了，由p1的状态决定p2的状态。所以，后面的then语句都变成针对后者（p1）。又过了 2 秒，p1变为rejected，导致触发catch方法指定的回调函数。

resolve和reject并不会终结Promise的参数函数的执行：

|  |
| --- |
| new Promise((resolve, reject) => {  resolve(1);  console.log(2);  }).then(r => {  console.log(r);  });  // 2  // 1 |

调用resolve(1)以后，后面的console.log(2)还是会执行，并且会首先打印出来。这是因为立即 resolved 的 Promise 是在本轮事件循环的末尾执行，总是晚于本轮循环的同步任务。

|  |
| --- |
| new Promise((resolve, reject) => {  return resolve(1);  // 后面的语句不会执行  console.log(2);  }) |

加上return后面的语句就不会执行了。

#### Promise.prototype.then()

then方法是定义在原型对象Promise.prototype上的。

then方法的第一个参数是resolved状态的回调函数，第二个参数（可选）是rejected状态的回调函数。

then方法返回的是一个新的Promise实例（注意，不是原来那个Promise实例）。因此可以采用链式写法，即then方法后面再调用另一个then方法。

#### Promise.prototype.catch()

Promise.prototype.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。

|  |
| --- |
| p.then((val) => console.log('fulfilled:', val))  .catch((err) => console.log('rejected', err));  // 等同于  p.then((val) => console.log('fulfilled:', val))  .then(null, (err) => console.log("rejected:", err)); |

如果 Promise 状态已经变成resolved，再抛出错误是无效的。

|  |
| --- |
| const promise = new Promise(function(resolve, reject) {  resolve('ok');  throw new Error('test');  });  promise  .then(function(value) { console.log(value) })  .catch(function(error) { console.log(error) });  // ok |

上面代码中，Promise 在resolve语句后面，再抛出错误，不会被捕获，等于没有抛出。因为 Promise 的状态一旦改变，就永久保持该状态，不会再变了。

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。也就是说，错误总是会被下一个catch语句捕获。

一般来说，不要在then方法里面定义 Reject 状态的回调函数（即then的第二个参数），总是使用catch方法。

|  |
| --- |
| // bad  promise  .then(function(data) {  // success  }, function(err) {  // error  });  // good  promise  .then(function(data) { //cb  // success  })  .catch(function(err) {  // error  }); |

上面代码中，第二种写法要好于第一种写法，理由是第二种写法可以捕获前面then方法执行中的错误。

如果没有使用catch方法指定错误处理的回调函数，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);  });  };  someAsyncThing().then(function() {  console.log('everything is great');  });  setTimeout(() => { console.log(123) }, 2000);  // Uncaught (in promise) ReferenceError: x is not defined  // 123 |

someAsyncThing函数产生的 Promise 对象，内部有语法错误。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、终止脚本执行，2 秒之后还是会输出123。

这就是说，Promise 内部的错误不会影响到 Promise 外部的代码，通俗的说法就是“Promise 会吃掉错误”。

tips: 这个脚本放在服务器执行，退出码就是0（即表示执行成功）。不过，Node 有一个unhandledRejection事件，专门监听未捕获的reject错误，上面的脚本会触发这个事件的监听函数，可以在监听函数里面抛出错误。

再看下面列子：

|  |
| --- |
| const promise = new Promise(function (resolve, reject) {  resolve('ok');  setTimeout(function () { throw new Error('test') }, 0)  });  promise.then(function (value) { console.log(value) });  // ok  // Uncaught Error: test |

Promise 指定在下一轮“事件循环”再抛出错误（setTimeout(func,0) 在下一轮时间循环执行），那时Promise运行已经结束了，所以这个错误是在Promise函数体外抛出的，会冒泡到最外层，成了为被捕获的错误。

一般总是建议，Promise 对象后面要跟catch方法，这样可以处理 Promise 内部发生的错误。catch方法返回的还是一个 Promise 对象，因此后面还可以接着调用then方法。

#### Promise.prototype.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。

finally方法的回调函数不接受任何参数，这意味着没有办法知道，前面的 Promise 状态到底是fulfilled还是rejected。这表明，finally方法里面的操作，应该是与状态无关的，不依赖于 Promise 的执行结果。

具体实现：

|  |
| --- |
| Promise.prototype.finally = function (callback) {  let P = this.constructor;  return this.then(  value => P.resolve(callback()).then(() => value),  reason => P.resolve(callback()).then(() => { throw reason })  );  }; |

不管前面的 Promise 是fulfilled还是rejected，都会执行回调函数callback。

从上面的实现还可以看到，finally方法总是会返回原来的值。

#### Promise.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]); |

Promise.all方法接受一个数组作为参数，p1、p2、p3都是 Promise 实例，如果不是，就会先调用下面讲到的Promise.resolve方法，将参数转为 Promise 实例，再进一步处理。

（Promise.all方法的参数可以不是数组，但必须具有 Iterator 接口，且返回的每个成员都是 Promise 实例。）

p的状态由p1、p2、p3决定，分成两种情况。

（1）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。

（2）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。

|  |
| --- |
| // 生成一个Promise对象的数组  const promises = [2, 3, 5, 7, 11, 13].map(function (id) {  return getJSON('/post/' + id + ".json");  });  Promise.all(promises).then(function (posts) {  // ...  }).catch(function(reason){  // ...  }); |

promises是包含 6 个 Promise 实例的数组，只有这 6 个实例的状态都变成fulfilled，或者其中有一个变为rejected，才会调用Promise.all方法后面的回调函数。

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] |

p1会resolved，p2首先会rejected，但是p2有自己的catch方法，该方法返回的是一个新的 Promise 实例，p2指向的实际上是这个实例。该实例执行完catch方法后，也会变成resolved，导致Promise.all()方法参数里面的两个实例都会resolved，因此会调用then方法指定的回调函数，而不会调用catch方法指定的回调函数。

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

#### Promise.race()

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

字面意思，多个Promise实例谁先改变状态就采用谁的状态。

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

上面代码中，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。那个率先改变的 Promise 实例的返回值，就传递给p的回调函数。

实例：

|  |
| --- |
| 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); |

上面代码，若5s内fetch没有返回，则抛出“request timeout”错误，变量p的状态就会变为rejected，从而触发catch方法；

#### Promise.resolve()

Promise.resolve将现有对象转为 Promise 对象；

等价于：

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

Promise.resolve方法的参数分成四种情况：

（1）参数是一个 Promise 实例

如果参数是 Promise 实例，那么Promise.resolve将不做任何修改、原封不动地返回这个实例。

（2）参数是一个thenable对象

thenable对象指的是具有then方法的对象，比如下面这个对象。

|  |
| --- |
| let thenable = {  then: function(resolve, reject) {  resolve(42);  }  }; |

Promise.resolve方法会将这个对象转为 Promise 对象，然后就立即执行thenable对象的then方法。

|  |
| --- |
| let thenable = {  then: function(resolve, reject) {  resolve(42);  }  };  let p1 = Promise.resolve(thenable);  p1.then(function(value) {  console.log(value); // 42  }); |

上面代码中，thenable对象的then方法执行后，对象p1的状态就变为resolved，从而立即执行最后那个then方法指定的回调函数，输出 42。

（3）参数不是具有then方法的对象，或根本就不是对象

如果参数是一个原始值，或者是一个不具有then方法的对象，则Promise.resolve方法返回一个新的 Promise 对象，状态为resolved。

|  |
| --- |
| const p = Promise.resolve('Hello');  p.then(function (s){  console.log(s)  });  // Hello |

由于字符串Hello不属于异步操作（判断方法是字符串对象不具有 then 方法），返回 Promise 实例的状态从一生成就是resolved，所以回调函数会立即执行。Promise.resolve方法的参数，会同时传给回调函数。

（4）不带有任何参数

Promise.resolve方法允许调用时不带参数，直接返回一个resolved状态的 Promise 对象。

注意：立即resolve的 Promise 对象，是在本轮“事件循环”（event loop）的结束时，而不是在下一轮“事件循环”的开始时。

|  |
| --- |
| setTimeout(function () {  console.log('three'); //下一轮事件循环的开始执行  }, 0);  Promise.resolve().then(function () {  console.log('two'); // 本轮事件循环的结尾执行  });  console.log('one');// 立即执行  // one  // two  // three |

#### Promise.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 |

注意，Promise.reject()方法的参数，会原封不动地作为reject的理由，变成后续方法的参数。这一点与Promise.resolve方法不一致。

### Iterator和for…of循环

#### 遍历器概念

任何数据结构只要部署 Iterator 接口，就可以完成遍历操作（即依次处理该数据结构的所有成员）。

Iterator 的作用有三个：一是为各种数据结构，提供一个统一的、简便的访问接口；二是使得数据结构的成员能够按某种次序排列；三是 ES6 创造了一种新的遍历命令for...of循环，Iterator 接口主要供for...of消费。

Iterator 的遍历过程是这样的。

（1）创建一个指针对象，指向当前数据结构的起始位置。也就是说，遍历器对象本质上，就是一个指针对象。

（2）第一次调用指针对象的next方法，可以将指针指向数据结构的第一个成员。

（3）第二次调用指针对象的next方法，指针就指向数据结构的第二个成员。

（4）不断调用指针对象的next方法，直到它指向数据结构的结束位置。

每次调用next,就会返回数据结构当前成员的信息，具体就是返回包含value和done两个属性的对象。其中，value属性是当前成员的值，done属性是个布尔值，表示遍历是否结束。

模拟next方法：

|  |
| --- |
| var it = makeIterator(['a', 'b']);  it.next() // { value: "a", done: false }  it.next() // { value: "b", done: false }  it.next() // { value: undefined, done: true }  function makeIterator(array) {  var nextIndex = 0;  return {  next: function() {  return nextIndex < array.length ?  {value: array[nextIndex++], done: false} :  {value: undefined, done: true};  }  };  } |

上面it就是遍历器对象（指针对象）

对于遍历器对象来说，done: false和value: undefined属性都是可以省略的，因此上面的makeIterator函数可以简写成下面的形式：

|  |
| --- |
| function makeIterator(array) {  var nextIndex = 0;  return {  next: function() {  return nextIndex < array.length ?  {value: array[nextIndex++]} :  {done: true};  }  };  } |

#### 默认 Iterator 接口

当使用for...of循环遍历某种数据结构时，该循环会自动去寻找 Iterator 接口。

一种数据结构只要部署了 Iterator 接口，我们就称这种数据结构是“可遍历的”（iterable）。

ES6 规定，默认的 Iterator 接口部署在数据结构的Symbol.iterator属性，或者说，一个数据结构只要具有Symbol.iterator属性，就可以认为是“可遍历的”（iterable）。

Symbol.iterator属性本身是一个函数，就是当前数据结构默认的遍历器生成函数。执行这个函数，就会返回一个遍历器。调用之后才会生成一个遍历器。

|  |
| --- |
| const obj = {  [Symbol.iterator] : function () {  return {  next: function () {  return {  value: 1,  done: true  };  }  };  }  };  let iterator = obj[Symbol.iterator]();  console.log(iterator.next());  *// { value: 1, done: true }* |

ES6 的有些数据结构原生具备 Iterator 接口（比如数组），即不用任何处理，就可以被for...of循环遍历。原因在于，这些数据结构原生部署了Symbol.iterator属性（详见下文），另外一些数据结构没有（比如对象）。凡是部署了Symbol.iterator属性的数据结构，就称为部署了遍历器接口。调用这个接口，就会返回一个遍历器对象。

原生具备 Iterator 接口的数据结构如下。

Array

Map

Set

String

TypedArray

函数的 arguments 对象

NodeList 对象

下面的例子是数组的Symbol.iterator属性:

|  |
| --- |
| let arr = ['a', 'b', 'c'];  let iter = arr[Symbol.iterator]();  iter.next() // { value: 'a', done: false }  iter.next() // { value: 'b', done: false }  iter.next() // { value: 'c', done: false }  iter.next() // { value: undefined, done: true } |

一个类部署 Iterator 接口的写法, Symbol.iterator属性对应一个函数，执行后返回当前对象的遍历器对象。

|  |
| --- |
| class RangeIterator {  constructor(start, stop) {  this.value = start;  this.stop = stop;  }  [Symbol.iterator]() { return this; }  next() {  var value = this.value;  if (value < this.stop) {  this.value++;  return {done: false, value: value};  }  return {done: true, value: undefined};  }  }  function range(start, stop) {  return new RangeIterator(start, stop);  }  for (var value of range(0, 3)) {  console.log(value); // 0, 1, 2  } |

下面是一个为对象添加Iterator接口的列子

|  |
| --- |
| let obj = {  data: [ 'hello', 'world' ],  [Symbol.iterator]() {  const self = this;  let index = 0;  return {  next() {  if (index < self.data.length) {  return {  value: self.data[index++],  done: false  };  } else {  return { value: undefined, done: true };  }  }  };  }  };  for(let i of obj){  console.log(i);  }  // hello  // world |

类似数组的对象，即存在数值键名（0，1，2…）和length属性，部署terator接口，有个简单方法：直接引用数组的Iterator接口。

|  |
| --- |
| let iterable = {  0: 'a',  1: 'b',  2: 'c',  length: 3,  [Symbol.iterator]: Array.prototype[Symbol.iterator]  };  for (let item of iterable) {  console.log(item); // 'a', 'b', 'c'  } |

注意，普通对象部署数组的Iterator接口，并无效果。

|  |
| --- |
| let iterable = {  a: 'a',  b: 'b',  c: 'c',  length: 3,  [Symbol.iterator]: Array.prototype[Symbol.iterator]  };  for (let item of iterable) {  console.log(item); // undefined, undefined, undefined  } |

#### 调用 Iterator 接口的场合

1）解构赋值

对数组和 Set 结构进行解构赋值时，会默认调用Symbol.iterator方法。

|  |
| --- |
| let set = new Set().add('a').add('b').add('c');  let [x,y] = set;  // x='a'; y='b'  let [first, ...rest] = set;  // first='a'; rest=['b','c']; |

（2）扩展运算符

扩展运算符（...）也会调用默认的 Iterator 接口。

|  |
| --- |
| // 例一  var str = 'hello';  [...str] // ['h','e','l','l','o']  // 例二  let arr = ['b', 'c'];  ['a', ...arr, 'd']  // ['a', 'b', 'c', 'd'] |

可以将任何部署了 Iterator 接口的数据结构，转为数组。

|  |
| --- |
| let arr = [...iterable]; |

4）其他场合

由于数组的遍历会调用遍历器接口，所以任何接受数组作为参数的场合，其实都调用了遍历器接口。下面是一些例子。

for...of

Array.from()

Map(), Set(), WeakMap(), WeakSet()（比如new Map([['a',1],['b',2]])）

Promise.all()

Promise.race()

#### 字符串的Iterator接口

字符串是一个类似数组的对象，也原生具有Iterator接口。

|  |
| --- |
| let str = 'hello';  console.log(typeof str[Symbol.iterator]);// function  let ite = str[Symbol.iterator]();  console.log(ite.next());// { value: "h", done: false }  console.log(ite.next());// { value: "e", done: false }  console.log(ite.next());// { value: "l", done: false } |

可以覆盖原生的Symbol.iterator方法，达到修改遍历器行为的目的：

|  |
| --- |
| var str = new String("hi");  [...str] // ["h", "i"]  str[Symbol.iterator] = function() {  return {  next: function() {  if (this.\_first) {  this.\_first = false;  return { value: "bye", done: false };  } else {  return { done: true };  }  },  \_first: true  };  };  [...str] // ["bye"]  str // "hi" |

#### 遍历器对象的 return()，throw()

遍历器对象除了具有next方法，还可以具有return方法和throw方法。如果你自己写遍历器对象生成函数，那么next方法是必须部署的，return方法和throw方法是否部署是可选的。

return方法的使用场合是，如果for...of循环提前退出（通常是因为出错，或者有break语句），就会调用return方法。如果一个对象在完成遍历前，需要清理或释放资源，就可以部署return方法。

|  |
| --- |
| function readLinesSync(file) {  return {  [Symbol.iterator]() {  return {  next() {  return { done: false };  },  return() {  file.close();  return { done: true };  }  };  },  };  } |

下面两种情况都会触发return方法：

|  |
| --- |
| // 情况一  for (let line of readLinesSync(fileName)) {  console.log(line);  break;  }  // 情况二  for (let line of readLinesSync(fileName)) {  console.log(line);  throw new Error();  } |

情况一会在打印第一行之后执行return方法，关闭文件；

情况二会在执行return方法关闭文件之后，再抛出错误；

注意：return方法必须返回一个对象；

#### for…of循环

for...of循环内部调用的是数据结构的Symbol.iterator方法。

for...of循环可以使用的范围包括数组、Set 和 Map 结构、某些类似数组的对象（比如arguments对象、DOM NodeList 对象）、后文的 Generator 对象，以及字符串。

|  |
| --- |
| let arr = ['a', 'b', 'c', 'd'];  for(let i of arr[Symbol.iterator]()){  console.log(i);  }  for(let i of arr){  console.log(i);  }  // 结果都是a b c d |

of 后面可以是以上结构，也可以是个迭代器对象（Iterator）

#### Set和Map结构

Set和Map结构原生具有Iterator接口，可以直接使用for…of循环。

|  |
| --- |
| var engines = new Set(["Gecko", "Trident", "Webkit", "Webkit"]);  for (var e of engines) {  console.log(e);  }  // Gecko  // Trident  // Webkit  var es6 = new Map();  es6.set("edition", 6);  es6.set("committee", "TC39");  es6.set("standard", "ECMA-262");  for (var [name, value] of es6) {  console.log(name + ": " + value);  }  // edition: 6  // committee: TC39  // standard: ECMA-262 |

首先，遍历的顺序是按照各个成员被添加进数据结构的顺序。

其次，Set 结构遍历时，返回的是一个值，而 Map 结构遍历时，返回的是一个数组，该数组的两个成员分别为当前 Map 成员的键名和键值。

|  |
| --- |
| let map = new Map().set('a', 1).set('b', 2);  for (let pair of map) {  console.log(pair);  }  // ['a', 1]  // ['b', 2]  for (let [key, value] of map) {  console.log(key + ' : ' + value);  }  // a : 1  // b : 2 |

#### 计算生成的数据结构

* entries() 返回一个遍历器对象（Iterator），用来遍历[键名, 键值]组成的数组。对于数组，键名就是索引值；对于 Set，键名与键值相同。Map 结构的 Iterator 接口，默认就是调用entries方法。
* keys() 返回一个遍历器对象，用来遍历所有的键名。
* values() 返回一个遍历器对象，用来遍历所有的键值。

#### 对象

对于普通对象，for…of结构不能直接使用，会报错，必须部署了Iterator接口后才能使用。

可以这样实现遍历：

|  |
| --- |
| let obj = {  name:'xx',  age: 28  };  for (let i of Object.values(obj)){  console.log(i);  }  for (let i of Object.keys(obj)){  console.log(i);  }  for (let [k,v] of Object.entries(obj)){  console.log(k,v);  } |

### Generator

#### 基本概念

调用 Generator 函数，返回一个遍历器对象，代表 Generator 函数的内部指针。以后，每次调用遍历器对象的next方法，就会返回一个有着value和done两个属性的对象。value属性表示当前的内部状态的值，是yield表达式后面那个表达式的值；done属性是一个布尔值，表示是否遍历结束。

|  |
| --- |
| function\* helloWorldGenerator() {  yield 'hello';  yield 'world';  return 'ending';  }    var hw = helloWorldGenerator();  console.log(hw.next());*//{ value: 'hello', done: false }*  console.log(hw.next());*//{ value: 'world', done: false }*  console.log(hw.next());*//{ value: 'ending', done: false }*  console.log(hw.next());*//{ value: undefined, done: true }* |

#### yield表达式

由于 Generator 函数返回的遍历器对象，只有调用next方法才会遍历下一个内部状态，所以其实提供了一种可以暂停执行的函数。yield表达式就是暂停标志。

遍历器对象的next方法的运行逻辑如下。

（1）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。

（2）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。

（3）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。

（4）如果该函数没有return语句，则返回的对象的value属性值为undefined。

Generator 函数可以不用yield表达式，这时就变成了一个单纯的暂缓执行函数。

|  |
| --- |
| function\* f() {  console.log('执行了！')  }    var generator = f();    console.log(generator.next()); |

yield表达式只能用在Generator函数里，用在其他地方都会报错。用在普通函数里就会报错：

|  |
| --- |
| (function (){  yield 1;  })()  // SyntaxError: Unexpected number |

错误例子：

|  |
| --- |
| var arr = [1, [[2, 3], 4], [5, 6]];  var flat = function\* (a) {  a.forEach(function (item) {  if (typeof item !== 'number') {  yield\* flat(item);  } else {  yield item;  }  });  };  for (var f of flat(arr)){  console.log(f);  } |

forEach的参数是一个普通函数，里面使用了yield会报错，这里可以改用for循环

|  |
| --- |
| var arr = [1, [[2, 3], 4], [5, 6]];  var flat = function\* (a) {  var length = a.length;  for (var i = 0; i < length; i++) {  var item = a[i];  if (typeof item !== 'number') {  yield\* flat(item);  } else {  yield item;  }  }  };  for (var f of flat(arr)) {  console.log(f);  }  // 1, 2, 3, 4, 5, 6 |

#### 与 Iterator 接口的关系

任意一个对象的Symbol.iterator方法，等于该对象的遍历器生成函数，调用该函数会返回该对象的一个遍历器对象。

由于 Generator 函数就是遍历器生成函数，因此可以把 Generator 赋值给对象的Symbol.iterator属性，从而使得该对象具有 Iterator 接口。

|  |
| --- |
| var myIterable = {};  myIterable[Symbol.iterator] = function\* () {  yield 1;  yield 2;  yield 3;  };  [...myIterable] // [1, 2, 3] |

Generator 函数执行后，返回一个遍历器对象，该对象本身也具有Symbol.iterator属性，执行后返回自身。

|  |
| --- |
| function\* gen(){  // some code  }  var g = gen();  g[Symbol.iterator]() === g  // true |

#### next方法参数

yield表达式本身没有返回值，或者说总是返回undefined。

next方法可以带一个参数，该参数就会被当做上一个yield表达式的返回值。

|  |
| --- |
| function\* f() {  for(var i = 0; true; i++) {  var reset = yield i;  if(reset) { i = -1; }  }  }  var g = f();  g.next() // { value: 0, done: false }  g.next() // { value: 1, done: false }  g.next(true) // { value: 0, done: false } |

因为yield表达式返回总是undefined，当next传入参数true时，reset即为true，所以i = -1,下次循环 i=0;

这个功能能让Generator函数从暂停到开始运行时，继续向其内部注入值，从而调整函数行为。

例子：

|  |
| --- |
| 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 } |

a: 第一个next没有参数,返回对象value为6；

第二个next没有参数，第一个yield 返回undefined，y = 2 \* undefined , y为NaN ,返回对象value值为 NaN / 3,还是NaN；

第三个next没有参数，第二个yield 返回undefined，z为undefined，返回对象value为x + y + z 即5 + NaN + undefined ,还是NaN；

b: 第一个next，返回对象value为6;

第二个next传入参数12，y = 2 \* 12 = 24;第二个yield 即24/3 = 8,所以第二个返回对象value为8；

第三next传入参数13，z = 13; 第三个返回对象value为 5+24+13=42，第三个返回对象的value为42；

注意：由于next方法的参数表示上一个yield表达式的返回值，所以在第一次使用next方法时，传递参数是无效的。

#### for…of循环

for…of循环可以自动遍历Generator函数生成的Iterator对象，所以不再需要调用next()方法；

for(let v of foo()){

…

}

foo()返回的是个Iterator，直接被for…of遍历

|  |
| --- |
| function\* foo() {  yield 1;  yield 2;  return 3;  }  let ite = foo();  console.log(ite.next());  console.log(ite.next());  console.log(ite.next());  //{ value: 1, done: false }  //{ value: 2, done: false }  //{ value: 3, done: true }  for (let v of foo()) {  console.log(v);  }  // 1  // 2 |

注意：一旦next方法的返回对象的done属性为true，for...of循环就会中止，且不包含该返回对象 （不包含done属性为true的这个对象），所以此处不会返回3；

利用for…of可以写出遍历任意对象的方法，原生的 JavaScript 对象没有遍历接口，无法使用for...of循环，通过 Generator 函数为它加上这个接口，就可以用了。

|  |
| --- |
| 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 |

还可以将 Generator 函数加到对象的Symbol.iterator属性上面：

|  |
| --- |
| function\* objectEntries() {  let propKeys = Object.keys(this);  for (let propKey of propKeys) {  yield [propKey, this[propKey]];  }  }  let jane = { first: 'Jane', last: 'Doe' };  jane[Symbol.iterator] = objectEntries;  for (let [key, value] of jane) {  console.log(`${key}: ${value}`);  }  // first: Jane  // last: Doe |

除了for...of循环以外，扩展运算符（...）、解构赋值和Array.from方法内部调用的，都是遍历器接口。这意味着，它们都可以将 Generator 函数返回的 Iterator 对象，作为参数。

|  |
| --- |
| function\* numbers () {  yield 1  yield 2  return 3  yield 4  }  // 扩展运算符  [...numbers()] // [1, 2]  // Array.from 方法  Array.from(numbers()) // [1, 2]  // 解构赋值  let [x, y] = numbers();  x // 1  y // 2  // for...of 循环  for (let n of numbers()) {  console.log(n)  }  // 1  // 2 |

#### Generator.prototype.throw()

Generator 函数返回的遍历器对象，都有一个throw方法，可以在函数体外抛出错误，然后在 Generator 函数体内捕获。

|  |
| --- |
| var g = function\* () {  try {  yield;  } catch (e) {  console.log('内部捕获', e);  }  };  var i = g();  i.next();  try {  i.throw('a');  i.throw('b');  } catch (e) {  console.log('外部捕获', e);  }  // 内部捕获 a  // 外部捕获 b |

i.throw('a') 被Generator 函数体内的catch语句捕获。i第二次抛出错误，由于 Generator 函数内部的catch语句已经执行过了，不会再捕捉到这个错误了，所以这个错误就被抛出了 Generator 函数体，被函数体外的catch语句捕获。

throw方法可以接受一个参数，该参数会被catch语句接收。

注意：不要混淆遍历器对象的throw方法和全局的throw命令；

遍历器对象的throw 可以被Generator函数内的catch捕捉，也能被外部的catch捕捉，但全局的throw命令只能被外部的catch捕捉。

throw方法抛出的错误要被内部捕获，前提是必须至少执行过一次next方法。

因为第一次执行next方法，等同于启动执行 Generator 函数的内部代码，否则 Generator 函数还没有开始执行，这时throw方法抛错只可能抛出在函数外部。

#### Generator.prototype.return()

Generator 函数返回的遍历器对象，还有一个return方法，可以返回给定的值，并且终结遍历 Generator 函数。

|  |
| --- |
| function\* gen() {  yield 1;  yield 2;  yield 3;  }  var g = gen();  g.next() // { value: 1, done: false }  g.return('foo') // { value: "foo", done: true }  g.next() // { value: undefined, done: true } |

上面代码中，遍历器对象g调用return方法后，返回值的value属性就是return方法的参数foo。

如果return方法调用时，不提供参数，则返回值的value属性为undefined。

|  |
| --- |
| function\* gen() {  yield 1;  yield 2;  yield 3;  }  var g = gen();  g.next() // { value: 1, done: false }  g.return() // { value: undefined, done: true } |

#### next()、throw()、return() 的共同点

三个方法是使用不同的语句替换yield表达式。

next()是将yield表达式替换成一个值。

|  |
| --- |
| const g = function\* (x, y) {  let result = yield x + y;  return result;  };  const gen = g(1, 2);  gen.next(); // Object {value: 3, done: false}  gen.next(1); // Object {value: 1, done: true}  // 相当于将 let result = yield x + y  // 替换成 let result = 1; |

若next不传参数，相当于替换成undefined。

throw()是将yield表达式替换成一个throw语句。

|  |
| --- |
| gen.throw(new Error('出错了')); // Uncaught Error: 出错了  // 相当于将 let result = yield x + y  // 替换成 let result = throw(new Error('出错了')); |

return()是将yield表达式替换成一个return语句。

|  |
| --- |
| gen.return(2); // Object {value: 2, done: true}  // 相当于将 let result = yield x + y  // 替换成 let result = return 2; |

#### yield\* 表达式

如果在 Generator 函数内部，调用另一个 Generator 函数，默认情况下是没有效果的。

yield\* 表达式：用来在一个 Generator 函数里面执行另一个 Generator 函数。

其实相当于：

yield\* Iterator

|  |
| --- |
| function\* foo() {  yield 'a';  yield 'b';  }  function\* bar() {  yield 'x';  yield\* foo();  yield 'y';  }  // 等同于  function\* bar() {  yield 'x';  yield 'a';  yield 'b';  yield 'y';  }  // 等同于  function\* bar() {  yield 'x';  for (let v of foo()) {  yield v;  }  yield 'y';  } |

yield\*后面的 Generator 函数（没有return语句时），等同于在 Generator 函数内部，部署一个for...of循环。

例子：

|  |
| --- |
| function\* inner() {  yield 'hello!';  }  function\* outer1() {  yield 'open';  yield inner();  yield 'close';  }  var gen = outer1()  gen.next().value // "open"  gen.next().value // 返回一个遍历器对象  gen.next().value // "close"  function\* outer2() {  yield 'open'  yield\* inner()  yield 'close'  }  var gen = outer2()  gen.next().value // "open"  gen.next().value // "hello!"  gen.next().value // "close" |

outer1返回一个遍历器对象，outer2返回遍历器对象的内部值；

从语法角度看，如果yield表达式后面跟的是一个遍历器对象，需要在yield表达式后面加上星号，表明它返回的是一个遍历器对象。这被称为yield\*表达式。

如果yield\*后面跟着一个数组，由于数组原生支持遍历器，因此就会遍历数组成员。

|  |
| --- |
| function\* gen(){  yield\* ["a", "b", "c"];  }  gen().next() // { value:"a", done:false } |

yield命令后面如果不加星号，返回的是整个数组，加了星号就表示返回的是数组的遍历器对象。

实际上，任何数据结构只要有 Iterator 接口，就可以被yield\*遍历。

所以，yield\* + 任何有Iterator接口的数据结构

在有return语句时，则需要用var value = yield\* iterator的形式获取return语句的值

|  |
| --- |
| function\* foo() {  yield 2;  yield 3;  return "foo";  }  function\* bar() {  yield 1;  var v = yield\* foo();  console.log("v: " + v);  yield 4;  }  var it = bar();  it.next()  // {value: 1, done: false}  it.next()  // {value: 2, done: false}  it.next()  // {value: 3, done: false}  it.next();  // "v: foo"  // {value: 4, done: false}  it.next()  // {value: undefined, done: true} |

yield\*命令可以很方便地取出嵌套数组的所有成员。

|  |
| --- |
| function\* iterTree(tree) {  if (Array.isArray(tree)) {  for(let i=0; i < tree.length; i++) {  yield\* iterTree(tree[i]);  }  } else {  yield tree;  }  }  const tree = [ 'a', ['b', 'c'], ['d', 'e'] ];  for(let x of iterTree(tree)) {  console.log(x);  }  // a  // b  // c  // d  // e |

由于扩展运算符...默认调用 Iterator 接口，所以上面这个函数也可以用于嵌套数组的平铺。

|  |
| --- |
| [...iterTree(tree)] // ["a", "b", "c", "d", "e"] |