# Let和const

## // 1.基本使用

**let arr = []**

**for(let i =0;i<10;i++){**

**// 变量i是let声明的，当前的i只在本轮循环有效，所以每一次循环的i其实都是一个新的变量**

**// for循环还有一个特别之处，就是设置循环变量的那部分是一个父作用域，而循环体内部是一个单独的子作用域。**

**// let i = "abc" // 不会报错**

**arr[i] = function(){**

**console.log(i)**

**}**

**}**

**arr[6]() // 6 如果用for中用var声明i 则输出10**

**// let 不存在变量提升**

**console.log(a) // undefined**

**var a = 3**

**// console.log(b) // Uncaught ReferenceError: b is not defined**

**let b = 6**

**// 暂时性死区**

**let tmp = 55**

**if (true) {**

**// 暂时性死区开始**

**// tmp = 'abc'; // ReferenceError**

**// console.log(tmp); // ReferenceError**

**let tmp; // 暂时性死区结束**

**console.log(tmp); // undefined**

**tmp = 123;**

**console.log(tmp); // 123**

**}**

**function bar(x = y,y=2){**

**console.log(x,y)**

**}**

**// bar() // Uncaught ReferenceError: y is not defined**

**// let c = c // Uncaught ReferenceError: c is not defined**

**// let不允许在相同作用域内，重复声明同一个变量。**

**// 报错**

**function func() {**

**let a = 10;**

**var a = 1;**

**}**

**// 报错**

**function func() {**

**let a = 10;**

**let a = 1;**

**}**

**//因此，不能在函数内部重新声明参数。**

**function func(arg) {**

**let arg; // 报错**

**}**

**function func(arg) {**

**{**

**let arg; // 不报错**

**}**

**}**

**// func()**

## // 2.块级作用域

**// 块级作用域的出现，实际上使得获得广泛应用的立即执行函数表达式（IIFE）不再必要了。**

**// IIFE 写法**

**(function () {**

**// var tmp = ...;**

**// ...**

**}());**

**// 块级作用域写法**

**{**

**// let tmp = ...;**

**// ...**

**}**

**// 块级作用域与函数声明**

**// ES5 规定，函数只能在顶层作用域和函数作用域之中声明，不能在块级作用域声明。**

**// 情况一**

**if (true) {**

**// function f() {}**

**}**

**// 情况二**

**try {**

**// function f() {}**

**} catch(e) {**

**// ...**

**}**

**// 以上都是非法的 但是，浏览器没有遵守这个规定，为了兼容以前的旧代码，还是支持在块级作用域之中声明函数，因此上面两种情况实际都能运行，不会报错。**

**// ES6 引入了块级作用域，明确允许在块级作用域之中声明函数。ES6 规定，块级作用域之中，函数声明语句的行为类似于let，在块级作用域之外不可引用。**

**// 另外，还有一个需要注意的地方。ES6 的块级作用域允许声明函数的规则，只在使用大括号的情况下成立，如果没有使用大括号，就会报错。**

**// 不报错**

**// 'use strict';**

**// if (true) {**

**// function f() {}**

**// }**

**// 报错**

**// 'use strict';**

**// if (true)**

**// function f() {}**

## // 3.const 命令

**// const声明一个只读的常量。一旦声明，常量的值就不能改变。const声明的变量不得改变值，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的作用域与let命令相同：只在声明所在的块级作用域内有效。const命令声明的常量也是不提升，同样存在暂时性死区，只能在声明的位置后面使用。const声明的常量，也与let一样不可重复声明。**

**// 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。因此，将一个对象声明为常量必须非常小心。**

**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**

**// 如果真的想将对象冻结，应该使用Object.freeze方法。**

**const fo = Object.freeze({});**

**// 常规模式时，下面一行不起作用；**

**// 严格模式时，该行会报错**

**fo.prop = 123;**

**// 上面代码中，常量foo指向一个冻结的对象，所以添加新属性不起作用，严格模式时还会报错。**

**// 除了将对象本身冻结，对象的属性也应该冻结。下面是一个将对象彻底冻结的函数。**

**var constantize = (obj) => {**

**Object.freeze(obj);**

**Object.keys(obj).forEach( (key, i) => {**

**if ( typeof obj[key] === 'object' ) {**

**constantize( obj[key] );**

**}**

**});**

**};**

## // 4.顶层对象的属性

**// 顶层对象的属性与全局变量挂钩，被认为是 JavaScript 语言最大的设计败笔之一。这样的设计带来了几个很大的问题，首先是没法在编译时就报出变量未声明的错误，只有运行时才能知道（因为全局变量可能是顶层对象的属性创造的，而属性的创造是动态的）；其次，程序员很容易不知不觉地就创建了全局变量（比如打字出错）；最后，顶层对象的属性是到处可以读写的，这非常不利于模块化编程。另一方面，window对象有实体含义，指的是浏览器的窗口对象，顶层对象是一个有实体含义的对象，也是不合适的。**

**// ES6 为了改变这一点，一方面规定，为了保持兼容性，var命令和function命令声明的全局变量，依旧是顶层对象的属性；另一方面规定，let命令、const命令、class命令声明的全局变量，不属于顶层对象的属性。也就是说，从 ES6 开始，全局变量将逐步与顶层对象的属性脱钩。**

**var e = 1;**

**// 如果在 Node 的 REPL 环境，可以写成 global.a**

**// 或者采用通用方法，写成 this.a**

**window.e // 1**

**let f = 1;**

**window.f // undefined**

**// 上面代码中，全局变量a由var命令声明，所以它是顶层对象的属性；全局变量b由let命令声明，所以它不是顶层对象的属性，返回undefined。**

# 变量的解构赋值

## // 1.数组的解构赋值

**// 以前，为变量赋值，只能直接指定值。**

**{**

**let a = 1;**

**let b = 2;**

**let c = 3;**

**// ES6 允许写成下面这样。**

**{**

**let [a, b, c] = [1, 2, 3];**

**}**

**}**

**// 上面代码表示，可以从数组中提取值，按照对应位置，对变量赋值。**

**// 本质上，这种写法属于“模式匹配”，只要等号两边的模式相同，左边的变量就会被赋予对应的值。下面是一些使用嵌套数组进行解构的例子。**

**{**

**let [foo, [[bar], baz]] = [1, [[2], 3]];**

**foo // 1**

**bar // 2**

**baz // 3**

**let [ , , third] = ["foo", "bar", "baz"];**

**third // "baz"**

**let [x, , y] = [1, 2, 3];**

**x // 1**

**y // 3**

**let [head, ...tail] = [1, 2, 3, 4];**

**head // 1**

**tail // [2, 3, 4]**

**{**

**let [x, y, ...z] = ['a'];**

**x // "a"**

**y // undefined**

**z // []**

**}**

**}**

**// 如果等号的右边不是数组（或者严格地说，不是可遍历的结构，参见《Iterator》一章），那么将会报错。**

**// 报错**

**{**

**// let [foo] = 1;**

**// let [foo] = false;**

**// let [foo] = NaN;**

**// let [foo] = undefined;**

**// let [foo] = null;**

**// let [foo] = {};**

**}**

**// 上面的语句都会报错，因为等号右边的值，要么转为对象以后不具备 Iterator 接口（前五个表达式），要么本身就不具备 Iterator 接口（最后一个表达式）。**

**// 解构赋值允许指定默认值。**

**{**

**let [foo = true] = [];**

**foo // true**

**let [x, y = 'b'] = ['a']; // x='a', y='b'**

**// let [x, y = 'b'] = ['a', undefined]; // x='a', y='b'**

**}**

**// 注意，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**

**}**

## // 2.对象的解构赋值

**// 对象的解构与数组有一个重要的不同。数组的元素是按次序排列的，变量的取值由它的位置决定；而对象的属性没有次序，变量必须与属性同名，才能取到正确的值。**

**{**

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

**foo // "aaa"**

**bar // "bbb"**

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

**baz // undefined**

**}**

**// 如果变量名与属性名不一致，必须写成下面这样。**

**{**

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

**baz // "aaa"**

**let obj = { first: 'hello', last: 'world' };**

**let { first: f, last: l } = obj;**

**f // 'hello'**

**l // 'world'**

**// 这实际上说明，对象的解构赋值是下面形式的简写（参见《对象的扩展》一章）。**

**{**

**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**

**}**

**// 上面代码中，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。**

**}**

**{**

**// 与数组一样，解构也可以用于嵌套结构的对象。**

**let obj = {**

**p: [**

**'Hello',**

**{ y: 'World' }**

**]**

**};**

**let { p: [x, { y }] } = obj;**

**x // "Hello"**

**y // "World"**

**// 注意，这时p是模式，不是变量，因此不会被赋值。如果p也要作为变量赋值，可以写成下面这样。**

**{**

**let obj = {**

**p: [**

**'Hello',**

**{ y: 'World' }**

**]**

**};**

**let { p, p: [x, { y }] } = obj;**

**x // "Hello"**

**y // "World"**

**p // ["Hello", {y: "World"}]**

**}**

**}**

**{**

**// 对象的解构也可以指定默认值。**

**var {x = 3} = {};**

**x // 3**

**// var {x, y = 5} = {x: 1};**

**// x // 1**

**// y // 5**

**// var {x: y = 3} = {};**

**// y // 3**

**// var {x: y = 3} = {x: 5};**

**// y // 5**

**var { message: msg = 'Something went wrong' } = {};**

**msg // "Something went wrong"**

**// 默认值生效的条件是，对象的属性值严格等于undefined。如果解构失败，变量的值等于undefined。**

**}**

**{**

**// 如果要将一个已经声明的变量用于解构赋值，必须非常小心。**

**// 错误的写法**

**// let x;**

**// {x} = {x: 1};**

**// SyntaxError: syntax error**

**// 上面代码的写法会报错，因为 JavaScript 引擎会将{x}理解成一个代码块，从而发生语法错误。只有不将大括号写在行首，避免 JavaScript 将其解释为代码块，才能解决这个问题。**

**// 正确的写法**

**let x;**

**({x} = {x: 1});**

**}**

**{**

**// 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;**

**// 上面代码将Math对象的对数、正弦、余弦三个方法，赋值到对应的变量上，使用起来就会方便很多。**

**// 由于数组本质是特殊的对象，因此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**

**// 上面代码对数组进行对象解构。数组arr的0键对应的值是1，[arr.length - 1]就是2键，对应的值是3。方括号这种写法，属于“属性名表达式”（参见《对象的扩展》一章）。**

**}**

## // 3.字符串的解构赋值

**{**

**// 字符串也可以解构赋值。这是因为此时，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"**

**// 类似数组的对象都有一个length属性，因此还可以对这个属性解构赋值。**

**let {length : len} = 'hello';**

**len // 5**

**}**

## // 4.数值和布尔值的解构赋值

**{**

**// 解构赋值时，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**

**// let {toString: s} = true;**

**// s === Boolean.prototype.toString // true**

**// 上面代码中，数值和布尔值的包装对象都有toString属性，因此变量s都能取到值。**

**// 解构赋值的规则是，只要等号右边的值不是对象或数组，就先将其转为对象。由于undefined和null无法转为对象，所以对它们进行解构赋值，都会报错。**

**// let { prop: x } = undefined; // TypeError**

**// let { prop: y } = null; // TypeError**

**}**

## // 5.函数参数的解构赋值

**{**

**// 函数的参数也可以使用解构赋值。**

**function add([x, y]){**

**return x + y;**

**}**

**add([1, 2]); // 3**

**// 上面代码中，函数add的参数表面上是一个数组，但在传入参数的那一刻，数组参数就被解构成变量x和y。对于函数内部的代码来说，它们能感受到的参数就是x和y。**

**// 下面是另一个例子。**

**[[1, 2], [3, 4]].map(([a, b]) => a + b);**

**// [ 3, 7 ]**

**}**

**{**

**// 函数参数的解构也可以使用默认值。**

**function move({x = 0, y = 0} = {}) {**

**return [x, y];**

**}**

**move({x: 3, y: 8}); // [3, 8]**

**move({x: 3}); // [3, 0]**

**move({}); // [0, 0]**

**move(); // [0, 0]**

**// 上面代码中，函数move的参数是一个对象，通过对这个对象进行解构，得到变量x和y的值。如果解构失败，x和y等于默认值。**

**}**

**{**

**// 注意，下面的写法会得到不一样的结果。**

**function move({x, y} = { x: 0, y: 0 }) {**

**return [x, y];**

**}**

**move({x: 3, y: 8}); // [3, 8]**

**move({x: 3}); // [3, undefined]**

**move({}); // [undefined, undefined]**

**move(); // [0, 0]**

**// 上面代码是为函数move的参数指定默认值，而不是为变量x和y指定默认值，所以会得到与前一种写法不同的结果。**

**// undefined就会触发函数参数的默认值。**

**[1, undefined, 3].map((x = 'yes') => x);**

**// [ 1, 'yes', 3 ]**

**}**

## // 6.圆括号问题

**// 解构赋值虽然很方便，但是解析起来并不容易。对于编译器来说，一个式子到底是模式，还是表达式，没有办法从一开始就知道，必须解析到（或解析不到）等号才能知道。**

**// 由此带来的问题是，如果模式中出现圆括号怎么处理。ES6 的规则是，只要有可能导致解构的歧义，就不得使用圆括号。**

**// 但是，这条规则实际上不那么容易辨别，处理起来相当麻烦。因此，建议只要有可能，就不要在模式中放置圆括号。**

**{**

**// 不能使用圆括号的情况**

**// 以下三种解构赋值不得使用圆括号。**

**// （1）变量声明语句**

**// 全部报错**

**/\*let [(a)] = [1];**

**let {x: (c)} = {};**

**let ({x: c}) = {};**

**let {(x: c)} = {};**

**let {(x): c} = {};\*/**

**// let { o: ({ p: p }) } = { o: { p: 2 } };**

**// 上面 6 个语句都会报错，因为它们都是变量声明语句，模式不能使用圆括号。**

**// （2）函数参数**

**// 函数参数也属于变量声明，因此不能带有圆括号。**

**// 报错**

**// function f([(z)]) { return z; }**

**// 报错**

**// function f([z,(x)]) { return x; }**

**// （3）赋值语句的模式**

**// 全部报错**

**// ({ p: a }) = { p: 42 };**

**// ([a]) = [5];**

**// 上面代码将整个模式放在圆括号之中，导致报错。**

**// 报错**

**// [({ p: a }), { x: c }] = [{}, {}];**

**// 上面代码将一部分模式放在圆括号之中，导致报错。**

**// 可以使用圆括号的情况**

**// 可以使用圆括号的情况只有一种：赋值语句的非模式部分，可以使用圆括号。**

**[(b)] = [3]; // 正确**

**({ p: (d) } = {}); // 正确**

**[(parseInt.prop)] = [3]; // 正确**

**// 上面三行语句都可以正确执行，因为首先它们都是赋值语句，而不是声明语句；其次它们的圆括号都不属于模式的一部分。第一行语句中，模式是取数组的第一个成员，跟圆括号无关；第二行语句中，模式是p，而不是d；第三行语句与第一行语句的性质一致。**

**}**

## // 7.用途

**{**

**// （1）交换变量的值**

**let x = 1;**

**let y = 2;**

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

**// 上面代码交换变量x和y的值，这样的写法不仅简洁，而且易读，语义非常清晰。**

**// （2）从函数返回多个值**

**// 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 example1() {**

**return [1, 2, 3];**

**}**

**let [a, b, c] = example1();**

**// 返回一个对象**

**function example2() {**

**return {**

**foo: 1,**

**bar: 2**

**};**

**}**

**let { foo, bar } = example2();**

**// （3）函数参数的定义**

**// 解构赋值可以方便地将一组参数与变量名对应起来。**

**// 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});**

**// （4）提取 JSON 数据**

**// 解构赋值对提取 JSON 对象中的数据，尤其有用。**

**let jsonData = {**

**id: 42,**

**status: "OK",**

**data: [867, 5309]**

**};**

**let { id, status, data: number } = jsonData;**

**console.log(id, status, number);**

**// 42, "OK", [867, 5309]**

**// 上面代码可以快速提取 JSON 数据的值。**

**// （5）函数参数的默认值**

**jQuery.ajax = function (url, {**

**async = true,**

**beforeSend = function () {},**

**cache = true,**

**complete = function () {},**

**crossDomain = false,**

**global = true,**

**// ... more config**

**} = {}) {**

**// ... do stuff**

**};**

**// 指定参数的默认值，就避免了在函数体内部再写var foo = config.foo || 'default foo';这样的语句。**

**// （6）遍历 Map 结构**

**// 任何部署了 Iterator 接口的对象，都可以用for...of循环遍历。Map 结构原生支持 Iterator 接口，配合变量的解构赋值，获取键名和键值就非常方便。**

**const map = new Map();**

**map.set('first', 'hello');**

**map.set('second', 'world');**

**for (let [key, value] of map) {**

**console.log(key + " is " + value);**

**}**

**// first is hello**

**// second is world**

**// 如果只想获取键名，或者只想获取键值，可以写成下面这样。**

**// 获取键名**

**for (let [key] of map) {**

**// ...**

**}**

**// 获取键值**

**for (let [,value] of map) {**

**// ...**

**}**

**// （7）输入模块的指定方法**

**// 加载模块时，往往需要指定输入哪些方法。解构赋值使得输入语句非常清晰。**

**const { SourceMapConsumer, SourceNode } = require("source-map");**

**}**

# 字符串的扩展

## // 1.字符的 Unicode 表示法

**{**

**// JavaScript 允许采用\uxxxx形式表示一个字符，其中xxxx表示字符的 Unicode 码点。**

**"\u0061"**

**// "a"**

**// 但是，这种表示法只限于码点在\u0000~\uFFFF之间的字符。超出这个范围的字符，必须用两个双字节的形式表示。**

**"\uD842\uDFB7"**

**// "𠮷"**

**"\u20BB7"**

**// " 7"**

**// 上面代码表示，如果直接在\u后面跟上超过0xFFFF的数值（比如\u20BB7），JavaScript 会理解成\u20BB+7。由于\u20BB是一个不可打印字符，所以只会显示一个空格，后面跟着一个7。**

**// ES6 对这一点做出了改进，只要将码点放入大括号，就能正确解读该字符。**

**"\u{20BB7}"**

**// "𠮷"**

**"\u{41}\u{42}\u{43}"**

**// "ABC"**

**let hello = 123;**

**hell\u{6F} // 123**

**'\u{1F680}' === '\uD83D\uDE80'**

**// true**

**// 上面代码中，最后一个例子表明，大括号表示法与四字节的 UTF-16 编码是等价的。**

**// 有了这种表示法之后，JavaScript 共有 6 种方法可以表示一个字符。**

**'\z' === 'z' // true**

**'\172' === 'z' // true**

**'\x7A' === 'z' // true**

**'\u007A' === 'z' // true**

**'\u{7A}' === 'z' // true**

**}**

## //2.codePointAt()

**{**

**// 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。**

**{**

**var s = "𠮷";**

**s.length // 2**

**s.charAt(0) // ''**

**s.charAt(1) // ''**

**s.charCodeAt(0) // 55362**

**s.charCodeAt(1) // 57271**

**}**

**// 上面代码中，汉字“𠮷”（注意，这个字不是“吉祥”的“吉”）的码点是0x20BB7，UTF-16 编码为0xD842 0xDFB7（十进制为55362 57271），需要4个字节储存。对于这种4个字节的字符，JavaScript 不能正确处理，字符串长度会误判为2，而且charAt方法无法读取整个字符，charCodeAt方法只能分别返回前两个字节和后两个字节的值。**

**// ES6 提供了codePointAt方法，能够正确处理 4 个字节储存的字符，返回一个字符的码点。**

**{**

**let s = '𠮷a';**

**s.codePointAt(0) // 134071**

**s.codePointAt(1) // 57271**

**s.codePointAt(2) // 97**

**}**

**// codePointAt方法的参数，是字符在字符串中的位置（从 0 开始）。上面代码中，JavaScript 将“𠮷a”视为三个字符，codePointAt 方法在第一个字符上，正确地识别了“𠮷”，返回了它的十进制码点 134071（即十六进制的20BB7）。在第二个字符（即“𠮷”的后两个字节）和第三个字符“a”上，codePointAt方法的结果与charCodeAt方法相同。**

**// 总之，codePointAt方法会正确返回 32 位的 UTF-16 字符的码点。对于那些两个字节储存的常规字符，它的返回结果与charCodeAt方法相同。**

**// codePointAt方法返回的是码点的十进制值，如果想要十六进制的值，可以使用toString方法转换一下。**

**{**

**let s = '𠮷a';**

**s.codePointAt(0).toString(16) // "20bb7"**

**s.codePointAt(2).toString(16) // "61"**

**}**

**// 你可能注意到了，codePointAt方法的参数，仍然是不正确的。比如，上面代码中，字符a在字符串s的正确位置序号应该是 1，但是必须向codePointAt方法传入 2。解决这个问题的一个办法是使用for...of循环，因为它会正确识别 32 位的 UTF-16 字符。**

**{**

**let s = '𠮷a';**

**for (let ch of s) {**

**console.log(ch.codePointAt(0).toString(16));**

**}**

**}**

**// 20bb7**

**// 61**

**// codePointAt方法是测试一个字符由两个字节还是由四个字节组成的最简单方法。**

**function is32Bit(c) {**

**return c.codePointAt(0) > 0xFFFF;**

**}**

**is32Bit("𠮷") // true**

**is32Bit("a") // false**

**}**

## //3.String.fromCodePoint()

**{**

**// ES5 提供String.fromCharCode方法，用于从码点返回对应字符，但是这个方法不能识别 32 位的 UTF-16 字符（Unicode 编号大于0xFFFF）。**

**String.fromCharCode(0x20BB7)**

**// "ஷ"**

**// 上面代码中，String.fromCharCode不能识别大于0xFFFF的码点，所以0x20BB7就发生了溢出，最高位2被舍弃了，最后返回码点U+0BB7对应的字符，而不是码点U+20BB7对应的字符。**

**// ES6 提供了String.fromCodePoint方法，可以识别大于0xFFFF的字符，弥补了String.fromCharCode方法的不足。在作用上，正好与codePointAt方法相反。**

**String.fromCodePoint(0x20BB7)**

**// "𠮷"**

**String.fromCodePoint(0x78, 0x1f680, 0x79) === 'x\uD83D\uDE80y'**

**// true**

**// 上面代码中，如果String.fromCodePoint方法有多个参数，则它们会被合并成一个字符串返回。**

**// 注意，fromCodePoint方法定义在String对象上，而codePointAt方法定义在字符串的实例对象上。**

**}**

## //4.字符串的遍历器接口

**{**

**// ES6 为字符串添加了遍历器接口（详见《Iterator》一章），使得字符串可以被for...of循环遍历。**

**for (let codePoint of 'foo') {**

**console.log(codePoint)**

**}**

**// "f"**

**// "o"**

**// "o"**

**// 除了遍历字符串，这个遍历器最大的优点是可以识别大于0xFFFF的码点，传统的for循环无法识别这样的码点。**

**let text = String.fromCodePoint(0x20BB7);**

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

**console.log(text[i]);**

**}**

**// " "**

**// " "**

**for (let i of text) {**

**console.log(i);**

**}**

**// "𠮷"**

**// 上面代码中，字符串text只有一个字符，但是for循环会认为它包含两个字符（都不可打印），而for...of循环会正确识别出这一个字符。**

**}**

## //5.at()

**{**

**// ES5 对字符串对象提供charAt方法，返回字符串给定位置的字符。该方法不能识别码点大于0xFFFF的字符。**

**'abc'.charAt(0) // "a"**

**'𠮷'.charAt(0) // "\uD842"**

**// 上面代码中的第二条语句，charAt方法期望返回的是用2个字节表示的字符，但汉字“𠮷”占用了4个字节，charAt(0)表示获取这4个字节中的前2个字节，很显然，这是无法正常显示的。**

**// 目前，有一个提案，提出字符串实例的at方法，可以识别 Unicode 编号大于0xFFFF的字符，返回正确的字符。**

**'abc'.at(0) // "a"**

**'𠮷'.at(0) // "𠮷"**

**// 这个方法可以通过垫片库实现。**

**}**

## // 6.includes(), startsWith(), endsWith()

**{**

**// 传统上，JavaScript 只有indexOf方法，可以用来确定一个字符串是否包含在另一个字符串中。ES6 又提供了三种新方法。**

**// includes()：返回布尔值，表示是否找到了参数字符串。**

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

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

**{**

**let s = 'Hello world!';**

**s.startsWith('Hello') // true**

**s.endsWith('!') // true**

**s.includes('o') // true**

**}**

**// 这三个方法都支持第二个参数，表示开始搜索的位置。**

**{**

**let s = 'Hello world!';**

**s.startsWith('world', 6) // true**

**s.endsWith('Hello', 5) // true**

**s.includes('Hello', 6) // false**

**}**

**// 上面代码表示，使用第二个参数n时，endsWith的行为与其他两个方法有所不同。它针对前n个字符，而其他两个方法针对从第n个位置直到字符串结束。**

**}**

## // 7.repeat()

**{**

**// repeat方法返回一个新字符串，表示将原字符串重复n次。**

**'x'.repeat(3) // "xxx"**

**'hello'.repeat(2) // "hellohello"**

**'na'.repeat(0) // ""**

**// 参数如果是小数，会被取整。**

**'na'.repeat(2.9) // "nana"**

**// 如果repeat的参数是负数或者Infinity，会报错。**

**'na'.repeat(Infinity)**

**// RangeError**

**'na'.repeat(-1)**

**// RangeError**

**// 但是，如果参数是 0 到-1 之间的小数，则等同于 0，这是因为会先进行取整运算。0 到-1 之间的小数，取整以后等于-0，repeat视同为 0。**

**'na'.repeat(-0.9) // ""**

**// 参数NaN等同于 0。**

**'na'.repeat(NaN) // ""**

**// 如果repeat的参数是字符串，则会先转换成数字。**

**'na'.repeat('na') // ""**

**'na'.repeat('3') // "nanana"**

**}**

## // 8.padStart()，padEnd()

**{**

**// ES2017 引入了字符串补全长度的功能。如果某个字符串不够指定长度，会在头部或尾部补全。padStart()用于头部补全，padEnd()用于尾部补全。**

**'x'.padStart(5, 'ab') // 'ababx'**

**'x'.padStart(4, 'ab') // 'abax'**

**'x'.padEnd(5, 'ab') // 'xabab'**

**'x'.padEnd(4, 'ab') // 'xaba'**

**// 上面代码中，padStart和padEnd一共接受两个参数，第一个参数用来指定字符串的最小长度，第二个参数是用来补全的字符串。**

**// 如果原字符串的长度，等于或大于指定的最小长度，则返回原字符串。**

**'xxx'.padStart(2, 'ab') // 'xxx'**

**'xxx'.padEnd(2, 'ab') // 'xxx'**

**// 如果用来补全的字符串与原字符串，两者的长度之和超过了指定的最小长度，则会截去超出位数的补全字符串。**

**'abc'.padStart(10, '0123456789')**

**// '0123456abc'**

**// 如果省略第二个参数，默认使用空格补全长度。**

**'x'.padStart(4) // ' x'**

**'x'.padEnd(4) // 'x '**

**// padStart的常见用途是为数值补全指定位数。下面代码生成 10 位的数值字符串。**

**'1'.padStart(10, '0') // "0000000001"**

**'12'.padStart(10, '0') // "0000000012"**

**'123456'.padStart(10, '0') // "0000123456"**

**// 另一个用途是提示字符串格式。**

**'12'.padStart(10, 'YYYY-MM-DD') // "YYYY-MM-12"**

**'09-12'.padStart(10, 'YYYY-MM-DD') // "YYYY-09-12"**

**}**

## // 9.模板字符串

**{**

**// 模板字符串（template string）是增强版的字符串，用反引号（`）标识。它可以当作普通字符串使用，也可以用来定义多行字符串，或者在字符串中嵌入变量。**

**// 普通字符串**

**`In JavaScript '\n' is a line-feed.`**

**// 多行字符串**

**`In JavaScript this is**

**not legal.`**

**console.log(`string text line 1**

**string text line 2`);**

**// 字符串中嵌入变量**

**let name = "Bob", time = "today";**

**`Hello ${name}, how are you ${time}?`**

**// 上面代码中的模板字符串，都是用反引号表示。如果在模板字符串中需要使用反引号，则前面要用反斜杠转义。**

**// 模板字符串中嵌入变量，需要将变量名写在${}之中。**

**// 大括号内部可以放入任意的 JavaScript 表达式，可以进行运算，以及引用对象属性。**

**let x = 1;**

**let y = 2;**

**`${x} + ${y} = ${x + y}`**

**// "1 + 2 = 3"**

**`${x} + ${y \* 2} = ${x + y \* 2}`**

**// "1 + 4 = 5"**

**let obj = {x: 1, y: 2};**

**`${obj.x + obj.y}`**

**// "3"**

**// 模板字符串之中还能调用函数。**

**function fn() {**

**return "Hello World";**

**}**

**`foo ${fn()} bar`**

**// foo Hello World bar**

**// 模板字符串甚至还能嵌套。**

**// 如果使用模板字符串表示多行字符串，所有的空格和缩进都会被保留在输出之中。**

**$('#list').html(`**

**<ul>**

**<li>first</li>**

**<li>second</li>**

**</ul>**

**`);**

**}**

## // 10.标签模板

**{**

**// 模板字符串的功能，不仅仅是上面这些。它可以紧跟在一个函数名后面，该函数将被调用来处理这个模板字符串。这被称为“标签模板”功能（tagged template）。**

**alert`123`**

**// 等同于**

**alert(123)**

**// 标签模板其实不是模板，而是函数调用的一种特殊形式。“标签”指的就是函数，紧跟在后面的模板字符串就是它的参数。**

**// 但是，如果模板字符里面有变量，就不是简单的调用了，而是会将模板字符串先处理成多个参数，再调用函数。**

**let a = 5;**

**let b = 10;**

**tag`Hello ${ a + b } world ${ a \* b }`;**

**// 等同于**

**tag(['Hello ', ' world ', ''], 15, 50);**

**// 上面代码中，模板字符串前面有一个标识名tag，它是一个函数。整个表达式的返回值，就是tag函数处理模板字符串后的返回值。**

**// 函数tag依次会接收到多个参数。**

**function tag(stringArr, value1, value2){**

**// ...**

**}**

**// 等同于**

**function tag(stringArr, ...values){**

**// ...**

**}**

**// tag函数的第一个参数是一个数组，该数组的成员是模板字符串中那些没有变量替换的部分，也就是说，变量替换只发生在数组的第一个成员与第二个成员之间、第二个成员与第三个成员之间，以此类推。**

**// tag函数的其他参数，都是模板字符串各个变量被替换后的值。由于本例中，模板字符串含有两个变量，因此tag会接受到value1和value2两个参数。**

**// tag函数所有参数的实际值如下。**

**// 第一个参数：['Hello ', ' world ', '']**

**// 第二个参数: 15**

**// 第三个参数：50**

**// 也就是说，tag函数实际上以下面的形式调用。**

**tag(['Hello ', ' world ', ''], 15, 50)**

**// 我们可以按照需要编写tag函数的代码。下面是tag函数的一种写法，以及运行结果。**

**let a = 5;**

**let b = 10;**

**function tag(s, v1, v2) {**

**console.log(s[0]);**

**console.log(s[1]);**

**console.log(s[2]);**

**console.log(v1);**

**console.log(v2);**

**return "OK";**

**}**

**tag`Hello ${ a + b } world ${ a \* b}`;**

**// "Hello "**

**// " world "**

**// ""**

**// 15**

**// 50**

**// "OK"**

**// 下面是一个更复杂的例子。**

**let total = 30;**

**let msg = passthru`The total is ${total} (${total\*1.05} with tax)`;**

**function passthru(literals) {**

**let result = '';**

**let i = 0;**

**while (i < literals.length) {**

**result += literals[i++];**

**if (i < arguments.length) {**

**result += arguments[i];**

**}**

**}**

**return result;**

**}**

**msg // "The total is 30 (31.5 with tax)"**

**// 上面这个例子展示了，如何将各个参数按照原来的位置拼合回去。**

**// “标签模板”的一个重要应用，就是过滤 HTML 字符串，防止用户输入恶意内容。**

**let message =**

**SaferHTML`<p>${sender} has sent you a message.</p>`;**

**function SaferHTML(templateData) {**

**let s = templateData[0];**

**for (let i = 1; i < arguments.length; i++) {**

**let arg = String(arguments[i]);**

**// Escape special characters in the substitution.**

**s += arg.replace(/&/g, "&amp;")**

**.replace(/</g, "&lt;")**

**.replace(/>/g, "&gt;");**

**// Don't escape special characters in the template.**

**s += templateData[i];**

**}**

**return s;**

**}**

**// 上面代码中，sender变量往往是用户提供的，经过SaferHTML函数处理，里面的特殊字符都会被转义。**

**let sender = '<script>alert("abc")</script>'; // 恶意代码**

**let message = SaferHTML`<p>${sender} has sent you a message.</p>`;**

**message**

**<!-- // <p>&lt;script&gt;alert("abc")&lt;/script&gt; has sent you a message.</p> -->**

**}**

# 数值的扩展

## //1.八进制和二进制表示法

**{**

**// ES6 提供了二进制和八进制数值的新的写法，分别用前缀0b（或0B）和0o（或0O）表示。**

**// 0b111110111 === 503 // true**

**// 0o767 === 503 // true**

**// 从 ES5 开始，在严格模式之中，八进制就不再允许使用前缀0表示，ES6 进一步明确，要使用前缀0o表示。**

**// 非严格模式**

**(function(){**

**console.log(0o11 === 011);**

**})() // true**

**// 严格模式**

**;(function(){**

**'use strict';**

**// console.log(0o11 === 011);**

**})() // Uncaught SyntaxError: Octal literals are not allowed in strict mode.**

**// 如果要将0b和0o前缀的字符串数值转为十进制，要使用Number方法。**

**Number('0b111') // 7**

**Number('0o10') // 8**

**console.log(0b11)**

**}**

## //2.Number.isFinite(), Number.isNaN()

**{**

**// ES6 在Number对象上，新提供了Number.isFinite()和Number.isNaN()两个方法。**

**// Number.isFinite()用来检查一个数值是否为有限的（finite），即不是Infinity。**

**Number.isFinite(15); // true**

**Number.isFinite(0.8); // true**

**Number.isFinite(NaN); // false**

**Number.isFinite(Infinity); // false**

**Number.isFinite(-Infinity); // false**

**Number.isFinite('foo'); // false**

**Number.isFinite('15'); // false**

**Number.isFinite(true); // false**

**// 注意，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。**

**Number.isNaN(NaN) // true**

**Number.isNaN(15) // false**

**Number.isNaN('15') // false**

**Number.isNaN(true) // false**

**Number.isNaN(9/NaN) // true**

**Number.isNaN('true' / 0) // true**

**Number.isNaN('true' / 'true') // true**

**// 如果参数类型不是NaN，Number.isNaN一律返回false。**

**// 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。**

**isFinite(25) // true**

**isFinite("25") // true**

**Number.isFinite(25) // true**

**Number.isFinite("25") // false**

**isNaN(NaN) // true**

**isNaN("NaN") // true**

**Number.isNaN(NaN) // true**

**Number.isNaN("NaN") // false**

**Number.isNaN(1) // false**

**}**

## //3.Number.parseInt(), Number.parseFloat()

**{**

**// ES6 将全局方法parseInt()和parseFloat()，移植到Number对象上面，行为完全保持不变。**

**// ES5的写法**

**parseInt('12.34') // 12**

**parseFloat('123.45#') // 123.45**

**// ES6的写法**

**Number.parseInt('12.34') // 12**

**Number.parseFloat('123.45#') // 123.45**

**// 这样做的目的，是逐步减少全局性方法，使得语言逐步模块化。**

**Number.parseInt === parseInt // true**

**Number.parseFloat === parseFloat // true**

**}**

## //4.Number.isInteger()

**{**

**// Number.isInteger()用来判断一个数值是否为整数。**

**Number.isInteger(25) // true**

**Number.isInteger(25.1) // false**

**// JavaScript 内部，整数和浮点数采用的是同样的储存方法，所以 25 和 25.0 被视为同一个值。**

**Number.isInteger(25) // true**

**Number.isInteger(25.0) // true**

**// 如果参数不是数值，Number.isInteger返回false。**

**Number.isInteger() // false**

**Number.isInteger(null) // false**

**Number.isInteger('15') // false**

**Number.isInteger(true) // false**

**// 注意，由于 JavaScript 采用 IEEE 754 标准，数值存储为64位双精度格式，数值精度最多可以达到 53 个二进制位（1 个隐藏位与 52 个有效位）。如果数值的精度超过这个限度，第54位及后面的位就会被丢弃，这种情况下，Number.isInteger可能会误判。**

**Number.isInteger(3.0000000000000002) // true**

**// 上面代码中，Number.isInteger的参数明明不是整数，但是会返回true。原因就是这个小数的精度达到了小数点后16个十进制位，转成二进制位超过了53个二进制位，导致最后的那个2被丢弃了。**

**// 类似的情况还有，如果一个数值的绝对值小于Number.MIN\_VALUE（5E-324），即小于 JavaScript 能够分辨的最小值，会被自动转为 0。这时，Number.isInteger也会误判。**

**Number.isInteger(5E-324) // false**

**Number.isInteger(5E-325) // true**

**// 上面代码中，5E-325由于值太小，会被自动转为0，因此返回true。**

**// 总之，如果对数据精度的要求较高，不建议使用Number.isInteger()判断一个数值是否为整数。**

**}**

## //5.Number.EPSILON

**{**

**// ES6 在Number对象上面，新增一个极小的常量Number.EPSILON。根据规格，它表示 1 与大于 1 的最小浮点数之间的差。**

**// 对于 64 位浮点数来说，大于 1 的最小浮点数相当于二进制的1.00..001，小数点后面有连续 51 个零。这个值减去 1 之后，就等于 2 的 -52 次方。**

**Number.EPSILON === Math.pow(2, -52)**

**// true**

**Number.EPSILON**

**// 2.220446049250313e-16**

**Number.EPSILON.toFixed(20)**

**// "0.00000000000000022204"**

**// Number.EPSILON实际上是 JavaScript 能够表示的最小精度。误差如果小于这个值，就可以认为已经没有意义了，即不存在误差了。**

**// 引入一个这么小的量的目的，在于为浮点数计算，设置一个误差范围。我们知道浮点数计算是不精确的。**

**0.1 + 0.2**

**// 0.30000000000000004**

**0.1 + 0.2 - 0.3**

**// 5.551115123125783e-17**

**5.551115123125783e-17.toFixed(20)**

**// '0.00000000000000005551'**

**// 上面代码解释了，为什么比较0.1 + 0.2与0.3得到的结果是false。**

**0.1 + 0.2 === 0.3 // false**

**// Number.EPSILON可以用来设置“能够接受的误差范围”。比如，误差范围设为 2 的-50 次方（即Number.EPSILON \* Math.pow(2, 2)），即如果两个浮点数的差小于这个值，我们就认为这两个浮点数相等。**

**5.551115123125783e-17 < Number.EPSILON \* Math.pow(2, 2)**

**// true**

**// 因此，Number.EPSILON的实质是一个可以接受的最小误差范围。**

**function withinErrorMargin (left, right) {**

**return Math.abs(left - right) < Number.EPSILON \* Math.pow(2, 2);**

**}**

**0.1 + 0.2 === 0.3 // false**

**withinErrorMargin(0.1 + 0.2, 0.3) // true**

**1.1 + 1.3 === 2.4 // false**

**withinErrorMargin(1.1 + 1.3, 2.4) // true**

**// 上面的代码为浮点数运算，部署了一个误差检查函数。**

**}**

## //6.安全整数和 Number.isSafeInteger()

**{**

**// JavaScript 能够准确表示的整数范围在-2^53到2^53之间（不含两个端点），超过这个范围，无法精确表示这个值。**

**Math.pow(2, 53) // 9007199254740992**

**9007199254740992 // 9007199254740992**

**9007199254740993 // 9007199254740992**

**Math.pow(2, 53) === Math.pow(2, 53) + 1**

**// true**

**// 上面代码中，超出 2 的 53 次方之后，一个数就不精确了。**

**// ES6 引入了Number.MAX\_SAFE\_INTEGER和Number.MIN\_SAFE\_INTEGER这两个常量，用来表示这个范围的上下限。**

**Number.MAX\_SAFE\_INTEGER === Math.pow(2, 53) - 1**

**// true**

**Number.MIN\_SAFE\_INTEGER === -Number.MAX\_SAFE\_INTEGER**

**// true**

**// Number.isSafeInteger()则是用来判断一个整数是否落在这个范围之内。如果参数不是数值，则返会false**

**}**

## //7.Math 对象的扩展

**{**

**//Math.trunc()**

**//Math.trunc方法用于去除一个数的小数部分，返回整数部分。对于非数值，Math.trunc内部使用Number方法将其先转为数值。对于空值和无法截取整数的值，返回NaN。**

**Math.trunc(4.1) // 4**

**Math.trunc(4.9) // 4**

**Math.trunc('123.456') // 123**

**Math.trunc(true) //1**

**Math.trunc(NaN); // NaN**

**Math.trunc('foo'); // NaN**

**// 对于没有部署这个方法的环境，可以用下面的代码模拟。**

**Math.trunc = Math.trunc || function(x) {**

**return x < 0 ? Math.ceil(x) : Math.floor(x);**

**};**

**// Math.sign**

**// Math.sign方法用来判断一个数到底是正数、负数、还是零。对于非数值，会先将其转换为数值。**

**// 它会返回五种值。**

**// 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。**

**// 对于没有部署这个方法的环境，可以用下面的代码模拟。**

**Math.sign = Math.sign || function(x) {**

**x = +x; // convert to a number**

**if (x === 0 || isNaN(x)) {**

**return x;**

**}**

**return x > 0 ? 1 : -1;**

**};**

**// Math.cbrt**

**// Math.cbrt方法用于计算一个数的立方根。对于非数值，Math.cbrt方法内部也是先使用Number方法将其转为数值。**

**// 对于没有部署这个方法的环境，可以用下面的代码模拟。**

**Math.cbrt = Math.cbrt || function(x) {**

**var y = Math.pow(Math.abs(x), 1/3);**

**return x < 0 ? -y : y;**

**};**

**}**

## //8.指数运算符

**{**

**// ES2016 新增了一个指数运算符（\*\*）。**

**2 \*\* 2 // 4**

**2 \*\* 3 // 8**

**// 指数运算符可以与等号结合，形成一个新的赋值运算符（\*\*=）。**

**let a = 1.5;**

**a \*\*= 2;**

**// 等同于 a = a \* a;**

**let b = 4;**

**b \*\*= 3;**

**// 等同于 b = b \* b \* b;**

**// 注意，在 V8 引擎中，指数运算符与Math.pow的实现不相同，对于特别大的运算结果，两者会有细微的差异。**

**Math.pow(99, 99)**

**// 3.697296376497263e+197**

**99 \*\* 99**

**// 3.697296376497268e+197**

**// 上面代码中，两个运算结果的最后一位有效数字是有差异的。**

**}**

# 函数的扩展

## // 1.rest 参数

**{**

**// ES6 引入 rest 参数（形式为...变量名），用于获取函数的多余参数，这样就不需要使用arguments对象了。rest 参数搭配的变量是一个数组，该变量将多余的参数放入数组中。**

**function add(...values) {**

**let sum = 0;**

**for (var val of values) {**

**sum += val;**

**}**

**return sum;**

**}**

**add(2, 5, 3) // 10**

**// arguments对象不是数组，而是一个类似数组的对象。所以为了使用数组的方法，必须使用Array.prototype.slice.call先将其转为数组。rest 参数就不存在这个问题，它就是一个真正的数组，数组特有的方法都可以使用。下面是一个利用 rest 参数改写数组push方法的例子。**

**function push(array, ...items) {**

**items.forEach(function(item) {**

**array.push(item);**

**console.log(item);**

**});**

**}**

**var a = [];**

**push(a, 1, 2, 3)**

**// 注意，rest 参数之后不能再有其他参数（即只能是最后一个参数），否则会报错。**

**// 报错**

**function f(a, ...b, c) {**

**// ...**

**}**

**// 函数的length属性，不包括 rest 参数。**

**(function(a) {}).length // 1**

**(function(...a) {}).length // 0**

**(function(a, ...b) {}).length // 1**

**}**

## //2.严格模式

**{**

**// 从 ES5 开始，函数内部可以设定为严格模式。ES2016 做了一点修改，规定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。**

**// 这样规定的原因是，函数内部的严格模式，同时适用于函数体和函数参数。但是，函数执行的时候，先执行函数参数，然后再执行函数体。这样就有一个不合理的地方，只有从函数体之中，才能知道参数是否应该以严格模式执行，但是参数却应该先于函数体执行。**

**// 两种方法可以规避这种限制。第一种是设定全局性的严格模式，这是合法的。**

**'use strict';**

**function doSomething(a, b = a) {**

**// code**

**}**

**// 第二种是把函数包在一个无参数的立即执行函数里面。**

**const doSomething = (function () {**

**'use strict';**

**return function(value = 42) {**

**return value;**

**};**

**}());**

**}**

## //3.箭头函数

**{**

**// ES6 允许使用“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 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; }**

**// 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" });**

**// 下面是一种特殊情况，虽然可以运行，但会得到错误的结果。**

**let foo = () => { a: 1 };**

**foo() // undefined**

**// 上面代码中，原始意图是返回一个对象{ a: 1 }，但是由于引擎认为大括号是代码块，所以执行了一行语句a: 1。这时，a可以被解释为语句的标签，因此实际执行的语句是1;，然后函数就结束了，没有返回值。**

**箭头函数可以与变量解构结合使用。**

**const full = ({ first, last }) => first + ' ' + last;**

**// 等同于**

**// function full(person) {**

**// return person.first + ' ' + person.last;**

**// }**

**下面是 rest 参数与箭头函数结合的例子。**

**const numbers = (...nums) => nums;**

**numbers(1, 2, 3, 4, 5)**

**// [1,2,3,4,5]**

**const headAndTail = (head, ...tail) => [head, tail];**

**headAndTail(1, 2, 3, 4, 5)**

**// [1,[2,3,4,5]]**

**/\*使用注意点**

**箭头函数有几个使用注意点。**

**（1）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。**

**（2）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。**

**（3）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。**

**（4）不可以使用yield命令，因此箭头函数不能用作 Generator 函数。**

**上面四点中，第一点尤其值得注意。this对象的指向是可变的，但是在箭头函数中，它是固定的。\*/**

**function foo() {**

**setTimeout(() => {**

**console.log('id:', this.id);**

**}, 100);**

**}**

**var id = 21;**

**foo.call({ id: 42 });**

**// id: 42**

**// 上面代码中，setTimeout的参数是一个箭头函数，这个箭头函数的定义生效是在foo函数生成时，而它的真正执行要等到 100 毫秒后。如果是普通函数，执行时this应该指向全局对象window，这时应该输出21。但是，箭头函数导致this总是指向函数定义生效时所在的对象（本例是{id: 42}），所以输出的是42。**

**// 箭头函数可以让setTimeout里面的this，绑定定义时所在的作用域，而不是指向运行时所在的作用域。下面是另一个例子。**

**function Timer() {**

**this.s1 = 0;**

**this.s2 = 0;**

**// 箭头函数**

**setInterval(() => this.s1++, 1000);**

**// 普通函数**

**setInterval(function () {**

**this.s2++;**

**}, 1000);**

**}**

**var timer = new Timer();**

**setTimeout(() => console.log('s1: ', timer.s1), 3100);**

**setTimeout(() => console.log('s2: ', timer.s2), 3100);**

**// s1: 3**

**// s2: 0**

**// 上面代码中，Timer函数内部设置了两个定时器，分别使用了箭头函数和普通函数。前者的this绑定定义时所在的作用域（即Timer函数），后者的this指向运行时所在的作用域（即全局对象）。所以，3100 毫秒之后，timer.s1被更新了 3 次，而timer.s2一次都没更新。**

**// 箭头函数可以让this指向固定化，这种特性很有利于封装回调函数。下面是一个例子，DOM 事件的回调函数封装在一个对象里面。**

**var handler = {**

**id: '123456',**

**init: function() {**

**document.addEventListener('click',**

**event => this.doSomething(event.type), false);**

**},**

**doSomething: function(type) {**

**console.log('Handling ' + type + ' for ' + this.id);**

**}**

**};**

**// 上面代码的init方法中，使用了箭头函数，这导致这个箭头函数里面的this，总是指向handler对象。否则，回调函数运行时，this.doSomething这一行会报错，因为此时this指向document对象。**

**// this指向的固定化，并不是因为箭头函数内部有绑定this的机制，实际原因是箭头函数根本没有自己的this，导致内部的this就是外层代码块的this。正是因为它没有this，所以也就不能用作构造函数。**

**// 所以，箭头函数转成 ES5 的代码如下。**

**// ES6**

**function foo() {**

**setTimeout(() => {**

**console.log('id:', this.id);**

**}, 100);**

**}**

**// ES5**

**function foo() {**

**var \_this = this;**

**setTimeout(function () {**

**console.log('id:', \_this.id);**

**}, 100);**

**}**

**// 上面代码中，转换后的 ES5 版本清楚地说明了，箭头函数里面根本没有自己的this，而是引用外层的this。**

**// 请问下面的代码之中有几个this？**

**function foo() {**

**return () => {**

**return () => {**

**return () => {**

**console.log('id:', this.id);**

**};**

**};**

**};**

**}**

**var f = foo.call({id: 1});**

**var t1 = f.call({id: 2})()(); // id: 1**

**var t2 = f().call({id: 3})(); // id: 1**

**var t3 = f()().call({id: 4}); // id: 1**

**// 上面代码之中，只有一个this，就是函数foo的this，所以t1、t2、t3都输出同样的结果。因为所有的内层函数都是箭头函数，都没有自己的this，它们的this其实都是最外层foo函数的this。**

**// 除了this，以下三个变量在箭头函数之中也是不存在的，指向外层函数的对应变量：arguments、super、new.target。**

**function foo() {**

**setTimeout(() => {**

**console.log('args:', arguments);**

**}, 100);**

**}**

**foo(2, 4, 6, 8)**

**// args: [2, 4, 6, 8]**

**// 上面代码中，箭头函数内部的变量arguments，其实是函数foo的arguments变量。**

**// 另外，由于箭头函数没有自己的this，所以当然也就不能用call()、apply()、bind()这些方法去改变this的指向。**

**(function() {**

**return [**

**(() => this.x).bind({ x: 'inner' })()**

**];**

**}).call({ x: 'outer' });**

**// ['outer']**

**// 上面代码中，箭头函数没有自己的this，所以bind方法无效，内部的this指向外部的this。**

**// 长期以来，JavaScript 语言的this对象一直是一个令人头痛的问题，在对象方法中使用this，必须非常小心。箭头函数”绑定”this，很大程度上解决了这个困扰。**

**}**

## //4.双冒号运算符

**{**

**// 箭头函数可以绑定this对象，大大减少了显式绑定this对象的写法（call、apply、bind）。但是，箭头函数并不适用于所有场合，所以现在有一个提案，提出了“函数绑定”（function bind）运算符，用来取代call、apply、bind调用。**

**// 函数绑定运算符是并排的两个冒号（::），双冒号左边是一个对象，右边是一个函数。该运算符会自动将左边的对象，作为上下文环境（即this对象），绑定到右边的函数上面。**

**foo::bar;**

**// 等同于**

**bar.bind(foo);**

**foo::bar(...arguments);**

**// 等同于**

**bar.apply(foo, arguments);**

**const hasOwnProperty = Object.prototype.hasOwnProperty;**

**function hasOwn(obj, key) {**

**return obj::hasOwnProperty(key);**

**}**

**// 如果双冒号左边为空，右边是一个对象的方法，则等于将该方法绑定在该对象上面。**

**var method = obj::obj.foo;**

**// 等同于**

**var method = ::obj.foo;**

**let log = ::console.log;**

**// 等同于**

**var log = console.log.bind(console);**

**// 如果双冒号运算符的运算结果，还是一个对象，就可以采用链式写法。**

**import { map, takeWhile, forEach } from "iterlib";**

**getPlayers()**

**::map(x => x.character())**

**::takeWhile(x => x.strength > 100)**

**::forEach(x => console.log(x));**

**}**

## //5.尾调用优化

**{**

**// 尾调用（Tail Call）是函数式编程的一个重要概念，本身非常简单，一句话就能说清楚，就是指某个函数的最后一步是调用另一个函数。**

**// 尾调用之所以与其他调用不同，就在于它的特殊的调用位置。我们知道，函数调用会在内存形成一个“调用记录”，又称“调用帧”（call frame），保存调用位置和内部变量等信息。如果在函数A的内部调用函数B，那么在A的调用帧上方，还会形成一个B的调用帧。等到B运行结束，将结果返回到A，B的调用帧才会消失。如果函数B内部还调用函数C，那就还有一个C的调用帧，以此类推。所有的调用帧，就形成一个“调用栈”（call stack）。尾调用由于是函数的最后一步操作，所以不需要保留外层函数的调用帧，因为调用位置、内部变量等信息都不会再用到了，只要直接用内层函数的调用帧，取代外层函数的调用帧就可以了。**

**function f() {**

**let m = 1;**

**let n = 2;**

**return g(m + n);**

**}**

**f();**

**// 等同于**

**function f() {**

**return g(3);**

**}**

**f();**

**// 等同于**

**g(3);**

**// 上面代码中，如果函数g不是尾调用，函数f就需要保存内部变量m和n的值、g的调用位置等信息。但由于调用g之后，函数f就结束了，所以执行到最后一步，完全可以删除f(x)的调用帧，只保留g(3)的调用帧。**

**// 这就叫做“尾调用优化”（Tail call optimization），即只保留内层函数的调用帧。如果所有函数都是尾调用，那么完全可以做到每次执行时，调用帧只有一项，这将大大节省内存。这就是“尾调用优化”的意义。**

**// 注意，只有不再用到外层函数的内部变量，内层函数的调用帧才会取代外层函数的调用帧，否则就无法进行“尾调用优化”。**

**function addOne(a){**

**var one = 1;**

**function inner(b){**

**return b + one;**

**}**

**return inner(a);**

**}**

**// 上面的函数不会进行尾调用优化，因为内层函数inner用到了外层函数addOne的内部变量one。**

**// 尾递归.函数调用自身，称为递归。如果尾调用自身，就称为尾递归。**

**// 递归非常耗费内存，因为需要同时保存成千上百个调用帧，很容易发生“栈溢出”错误（stack overflow）。但对于尾递归来说，由于只存在一个调用帧，所以永远不会发生“栈溢出”错误。**

**function factorial(n) {**

**if (n === 1) return 1;**

**return n \* factorial(n - 1);**

**}**

**factorial(5) // 120**

**// 上面代码是一个阶乘函数，计算n的阶乘，最多需要保存n个调用记录，复杂度 O(n) 。**

**//如果改写成尾递归，只保留一个调用记录，复杂度 O(1) 。**

**function factorial(n, total) {**

**if (n === 1) return total;**

**return factorial(n - 1, n \* total);**

**}**

**factorial(5, 1) // 120**

**// 递归函数的改写**

**// 尾递归的实现，往往需要改写递归函数，确保最后一步只调用自身。做到这一点的方法，就是把所有用到的内部变量改写成函数的参数。比如上面的例子，阶乘函数 factorial 需要用到一个中间变量total，那就把这个中间变量改写成函数的参数。这样做的缺点就是不太直观，第一眼很难看出来，为什么计算5的阶乘，需要传入两个参数5和1？**

**// 两个方法可以解决这个问题。方法一是在尾递归函数之外，再提供一个正常形式的函数。**

**function tailFactorial(n, total) {**

**if (n === 1) return total;**

**return tailFactorial(n - 1, n \* total);**

**}**

**function factorial(n) {**

**return tailFactorial(n, 1);**

**}**

**factorial(5) // 120**

**// 上面代码通过一个正常形式的阶乘函数factorial，调用尾递归函数tailFactorial，看起来就正常多了。**

**// 函数式编程有一个概念，叫做柯里化（currying），意思是将多参数的函数转换成单参数的形式。这里也可以使用柯里化。**

**function currying(fn, n) {**

**return function (m) {**

**return fn.call(this, m, n);**

**};**

**}**

**function tailFactorial(n, total) {**

**if (n === 1) return total;**

**return tailFactorial(n - 1, n \* total);**

**}**

**const factorial = currying(tailFactorial, 1);**

**factorial(5) // 120**

**// 上面代码通过柯里化，将尾递归函数tailFactorial变为只接受一个参数的factorial。**

**// 第二种方法就简单多了，就是采用 ES6 的函数默认值。**

**function factorial(n, total = 1) {**

**if (n === 1) return total;**

**return factorial(n - 1, n \* total);**

**}**

**factorial(5) // 120**

**// 上面代码中，参数total有默认值1，所以调用时不用提供这个值。**

**// 总结一下，递归本质上是一种循环操作。纯粹的函数式编程语言没有循环操作命令，所有的循环都用递归实现，这就是为什么尾递归对这些语言极其重要。对于其他支持“尾调用优化”的语言（比如 Lua，ES6），只需要知道循环可以用递归代替，而一旦使用递归，就最好使用尾递归。**

**// 严格模式**

**// ES6 的尾调用优化只在严格模式下开启，正常模式是无效的。**

**// 这是因为在正常模式下，函数内部有两个变量，可以跟踪函数的调用栈。**

**// func.arguments：返回调用时函数的参数。**

**// func.caller：返回调用当前函数的那个函数。**

**// 尾调用优化发生时，函数的调用栈会改写，因此上面两个变量就会失真。严格模式禁用这两个变量，所以尾调用模式仅在严格模式下生效。**

**function restricted() {**

**'use strict';**

**restricted.caller; // 报错**

**restricted.arguments; // 报错**

**}**

**restricted();**

**//尾递归优化的实现**

**// 尾递归优化只在严格模式下生效，那么正常模式下，或者那些不支持该功能的环境中，有没有办法也使用尾递归优化呢？回答是可以的，就是自己实现尾递归优化。**

**// 它的原理非常简单。尾递归之所以需要优化，原因是调用栈太多，造成溢出，那么只要减少调用栈，就不会溢出。怎么做可以减少调用栈呢？就是采用“循环”换掉“递归”。**

**// 下面是一个正常的递归函数。**

**function sum(x, y) {**

**if (y > 0) {**

**return sum(x + 1, y - 1);**

**} else {**

**return x;**

**}**

**}**

**sum(1, 100000)**

**// Uncaught RangeError: Maximum call stack size exceeded(…)**

**// 上面代码中，sum是一个递归函数，参数x是需要累加的值，参数y控制递归次数。一旦指定sum递归 100000 次，就会报错，提示超出调用栈的最大次数。**

**// 蹦床函数（trampoline）可以将递归执行转为循环执行。**

**function trampoline(f) {**

**while (f && f instanceof Function) {**

**f = f();**

**}**

**return f;**

**}**

**// 上面就是蹦床函数的一个实现，它接受一个函数f作为参数。只要f执行后返回一个函数，就继续执行。注意，这里是返回一个函数，然后执行该函数，而不是函数里面调用函数，这样就避免了递归执行，从而就消除了调用栈过大的问题。**

**// 然后，要做的就是将原来的递归函数，改写为每一步返回另一个函数。**

**function sum(x, y) {**

**if (y > 0) {**

**return sum.bind(null, x + 1, y - 1);**

**} else {**

**return x;**

**}**

**}**

**// 上面代码中，sum函数的每次执行，都会返回自身的另一个版本。**

**// 现在，使用蹦床函数执行sum，就不会发生调用栈溢出。**

**trampoline(sum(1, 100000))**

**// 100001**

**// 蹦床函数并不是真正的尾递归优化，下面的实现才是。**

**function tco(f) {**

**var value;**

**var active = false;**

**var accumulated = [];**

**return function accumulator() {**

**accumulated.push(arguments);**

**if (!active) {**

**active = true;**

**while (accumulated.length) {**

**value = f.apply(this, accumulated.shift());**

**}**

**active = false;**

**return value;**

**}**

**};**

**}**

**var sum = tco(function(x, y) {**

**if (y > 0) {**

**return sum(x + 1, y - 1)**

**}**

**else {**

**return x**

**}**

**});**

**sum(1, 100000)**

**// 100001**

**// 上面代码中，tco函数是尾递归优化的实现，它的奥妙就在于状态变量active。默认情况下，这个变量是不激活的。一旦进入尾递归优化的过程，这个变量就激活了。然后，每一轮递归sum返回的都是undefined，所以就避免了递归执行；而accumulated数组存放每一轮sum执行的参数，总是有值的，这就保证了accumulator函数内部的while循环总是会执行。这样就很巧妙地将“递归”改成了“循环”，而后一轮的参数会取代前一轮的参数，保证了调用栈只有一层。**

**}**

## //6.函数参数的尾逗号

**{**

**// ES2017 允许函数的最后一个参数有尾逗号（trailing comma）。**

**// 此前，函数定义和调用时，都不允许最后一个参数后面出现逗号。**

**function clownsEverywhere(**

**param1,**

**param2**

**) { /\* ... \*/ }**

**clownsEverywhere(**

**'foo',**

**'bar'**

**);**

**// 上面代码中，如果在param2或bar后面加一个逗号，就会报错。**

**// 如果像上面这样，将参数写成多行（即每个参数占据一行），以后修改代码的时候，想为函数clownsEverywhere添加第三个参数，或者调整参数的次序，就势必要在原来最后一个参数后面添加一个逗号。这对于版本管理系统来说，就会显示添加逗号的那一行也发生了变动。这看上去有点冗余，因此新的语法允许定义和调用时，尾部直接有一个逗号。**

**function clownsEverywhere(**

**param1,**

**param2,**

**) { /\* ... \*/ }**

**clownsEverywhere(**

**'foo',**

**'bar',**

**);**

**// 这样的规定也使得，函数参数与数组和对象的尾逗号规则，保持一致了。**

**}**

# 数组的扩展

## // 1.扩展运算符

**{**

**// 扩展运算符（spread）是三个点（...）。它好比 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>]**

**// 该运算符主要用于函数调用。**

**function push(array, ...items) {**

**array.push(...items);**

**}**

**function add(x, y) {**

**return x + y;**

**}**

**const numbers = [4, 38];**

**add(...numbers) // 42**

**// 上面代码中，array.push(...items)和add(...numbers)这两行，都是函数的调用，它们的都使用了扩展运算符。该运算符将一个数组，变为参数序列。**

**// 扩展运算符与正常的函数参数可以结合使用，非常灵活。**

**function f(v, w, x, y, z) { }**

**const args = [0, 1];**

**f(-1, ...args, 2, ...[3]);**

**// 扩展运算符后面还可以放置表达式。**

**const arr = [**

**...(x > 0 ? ['a'] : []),**

**'b',**

**];**

**// 如果扩展运算符后面是一个空数组，则不产生任何效果。**

**[...[], 1]**

**// [1]**

**// 替代函数的 apply 方法**

**// 由于扩展运算符可以展开数组，所以不再需要apply方法，将数组转为函数的参数了。**

**// ES5 的写法**

**function f(x, y, z) {**

**// ...**

**}**

**var args = [0, 1, 2];**

**f.apply(null, args);**

**// ES6的写法**

**function f(x, y, z) {**

**// ...**

**}**

**let args = [0, 1, 2];**

**f(...args);**

**// 下面是扩展运算符取代apply方法的一个实际的例子，应用Math.max方法，简化求出一个数组最大元素的写法。**

**// ES5 的写法**

**Math.max.apply(null, [14, 3, 77])**

**// ES6 的写法**

**Math.max(...[14, 3, 77])**

**// 等同于**

**Math.max(14, 3, 77);**

**// 扩展运算符的应用**

**// （1）复制数组**

**// 数组是复合的数据类型，直接复制的话，只是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。**

**const a1 = [1, 2];**

**const a2 = a1;**

**a2[0] = 2;**

**a1 // [2, 2]**

**// 上面代码中，a2并不是a1的克隆，而是指向同一份数据的另一个指针。修改a2，会直接导致a1的变化。**

**// ES5 只能用变通方法来复制数组。**

**const a1 = [1, 2];**

**const a2 = a1.concat();**

**a2[0] = 2;**

**a1 // [1, 2]**

**// 上面代码中，a1会返回原数组的克隆，再修改a2就不会对a1产生影响。**

**// 扩展运算符提供了复制数组的简便写法。**

**const a1 = [1, 2];**

**// 写法一**

**const a2 = [...a1];**

**// 写法二**

**const [...a2] = a1;**

**// 上面的两种写法，a2都是a1的克隆。**

**// （2）合并数组**

**// 扩展运算符提供了数组合并的新写法。**

**// ES5**

**[1, 2].concat(more)**

**// ES6**

**[1, 2, ...more]**

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

**var arr2 = ['c'];**

**var arr3 = ['d', 'e'];**

**// ES5的合并数组**

**arr1.concat(arr2, arr3);**

**// [ 'a', 'b', 'c', 'd', 'e' ]**

**// ES6的合并数组**

**[...arr1, ...arr2, ...arr3]**

**// [ 'a', 'b', 'c', 'd', 'e' ]**

**// （3）与解构赋值结合**

**// 扩展运算符可以与解构赋值结合起来，用于生成数组。**

**// ES5**

**a = list[0], rest = list.slice(1)**

**// ES6**

**[a, ...rest] = list**

**// 下面是另外一些例子。**

**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**

**// （4）字符串**

**// 扩展运算符还可以将字符串转为真正的数组。**

**[...'hello']**

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

**// 上面的写法，有一个重要的好处，那就是能够正确识别四个字节的 Unicode 字符。**

**'x\uD83D\uDE80y'.length // 4**

**[...'x\uD83D\uDE80y'].length // 3**

**//上面代码的第一种写法，JavaScript 会将四个字节的 Unicode 字符，识别为 2 个字符，采用扩展运算符就没有这个问题。因此，正确返回字符串长度的函数，可以像下面这样写。**

**function length(str) {**

**return [...str].length;**

**}**

**length('x\uD83D\uDE80y') // 3**

**// 凡是涉及到操作四个字节的 Unicode 字符的函数，都有这个问题。因此，最好都用扩展运算符改写。**

**let str = 'x\uD83D\uDE80y';**

**str.split('').reverse().join('')**

**// 'y\uDE80\uD83Dx'**

**[...str].reverse().join('')**

**// 'y\uD83D\uDE80x'**

**// 上面代码中，如果不用扩展运算符，字符串的reverse操作就不正确。**

**// （5）实现了 Iterator 接口的对象**

**// 任何 Iterator 接口的对象（参阅 Iterator 一章），都可以用扩展运算符转为真正的数组。**

**let nodeList = document.querySelectorAll('div');**

**let array = [...nodeList];**

**// 上面代码中，querySelectorAll方法返回的是一个nodeList对象。它不是数组，而是一个类似数组的对象。这时，扩展运算符可以将其转为真正的数组，原因就在于NodeList对象实现了 Iterator 。**

**// 对于那些没有部署 Iterator 接口的类似数组的对象，扩展运算符就无法将其转为真正的数组。**

**let arrayLike = {**

**'0': 'a',**

**'1': 'b',**

**'2': 'c',**

**length: 3**

**};**

**// TypeError: Cannot spread non-iterable object.**

**let arr = [...arrayLike];**

**// 上面代码中，arrayLike是一个类似数组的对象，但是没有部署 Iterator 接口，扩展运算符就会报错。这时，可以改为使用Array.from方法将arrayLike转为真正的数组。**

**// （6）Map 和 Set 结构，Generator 函数**

**// 扩展运算符内部调用的是数据结构的 Iterator 接口，因此只要具有 Iterator 接口的对象，都可以使用扩展运算符，比如 Map 结构。**

**let map = new Map([**

**[1, 'one'],**

**[2, 'two'],**

**[3, 'three'],**

**]);**

**let arr = [...map.keys()]; // [1, 2, 3]**

**// Generator 函数运行后，返回一个遍历器对象，因此也可以使用扩展运算符。**

**const go = function\*(){**

**yield 1;**

**yield 2;**

**yield 3;**

**};**

**[...go()] // [1, 2, 3]**

**// 上面代码中，变量go是一个 Generator 函数，执行后返回的是一个遍历器对象，对这个遍历器对象执行扩展运算符，就会将内部遍历得到的值，转为一个数组。**

**// 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.Array.from()

**{**

**// Array.from方法用于将两类对象转为真正的数组：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）。**

**// 下面是一个类似数组的对象，Array.from将它转为真正的数组。**

**let arrayLike = {**

**'0': 'a',**

**'1': 'b',**

**'2': 'c',**

**length: 3**

**};**

**// ES5的写法**

**var arr1 = [].slice.call(arrayLike); // ['a', 'b', 'c']**

**// ES6的写法**

**let arr2 = Array.from(arrayLike); // ['a', 'b', 'c']**

**// 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。Array.from都可以将它们转为真正的数组。**

**// NodeList对象**

**let ps = document.querySelectorAll('p');**

**Array.from(ps).filter(p => {**

**return p.textContent.length > 100;**

**});**

**// arguments对象**

**function foo() {**

**var args = Array.from(arguments);**

**// ...**

**}**

**// 上面代码中，querySelectorAll方法返回的是一个类似数组的对象，可以将这个对象转为真正的数组，再使用filter方法。**

**// 只要是部署了 Iterator 接口的数据结构，Array.from都能将其转为数组。如果参数是一个真正的数组，Array.from会返回一个一模一样的新数组。**

**// 值得提醒的是，扩展运算符（...）也可以将某些数据结构转为数组。**

**// arguments对象**

**function foo() {**

**const args = [...arguments];**

**}**

**// NodeList对象**

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

**// 扩展运算符背后调用的是遍历器接口（Symbol.iterator），如果一个对象没有部署这个接口，就无法转换。Array.from方法还支持类似数组的对象。所谓类似数组的对象，本质特征只有一点，即必须有length属性。因此，任何有length属性的对象，都可以通过Array.from方法转为数组，而此时扩展运算符就无法转换。**

**Array.from({ length: 3 });**

**// [ undefined, undefined, undefined ]**

**// 上面代码中，Array.from返回了一个具有三个成员的数组，每个位置的值都是undefined。扩展运算符转换不了这个对象。**

**// 对于还没有部署该方法的浏览器，可以用Array.prototype.slice方法替代。**

**const toArray = (() =>**

**Array.from ? Array.from : obj => [].slice.call(obj)**

**)();**

**// Array.from还可以接受第二个参数，作用类似于数组的map方法，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(arrayLike, x => x \* x);**

**// 等同于**

**Array.from(arrayLike).map(x => x \* x);**

**Array.from([1, 2, 3], (x) => x \* x)**

**// [1, 4, 9]**

**// 下面的例子是取出一组 DOM 节点的文本内容。**

**let spans = document.querySelectorAll('span.name');**

**// map()**

**let names1 = Array.prototype.map.call(spans, s => s.textContent);**

**// Array.from()**

**let names2 = Array.from(spans, s => s.textContent)**

**// 如果map函数里面用到了this关键字，还可以传入Array.from的第三个参数，用来绑定this。**

**// Array.from()可以将各种值转为真正的数组，并且还提供map功能。这实际上意味着，只要有一个原始的数据结构，你就可以先对它的值进行处理，然后转成规范的数组结构，进而就可以使用数量众多的数组方法。**

**Array.from({ length: 2 }, () => 'jack')**

**// ['jack', 'jack']**

**// 上面代码中，Array.from的第一个参数指定了第二个参数运行的次数。这种特性可以让该方法的用法变得非常灵活。**

**// Array.from()的另一个应用是，将字符串转为数组，然后返回字符串的长度。因为它能正确处理各种 Unicode 字符，可以避免 JavaScript 将大于\uFFFF的 Unicode 字符，算作两个字符的 bug。**

**function countSymbols(string) {**

**return Array.from(string).length;**

**}**

**}**

## // 3.Array.of()

**{**

**// Array.of方法用于将一组值，转换为数组。**

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

**Array.of(3) // [3]**

**Array.of(3).length // 1**

**// 这个方法的主要目的，是弥补数组构造函数Array()的不足。因为参数个数的不同，会导致Array()的行为有差异。**

**Array() // []**

**Array(3) // [, , ,]**

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

**// 上面代码中，Array方法没有参数、一个参数、三个参数时，返回结果都不一样。只有当参数个数不少于 2 个时，Array()才会返回由参数组成的新数组。参数个数只有一个时，实际上是指定数组的长度。**

**// Array.of基本上可以用来替代Array()或new Array()，并且不存在由于参数不同而导致的重载。它的行为非常统一。**

**Array.of() // []**

**Array.of(undefined) // [undefined]**

**Array.of(1) // [1]**

**Array.of(1, 2) // [1, 2]**

**// Array.of总是返回参数值组成的数组。如果没有参数，就返回一个空数组。**

**// Array.of方法可以用下面的代码模拟实现。**

**function ArrayOf(){**

**return [].slice.call(arguments);**

**}**

**}**

## // 4.数组实例的 copyWithin()

**{**

**// 数组实例的copyWithin方法，在当前数组内部，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。也就是说，使用这个方法，会修改当前数组。**

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

**// 它接受三个参数。**

**// 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。**

**// 这三个参数都应该是数值，如果不是，会自动转为数值。**

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

**// [4, 5, 3, 4, 5]**

**// 上面代码表示将从 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。**

**// 下面是更多例子。**

**// 将3号位复制到0号位**

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

**// [4, 2, 3, 4, 5]**

**// -2相当于3号位，-1相当于4号位**

**[1, 2, 3, 4, 5].copyWithin(0, -2, -1)**

**// [4, 2, 3, 4, 5]**

**// 将3号位复制到0号位**

**[].copyWithin.call({length: 5, 3: 1}, 0, 3)**

**// {0: 1, 3: 1, length: 5}**

**// 将2号位到数组结束，复制到0号位**

**let i32a = new Int32Array([1, 2, 3, 4, 5]);**

**i32a.copyWithin(0, 2);**

**// Int32Array [3, 4, 5, 4, 5]**

**// 对于没有部署 TypedArray 的 copyWithin 方法的平台**

**// 需要采用下面的写法**

**[].copyWithin.call(new Int32Array([1, 2, 3, 4, 5]), 0, 3, 4);**

**// Int32Array [4, 2, 3, 4, 5]**

**}**

## // 5.数组实例的 find() 和 findIndex()

**{**

**// 数组实例的find方法，用于找出第一个符合条件的数组成员。它的参数是一个回调函数，所有数组成员依次执行该回调函数，直到找出第一个返回值为true的成员，然后返回该成员。如果没有符合条件的成员，则返回undefined。**

**[1, 4, -5, 10].find((n) => n < 0)**

**// -5**

**// 上面代码找出数组中第一个小于 0 的成员。**

**[1, 5, 10, 15].find(function(value, index, arr) {**

**return value > 9;**

**}) // 10**

**// 上面代码中，find方法的回调函数可以接受三个参数，依次为当前的值、当前的位置和原数组。**

**// 数组实例的findIndex方法的用法与find方法非常类似，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**

**// 上面的代码中，find函数接收了第二个参数person对象，回调函数中的this对象指向person对象。**

**// 另外，这两个方法都可以发现NaN，弥补了数组的indexOf方法的不足。**

**[NaN].indexOf(NaN)**

**// -1**

**[NaN].findIndex(y => Object.is(NaN, y))**

**// 0**

**// 上面代码中，indexOf方法无法识别数组的NaN成员，但是findIndex方法可以借助Object.is方法做到。**

**}**

## // 6.数组实例的 fill()

**{**

**// fill方法使用给定值，填充一个数组。**

**['a', 'b', 'c'].fill(7)**

**// [7, 7, 7]**

**new Array(3).fill(7)**

**// [7, 7, 7]**

**// 上面代码表明，fill方法用于空数组的初始化非常方便。数组中已有的元素，会被全部抹去。**

**// fill方法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']**

**// 上面代码表示，fill方法从 1 号位开始，向原数组填充 7，到 2 号位之前结束。**

**// 注意，如果填充的类型为对象，那么被赋值的是同一个内存地址的对象，而不是深拷贝对象。**

**let arr = new Array(3).fill({name: "Mike"});**

**arr[0].name = "Ben";**

**arr**

**// [{name: "Ben"}, {name: "Ben"}, {name: "Ben"}]**

**let arr = new Array(3).fill([]);**

**arr[0].push(5);**

**arr**

**// [[5], [5], [5]]**

**}**

## // 7.数组实例的 entries()，keys() 和 values()

**{**

**// ES6 提供三个新的方法——entries()，keys()和values()——用于遍历数组。它们都返回一个遍历器对象（详见《Iterator》一章），可以用for...of循环进行遍历，唯一的区别是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。**

**for (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"**

**// 1 "b"**

**// 如果不使用for...of循环，可以手动调用遍历器对象的next方法，进行遍历。**

**let letter = ['a', 'b', 'c'];**

**let entries = letter.entries();**

**console.log(entries.next().value); // [0, 'a']**

**console.log(entries.next().value); // [1, 'b']**

**console.log(entries.next().value); // [2, 'c']**

**}**

## // 8.数组实例的 includes()

**{**

**// Array.prototype.includes方法返回一个布尔值，表示某个数组是否包含给定的值，与字符串的includes方法类似。ES2016 引入了该方法。**

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

**[1, 2, 3].includes(4) // false**

**[1, 2, NaN].includes(NaN) // true**

**// 该方法的第二个参数表示搜索的起始位置，默认为0。如果第二个参数为负数，则表示倒数的位置，如果这时它大于数组长度（比如第二个参数为-4，但数组长度为3），则会重置为从0开始。**

**[1, 2, 3].includes(3, 3); // false**

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

**// 没有该方法之前，我们通常使用数组的indexOf方法，检查是否包含某个值。**

**if (arr.indexOf(el) !== -1) {**

**// ...**

**}**

**// indexOf方法有两个缺点，一是不够语义化，它的含义是找到参数值的第一个出现位置，所以要去比较是否不等于-1，表达起来不够直观。二是，它内部使用严格相等运算符（===）进行判断，这会导致对NaN的误判。**

**[NaN].indexOf(NaN)**

**// -1**

**// includes使用的是不一样的判断算法，就没有这个问题。**

**[NaN].includes(NaN)**

**// true**

**// 下面代码用来检查当前环境是否支持该方法，如果不支持，部署一个简易的替代版本。**

**const contains = (() =>**

**Array.prototype.includes**

**? (arr, value) => arr.includes(value)**

**: (arr, value) => arr.some(el => el === value)**

**)();**

**contains(['foo', 'bar'], 'baz'); // => false**

**// 另外，Map 和 Set 数据结构有一个has方法，需要注意与includes区分。**

**// Map 结构的has方法，是用来查找键名的，比如Map.prototype.has(key)、WeakMap.prototype.has(key)、Reflect.has(target, propertyKey)。**

**// Set 结构的has方法，是用来查找值的，比如Set.prototype.has(value)、WeakSet.prototype.has(value)。**

**}**

## // 9.数组的空位

**{**

**// 数组的空位指，数组的某一个位置没有任何值。比如，Array构造函数返回的数组都是空位。**

**Array(3) // [, , ,]**

**// 上面代码中，Array(3)返回一个具有 3 个空位的数组。**

**// 注意，空位不是undefined，一个位置的值等于undefined，依然是有值的。空位是没有任何值，in运算符可以说明这一点。**

**0 in [undefined, undefined, undefined] // true**

**0 in [, , ,] // false**

**// 上面代码说明，第一个数组的 0 号位置是有值的，第二个数组的 0 号位置没有值。**

**// ES5 对空位的处理，已经很不一致了，大多数情况下会忽略空位。**

**// forEach(), filter(), reduce(), every() 和some()都会跳过空位。**

**// map()会跳过空位，但会保留这个值**

**// join()和toString()会将空位视为undefined，而undefined和null会被处理成空字符串。**

**// forEach方法**

**[,'a'].forEach((x,i) => console.log(i)); // 1**

**// filter方法**

**['a',,'b'].filter(x => true) // ['a','b']**

**// every方法**

**[,'a'].every(x => x==='a') // true**

**// reduce方法**

**[1,,2].reduce((x,y) => return x+y) // 3**

**// some方法**

**[,'a'].some(x => x !== 'a') // false**

**// map方法**

**[,'a'].map(x => 1) // [,1]**

**// join方法**

**[,'a',undefined,null].join('#') // "#a##"**

**// toString方法**

**[,'a',undefined,null].toString() // ",a,,"**

**// ES6 则是明确将空位转为undefined。**

**// Array.from方法会将数组的空位，转为undefined，也就是说，这个方法不会忽略空位。**

**Array.from(['a',,'b'])**

**// [ "a", undefined, "b" ]**

**// 扩展运算符（...）也会将空位转为undefined。**

**[...['a',,'b']]**

**// [ "a", undefined, "b" ]**

**// copyWithin()会连空位一起拷贝。**

**[,'a','b',,].copyWithin(2,0) // [,"a",,"a"]**

**// fill()会将空位视为正常的数组位置。**

**new Array(3).fill('a') // ["a","a","a"]**

**// for...of循环也会遍历空位。**

**let arr = [, ,];**

**for (let i of arr) {**

**console.log(1);**

**}**

**// 1**

**// 1**

**// 上面代码中，数组arr有两个空位，for...of并没有忽略它们。如果改成map方法遍历，空位是会跳过的。**

**// entries()、keys()、values()、find()和findIndex()会将空位处理成undefined。**

**// entries()**

**[...[,'a'].entries()] // [[0,undefined], [1,"a"]]**

**// keys()**

**[...[,'a'].keys()] // [0,1]**

**// values()**

**[...[,'a'].values()] // [undefined,"a"]**

**// find()**

**[,'a'].find(x => true) // undefined**

**// findIndex()**

**[,'a'].findIndex(x => true) // 0**

**// 由于空位的处理规则非常不统一，所以建议避免出现空位。**

**}**

# 对象的扩展

## // 1.属性的简洁表示法

**{**

**// ES6 允许直接写入变量和函数，作为对象的属性和方法。这样的书写更加简洁。**

**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); }**

**};**

**}**

## // 2.属性名表达式

**{**

**// JavaScript 定义对象的属性，有两种方法。**

**// 方法一**

**obj.foo = true;**

**// 方法二**

**obj['a' + 'bc'] = 123;**

**// 上面代码的方法一是直接用标识符作为属性名，方法二是用表达式作为属性名，这时要将表达式放在方括号之内。**

**// 但是，如果使用字面量方式定义对象（使用大括号），在 ES5 中只能使用方法一（标识符）定义属性。**

**var obj = {**

**foo: true,**

**abc: 123**

**};**

**// ES6 允许字面量定义对象时，用方法二（表达式）作为对象的属性名，即把表达式放在方括号内。**

**let propKey = 'foo';**

**let obj = {**

**[propKey]: true,**

**['a' + 'bc']: 123**

**};**

**// 下面是另一个例子。**

**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]，这一点要特别小心。**

**const keyA = {a: 1};**

**const keyB = {b: 2};**

**const myObject = {**

**[keyA]: 'valueA',**

**[keyB]: 'valueB'**

**};**

**myObject // Object {[object Object]: "valueB"}**

**// 上面代码中，[keyA]和[keyB]得到的都是[object Object]，所以[keyB]会把[keyA]覆盖掉，而myObject最后只有一个[object Object]属性。**

**}**

## // 3.方法的 name 属性

**{**

**// 函数的name属性，返回函数名。对象方法也是函数，因此也有name属性。**

**const person = {**

**sayName() {**

**console.log('hello!');**

**},**

**};**

**// person.sayName.name // "sayName"**

**// 上面代码中，方法的name属性返回函数名（即方法名）。**

**// 如果对象的方法使用了取值函数（getter）和存值函数（setter），则name属性不是在该方法上面，而是该方法的属性的描述对象的get和set属性上面，返回值是方法名前加上get和set。**

**const obj = {**

**get foo() {},**

**set foo(x) {}**

**};**

**obj.foo.name**

**// TypeError: Cannot read property 'name' of undefined**

**const descriptor = Object.getOwnPropertyDescriptor(obj, 'foo');**

**descriptor.get.name // "get foo"**

**descriptor.set.name // "set foo"**

**// 有两种特殊情况：bind方法创造的函数，name属性返回bound加上原函数的名字；Function构造函数创造的函数，name属性返回anonymous。**

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

**var doSomething = function() {**

**// ...**

**};**

**doSomething.bind().name // "bound doSomething"**

**// 如果对象的方法是一个 Symbol 值，那么name属性返回的是这个 Symbol 值的描述。**

**const key1 = Symbol('description');**

**const key2 = Symbol();**

**let obj = {**

**[key1]() {},**

**[key2]() {},**

**};**

**obj[key1].name // "[description]"**

**obj[key2].name // ""**

**// 上面代码中，key1对应的 Symbol 值有描述，key2没有。**

**}**

## // 4.Object.is()

**{**

**// ES5 比较两个值是否相等，只有两个运算符：相等运算符（==）和严格相等运算符（===）。它们都有缺点，前者会自动转换数据类型，后者的NaN不等于自身，以及+0等于-0。JavaScript 缺乏一种运算，在所有环境中，只要两个值是一样的，它们就应该相等。**

**// ES6 提出“Same-value equality”（同值相等）算法，用来解决这个问题。Object.is就是部署这个算法的新方法。它用来比较两个值是否严格相等，与严格比较运算符（===）的行为基本一致。**

**Object.is('foo', 'foo')**

**// true**

**Object.is({}, {})**

**// false**

**// 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**

**// ES5 可以通过下面的代码，部署Object.is。**

**Object.defineProperty(Object, 'is', {**

**value: function(x, y) {**

**if (x === y) {**

**// 针对+0 不等于 -0的情况**

**return x !== 0 || 1 / x === 1 / y;**

**}**

**// 针对NaN的情况**

**return x !== x && y !== y;**

**},**

**configurable: true,**

**enumerable: false,**

**writable: true**

**});**

**}**

## // 5.Object.assign()

## ES 5扩展

**Object.create(proto, [propertiesObject])**

**proto**

**新创建对象的原型对象。**

**propertiesObject**

**可选。如果没有指定为 undefined，则是要添加到新创建对象的可枚举属性（即其自身定义的属性，而不是其原型链上的枚举属性）对象的属性描述符以及相应的属性名称。这些属性对应Object.defineProperties()的第二个参数。**

**返回值：一个新对象，带着指定的原型对象和属性。**

**var obj = {**

**name : "tom",**

**age : 20**

**};**

**var obj2 = {};**

**obj2 = Object.create(obj,{});**

**console.log(obj2);**

**obj2 = Object.create(obj,{**

**sex : {**

**value : "male",**

**writable : true,**

**configurable : true,**

**enumerable : true**

**}**

**});**

**{**

**// 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）。**

**const target = { a: 1 };**

**const source1 = { b: 2 };**

**const source2 = { c: 3 };**

**Object.assign(target, source1, source2);**

**target // {a:1, b:2, c:3}**

**// 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。**

**// 注意，如果目标对象与源对象有同名属性，或多个源对象有同名属性，则后面的属性会覆盖前面的属性。**

**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}**

**// 如果只有一个参数，Object.assign会直接返回该参数。**

**const obj = {a: 1};**

**Object.assign(obj) === obj // true**

**// 如果该参数不是对象，则会先转成对象，然后返回。**

**typeof Object.assign(2) // "object"**

**// 由于undefined和null无法转成对象，所以如果它们作为参数，就会报错。**

**Object.assign(undefined) // 报错**

**Object.assign(null) // 报错**

**// 如果非对象参数出现在源对象的位置（即非首参数），那么处理规则有所不同。首先，这些参数都会转成对象，如果无法转成对象，就会跳过。这意味着，如果undefined和null不在首参数，就不会报错。**

**let obj = {a: 1};**

**Object.assign(obj, undefined) === obj // true**

**Object.assign(obj, null) === obj // true**

**// 其他类型的值（即数值、字符串和布尔值）不在首参数，也不会报错。但是，除了字符串会以数组形式，拷贝入目标对象，其他值都不会产生效果。**

**const v1 = 'abc';**

**const v2 = true;**

**const v3 = 10;**

**const obj = Object.assign({}, v1, v2, v3);**

**console.log(obj); // { "0": "a", "1": "b", "2": "c" }**

**// 上面代码中，v1、v2、v3分别是字符串、布尔值和数值，结果只有字符串合入目标对象（以字符数组的形式），数值和布尔值都会被忽略。这是因为只有字符串的包装对象，会产生可枚举属性。**

**// Object.assign拷贝的属性是有限制的，只拷贝源对象的自身属性（不拷贝继承属性），也不拷贝不可枚举的属性（enumerable: false）。**

**Object.assign({b: 'c'},**

**Object.defineProperty({}, 'invisible', {**

**enumerable: false,**

**value: 'hello'**

**})**

**)**

**// { b: 'c' }**

**// 上面代码中，Object.assign要拷贝的对象只有一个不可枚举属性invisible，这个属性并没有被拷贝进去。**

**// 属性名为 Symbol 值的属性，也会被Object.assign拷贝。**

**Object.assign({ a: 'b' }, { [Symbol('c')]: 'd' })**

**// { a: 'b', Symbol(c): 'd' }**

**// 注意点**

**// （1）浅拷贝**

**// Object.assign方法实行的是浅拷贝，而不是深拷贝。也就是说，如果源对象某个属性的值是对象，那么目标对象拷贝得到的是这个对象的引用。**

**const obj1 = {a: {b: 1}};**

**const obj2 = Object.assign({}, obj1);**

**obj1.a.b = 2;**

**obj2.a.b // 2**

**// 上面代码中，源对象obj1的a属性的值是一个对象，Object.assign拷贝得到的是这个对象的引用。这个对象的任何变化，都会反映到目标对象上面。**

**// （2）同名属性的替换**

**// 对于这种嵌套的对象，一旦遇到同名属性，Object.assign的处理方法是替换，而不是添加。**

**const target = { a: { b: 'c', d: 'e' } }**

**const source = { a: { b: 'hello' } }**

**Object.assign(target, source)**

**// { a: { b: 'hello' } }**

**// 上面代码中，target对象的a属性被source对象的a属性整个替换掉了，而不会得到{ a: { b: 'hello', d: 'e' } }的结果。这通常不是开发者想要的，需要特别小心。**

**// 一些函数库提供Object.assign的定制版本（比如 Lodash 的\_.defaultsDeep方法），可以得到深拷贝的合并。**

**// （3）数组的处理**

**// Object.assign可以用来处理数组，但是会把数组视为对象。**

**Object.assign([1, 2, 3], [4, 5])**

**// [4, 5, 3]**

**// 上面代码中，Object.assign把数组视为属性名为 0、1、2 的对象，因此源数组的 0 号属性4覆盖了目标数组的 0 号属性1。**

**// （4）取值函数的处理**

**// Object.assign只能进行值的复制，如果要复制的值是一个取值函数，那么将求值后再复制。**

**const source = {**

**get foo() { return 1 }**

**};**

**const target = {};**

**Object.assign(target, source)**

**// { foo: 1 }**

**// 上面代码中，source对象的foo属性是一个取值函数，Object.assign不会复制这个取值函数，只会拿到值以后，将这个值复制过去。**

**// 常见用途**

**// Object.assign方法有很多用处。**

**// （1）为对象添加属性**

**class Point {**

**constructor(x, y) {**

**Object.assign(this, {x, y});**

**}**

**}**

**// 上面方法通过Object.assign方法，将x属性和y属性添加到Point类的对象实例。**

**// （2）为对象添加方法**

**Object.assign(SomeClass.prototype, {**

**someMethod(arg1, arg2) {**

**···**

**},**

**anotherMethod() {**

**···**

**}**

**});**

**// 等同于下面的写法**

**SomeClass.prototype.someMethod = function (arg1, arg2) {**

**···**

**};**

**SomeClass.prototype.anotherMethod = function () {**

**···**

**};**

**// 上面代码使用了对象属性的简洁表示法，直接将两个函数放在大括号中，再使用assign方法添加到SomeClass.prototype之中。**

**// （3）克隆对象**

**function clone(origin) {**

**return Object.assign({}, origin);**

**}**

**// 上面代码将原始对象拷贝到一个空对象，就得到了原始对象的克隆。**

**// 不过，采用这种方法克隆，只能克隆原始对象自身的值，不能克隆它继承的值。如果想要保持继承链，可以采用下面的代码。**

**function clone(origin) {**

**let originProto = Object.getPrototypeOf(origin);**

**return Object.assign(Object.create(originProto), origin);**

**}**

**// （4）合并多个对象**

**// 将多个对象合并到某个对象。**

**const merge =**

**(target, ...sources) => Object.assign(target, ...sources);**

**// 如果希望合并后返回一个新对象，可以改写上面函数，对一个空对象合并。**

**const merge =**

**(...sources) => Object.assign({}, ...sources);**

**// （5）为属性指定默认值**

**const DEFAULTS = {**

**logLevel: 0,**

**outputFormat: 'html'**

**};**

**function processContent(options) {**

**options = Object.assign({}, DEFAULTS, options);**

**console.log(options);**

**// ...**

**}**

**// 上面代码中，DEFAULTS对象是默认值，options对象是用户提供的参数。Object.assign方法将DEFAULTS和options合并成一个新对象，如果两者有同名属性，则option的属性值会覆盖DEFAULTS的属性值。**

**// 注意，由于存在浅拷贝的问题，DEFAULTS对象和options对象的所有属性的值，最好都是简单类型，不要指向另一个对象。否则，DEFAULTS对象的该属性很可能不起作用。**

**const DEFAULTS = {**

**url: {**

**host: 'example.com',**

**port: 7070**

**},**

**};**

**processContent({ url: {port: 8000} })**

**// {**

**// url: {port: 8000}**

**// }**

**// 上面代码的原意是将url.port改成 8000，url.host不变。实际结果却是options.url覆盖掉DEFAULTS.url，所以url.host就不存在了。**

**}**

## // 6.属性的可枚举性和遍历

**{**

**// 可枚举性**

**// 对象的每个属性都有一个描述对象（Descriptor），用来控制该属性的行为。Object.getOwnPropertyDescriptor方法可以获取该属性的描述对象。**

**let obj = { foo: 123 };**

**Object.getOwnPropertyDescriptor(obj, 'foo')**

**// {**

**// value: 123,**

**// writable: true,**

**// enumerable: true,**

**// configurable: true**

**// }**

**// 描述对象的enumerable属性，称为”可枚举性“，如果该属性为false，就表示某些操作会忽略当前属性。**

**// 目前，有四个操作会忽略enumerable为false的属性。**

**// for...in循环：只遍历对象自身的和继承的可枚举的属性。**

**// Object.keys()：返回对象自身的所有可枚举的属性的键名。**

**// JSON.stringify()：只串行化对象自身的可枚举的属性。**

**// Object.assign()： 忽略enumerable为false的属性，只拷贝对象自身的可枚举的属性。**

**// 这四个操作之中，前三个是 ES5 就有的，最后一个Object.assign()是 ES6 新增的。其中，只有for...in会返回继承的属性，其他三个方法都会忽略继承的属性，只处理对象自身的属性。实际上，引入“可枚举”（enumerable）这个概念的最初目的，就是让某些属性可以规避掉for...in操作，不然所有内部属性和方法都会被遍历到。比如，对象原型的toString方法，以及数组的length属性，就通过“可枚举性”，从而避免被for...in遍历到。**

**Object.getOwnPropertyDescriptor(Object.prototype, 'toString').enumerable**

**// false**

**Object.getOwnPropertyDescriptor([], 'length').enumerable**

**// false**

**// 上面代码中，toString和length属性的enumerable都是false，因此for...in不会遍历到这两个继承自原型的属性。**

**// 另外，ES6 规定，所有 Class 的原型的方法都是不可枚举的。**

**Object.getOwnPropertyDescriptor(class {foo() {}}.prototype, 'foo').enumerable**

**// false**

**// 总的来说，操作中引入继承的属性会让问题复杂化，大多数时候，我们只关心对象自身的属性。所以，尽量不要用for...in循环，而用Object.keys()代替。**

**// 属性的遍历**

**// ES6 一共有 5 种方法可以遍历对象的属性。**

**// （1）for...in**

**// 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）。**

**// （2）Object.keys(obj)**

**// Object.keys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。**

**// （3）Object.getOwnPropertyNames(obj)**

**// Object.getOwnPropertyNames返回一个数组，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。**

**// （4）Object.getOwnPropertySymbols(obj)**

**// Object.getOwnPropertySymbols返回一个数组，包含对象自身的所有 Symbol 属性的键名。**

**// （5）Reflect.ownKeys(obj)**

**// Reflect.ownKeys返回一个数组，包含对象自身的所有键名，不管键名是 Symbol 或字符串，也不管是否可枚举。**

**// 以上的 5 种方法遍历对象的键名，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列。**

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

**// // ['2', '10', 'b', 'a', Symbol()]**

**// 上面代码中，Reflect.ownKeys方法返回一个数组，包含了参数对象的所有属性。这个数组的属性次序是这样的，首先是数值属性2和10，其次是字符串属性b和a，最后是 Symbol 属性。**

**}**

## // 7.Object.getOwnPropertyDescriptors()

**{**

**// 前面说过，Object.getOwnPropertyDescriptor方法会返回某个对象属性的描述对象（descriptor）。ES2017 引入了Object.getOwnPropertyDescriptors方法，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 } }**

**// 上面代码中，Object.getOwnPropertyDescriptors方法返回一个对象，所有原对象的属性名都是该对象的属性名，对应的属性值就是该属性的描述对象。**

**// 该方法的实现非常容易。**

**function getOwnPropertyDescriptors(obj) {**

**const result = {};**

**for (let key of Reflect.ownKeys(obj)) {**

**result[key] = Object.getOwnPropertyDescriptor(obj, key);**

**}**

**return result;**

**}**

**}**

## // 8.\_\_proto\_\_属性，Object.setPrototypeOf()，Object.getPrototypeOf()

**{**

**// \_\_proto\_\_属性**

**// \_\_proto\_\_属性（前后各两个下划线），用来读取或设置当前对象的prototype对象。目前，所有浏览器（包括 IE11）都部署了这个属性。**

**// es5 的写法**

**const obj = {**

**method: function() { ... }**

**};**

**obj.\_\_proto\_\_ = someOtherObj;**

**// es6 的写法**

**var obj = Object.create(someOtherObj);**

**obj.method = function() { ... };**

**// 该属性没有写入 ES6 的正文，而是写入了附录，原因是\_\_proto\_\_前后的双下划线，说明它本质上是一个内部属性，而不是一个正式的对外的 API，只是由于浏览器广泛支持，才被加入了 ES6。标准明确规定，只有浏览器必须部署这个属性，其他运行环境不一定需要部署，而且新的代码最好认为这个属性是不存在的。因此，无论从语义的角度，还是从兼容性的角度，都不要使用这个属性，而是使用下面的Object.setPrototypeOf()（写操作）、Object.getPrototypeOf()（读操作）、Object.create()（生成操作）代替。**

**// 实现上，\_\_proto\_\_调用的是Object.prototype.\_\_proto\_\_，具体实现如下。**

**Object.defineProperty(Object.prototype, '\_\_proto\_\_', {**

**get() {**

**let \_thisObj = Object(this);**

**return Object.getPrototypeOf(\_thisObj);**

**},**

**set(proto) {**

**if (this === undefined || this === null) {**

**throw new TypeError();**

**}**

**if (!isObject(this)) {**

**return undefined;**

**}**

**if (!isObject(proto)) {**

**return undefined;**

**}**

**let status = Reflect.setPrototypeOf(this, proto);**

**if (!status) {**

**throw new TypeError();**

**}**

**},**

**});**

**function isObject(value) {**

**return Object(value) === value;**

**}**

**// 如果一个对象本身部署了\_\_proto\_\_属性，该属性的值就是对象的原型。**

**Object.getPrototypeOf({ \_\_proto\_\_: null })**

**// null**

**Object.setPrototypeOf()**

**// Object.setPrototypeOf方法的作用与\_\_proto\_\_相同，用来设置一个对象的prototype对象，返回参数对象本身。它是 ES6 正式推荐的设置原型对象的方法。**

**// 格式**

**Object.setPrototypeOf(object, prototype)**

**// 用法**

**const o = Object.setPrototypeOf({}, null);**

**// 该方法等同于下面的函数。**

**function (obj, proto) {**

**obj.\_\_proto\_\_ = proto;**

**return obj;**

**}**

**// 下面是一个例子。**

**let proto = {};**

**let obj = { x: 10 };**

**Object.setPrototypeOf(obj, proto);**

**proto.y = 20;**

**proto.z = 40;**

**obj.x // 10**

**obj.y // 20**

**obj.z // 40**

**// 上面代码将proto对象设为obj对象的原型，所以从obj对象可以读取proto对象的属性。**

**// 如果第一个参数不是对象，会自动转为对象。但是由于返回的还是第一个参数，所以这个操作不会产生任何效果。**

**Object.setPrototypeOf(1, {}) === 1 // true**

**Object.setPrototypeOf('foo', {}) === 'foo' // true**

**Object.setPrototypeOf(true, {}) === true // true**

**// 由于undefined和null无法转为对象，所以如果第一个参数是undefined或null，就会报错。**

**Object.setPrototypeOf(undefined, {})**

**// TypeError: Object.setPrototypeOf called on null or undefined**

**Object.setPrototypeOf(null, {})**

**// TypeError: Object.setPrototypeOf called on null or undefined**

**Object.getPrototypeOf()**

**// 该方法与Object.setPrototypeOf方法配套，用于读取一个对象的原型对象。**

**Object.getPrototypeOf(obj);**

**// 下面是一个例子。**

**function Rectangle() {**

**// ...**

**}**

**const rec = new Rectangle();**

**Object.getPrototypeOf(rec) === Rectangle.prototype**

**// true**

**Object.setPrototypeOf(rec, Object.prototype);**

**Object.getPrototypeOf(rec) === Rectangle.prototype**

**// false**

**// 如果参数不是对象，会被自动转为对象。**

**// 等同于 Object.getPrototypeOf(Number(1))**

**Object.getPrototypeOf(1)**

**// Number {[[PrimitiveValue]]: 0}**

**// 等同于 Object.getPrototypeOf(String('foo'))**

**Object.getPrototypeOf('foo')**

**// String {length: 0, [[PrimitiveValue]]: ""}**

**// 等同于 Object.getPrototypeOf(Boolean(true))**

**Object.getPrototypeOf(true)**

**// Boolean {[[PrimitiveValue]]: false}**

**Object.getPrototypeOf(1) === Number.prototype // true**

**Object.getPrototypeOf('foo') === String.prototype // true**

**Object.getPrototypeOf(true) === Boolean.prototype // true**

**// 如果参数是undefined或null，它们无法转为对象，所以会报错。**

**Object.getPrototypeOf(null)**

**// TypeError: Cannot convert undefined or null to object**

**Object.getPrototypeOf(undefined)**

**// TypeError: Cannot convert undefined or null to object**

**}**

## // 9.super 关键字

**{**

**// 我们知道，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);**

**obj.find() // "hello"**

**// 上面代码中，对象obj的find方法之中，通过super.foo引用了原型对象proto的foo属性。**

**// 注意，super关键字表示原型对象时，只能用在对象的方法之中，用在其他地方都会报错。**

**// 报错**

**const obj = {**

**foo: super.foo**

**}**

**// 报错**

**const obj = {**

**foo: () => super.foo**

**}**

**// 报错**

**const obj = {**

**foo: function () {**

**return super.foo**

**}**

**}**

**// 上面三种super的用法都会报错，因为对于 JavaScript 引擎来说，这里的super都没有用在对象的方法之中。第一种写法是super用在属性里面，第二种和第三种写法是super用在一个函数里面，然后赋值给foo属性。目前，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。**

**// JavaScript 引擎内部，super.foo等同于Object.getPrototypeOf(this).foo（属性）或Object.getPrototypeOf(this).foo.call(this)（方法）。**

**const proto = {**

**x: 'hello',**

**foo() {**

**console.log(this.x);**

**},**

**};**

**const obj = {**

**x: 'world',**

**foo() {**

**super.foo();**

**}**

**}**

**Object.setPrototypeOf(obj, proto);**

**obj.foo() // "world"**

**// 上面代码中，super.foo指向原型对象proto的foo方法，但是绑定的this却还是当前对象obj，因此输出的就是world。**

**}**

## // 10.Object.keys()，Object.values()，Object.entries()

**{**

**// Object.keys()**

**// ES5 引入了Object.keys方法，返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（enumerable）属性的键名。**

**var obj = { foo: 'bar', baz: 42 };**

**Object.keys(obj)**

**// ["foo", "baz"]**

**// ES2017 引入了跟Object.keys配套的Object.values和Object.entries，作为遍历一个对象的补充手段，供for...of循环使用。**

**let {keys, values, entries} = Object;**

**let obj = { a: 1, b: 2, c: 3 };**

**for (let key of keys(obj)) {**

**console.log(key); // 'a', 'b', 'c'**

**}**

**for (let value of values(obj)) {**

**console.log(value); // 1, 2, 3**

**}**

**for (let [key, value] of entries(obj)) {**

**console.log([key, value]); // ['a', 1], ['b', 2], ['c', 3]**

**}**

**Object.values()**

**// Object.values方法返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（enumerable）属性的键值。**

**const obj = { foo: 'bar', baz: 42 };**

**Object.values(obj)**

**// ["bar", 42]**

**// 返回数组的成员顺序，与本章的《属性的遍历》部分介绍的排列规则一致。**

**const obj = { 100: 'a', 2: 'b', 7: 'c' };**

**Object.values(obj)**

**// ["b", "c", "a"]**

**// 上面代码中，属性名为数值的属性，是按照数值大小，从小到大遍历的，因此返回的顺序是b、c、a。**

**// Object.values只返回对象自身的可遍历属性。**

**const obj = Object.create({}, {p: {value: 42}});**

**Object.values(obj) // []**

**// 上面代码中，Object.create方法的第二个参数添加的对象属性（属性p），如果不显式声明，默认是不可遍历的，因为p的属性描述对象的enumerable默认是false，Object.values不会返回这个属性。只要把enumerable改成true，Object.values就会返回属性p的值。**

**const obj = Object.create({}, {p:**

**{**

**value: 42,**

**enumerable: true**

**}**

**});**

**Object.values(obj) // [42]**

**// Object.values会过滤属性名为 Symbol 值的属性。**

**Object.values({ [Symbol()]: 123, foo: 'abc' });**

**// ['abc']**

**// 如果Object.values方法的参数是一个字符串，会返回各个字符组成的一个数组。**

**Object.values('foo')**

**// ['f', 'o', 'o']**

**// 上面代码中，字符串会先转成一个类似数组的对象。字符串的每个字符，就是该对象的一个属性。因此，Object.values返回每个属性的键值，就是各个字符组成的一个数组。**

**// 如果参数不是对象，Object.values会先将其转为对象。由于数值和布尔值的包装对象，都不会为实例添加非继承的属性。所以，Object.values会返回空数组。**

**Object.values(42) // []**

**Object.values(true) // []**

**Object.entries**

**// Object.entries方法返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（enumerable）属性的键值对数组。**

**const obj = { foo: 'bar', baz: 42 };**

**Object.entries(obj)**

**// [ ["foo", "bar"], ["baz", 42] ]**

**// 除了返回值不一样，该方法的行为与Object.values基本一致。**

**// 如果原对象的属性名是一个 Symbol 值，该属性会被忽略。**

**Object.entries({ [Symbol()]: 123, foo: 'abc' });**

**// [ [ 'foo', 'abc' ] ]**

**// 上面代码中，原对象有两个属性，Object.entries只输出属性名非 Symbol 值的属性。将来可能会有Reflect.ownEntries()方法，返回对象自身的所有属性。**

**// Object.entries的基本用途是遍历对象的属性。**

**let obj = { one: 1, two: 2 };**

**for (let [k, v] of Object.entries(obj)) {**

**console.log(**

**`${JSON.stringify(k)}: ${JSON.stringify(v)}`**

**);**

**}**

**// "one": 1**

**// "two": 2**

**// Object.entries方法的另一个用处是，将对象转为真正的Map结构。**

**const obj = { foo: 'bar', baz: 42 };**

**const map = new Map(Object.entries(obj));**

**map // Map { foo: "bar", baz: 42 }**

**// 自己实现Object.entries方法，非常简单。**

**// Generator函数的版本**

**function\* entries(obj) {**

**for (let key of Object.keys(obj)) {**

**yield [key, obj[key]];**

**}**

**}**

**// 非Generator函数的版本**

**function entries(obj) {**

**let arr = [];**

**for (let key of Object.keys(obj)) {**

**arr.push([key, obj[key]]);**

**}**

**return arr;**

**}**

**}**

## // 11.对象的扩展运算符

**{**

**// 解构赋值**

**// 对象的解构赋值用于从一个对象取值，相当于将目标对象自身的所有可遍历的（enumerable）、但尚未被读取的属性，分配到指定的对象上面。所有的键和它们的值，都会拷贝到新对象上面。**

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

**x // 1**

**y // 2**

**z // { a: 3, b: 4 }**

**// 上面代码中，变量z是解构赋值所在的对象。它获取等号右边的所有尚未读取的键（a和b），将它们连同值一起拷贝过来。**

**// 由于解构赋值要求等号右边是一个对象，所以如果等号右边是undefined或null，就会报错，因为它们无法转为对象。**

**let { x, y, ...z } = null; // 运行时错误**

**let { x, y, ...z } = undefined; // 运行时错误**

**// 解构赋值必须是最后一个参数，否则会报错。**

**let { ...x, y, z } = obj; // 句法错误**

**let { x, ...y, ...z } = obj; // 句法错误**

**// 上面代码中，解构赋值不是最后一个参数，所以会报错。**

**// 注意，解构赋值的拷贝是浅拷贝，即如果一个键的值是复合类型的值（数组、对象、函数）、那么解构赋值拷贝的是这个值的引用，而不是这个值的副本。**

**let obj = { a: { b: 1 } };**

**let { ...x } = obj;**

**obj.a.b = 2;**

**x.a.b // 2**

**// 上面代码中，x是解构赋值所在的对象，拷贝了对象obj的a属性。a属性引用了一个对象，修改这个对象的值，会影响到解构赋值对它的引用。**

**// 另外，扩展运算符的解构赋值，不能复制继承自原型对象的属性。**

**let o1 = { a: 1 };**

**let o2 = { b: 2 };**

**o2.\_\_proto\_\_ = o1;**

**let { ...o3 } = o2;**

**o3 // { b: 2 }**

**o3.a // undefined**

**// 上面代码中，对象o3复制了o2，但是只复制了o2自身的属性，没有复制它的原型对象o1的属性。**

**// 下面是另一个例子。**

**const o = Object.create({ x: 1, y: 2 });**

**o.z = 3;**

**let { x, ...newObj } = o;**

**let { y, z } = newObj;**

**x // 1**

**y // undefined**

**z // 3**

**// 上面代码中，变量x是单纯的解构赋值，所以可以读取对象o继承的属性；变量y和z是扩展运算符的解构赋值，只能读取对象o自身的属性，所以变量z可以赋值成功，变量y取不到值。ES6 规定，变量声明语句之中，如果使用解构赋值，扩展运算符后面必须是一个变量名，而不能是一个解构赋值表达式，所以上面代码引入了中间变量newObj，如果写成下面这样会报错。**

**let { x, ...{ y, z } } = o;**

**// SyntaxError: ... must be followed by an identifier in declaration contexts**

**// 解构赋值的一个用处，是扩展某个函数的参数，引入其他操作。**

**function baseFunction({ a, b }) {**

**// ...**

**}**

**function wrapperFunction({ x, y, ...restConfig }) {**

**// 使用 x 和 y 参数进行操作**

**// 其余参数传给原始函数**

**return baseFunction(restConfig);**

**}**

**// 上面代码中，原始函数baseFunction接受a和b作为参数，函数wrapperFunction在baseFunction的基础上进行了扩展，能够接受多余的参数，并且保留原始函数的行为。**

**// 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 }**

**// 这等同于使用Object.assign方法。**

**let aClone = { ...a };**

**// 等同于**

**let aClone = Object.assign({}, a);**

**// 上面的例子只是拷贝了对象实例的属性，如果想完整克隆一个对象，还拷贝对象原型的属性，可以采用下面的写法。**

**// 写法一**

**const clone1 = {**

**\_\_proto\_\_: Object.getPrototypeOf(obj),**

**...obj**

**};**

**// 写法二**

**const clone2 = Object.assign(**

**Object.create(Object.getPrototypeOf(obj)),**

**obj**

**);**

**// 写法三**

**const clone3 = Object.create(**

**Object.getPrototypeOf(obj),**

**Object.getOwnPropertyDescriptors(obj)**

**)**

**// 上面代码中，写法一的\_\_proto\_\_属性在非浏览器的环境不一定部署，因此推荐使用写法二和写法三。**

**// 扩展运算符可以用于合并两个对象。**

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

**// 等同于**

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

**// 如果用户自定义的属性，放在扩展运算符后面，则扩展运算符内部的同名属性会被覆盖掉。**

**let aWithOverrides = { ...a, x: 1, y: 2 };**

**// 等同于**

**let aWithOverrides = { ...a, ...{ x: 1, y: 2 } };**

**// 等同于**

**let x = 1, y = 2, aWithOverrides = { ...a, x, y };**

**// 等同于**

**let aWithOverrides = Object.assign({}, a, { x: 1, y: 2 });**

**// 上面代码中，a对象的x属性和y属性，拷贝到新对象后会被覆盖掉。**

**// 这用来修改现有对象部分的属性就很方便了。**

**let newVersion = {**

**...previousVersion,**

**name: 'New Name' // Override the name property**

**};**

**// 上面代码中，newVersion对象自定义了name属性，其他属性全部复制自previousVersion对象。**

**// 如果把自定义属性放在扩展运算符前面，就变成了设置新对象的默认属性值。**

**let aWithDefaults = { x: 1, y: 2, ...a };**

**// 等同于**

**// even if property keys don’t clash, because objects record insertion order:**

**let aWithDefaults = Object.assign({}, { x: 1, y: 2 }, a);**

**// 等同于**

**let aWithDefaults = Object.assign({ x: 1, y: 2 }, a);**

**// 与数组的扩展运算符一样，对象的扩展运算符后面可以跟表达式。**

**const obj = {**

**...(x > 1 ? {a: 1} : {}),**

**b: 2,**

**};**

**// 如果扩展运算符后面是一个空对象，则没有任何效果。**

**{...{}, a: 1}**

**// { a: 1 }**

**// even if property keys don’t clash, because objects record insertion order:**

**// 如果扩展运算符的参数是null或undefined，这两个值会被忽略，不会报错。**

**let emptyObject = { ...null, ...undefined }; // 不报错**

**// 扩展运算符的参数对象之中，如果有取值函数get，这个函数是会执行的。**

**// 并不会抛出错误，因为 x 属性只是被定义，但没执行**

**let aWithXGetter = {**

**...a,**

**get x() {**

**throw new Error('not throw yet');**

**}**

**};**

**// 会抛出错误，因为 x 属性被执行了**

**let runtimeError = {**

**...a,**

**...{**

**get x() {**

**throw new Error('throw now');**

**}**

**}**

**};**

**}**

# Symbol

## // 1.概述

**{**

**// ES5 的对象属性名都是字符串，这容易造成属性名的冲突。ES6 引入了一种新的原始数据类型Symbol，表示独一无二的值。它是 JavaScript 语言的第七种数据类型，前六种是：undefined、null、布尔值（Boolean）、字符串（String）、数值（Number）、对象（Object）。Symbol 值通过Symbol函数生成。这就是说，对象的属性名现在可以有两种类型，一种是原来就有的字符串，另一种就是新增的 Symbol 类型。凡是属性名属于 Symbol 类型，就都是独一无二的，可以保证不会与其他属性名产生冲突。**

**let s = Symbol();**

**typeof s**

**// "symbol"**

**// 注意，Symbol函数前不能使用new命令，否则会报错。这是因为生成的 Symbol 是一个原始类型的值，不是对象。也就是说，由于 Symbol 值不是对象，所以不能添加属性。基本上，它是一种类似于字符串的数据类型。**

**// Symbol函数可以接受一个字符串作为参数，表示对 Symbol 实例的描述，主要是为了在控制台显示，或者转为字符串时，比较容易区分。**

**let s1 = Symbol('foo');**

**let s2 = Symbol('bar');**

**s1 // Symbol(foo)**

**s2 // Symbol(bar)**

**s1.toString() // "Symbol(foo)"**

**s2.toString() // "Symbol(bar)"**

**// 如果 Symbol 的参数是一个对象，就会调用该对象的toString方法，将其转为字符串，然后才生成一个 Symbol 值。**

**// 注意，Symbol函数的参数只是表示对当前 Symbol 值的描述，因此相同参数的Symbol函数的返回值是不相等的。**

**// 没有参数的情况**

**let s1 = Symbol();**

**let s2 = Symbol();**

**s1 === s2 // false**

**// 有参数的情况**

**let s1 = Symbol('foo');**

**let s2 = Symbol('foo');**

**s1 === s2 // false**

**// Symbol 值不能与其他类型的值进行运算，会报错。**

**let sym = Symbol('My symbol');**

**"your symbol is " + sym**

**// TypeError: can't convert symbol to string**

**`your symbol is ${sym}`**

**// TypeError: can't convert symbol to string**

**// 但是，Symbol 值可以显式转为字符串。**

**let sym = Symbol('My symbol');**

**String(sym) // 'Symbol(My symbol)'**

**sym.toString() // 'Symbol(My symbol)'**

**// 另外，Symbol 值也可以转为布尔值，但是不能转为数值。**

**let sym = Symbol();**

**Boolean(sym) // true**

**!sym // false**

**if (sym) {**

**// ...**

**}**

**Number(sym) // TypeError**

**sym + 2 // TypeError**

**}**

## // 2.作为属性名的 Symbol

**{**

**// 由于每一个 Symbol 值都是不相等的，这意味着 Symbol 值可以作为标识符，用于对象的属性名，就能保证不会出现同名的属性。这对于一个对象由多个模块构成的情况非常有用，能防止某一个键被不小心改写或覆盖。**

**let mySymbol = Symbol();**

**// 第一种写法**

**let a = {};**

**a[mySymbol] = 'Hello!';**

**// 第二种写法**

**let a = {**

**[mySymbol]: 'Hello!'**

**};**

**// 第三种写法**

**let a = {};**

**Object.defineProperty(a, mySymbol, { value: 'Hello!' });**

**// 以上写法都得到同样结果**

**a[mySymbol] // "Hello!"**

**// 注意，Symbol 值作为对象属性名时，不能用点运算符。**

**const mySymbol = Symbol();**

**const a = {};**

**a.mySymbol = 'Hello!';**

**a[mySymbol] // undefined**

**a['mySymbol'] // "Hello!"**

**// 同理，在对象的内部，使用 Symbol 值定义属性时，Symbol 值必须放在方括号之中。**

**let s = Symbol();**

**let obj = {**

**[s]: function (arg) { ... }**

**};**

**obj[s](123);**

**// 上面代码中，如果s不放在方括号中，该属性的键名就是字符串s，而不是s所代表的那个 Symbol 值。**

**// 采用增强的对象写法，上面代码的obj对象可以写得更简洁一些。**

**let obj = {**

**[s](arg) { ... }**

**};**

**// 还有一点需要注意，Symbol 值作为属性名时，该属性还是公开属性，不是私有属性。**

**}**

## // 3.实例：消除魔术字符串

**{**

**// 魔术字符串指的是，在代码之中多次出现、与代码形成强耦合的某一个具体的字符串或者数值。风格良好的代码，应该尽量消除魔术字符串，改由含义清晰的变量代替。**

**function getArea(shape, options) {**

**let area = 0;**

**switch (shape) {**

**case 'Triangle': // 魔术字符串**

**area = .5 \* options.width \* options.height;**

**break;**

**/\* ... more code ... \*/**

**}**

**return area;**

**}**

**getArea('Triangle', { width: 100, height: 100 }); // 魔术字符串**

**// 上面代码中，字符串Triangle就是一个魔术字符串。它多次出现，与代码形成“强耦合”，不利于将来的修改和维护。**

**// 常用的消除魔术字符串的方法，就是把它写成一个变量。**

**const shapeType = {**

**triangle: 'Triangle'**

**};**

**function getArea(shape, options) {**

**let area = 0;**

**switch (shape) {**

**case shapeType.triangle:**

**area = .5 \* options.width \* options.height;**

**break;**

**}**

**return area;**

**}**

**getArea(shapeType.triangle, { width: 100, height: 100 });**

**// 上面代码中，我们把Triangle写成shapeType对象的triangle属性，这样就消除了强耦合。**

**// 如果仔细分析，可以发现shapeType.triangle等于哪个值并不重要，只要确保不会跟其他shapeType属性的值冲突即可。因此，这里就很适合改用 Symbol 值。**

**const shapeType = {**

**triangle: Symbol()**

**};**

**// 上面代码中，除了将shapeType.triangle的值设为一个 Symbol，其他地方都不用修改。**

**}**

## // 4.属性名的遍历

**{**

**// Symbol 作为属性名，该属性不会出现在for...in、for...of循环中，也不会被Object.keys()、Object.getOwnPropertyNames()、JSON.stringify()返回。但是，它也不是私有属性，有一个Object.getOwnPropertySymbols方法，可以获取指定对象的所有 Symbol 属性名。**

**// Object.getOwnPropertySymbols方法返回一个数组，成员是当前对象的所有用作属性名的 Symbol 值。**

**const obj = {};**

**let a = Symbol('a');**

**let b = Symbol('b');**

**obj[a] = 'Hello';**

**obj[b] = 'World';**

**const objectSymbols = Object.getOwnPropertySymbols(obj);**

**objectSymbols**

**// [Symbol(a), Symbol(b)]**

**// 下面是另一个例子，Object.getOwnPropertySymbols方法与for...in循环、Object.getOwnPropertyNames方法进行对比的例子。**

**const obj = {};**

**let foo = Symbol("foo");**

**Object.defineProperty(obj, foo, {**

**value: "foobar",**

**});**

**for (let i in obj) {**

**console.log(i); // 无输出**

**}**

**Object.getOwnPropertyNames(obj)**

**// []**

**Object.getOwnPropertySymbols(obj)**

**// [Symbol(foo)]**

**// 上面代码中，使用Object.getOwnPropertyNames方法得不到Symbol属性名，需要使用Object.getOwnPropertySymbols方法。**

**// 另一个新的 API，Reflect.ownKeys方法可以返回所有类型的键名，包括常规键名和 Symbol 键名。**

**let obj = {**

**[Symbol('my\_key')]: 1,**

**enum: 2,**

**nonEnum: 3**

**};**

**Reflect.ownKeys(obj)**

**// ["enum", "nonEnum", Symbol(my\_key)]**

**// 由于以 Symbol 值作为名称的属性，不会被常规方法遍历得到。我们可以利用这个特性，为对象定义一些非私有的、但又希望只用于内部的方法。**

**let size = Symbol('size');**

**class Collection {**

**constructor() {**

**this[size] = 0;**

**}**

**add(item) {**

**this[this[size]] = item;**

**this[size]++;**

**}**

**static sizeOf(instance) {**

**return instance[size];**

**}**

**}**

**let x = new Collection();**

**Collection.sizeOf(x) // 0**

**x.add('foo');**

**Collection.sizeOf(x) // 1**

**Object.keys(x) // ['0']**

**Object.getOwnPropertyNames(x) // ['0']**

**Object.getOwnPropertySymbols(x) // [Symbol(size)]**

**// 上面代码中，对象x的size属性是一个 Symbol 值，所以Object.keys(x)、Object.getOwnPropertyNames(x)都无法获取它。这就造成了一种非私有的内部方法的效果。**

**}**

## // 5.Symbol.for()，Symbol.keyFor()

**{**

**// 有时，我们希望重新使用同一个 Symbol 值，Symbol.for方法可以做到这一点。它接受一个字符串作为参数，然后搜索有没有以该参数作为名称的 Symbol 值。如果有，就返回这个 Symbol 值，否则就新建并返回一个以该字符串为名称的 Symbol 值。**

**let s1 = Symbol.for('foo');**

**let s2 = Symbol.for('foo');**

**s1 === s2 // true**

**// 上面代码中，s1和s2都是 Symbol 值，但是它们都是同样参数的Symbol.for方法生成的，所以实际上是同一个值。**

**// Symbol.for()与Symbol()这两种写法，都会生成新的 Symbol。它们的区别是，前者会被登记在全局环境中供搜索，后者不会。Symbol.for()不会每次调用就返回一个新的 Symbol 类型的值，而是会先检查给定的key是否已经存在，如果不存在才会新建一个值。比如，如果你调用Symbol.for("cat")30 次，每次都会返回同一个 Symbol 值，但是调用Symbol("cat")30 次，会返回 30 个不同的 Symbol 值。**

**Symbol.for("bar") === Symbol.for("bar")**

**// true**

**Symbol("bar") === Symbol("bar")**

**// false**

**// Symbol.keyFor方法返回一个已登记的 Symbol 类型值的key。**

**let s1 = Symbol.for("foo");**

**Symbol.keyFor(s1) // "foo"**

**let s2 = Symbol("foo");**

**Symbol.keyFor(s2) // undefined**

**// 上面代码中，变量s2属于未登记的 Symbol 值，所以返回undefined。**

**// 需要注意的是，Symbol.for为 Symbol 值登记的名字，是全局环境的，可以在不同的 iframe 或 service worker 中取到同一个值。**

**iframe = document.createElement('iframe');**

**iframe.src = String(window.location);**

**document.body.appendChild(iframe);**

**iframe.contentWindow.Symbol.for('foo') === Symbol.for('foo')**

**// true**

**// 上面代码中，iframe 窗口生成的 Symbol 值，可以在主页面得到。**

**}**

## // 6.实例：模块的 Singleton 模式

**{**

**// Singleton 模式指的是调用一个类，任何时候返回的都是同一个实例。**

**// 对于 Node 来说，模块文件可以看成是一个类。怎么保证每次执行这个模块文件，返回的都是同一个实例呢？**

**// 很容易想到，可以把实例放到顶层对象global。**

**// mod.js**

**function A() {**

**this.foo = 'hello';**

**}**

**if (!global.\_foo) {**

**global.\_foo = new A();**

**}**

**module.exports = global.\_foo;**

**// 然后，加载上面的mod.js。**

**const a = require('./mod.js');**

**console.log(a.foo);**

**// 上面代码中，变量a任何时候加载的都是A的同一个实例。**

**// 但是，这里有一个问题，全局变量global.\_foo是可写的，任何文件都可以修改。**

**global.\_foo = { foo: 'world' };**

**const a = require('./mod.js');**

**console.log(a.foo);**

**// 上面的代码，会使得加载mod.js的脚本都失真。**

**// 为了防止这种情况出现，我们就可以使用 Symbol。**

**// mod.js**

**const FOO\_KEY = Symbol.for('foo');**

**function A() {**

**this.foo = 'hello';**

**}**

**if (!global[FOO\_KEY]) {**

**global[FOO\_KEY] = new A();**

**}**

**module.exports = global[FOO\_KEY];**

**// 上面代码中，可以保证global[FOO\_KEY]不会被无意间覆盖，但还是可以被改写。**

**global[Symbol.for('foo')] = { foo: 'world' };**

**const a = require('./mod.js');**

**// 如果键名使用Symbol方法生成，那么外部将无法引用这个值，当然也就无法改写。**

**// mod.js**

**const FOO\_KEY = Symbol('foo');**

**// 后面代码相同 ……**

**// 上面代码将导致其他脚本都无法引用FOO\_KEY。但这样也有一个问题，就是如果多次执行这个脚本，每次得到的FOO\_KEY都是不一样的。虽然 Node 会将脚本的执行结果缓存，一般情况下，不会多次执行同一个脚本，但是用户可以手动清除缓存，所以也不是绝对可靠。**

**}**

## // 7.内置的 Symbol 值

**{**

**}**

# Set和map

## // 1.Set

**{**

**// ES6 提供了新的数据结构 Set。它类似于数组，但是成员的值都是唯一的，没有重复的值。**

**// Set 本身是一个构造函数，用来生成 Set 数据结构。**

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

**[2, 3, 5, 4, 5, 2, 2].forEach(x => s.add(x));**

**for (let i of s) {**

**console.log(i);**

**}**

**// 2 3 5 4**

**// Set 函数可以接受一个数组（或者具有 iterable 接口的其他数据结构）作为参数，用来初始化。**

**// 例一**

**const set = new Set([1, 2, 3, 4, 4]);**

**[...set]**

**// [1, 2, 3, 4]**

**// 例二**

**const items = new Set([1, 2, 3, 4, 5, 5, 5, 5]);**

**items.size // 5**

**// 例三**

**const set = new Set(document.querySelectorAll('div'));**

**set.size // 56**

**// 类似于**

**const set = new Set();**

**document**

**.querySelectorAll('div')**

**.forEach(div => set.add(div));**

**set.size // 56**

**// 上面代码也展示了一种去除数组重复成员的方法。**

**// 去除数组的重复成员**

**[...new Set(array)]**

**// 向 Set 加入值的时候，不会发生类型转换，所以5和"5"是两个不同的值。Set 内部判断两个值是否不同，使用的算法叫做“Same-value-zero equality”，它类似于精确相等运算符（===），主要的区别是NaN等于自身，而精确相等运算符认为NaN不等于自身。**

**// 另外，两个对象总是不相等的。**

**let set = new Set();**

**set.add({});**

**set.size // 1**

**set.add({});**

**set.size // 2**

**// Set 实例的属性和方法**

**/\*Set 结构的实例有以下属性。**

**Set.prototype.constructor：构造函数，默认就是Set函数。**

**Set.prototype.size：返回Set实例的成员总数。**

**Set 实例的方法分为两大类：操作方法（用于操作数据）和遍历方法（用于遍历成员）。下面先介绍四个操作方法。**

**add(value)：添加某个值，返回 Set 结构本身。**

**delete(value)：删除某个值，返回一个布尔值，表示删除是否成功。**

**has(value)：返回一个布尔值，表示该值是否为Set的成员。**

**clear()：清除所有成员，没有返回值。\*/**

**// 上面这些属性和方法的实例如下。**

**s.add(1).add(2).add(2);**

**// 注意2被加入了两次**

**s.size // 2**

**s.has(1) // true**

**s.has(2) // true**

**s.has(3) // false**

**s.delete(2);**

**s.has(2) // false**

**// 下面是一个对比，看看在判断是否包括一个键上面，Object结构和Set结构的写法不同。**

**// 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');**

**properties.add('height');**

**if (properties.has(someName)) {**

**// do something**

**}**

**// Array.from方法可以将 Set 结构转为数组。**

**const items = new Set([1, 2, 3, 4, 5]);**

**const array = Array.from(items);**

**// 这就提供了去除数组重复成员的另一种方法。**

**function dedupe(array) {**

**return Array.from(new Set(array));**

**}**

**dedupe([1, 1, 2, 3]) // [1, 2, 3]**

**/\*Set 结构的实例有四个遍历方法，可以用于遍历成员。**

**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**

**需要特别指出的是，Set的遍历顺序就是插入顺序。这个特性有时非常有用，比如使用 Set 保存一个回调函数列表，调用时就能保证按照添加顺序调用。\*/**

**// （1）keys()，values()，entries()**

**// keys方法、values方法、entries方法返回的都是遍历器对象（详见《Iterator 对象》一章）。由于 Set 结构没有键名，只有键值（或者说键名和键值是同一个值），所以keys方法和values方法的行为完全一致。**

**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"]**

**// 上面代码中，entries方法返回的遍历器，同时包括键名和键值，所以每次输出一个数组，它的两个成员完全相等。**

**// Set 结构的实例默认可遍历，它的默认遍历器生成函数就是它的values方法。**

**Set.prototype[Symbol.iterator] === Set.prototype.values**

**// true**

**// 这意味着，可以省略values方法，直接用for...of循环遍历 Set。**

**let set = new Set(['red', 'green', 'blue']);**

**for (let x of set) {**

**console.log(x);**

**}**

**// red**

**// green**

**// blue**

**// （2）forEach()**

**// Set 结构的实例与数组一样，也拥有forEach方法，用于对每个成员执行某种操作，没有返回值。**

**set = new Set([1, 4, 9]);**

**set.forEach((value, key) => console.log(key + ' : ' + value))**

**// 1 : 1**

**// 4 : 4**

**// 9 : 9**

**// 上面代码说明，forEach方法的参数就是一个处理函数。该函数的参数与数组的forEach一致，依次为键值、键名、集合本身（上例省略了该参数）。这里需要注意，Set 结构的键名就是键值（两者是同一个值），因此第一个参数与第二个参数的值永远都是一样的。**

**// 另外，forEach方法还可以有第二个参数，表示绑定处理函数内部的this对象。**

**// （3）遍历的应用**

**// 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']**

**// 扩展运算符和 Set 结构相结合，就可以去除数组的重复成员。**

**let arr = [3, 5, 2, 2, 5, 5];**

**let unique = [...new Set(arr)];**

**// [3, 5, 2]**

**// 而且，数组的map和filter方法也可以间接用于 Set 了。**

**let set = new Set([1, 2, 3]);**

**set = new Set([...set].map(x => x \* 2));**

**// 返回Set结构：{2, 4, 6}**

**let set = new Set([1, 2, 3, 4, 5]);**

**set = new Set([...set].filter(x => (x % 2) == 0));**

**// 返回Set结构：{2, 4}**

**// 因此使用 Set 可以很容易地实现并集（Union）、交集（Intersect）和差集（Difference）。**

**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}**

**// 差集**

**let difference = new Set([...a].filter(x => !b.has(x)));**

**// Set {1}**

**// 如果想在遍历操作中，同步改变原来的 Set 结构，目前没有直接的方法，但有两种变通方法。一种是利用原 Set 结构映射出一个新的结构，然后赋值给原来的 Set 结构；另一种是利用Array.from方法。**

**// 方法一**

**let set = new Set([1, 2, 3]);**

**set = new Set([...set].map(val => val \* 2));**

**// set的值是2, 4, 6**

**// 方法二**

**let set = new Set([1, 2, 3]);**

**set = new Set(Array.from(set, val => val \* 2));**

**// set的值是2, 4, 6**

**// 上面代码提供了两种方法，直接在遍历操作中改变原来的 Set 结构。**

**}**

## // 2.WeakSet

**{**

**// WeakSet 结构与 Set 类似，也是不重复的值的集合。但是，它与 Set 有两个区别。**

**// 首先，WeakSet 的成员只能是对象，而不能是其他类型的值。**

**// 其次，WeakSet 中的对象都是弱引用，即垃圾回收机制不考虑 WeakSet 对该对象的引用，也就是说，如果其他对象都不再引用该对象，那么垃圾回收机制会自动回收该对象所占用的内存，不考虑该对象还存在于 WeakSet 之中。**

**}**

## // 3.Map

**{**

**// 含义和基本用法**

**// JavaScript 的对象（Object），本质上是键值对的集合（Hash 结构），但是传统上只能用字符串当作键。这给它的使用带来了很大的限制。**

**const data = {};**

**const element = document.getElementById('myDiv');**

**data[element] = 'metadata';**

**data['[object HTMLDivElement]'] // "metadata"**

**// 上面代码原意是将一个 DOM 节点作为对象data的键，但是由于对象只接受字符串作为键名，所以element被自动转为字符串[object HTMLDivElement]。**

**// 为了解决这个问题，ES6 提供了 Map 数据结构。它类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。也就是说，Object 结构提供了“字符串—值”的对应，Map 结构提供了“值—值”的对应，是一种更完善的 Hash 结构实现。如果你需要“键值对”的数据结构，Map 比 Object 更合适。**

**// 作为构造函数，Map 也可以接受一个数组作为参数。该数组的成员是一个个表示键值对的数组。**

**const map = new Map([**

**['name', '张三'],**

**['title', 'Author']**

**]);**

**map.size // 2**

**map.has('name') // true**

**map.get('name') // "张三"**

**map.has('title') // true**

**map.get('title') // "Author"**

**// 上面代码在新建 Map 实例时，就指定了两个键name和title。**

**// Map构造函数接受数组作为参数，实际上执行的是下面的算法。**

**const items = [**

**['name', '张三'],**

**['title', 'Author']**

**];**

**const map = new Map();**

**items.forEach(**

**([key, value]) => map.set(key, value)**

**);**

**// 事实上，不仅仅是数组，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（详见《Iterator》一章）都可以当作Map构造函数的参数。这就是说，Set和Map都可以用来生成新的 Map。**

**const set = new Set([**

**['foo', 1],**

**['bar', 2]**

**]);**

**const m1 = new Map(set);**

**m1.get('foo') // 1**

**const m2 = new Map([['baz', 3]]);**

**const m3 = new Map(m2);**

**m3.get('baz') // 3**

**// 如果对同一个键多次赋值，后面的值将覆盖前面的值。如果读取一个未知的键，则返回undefined。**

**// 注意，只有对同一个对象的引用，Map 结构才将其视为同一个键。这一点要非常小心。**

**const map = new Map();**

**map.set(['a'], 555);**

**map.get(['a']) // undefined**

**// 上面代码的set和get方法，表面是针对同一个键，但实际上这是两个值，内存地址是不一样的，因此get方法无法读取该键，返回undefined。**

**// 同理，同样的值的两个实例，在 Map 结构中被视为两个键。**

**const map = new Map();**

**const k1 = ['a'];**

**const k2 = ['a'];**

**map**

**.set(k1, 111)**

**.set(k2, 222);**

**map.get(k1) // 111**

**map.get(k2) // 222**

**// 由上可知，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。这就解决了同名属性碰撞（clash）的问题，我们扩展别人的库的时候，如果使用对象作为键名，就不用担心自己的属性与原作者的属性同名。**

**// 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，比如0和-0就是一个键，布尔值true和字符串true则是两个不同的键。另外，undefined和null也是两个不同的键。虽然NaN不严格相等于自身，但 Map 将其视为同一个键。**

**let map = new Map();**

**map.set(-0, 123);**

**map.get(+0) // 123**

**map.set(true, 1);**

**map.set('true', 2);**

**map.get(true) // 1**

**map.set(undefined, 3);**

**map.set(null, 4);**

**map.get(undefined) // 3**

**map.set(NaN, 123);**

**map.get(NaN) // 123**

**// 实例的属性和操作方法**

**// Map 结构的实例有以下属性和操作方法。**

**// （1）size 属性**

**// 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**

**// （2）set(key, value)**

**// set方法设置键名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。**

**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**

**// set方法返回的是当前的Map对象，因此可以采用链式写法。**

**let map = new Map()**

**.set(1, 'a')**

**.set(2, 'b')**

**.set(3, 'c');**

**// （3）get(key)**

**// 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!**

**// （4）has(key)**

**// 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**

**// （5）delete(key)**

**// 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**

**// （6）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**

**/\*遍历方法**

**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。**

**需要特别注意的是，Map 的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"**

**// 上面代码最后的那个例子，表示 Map 结构的默认遍历器接口（Symbol.iterator属性），就是entries方法。**

**map[Symbol.iterator] === map.entries**

**// true**

**// Map 结构转为数组结构，比较快速的方法是使用扩展运算符（...）。**

**const map = new Map([**

**[1, 'one'],**

**[2, 'two'],**

**[3, 'three'],**

**]);**

**[...map.keys()]**

**// [1, 2, 3]**

**[...map.values()]**

**// ['one', 'two', 'three']**

**[...map.entries()]**

**// [[1,'one'], [2, 'two'], [3, 'three']]**

**[...map]**

**// [[1,'one'], [2, 'two'], [3, 'three']]**

**// 结合数组的map方法、filter方法，可以实现 Map 的遍历和过滤（Map 本身没有map和filter方法）。**

**const map0 = new Map()**

**.set(1, 'a')**

**.set(2, 'b')**

**.set(3, 'c');**

**const map1 = new Map(**

**[...map0].filter(([k, v]) => k < 3)**

**);**

**// 产生 Map 结构 {1 => 'a', 2 => 'b'}**

**const map2 = new Map(**

**[...map0].map(([k, v]) => [k \* 2, '\_' + v])**

**);**

**// 产生 Map 结构 {2 => '\_a', 4 => '\_b', 6 => '\_c'}**

**// 此外，Map 还有一个forEach方法，与数组的forEach方法类似，也可以实现遍历。**

**map.forEach(function(value, key, map) {**

**console.log("Key: %s, Value: %s", key, value);**

**});**

**// forEach方法还可以接受第二个参数，用来绑定this。**

**const reporter = {**

**report: function(key, value) {**

**console.log("Key: %s, Value: %s", key, value);**

**}**

**};**

**map.forEach(function(value, key, map) {**

**this.report(key, value);**

**}, reporter);**

**// 上面代码中，forEach方法的回调函数的this，就指向reporter。**

**// 与其他数据结构的互相转换**

**// （1）Map 转为数组**

**// 前面已经提过，Map 转为数组最方便的方法，就是使用扩展运算符（...）。**

**const myMap = new Map()**

**.set(true, 7)**

**.set({foo: 3}, ['abc']);**

**[...myMap]**

**// [ [ true, 7 ], [ { foo: 3 }, [ 'abc' ] ] ]**

**// （2）数组 转为 Map**

**// 将数组传入 Map 构造函数，就可以转为 Map。**

**new Map([**

**[true, 7],**

**[{foo: 3}, ['abc']]**

**])**

**// Map {**

**// true => 7,**

**// Object {foo: 3} => ['abc']**

**// }**

**// （3）Map 转为对象**

**// 如果所有 Map 的键都是字符串，它可以无损地转为对象。**

**function strMapToObj(strMap) {**

**let obj = Object.create(null);**

**for (let [k,v] of strMap) {**

**obj[k] = v;**

**}**

**return obj;**

**}**

**const myMap = new Map()**

**.set('yes', true)**

**.set('no', false);**

**strMapToObj(myMap)**

**// { yes: true, no: false }**

**// 如果有非字符串的键名，那么这个键名会被转成字符串，再作为对象的键名。**

**// （4）对象转为 Map**

**function objToStrMap(obj) {**

**let strMap = new Map();**

**for (let k of Object.keys(obj)) {**

**strMap.set(k, obj[k]);**

**}**

**return strMap;**

**}**

**objToStrMap({yes: true, no: false})**

**// Map {"yes" => true, "no" => false}**

**// （5）Map 转为 JSON**

**//Map 转为 JSON 要区分两种情况。一种情况是，Map 的键名都是字符串，这时可以选择转为对象 JSON。**

**function strMapToJson(strMap) {**

**return JSON.stringify(strMapToObj(strMap));**

**}**

**let myMap = new Map().set('yes', true).set('no', false);**

**strMapToJson(myMap)**

**// '{"yes":true,"no":false}'**

**// 另一种情况是，Map 的键名有非字符串，这时可以选择转为数组 JSON。**

**function mapToArrayJson(map) {**

**return JSON.stringify([...map]);**

**}**

**let myMap = new Map().set(true, 7).set({foo: 3}, ['abc']);**

**mapToArrayJson(myMap)**

**// '[[true,7],[{"foo":3},["abc"]]]'**

**// （6）JSON 转为 Map**

**// JSON 转为 Map，正常情况下，所有键名都是字符串。**

**function jsonToStrMap(jsonStr) {**

**return objToStrMap(JSON.parse(jsonStr));**

**}**

**jsonToStrMap('{"yes": true, "no": false}')**

**// Map {'yes' => true, 'no' => false}**

**// 但是，有一种特殊情况，整个 JSON 就是一个数组，且每个数组成员本身，又是一个有两个成员的数组。这时，它可以一一对应地转为 Map。这往往是 Map 转为数组 JSON 的逆操作。**

**function jsonToMap(jsonStr) {**

**return new Map(JSON.parse(jsonStr));**

**}**

**jsonToMap('[[true,7],[{"foo":3},["abc"]]]')**

**// Map {true => 7, Object {foo: 3} => ['abc']}**

**}**

## // 4.WeakMap

**{**

**// WeakMap与Map的区别有两点。**

**// 首先，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。其次，WeakMap的键名所指向的对象，不计入垃圾回收机制。**

**}**

# Promise

## // 1.Promise 的含义

**{**

**/\*Promise 是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理和更强大。它由社区最早提出和实现，ES6 将其写进了语言标准，统一了用法，原生提供了Promise对象。**

**所谓Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。Promise 提供统一的 API，各种异步操作都可以用同样的方法进行处理。**

**Promise对象有以下两个特点。**

**（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，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。这也是Promise这个名字的由来，它的英语意思就是“承诺”，表示其他手段无法改变。**

**（2）一旦状态改变，就不会再变，任何时候都可以得到这个结果。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）。如果改变已经发生了，你再对Promise对象添加回调函数，也会立即得到这个结果。这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。**

**注意，为了行文方便，本章后面的resolved统一只指fulfilled状态，不包含rejected状态。**

**有了Promise对象，就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。此外，Promise对象提供统一的接口，使得控制异步操作更加容易。**

**Promise也有一些缺点。首先，无法取消Promise，一旦新建它就会立即执行，无法中途取消。其次，如果不设置回调函数，Promise内部抛出的错误，不会反应到外部。第三，当处于pending状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。**

**如果某些事件不断地反复发生，一般来说，使用 Stream 模式是比部署Promise更好的选择。\*/**

**}**

## // 2.基本用法

**{**

**// ES6 规定，Promise对象是一个构造函数，用来生成Promise实例。**

**// 下面代码创造了一个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”（即从 pending 变为 resolved），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”（即从 pending 变为 rejected），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。这两个函数都接受Promise对象传出的值作为参数。**

**// Promise 新建后就会立即执行。**

**let promise = new Promise(function(resolve, reject) {**

**console.log('Promise');**

**resolve();**

**});**

**promise.then(function() {**

**console.log('resolved.');**

**});**

**console.log('Hi!');**

**// Promise**

**// Hi!**

**// resolved**

**// 上面代码中，Promise 新建后立即执行，所以首先输出的是Promise。然后，then方法指定的回调函数，将在当前脚本所有同步任务执行完才会执行，所以resolved最后输出。**

**// 下面是异步加载图片的例子。**

**function loadImageAsync(url) {**

**return new Promise(function(resolve, reject) {**

**const image = new Image();**

**image.onload = function() {**

**resolve(image);**

**};**

**image.onerror = function() {**

**reject(new Error('Could not load image at ' + url));**

**};**

**image.src = url;**

**});**

**}**

**// 下面是一个用Promise对象实现的 Ajax 操作的例子。**

**const getJSON = function(url) {**

**const promise = new Promise(function(resolve, reject){**

**const handler = function() {**

**if (this.readyState !== 4) {**

**return;**

**}**

**if (this.status === 200) {**

**resolve(this.response);**

**} else {**

**reject(new Error(this.statusText));**

**}**

**};**

**const client = new XMLHttpRequest();**

**client.open("GET", url);**

**client.onreadystatechange = handler;**

**client.responseType = "json";**

**client.setRequestHeader("Accept", "application/json");**

**client.send();**

**});**

**return promise;**

**};**

**getJSON("/posts.json").then(function(json) {**

**console.log('Contents: ' + json);**

**}, function(error) {**

**console.error('出错了', error);**

**});**

**// 如果调用resolve函数和reject函数时带有参数，那么它们的参数会被传递给回调函数。reject函数的参数通常是Error对象的实例，表示抛出的错误；resolve函数的参数除了正常的值以外，还可能是另一个 Promise 实例，比如像下面这样。**

**const p1 = new Promise(function (resolve, reject) {**

**// ...**

**});**

**const p2 = new Promise(function (resolve, reject) {**

**// ...**

**resolve(p1);**

**})**

**// 上面代码中，p1和p2都是 Promise 的实例，但是p2的resolve方法将p1作为参数，即一个异步操作的结果是返回另一个异步操作。**

**// 注意，这时p1的状态就会传递给p2，也就是说，p1的状态决定了p2的状态。如果p1的状态是pending，那么p2的回调函数就会等待p1的状态改变；如果p1的状态已经是resolved或者rejected，那么p2的回调函数将会立刻执行。**

**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。**

**// 注意，调用resolve或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**

**// 上面代码中，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。**

**// 一般来说，调用resolve或reject以后，Promise 的使命就完成了，后继操作应该放到then方法里面，而不应该直接写在resolve或reject的后面。所以，最好在它们前面加上return语句，这样就不会有意外。**

**new Promise((resolve, reject) => {**

**return resolve(1);**

**// 后面的语句不会执行**

**console.log(2);**

**})**

**}**

## // 3.Promise.prototype.then()

**{**

**// Promise 实例具有then方法，也就是说，then方法是定义在原型对象Promise.prototype上的。它的作用是为 Promise 实例添加状态改变时的回调函数。前面说过，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。**

**getJSON("/posts.json").then(function(json) {**

**return json.post;**

**}).then(function(post) {**

**// ...**

**});**

**// 上面的代码使用then方法，依次指定了两个回调函数。第一个回调函数完成以后，会将返回结果作为参数，传入第二个回调函数。**

**// 采用链式的then，可以指定一组按照次序调用的回调函数。这时，前一个回调函数，有可能返回的还是一个Promise对象（即有异步操作），这时后一个回调函数，就会等待该Promise对象的状态发生变化，才会被调用。**

**getJSON("/post/1.json").then(function(post) {**

**return getJSON(post.commentURL);**

**}).then(function funcA(comments) {**

**console.log("resolved: ", comments);**

**}, function funcB(err){**

**console.log("rejected: ", err);**

**});**

**// 上面代码中，第一个then方法指定的回调函数，返回的是另一个Promise对象。这时，第二个then方法指定的回调函数，就会等待这个新的Promise对象状态发生变化。如果变为resolved，就调用funcA，如果状态变为rejected，就调用funcB。**

**// 如果采用箭头函数，上面的代码可以写得更简洁。**

**getJSON("/post/1.json").then(**

**post => getJSON(post.commentURL)**

**).then(**

**comments => console.log("resolved: ", comments),**

**err => console.log("rejected: ", err)**

**);**

**}**

## // 4.Promise.prototype.catch()

**{**

**// Promise.prototype.catch方法是.then(null, 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);**

**});**

**// 上面代码中，getJSON方法返回一个 Promise 对象，如果该对象状态变为resolved，则会调用then方法指定的回调函数；如果异步操作抛出错误，状态就会变为rejected，就会调用catch方法指定的回调函数，处理这个错误。另外，then方法指定的回调函数，如果运行中抛出错误，也会被catch方法捕获。**

**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));**

**// 下面是一个例子。**

**const promise = new Promise(function(resolve, reject) {**

**throw new Error('test');**

**});**

**promise.catch(function(error) {**

**console.log(error);**

**});**

**// Error: test**

**// 上面代码中，promise抛出一个错误，就被catch方法指定的回调函数捕获。注意，上面的写法与下面两种写法是等价的。**

**// 写法一**

**const promise = new Promise(function(resolve, reject) {**

**try {**

**throw new Error('test');**

**} catch(e) {**

**reject(e);**

**}**

**});**

**promise.catch(function(error) {**

**console.log(error);**

**});**

**// 写法二**

**const promise = new Promise(function(resolve, reject) {**

**reject(new Error('test'));**

**});**

**promise.catch(function(error) {**

**console.log(error);**

**});**

**// 比较上面两种写法，可以发现reject方法的作用，等同于抛出错误。**

**// 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。**

**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**

**});**

**// 上面代码中，一共有三个 Promise 对象：一个由getJSON产生，两个由then产生。它们之中任何一个抛出的错误，都会被最后一个catch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，也更接近同步的写法（try/catch）。因此，建议总是使用catch方法，而不使用then方法的第二个参数。**

**// 跟传统的try/catch代码块不同的是，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**

**// 上面代码中，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”。**

**// 再看下面的例子。**

**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**

**// 上面代码中，Promise 指定在下一轮“事件循环”再抛出错误。到了那个时候，Promise 的运行已经结束了，所以这个错误是在 Promise 函数体外抛出的，会冒泡到最外层，成了未捕获的错误。**

**// 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。**

**// catch方法之中，还能再抛出错误。**

**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() {**

**return someOtherAsyncThing();**

**}).catch(function(error) {**

**console.log('oh no', error);**

**// 下面一行会报错，因为 y 没有声明**

**y + 2;**

**}).then(function() {**

**console.log('carry on');**

**});**

**// oh no [ReferenceError: x is not defined]**

**// 上面代码中，catch方法抛出一个错误，因为后面没有别的catch方法了，导致这个错误不会被捕获，也不会传递到外层。如果改写一下，结果就不一样了。**

**someAsyncThing().then(function() {**

**return someOtherAsyncThing();**

**}).catch(function(error) {**

**console.log('oh no', error);**

**// 下面一行会报错，因为y没有声明**

**y + 2;**

**}).catch(function(error) {**

**console.log('carry on', error);**

**});**

**// oh no [ReferenceError: x is not defined]**

**// carry on [ReferenceError: y is not defined]**

**// 上面代码中，第二个catch方法用来捕获前一个catch方法抛出的错误。**

**}**

## // 5.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。该方法是 ES2018 引入标准的。**

**promise**

**.then(result => {···})**

**.catch(error => {···})**

**.finally(() => {···});**

**// 上面代码中，不管promise最后的状态，在执行完then或catch指定的回调函数以后，都会执行finally方法指定的回调函数。**

**// 下面是一个例子，服务器使用 Promise 处理请求，然后使用finally方法关掉服务器。**

**server.listen(port)**

**.then(function () {**

**// ...**

**})**

**.finally(server.stop);**

**// 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。**

**// finally本质上是then方法的特例。**

**promise**

**.finally(() => {**

**// 语句**

**});**

**// 等同于**

**promise**

**.then(**

**result => {**

**// 语句**

**return result;**

**},**

**error => {**

**// 语句**

**throw error;**

**}**

**);**

**// 上面代码中，如果不使用finally方法，同样的语句需要为成功和失败两种情况各写一次。有了finally方法，则只需要写一次。**

**// 它的实现也很简单。**

**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 })**

**);**

**};**

**// 上面代码中，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。**

**// resolve 的值是 undefined**

**Promise.resolve(2).then(() => {}, () => {})**

**// resolve 的值是 2**

**Promise.resolve(2).finally(() => {})**

**// reject 的值是 undefined**

**Promise.reject(3).then(() => {}, () => {})**

**// reject 的值是 3**

**Promise.reject(3).finally(() => {})**

**}**

## // 6.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]);**

**// 上面代码中，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。**

**// 下面是一个具体的例子。**

**// 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){**

**// ...**

**});**

**// 上面代码中，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。**

**// 下面是另一个例子。**

**const databasePromise = connectDatabase();**

**const booksPromise = databasePromise**

**.then(findAllBooks);**

**const userPromise = databasePromise**

**.then(getCurrentUser);**

**Promise.all([**

**booksPromise,**

**userPromise**

**])**

**.then(([books, user]) => pickTopRecommentations(books, user));**

**// 上面代码中，booksPromise和userPromise是两个异步操作，只有等到它们的结果都返回了，才会触发pickTopRecommentations这个回调函数。**

**// 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]**

**// 上面代码中，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。**

**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**

**}**

## // 7.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。**

**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。**

**// Promise.race方法的参数与Promise.all方法一样，如果不是 Promise 实例，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。**

**// 下面是一个例子，如果指定时间内没有获得结果，就将 Promise 的状态变为reject，否则变为resolve。**

**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);**

**// 上面代码中，如果 5 秒之内fetch方法无法返回结果，变量p的状态就会变为rejected，从而触发catch方法指定的回调函数。**

**}**

## // 8.Promise.resolve()

**{**

**// 有时需要将现有对象转为 Promise 对象，Promise.resolve方法就起到这个作用。**

**const jsPromise = Promise.resolve($.ajax('/whatever.json'));**

**// 上面代码将 jQuery 生成的deferred对象，转为一个新的 Promise 对象。**

**// Promise.resolve等价于下面的写法。**

**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**

**// 上面代码生成一个新的 Promise 对象的实例p。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。**

**// 所以，如果希望得到一个 Promise 对象，比较方便的方法就是直接调用Promise.resolve方法。**

**const p = Promise.resolve();**

**p.then(function () {**

**// ...**

**});**

**// 上面代码的变量p就是一个 Promise 对象。**

**// 需要注意的是，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');**

**}, 0);**

**Promise.resolve().then(function () {**

**console.log('two');**

**});**

**console.log('one');**

**// one**

**// two**

**// three**

**// 上面代码中，setTimeout(fn, 0)在下一轮“事件循环”开始时执行，Promise.resolve()在本轮“事件循环”结束时执行，console.log('one')则是立即执行，因此最先输出。**

**}**

## // 9.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**

**// 上面代码生成一个 Promise 对象的实例p，状态为rejected，回调函数会立即执行。**

**// 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。**

**const thenable = {**

**then(resolve, reject) {**

**reject('出错了');**

**}**

**};**

**Promise.reject(thenable)**

**.catch(e => {**

**console.log(e === thenable)**

**})**

**// true**

**// 上面代码中，Promise.reject方法的参数是一个thenable对象，执行以后，后面catch方法的参数不是reject抛出的“出错了”这个字符串，而是thenable对象。**

**}**

## // 10.应用

**{**

**// 加载图片**

**// 我们可以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;**

**});**

**};**

**// Generator 函数与 Promise 的结合**

**// 使用 Generator 函数管理流程，遇到异步操作的时候，通常返回一个Promise对象。**

**function getFoo () {**

**return new Promise(function (resolve, reject){**

**resolve('foo');**

**});**

**}**

**const g = function\* () {**

**try {**

**const foo = yield getFoo();**

**console.log(foo);**

**} catch (e) {**

**console.log(e);**

**}**

**};**

**function run (generator) {**

**const it = generator();**

**function go(result) {**

**if (result.done) return result.value;**

**return result.value.then(function (value) {**

**return go(it.next(value));**

**}, function (error) {**

**return go(it.throw(error));**

**});**

**}**

**go(it.next());**

**}**

**run(g);**

**// 上面代码的 Generator 函数g之中，有一个异步操作getFoo，它返回的就是一个Promise对象。函数run用来处理这个Promise对象，并调用下一个next方法。**

**}**

## // 11.Promise.try()

**{**

**// 实际开发中，经常遇到一种情况：不知道或者不想区分，函数f是同步函数还是异步操作，但是想用 Promise 来处理它。因为这样就可以不管f是否包含异步操作，都用then方法指定下一步流程，用catch方法处理f抛出的错误。一般就会采用下面的写法。**

**Promise.resolve().then(f)**

**// 上面的写法有一个缺点，就是如果f是同步函数，那么它会在本轮事件循环的末尾执行。**

**const f = () => console.log('now');**

**Promise.resolve().then(f);**

**console.log('next');**

**// next**

**// now**

**// 上面代码中，函数f是同步的，但是用 Promise 包装了以后，就变成异步执行了。**

**// 那么有没有一种方法，让同步函数同步执行，异步函数异步执行，并且让它们具有统一的 API 呢？回答是可以的，并且还有两种写法。第一种写法是用async函数来写。**

**const f = () => console.log('now');**

**(async () => f())();**

**console.log('next');**

**// now**

**// next**

**// 上面代码中，第二行是一个立即执行的匿名函数，会立即执行里面的async函数，因此如果f是同步的，就会得到同步的结果；如果f是异步的，就可以用then指定下一步，就像下面的写法。**

**(async () => f())()**

**.then(...)**

**// 需要注意的是，async () => f()会吃掉f()抛出的错误。所以，如果想捕获错误，要使用promise.catch方法。**

**(async () => f())()**

**.then(...)**

**.catch(...)**

**// 第二种写法是使用new Promise()。**

**const f = () => console.log('now');**

**(**

**() => new Promise(**

**resolve => resolve(f())**

**)**

**)();**

**console.log('next');**

**// now**

**// next**

**// 上面代码也是使用立即执行的匿名函数，执行new Promise()。这种情况下，同步函数也是同步执行的。**

**// 鉴于这是一个很常见的需求，所以现在有一个提案，提供Promise.try方法替代上面的写法。**

**const f = () => console.log('now');**

**Promise.try(f);**

**console.log('next');**

**// now**

**// next**

**// 事实上，Promise.try存在已久，Promise 库Bluebird、Q和when，早就提供了这个方法。**

**// 由于Promise.try为所有操作提供了统一的处理机制，所以如果想用then方法管理流程，最好都用Promise.try包装一下。这样有许多好处，其中一点就是可以更好地管理异常。**

**function getUsername(userId) {**

**return database.users.get({id: userId})**

**.then(function(user) {**

**return user.name;**

**});**

**}**

**// 上面代码中，database.users.get()返回一个 Promise 对象，如果抛出异步错误，可以用catch方法捕获，就像下面这样写。**

**database.users.get({id: userId})**

**.then(...)**

**.catch(...)**

**// 但是database.users.get()可能还会抛出同步错误（比如数据库连接错误，具体要看实现方法），这时你就不得不用try...catch去捕获。**

**try {**

**database.users.get({id: userId})**

**.then(...)**

**.catch(...)**

**} catch (e) {**

**// ...**

**}**

**// 上面这样的写法就很笨拙了，这时就可以统一用promise.catch()捕获所有同步和异步的错误。**

**Promise.try(database.users.get({id: userId}))**

**.then(...)**

**.catch(...)**

**// 事实上，Promise.try就是模拟try代码块，就像promise.catch模拟的是catch代码块。**

**}**

# Iterator和for…of循环

## // 1.Iterator（遍历器）的概念

**{**

**/\*JavaScript 原有的表示“集合”的数据结构，主要是数组（Array）和对象（Object），ES6 又添加了Map和Set。这样就有了四种数据集合，用户还可以组合使用它们，定义自己的数据结构，比如数组的成员是Map，Map的成员是对象。这样就需要一种统一的接口机制，来处理所有不同的数据结构。**

**遍历器（Iterator）就是这样一种机制。它是一种接口，为各种不同的数据结构提供统一的访问机制。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。**

**每一次调用next方法，都会返回数据结构的当前成员的信息。具体来说，就是返回一个包含value和done两个属性的对象。其中，value属性是当前成员的值，done属性是一个布尔值，表示遍历是否结束。**

**下面是一个模拟next方法返回值的例子。\*/**

**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};**

**}**

**};**

**}**

**}**

## // 2.默认 Iterator 接口

**{**

**// Iterator 接口的目的，就是为所有数据结构，提供了一种统一的访问机制，即for...of循环（详见下文）。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。至于属性名Symbol.iterator，它是一个表达式，返回Symbol对象的iterator属性，这是一个预定义好的、类型为 Symbol 的特殊值，所以要放在方括号内（参见《Symbol》一章）。**

**// 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**

**// 对于原生部署 Iterator 接口的数据结构，不用自己写遍历器生成函数，for...of循环会自动遍历它们。除此之外，其他数据结构（主要是对象）的 Iterator 接口，都需要自己在Symbol.iterator属性上面部署，这样才会被for...of循环遍历。**

**// 对象（Object）之所以没有默认部署 Iterator 接口，是因为对象的哪个属性先遍历，哪个属性后遍历是不确定的，需要开发者手动指定。本质上，遍历器是一种线性处理，对于任何非线性的数据结构，部署遍历器接口，就等于部署一种线性转换。不过，严格地说，对象部署遍历器接口并不是很必要，因为这时对象实际上被当作 Map 结构使用，ES5 没有 Map 结构，而 ES6 原生提供了。**

**// 一个对象如果要具备可被for...of循环调用的 Iterator 接口，就必须在Symbol.iterator的属性上部署遍历器生成方法（原型链上的对象具有该方法也可）。**

**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**

**}**

**// 上面代码是一个类部署 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。**

**// 下面是通过遍历器实现指针结构的例子。**

**function Obj(value) {**

**this.value = value;**

**this.next = null;**

**}**

**Obj.prototype[Symbol.iterator] = function() {**

**var iterator = { next: next };**

**var current = this;**

**function next() {**

**if (current) {**

**var value = current.value;**

**current = current.next;**

**return { done: false, value: value };**

**} else {**

**return { done: true };**

**}**

**}**

**return iterator;**

**}**

**var one = new Obj(1);**

**var two = new Obj(2);**

**var three = new Obj(3);**

**one.next = two;**

**two.next = three;**

**for (var i of one){**

**console.log(i); // 1, 2, 3**

**}**

**// 上面代码首先在构造函数的原型链上部署Symbol.iterator方法，调用该方法会返回遍历器对象iterator，调用该对象的next方法，在返回一个值的同时，自动将内部指针移到下一个实例。**

**// 下面是另一个为对象添加 Iterator 接口的例子。**

**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 };**

**}**

**}**

**};**

**}**

**};**

**// 对于类似数组的对象（存在数值键名和length属性），部署 Iterator 接口，有一个简便方法，就是Symbol.iterator方法直接引用数组的 Iterator 接口。**

**NodeList.prototype[Symbol.iterator] = Array.prototype[Symbol.iterator];**

**// 或者**

**NodeList.prototype[Symbol.iterator] = [][Symbol.iterator];**

**[...document.querySelectorAll('div')] // 可以执行了**

**// NodeList 对象是类似数组的对象，本来就具有遍历接口，可以直接遍历。上面代码中，我们将它的遍历接口改成数组的Symbol.iterator属性，可以看到没有任何影响。**

**// 下面是另一个类似数组的对象调用数组的Symbol.iterator方法的例子。**

**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Symbol.iterator方法，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**

**}**

**}**

## // 3.调用 Iterator 接口的场合

**{**

**// 有一些场合会默认调用 Iterator 接口（即Symbol.iterator方法），除了下文会介绍的for...of循环，还有几个别的场合。**

**// （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']**

**// 上面代码的扩展运算符内部就调用 Iterator 接口。**

**// 实际上，这提供了一种简便机制，可以将任何部署了 Iterator 接口的数据结构，转为数组。也就是说，只要某个数据结构部署了 Iterator 接口，就可以对它使用扩展运算符，将其转为数组。**

**let arr = [...iterable];**

**// （3）yield\***

**// yield\*后面跟的是一个可遍历的结构，它会调用该结构的遍历器接口。**

**let generator = function\* () {**

**yield 1;**

**yield\* [2,3,4];**

**yield 5;**

**};**

**var iterator = generator();**

**iterator.next() // { value: 1, done: false }**

**iterator.next() // { value: 2, done: false }**

**iterator.next() // { value: 3, done: false }**

**iterator.next() // { value: 4, done: false }**

**iterator.next() // { value: 5, done: false }**

**iterator.next() // { value: undefined, done: true }**

**// （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()**

**}**

## // 4.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。**

**var someString = "hi";**

**typeof someString[Symbol.iterator]**

**// "function"**

**var iterator = someString[Symbol.iterator]();**

**iterator.next() // { value: "h", done: false }**

**iterator.next() // { value: "i", done: false }**

**iterator.next() // { value: undefined, done: true }**

**// 上面代码中，调用Symbol.iterator方法返回一个遍历器对象，在这个遍历器上可以调用 next 方法，实现对于字符串的遍历。**

**// 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"**

**// 上面代码中，字符串 str 的Symbol.iterator方法被修改了，所以扩展运算符（...）返回的值变成了bye，而字符串本身还是hi。**

**}**

## // 5.Iterator 接口与 Generator 函数

**{**

**// Symbol.iterator方法的最简单实现，还是使用下一章要介绍的 Generator 函数。**

**let myIterable = {**

**[Symbol.iterator]: function\* () {**

**yield 1;**

**yield 2;**

**yield 3;**

**}**

**}**

**[...myIterable] // [1, 2, 3]**

**// 或者采用下面的简洁写法**

**let obj = {**

**\* [Symbol.iterator]() {**

**yield 'hello';**

**yield 'world';**

**}**

**};**

**for (let x of obj) {**

**console.log(x);**

**}**

**// "hello"**

**// "world"**

**// 上面代码中，Symbol.iterator方法几乎不用部署任何代码，只要用 yield 命令给出每一步的返回值即可。**

**}**

## // 6.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语句或continue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 };**

**}**

**};**

**},**

**};**

**}**

**// 上面代码中，函数readLinesSync接受一个文件对象作为参数，返回一个遍历器对象，其中除了next方法，还部署了return方法。下面的三种情况，都会触发执行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);**

**continue;**

**}**

**// 情况三**

**for (let line of readLinesSync(fileName)) {**

**console.log(line);**

**throw new Error();**

**}**

**// 上面代码中，情况一输出文件的第一行以后，就会执行return方法，关闭这个文件；情况二输出所有行以后，执行return方法，关闭该文件；情况三会在执行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，这是 Generator 规格决定的。**

**// throw方法主要是配合 Generator 函数使用，一般的遍历器对象用不到这个方法。请参阅《Generator 函数》一章**

**}**

## // 7.for...of 循环

**{**

**// ES6 借鉴 C++、Java、C# 和 Python 语言，引入了for...of循环，作为遍历所有数据结构的统一的方法。**

**// 一个数据结构只要部署了Symbol.iterator属性，就被视为具有 iterator 接口，就可以用for...of循环遍历它的成员。也就是说，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。**

**// 数组**

**// 数组原生具备iterator接口（即默认部署了Symbol.iterator属性），for...of循环本质上就是调用这个接口产生的遍历器，可以用下面的代码证明。**

**const arr = ['red', 'green', 'blue'];**

**for(let v of arr) {**

**console.log(v); // red green blue**

**}**

**const obj = {};**

**obj[Symbol.iterator] = arr[Symbol.iterator].bind(arr);**

**for(let v of obj) {**

**console.log(v); // red green blue**

**}**

**// 上面代码中，空对象obj部署了数组arr的Symbol.iterator属性，结果obj的for...of循环，产生了与arr完全一样的结果。**

**// for...of循环可以代替数组实例的forEach方法。**

**const arr = ['red', 'green', 'blue'];**

**arr.forEach(function (element, index) {**

**console.log(element); // red green blue**

**console.log(index); // 0 1 2**

**});**

**// JavaScript 原有的for...in循环，只能获得对象的键名，不能直接获取键值。ES6 提供for...of循环，允许遍历获得键值。**

**var arr = ['a', 'b', 'c', 'd'];**

**for (let a in arr) {**

**console.log(a); // 0 1 2 3**

**}**

**for (let a of arr) {**

**console.log(a); // a b c d**

**}**

**// 上面代码表明，for...in循环读取键名，for...of循环读取键值。如果要通过for...of循环，获取数组的索引，可以借助数组实例的entries方法和keys方法（参见《数组的扩展》一章）。**

**// for...of循环调用遍历器接口，数组的遍历器接口只返回具有数字索引的属性。这一点跟for...in循环也不一样。**

**let arr = [3, 5, 7];**

**arr.foo = 'hello';**

**for (let i in arr) {**

**console.log(i); // "0", "1", "2", "foo"**

**}**

**for (let i of arr) {**

**console.log(i); // "3", "5", "7"**

**}**

**// 上面代码中，for...of循环不会返回数组arr的foo属性。**

**// Set 和 Map 结构**

**// Set 和 Map 结构也原生具有 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**

**// 上面代码演示了如何遍历 Set 结构和 Map 结构。值得注意的地方有两个，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**

**// 有些数据结构是在现有数据结构的基础上，计算生成的。比如，ES6 的数组、Set、Map 都部署了以下三个方法，调用后都返回遍历器对象。**

**// entries() 返回一个遍历器对象，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。**

**// 这三个方法调用后生成的遍历器对象，所遍历的都是计算生成的数据结构。**

**let arr = ['a', 'b', 'c'];**

**for (let pair of arr.entries()) {**

**console.log(pair);**

**}**

**// [0, 'a']**

**// [1, 'b']**

**// [2, 'c']**

**// 类似数组的对象**

**// 类似数组的对象包括好几类。下面是for...of循环用于字符串、DOM NodeList 对象、arguments对象的例子。**

**// 字符串**

**let str = "hello";**

**for (let s of str) {**

**console.log(s); // h e l l o**

**}**

**// DOM NodeList对象**

**let paras = document.querySelectorAll("p");**

**for (let p of paras) {**

**p.classList.add("test");**

**}**

**// arguments对象**

**function printArgs() {**

**for (let x of arguments) {**

**console.log(x);**

**}**

**}**

**printArgs('a', 'b');**

**// 'a'**

**// 'b'**

**// 对于字符串来说，for...of循环还有一个特点，就是会正确识别 32 位 UTF-16 字符。**

**for (let x of 'a\uD83D\uDC0A') {**

**console.log(x);**

**}**

**// 'a'**

**// '\uD83D\uDC0A'**

**// 并不是所有类似数组的对象都具有 Iterator 接口，一个简便的解决方法，就是使用Array.from方法将其转为数组。**

**let arrayLike = { length: 2, 0: 'a', 1: 'b' };**

**// 报错**

**for (let x of arrayLike) {**

**console.log(x);**

**}**

**// 正确**

**for (let x of Array.from(arrayLike)) {**

**console.log(x);**

**}**

**// 对象**

**// 对于普通的对象，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。但是，这样情况下，for...in循环依然可以用来遍历键名。**

**let es6 = {**

**edition: 6,**

**committee: "TC39",**

**standard: "ECMA-262"**

**};**

**for (let e in es6) {**

**console.log(e);**

**}**

**// edition**

**// committee**

**// standard**

**for (let e of es6) {**

**console.log(e);**

**}**

**// TypeError: es6[Symbol.iterator] is not a function**

**// 上面代码表示，对于普通的对象，for...in循环可以遍历键名，for...of循环会报错。**

**// 一种解决方法是，使用Object.keys方法将对象的键名生成一个数组，然后遍历这个数组。**

**for (var key of Object.keys(someObject)) {**

**console.log(key + ': ' + someObject[key]);**

**}**

**// 另一个方法是使用 Generator 函数将对象重新包装一下。**

**function\* entries(obj) {**

**for (let key of Object.keys(obj)) {**

**yield [key, obj[key]];**

**}**

**}**

**for (let [key, value] of entries(obj)) {**

**console.log(key, '->', value);**

**}**

**// a -> 1**

**// b -> 2**

**// c -> 3**

**// 与其他遍历语法的比较**

**// 以数组为例，JavaScript 提供多种遍历语法。最原始的写法就是for循环。**

**for (var index = 0; index < myArray.length; index++) {**

**console.log(myArray[index]);**

**}**

**// 这种写法比较麻烦，因此数组提供内置的forEach方法。**

**myArray.forEach(function (value) {**

**console.log(value);**

**});**

**// 这种写法的问题在于，无法中途跳出forEach循环，break命令或return命令都不能奏效。**

**// for...in循环可以遍历数组的键名。**

**for (var index in myArray) {**

**console.log(myArray[index]);**

**}**

**// for...in循环有几个缺点。**

**// 数组的键名是数字，但是for...in循环是以字符串作为键名“0”、“1”、“2”等等。**

**// for...in循环不仅遍历数字键名，还会遍历手动添加的其他键，甚至包括原型链上的键。**

**// 某些情况下，for...in循环会以任意顺序遍历键名。**

**// 总之，for...in循环主要是为遍历对象而设计的，不适用于遍历数组。**

**// for...of循环相比上面几种做法，有一些显著的优点。**

**for (let value of myArray) {**

**console.log(value);**

**}**

**// 有着同for...in一样的简洁语法，但是没有for...in那些缺点。**

**// 不同于forEach方法，它可以与break、continue和return配合使用。**

**// 提供了遍历所有数据结构的统一操作接口。**

**// 下面是一个使用 break 语句，跳出for...of循环的例子。**

**for (var n of fibonacci) {**

**if (n > 1000)**

**break;**

**console.log(n);**

**}**

**// 上面的例子，会输出斐波纳契数列小于等于 1000 的项。如果当前项大于 1000，就会使用break语句跳出for...of循环。**

**}**

# Generator函数的语法

## // 1.简介

**{**

**// 基本概念**

**// 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。本章详细介绍 Generator 函数的语法和 API，它的异步编程应用请看《Generator 函数的异步应用》一章。**

**// Generator 函数有多种理解角度。语法上，首先可以把它理解成，Generator 函数是一个状态机，封装了多个内部状态。**

**// 执行 Generator 函数会返回一个遍历器对象，也就是说，Generator 函数除了状态机，还是一个遍历器对象生成函数。返回的遍历器对象，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。一是，function关键字与函数名之间有一个星号；二是，函数体内部使用yield表达式，定义不同的内部状态（yield在英语里的意思就是“产出”）。**

**function\* helloWorldGenerator() {**

**yield 'hello';**

**yield 'world';**

**return 'ending';**

**}**

**var hw = helloWorldGenerator();**

**// 上面代码定义了一个 Generator 函数helloWorldGenerator，它内部有两个yield表达式（hello和world），即该函数有三个状态：hello，world 和 return 语句（结束执行）。**

**// 然后，Generator 函数的调用方法与普通函数一样，也是在函数名后面加上一对圆括号。不同的是，调用 Generator 函数后，该函数并不执行，返回的也不是函数运行结果，而是一个指向内部状态的指针对象，也就是上一章介绍的遍历器对象（Iterator Object）。**

**// 下一步，必须调用遍历器对象的next方法，使得指针移向下一个状态。也就是说，每次调用next方法，内部指针就从函数头部或上一次停下来的地方开始执行，直到遇到下一个yield表达式（或return语句）为止。换言之，Generator 函数是分段执行的，yield表达式是暂停执行的标记，而next方法可以恢复执行。**

**hw.next()**

**// { value: 'hello', done: false }**

**hw.next()**

**// { value: 'world', done: false }**

**hw.next()**

**// { value: 'ending', done: true }**

**hw.next()**

**// { value: undefined, done: true }**

**// 上面代码一共调用了四次next方法。**

**// 第一次调用，Generator 函数开始执行，直到遇到第一个yield表达式为止。next方法返回一个对象，它的value属性就是当前yield表达式的值hello，done属性的值false，表示遍历还没有结束。**

**// 第二次调用，Generator 函数从上次yield表达式停下的地方，一直执行到下一个yield表达式。next方法返回的对象的value属性就是当前yield表达式的值world，done属性的值false，表示遍历还没有结束。**

**// 第三次调用，Generator 函数从上次yield表达式停下的地方，一直执行到return语句（如果没有return语句，就执行到函数结束）。next方法返回的对象的value属性，就是紧跟在return语句后面的表达式的值（如果没有return语句，则value属性的值为undefined），done属性的值true，表示遍历已经结束。**

**// 第四次调用，此时 Generator 函数已经运行完毕，next方法返回对象的value属性为undefined，done属性为true。以后再调用next方法，返回的都是这个值。**

**// 总结一下，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。**

**// ES6 没有规定，function关键字与函数名之间的星号，写在哪个位置。这导致下面的写法都能通过。**

**function \* foo(x, y) { ··· }**

**function \*foo(x, y) { ··· }**

**function\* foo(x, y) { ··· }**

**function\*foo(x, y) { ··· }**

**// 由于 Generator 函数仍然是普通函数，所以一般的写法是上面的第三种，即星号紧跟在function关键字后面。本书也采用这种写法。**

**/\*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。**

**需要注意的是，yield表达式后面的表达式，只有当调用next方法、内部指针指向该语句时才会执行，因此等于为 JavaScript 提供了手动的“惰性求值”（Lazy Evaluation）的语法功能。\*/**

**function\* gen() {**

**yield 123 + 456;**

**}**

**// 上面代码中，yield后面的表达式123 + 456，不会立即求值，只会在next方法将指针移到这一句时，才会求值。**

**// yield表达式与return语句既有相似之处，也有区别。相似之处在于，都能返回紧跟在语句后面的那个表达式的值。区别在于每次遇到yield，函数暂停执行，下一次再从该位置继续向后执行，而return语句不具备位置记忆的功能。一个函数里面，只能执行一次（或者说一个）return语句，但是可以执行多次（或者说多个）yield表达式。正常函数只能返回一个值，因为只能执行一次return；Generator 函数可以返回一系列的值，因为可以有任意多个yield。从另一个角度看，也可以说 Generator 生成了一系列的值，这也就是它的名称的来历（英语中，generator 这个词是“生成器”的意思）。**

**// 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();**

**setTimeout(function () {**

**generator.next()**

**}, 2000);**

**// 上面代码中，函数f如果是普通函数，在为变量generator赋值时就会执行。但是，函数f是一个 Generator 函数，就变成只有调用next方法时，函数f才会执行。**

**// 另外需要注意，yield表达式只能用在 Generator 函数里面，用在其他地方都会报错。**

**(function (){**

**yield 1;**

**})()**

**// SyntaxError: Unexpected number**

**// 上面代码在一个普通函数中使用yield表达式，结果产生一个句法错误。**

**// 下面是另外一个例子。**

**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);**

**}**

**// 上面代码也会产生句法错误，因为forEach方法的参数是一个普通函数，但是在里面使用了yield表达式（这个函数里面还使用了yield\*表达式，详细介绍见后文）。一种修改方法是改用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**

**// 另外，yield表达式如果用在另一个表达式之中，必须放在圆括号里面。**

**function\* demo() {**

**console.log('Hello' + yield); // SyntaxError**

**console.log('Hello' + yield 123); // SyntaxError**

**console.log('Hello' + (yield)); // OK**

**console.log('Hello' + (yield 123)); // OK**

**}**

**// yield表达式用作函数参数或放在赋值表达式的右边，可以不加括号。**

**function\* demo() {**

**foo(yield 'a', yield 'b'); // OK**

**let input = yield; // OK**

**}**

**// 与 Iterator 接口的关系**

**// 上一章说过，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]**

**// 上面代码中，Generator 函数赋值给Symbol.iterator属性，从而使得myIterable对象具有了 Iterator 接口，可以被...运算符遍历了。**

**// 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**

**// 上面代码中，gen是一个 Generator 函数，调用它会生成一个遍历器对象g。它的Symbol.iterator属性，也是一个遍历器对象生成函数，执行后返回它自己。**

**}**

## // 2.next 方法的参数

**{**

**// 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，该参数就会被当作上一个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 }**

**// 上面代码先定义了一个可以无限运行的 Generator 函数f，如果next方法没有参数，每次运行到yield表达式，变量reset的值总是undefined。当next方法带一个参数true时，变量reset就被重置为这个参数（即true），因此i会等于-1，下一轮循环就会从-1开始递增。**

**// 这个功能有很重要的语法意义。Generator 函数从暂停状态到恢复运行，它的上下文状态（context）是不变的。通过next方法的参数，就有办法在 Generator 函数开始运行之后，继续向函数体内部注入值。也就是说，可以在 Generator 函数运行的不同阶段，从外部向内部注入不同的值，从而调整函数行为。**

**// 再看一个例子。**

**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 }**

**// 上面代码中，第二次运行next方法的时候不带参数，导致 y 的值等于2 \* undefined（即NaN），除以 3 以后还是NaN，因此返回对象的value属性也等于NaN。第三次运行Next方法的时候不带参数，所以z等于undefined，返回对象的value属性等于5 + NaN + undefined，即NaN。**

**// 如果向next方法提供参数，返回结果就完全不一样了。上面代码第一次调用b的next方法时，返回x+1的值6；第二次调用next方法，将上一次yield表达式的值设为12，因此y等于24，返回y / 3的值8；第三次调用next方法，将上一次yield表达式的值设为13，因此z等于13，这时x等于5，y等于24，所以return语句的值等于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。V8 引擎直接忽略第一次使用next方法时的参数，只有从第二次使用next方法开始，参数才是有效的。从语义上讲，第一个next方法用来启动遍历器对象，所以不用带有参数。**

**// 再看一个通过next方法的参数，向 Generator 函数内部输入值的例子。**

**function\* dataConsumer() {**

**console.log('Started');**

**console.log(`1. ${yield}`);**

**console.log(`2. ${yield}`);**

**return 'result';**

**}**

**let genObj = dataConsumer();**

**genObj.next();**

**// Started**

**genObj.next('a')**

**// 1. a**

**genObj.next('b')**

**// 2. b**

**// 上面代码是一个很直观的例子，每次通过next方法向 Generator 函数输入值，然后打印出来。**

**// 如果想要第一次调用next方法时，就能够输入值，可以在 Generator 函数外面再包一层。**

**function wrapper(generatorFunction) {**

**return function (...args) {**

**let generatorObject = generatorFunction(...args);**

**generatorObject.next();**

**return generatorObject;**

**};**

**}**

**const wrapped = wrapper(function\* () {**

**console.log(`First input: ${yield}`);**

**return 'DONE';**

**});**

**wrapped().next('hello!')**

**// First input: hello!**

**// 上面代码中，Generator 函数如果不用wrapper先包一层，是无法第一次调用next方法，就输入参数的。**

**}**

## // 3.for...of 循环

**{**

**// for...of循环可以自动遍历 Generator 函数时生成的Iterator对象，且此时不再需要调用next方法。**

**function\* foo() {**

**yield 1;**

**yield 2;**

**yield 3;**

**yield 4;**

**yield 5;**

**return 6;**

**}**

**for (let v of foo()) {**

**console.log(v);**

**}**

**// 1 2 3 4 5**

**// 上面代码使用for...of循环，依次显示 5 个yield表达式的值。这里需要注意，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，所以上面代码的return语句返回的6，不包括在for...of循环之中。**

**// 下面是一个利用 Generator 函数和for...of循环，实现斐波那契数列的例子。**

**function\* fibonacci() {**

**let [prev, curr] = [0, 1];**

**for (;;) {**

**yield curr;**

**[prev, curr] = [curr, prev + curr];**

**}**

**}**

**for (let n of fibonacci()) {**

**if (n > 1000) break;**

**console.log(n);**

**}**

**// 从上面代码可见，使用for...of语句时不需要使用next方法。**

**// 利用for...of循环，可以写出遍历任意对象（object）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**

**// 上面代码中，对象jane原生不具备 Iterator 接口，无法用for...of遍历。这时，我们通过 Generator 函数objectEntries为它加上遍历器接口，就可以用for...of遍历了。加上遍历器接口的另一种写法是，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**

**}**

## // 4.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**

**// 上面代码中，遍历器对象i连续抛出两个错误。第一个错误被 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，建议抛出Error对象的实例。**

**var g = function\* () {**

**try {**

**yield;**

**} catch (e) {**

**console.log(e);**

**}**

**};**

**var i = g();**

**i.next();**

**i.throw(new Error('出错了！'));**

**// Error: 出错了！(…)**

**// 注意，不要混淆遍历器对象的throw方法和全局的throw命令。上面代码的错误，是用遍历器对象的throw方法抛出的，而不是用throw命令抛出的。后者只能被函数体外的catch语句捕获。**

**var g = function\* () {**

**while (true) {**

**try {**

**yield;**

**} catch (e) {**

**if (e != 'a') throw e;**

**console.log('内部捕获', e);**

**}**

**}**

**};**

**var i = g();**

**i.next();**

**try {**

**throw new Error('a');**

**throw new Error('b');**

**} catch (e) {**

**console.log('外部捕获', e);**

**}**

**// 外部捕获 [Error: a]**

**// 上面代码之所以只捕获了a，是因为函数体外的catch语句块，捕获了抛出的a错误以后，就不会再继续try代码块里面剩余的语句了。**

**// 如果 Generator 函数内部没有部署try...catch代码块，那么throw方法抛出的错误，将被外部try...catch代码块捕获。**

**var g = function\* () {**

**while (true) {**

**yield;**

**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);**

**}**

**// 外部捕获 a**

**// 上面代码中，Generator 函数g内部没有部署try...catch代码块，所以抛出的错误直接被外部catch代码块捕获。**

**// 如果 Generator 函数内部和外部，都没有部署try...catch代码块，那么程序将报错，直接中断执行。**

**var gen = function\* gen(){**

**yield console.log('hello');**

**yield console.log('world');**

**}**

**var g = gen();**

**g.next();**

**g.throw();**

**// hello**

**// Uncaught undefined**

**// 上面代码中，g.throw抛出错误以后，没有任何try...catch代码块可以捕获这个错误，导致程序报错，中断执行。**

**// 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。**

**function\* gen() {**

**try {**

**yield 1;**

**} catch (e) {**

**console.log('内部捕获');**

**}**

**}**

**var g = gen();**

**g.throw(1);**

**// Uncaught 1**

**// 上面代码中，g.throw(1)执行时，next方法一次都没有执行过。这时，抛出的错误不会被内部捕获，而是直接在外部抛出，导致程序出错。这种行为其实很好理解，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。**

**// throw方法被捕获以后，会附带执行下一条yield表达式。也就是说，会附带执行一次next方法。**

**var gen = function\* gen(){**

**try {**

**yield console.log('a');**

**} catch (e) {**

**// ...**

**}**

**yield console.log('b');**

**yield console.log('c');**

**}**

**var g = gen();**

**g.next() // a**

**g.throw() // b**

**g.next() // c**

**// 上面代码中，g.throw方法被捕获以后，自动执行了一次next方法，所以会打印b。另外，也可以看到，只要 Generator 函数内部部署了try...catch代码块，那么遍历器的throw方法抛出的错误，不影响下一次遍历。**

**// 另外，throw命令与g.throw方法是无关的，两者互不影响。**

**var gen = function\* gen(){**

**yield console.log('hello');**

**yield console.log('world');**

**}**

**var g = gen();**

**g.next();**

**try {**

**throw new Error();**

**} catch (e) {**

**g.next();**

**}**

**// hello**

**// world**

**// 上面代码中，throw命令抛出的错误不会影响到遍历器的状态，所以两次执行next方法，都进行了正确的操作。**

**// 这种函数体内捕获错误的机制，大大方便了对错误的处理。多个yield表达式，可以只用一个try...catch代码块来捕获错误。如果使用回调函数的写法，想要捕获多个错误，就不得不为每个函数内部写一个错误处理语句，现在只在 Generator 函数内部写一次catch语句就可以了。**

**// Generator 函数体外抛出的错误，可以在函数体内捕获；反过来，Generator 函数体内抛出的错误，也可以被函数体外的catch捕获。**

**function\* foo() {**

**var x = yield 3;**

**var y = x.toUpperCase();**

**yield y;**

**}**

**var it = foo();**

**it.next(); // { value:3, done:false }**

**try {**

**it.next(42);**

**} catch (err) {**

**console.log(err);**

**}**

**// 上面代码中，第二个next方法向函数体内传入一个参数 42，数值是没有toUpperCase方法的，所以会抛出一个 TypeError 错误，被函数体外的catch捕获。**

**// 一旦 Generator 执行过程中抛出错误，且没有被内部捕获，就不会再执行下去了。如果此后还调用next方法，将返回一个value属性等于undefined、done属性等于true的对象，即 JavaScript 引擎认为这个 Generator 已经运行结束了。**

**function\* g() {**

**yield 1;**

**console.log('throwing an exception');**

**throw new Error('generator broke!');**

**yield 2;**

**yield 3;**

**}**

**function log(generator) {**

**var v;**

**console.log('starting generator');**

**try {**

**v = generator.next();**

**console.log('第一次运行next方法', v);**

**} catch (err) {**

**console.log('捕捉错误', v);**

**}**

**try {**

**v = generator.next();**

**console.log('第二次运行next方法', v);**

**} catch (err) {**

**console.log('捕捉错误', v);**

**}**

**try {**

**v = generator.next();**

**console.log('第三次运行next方法', v);**

**} catch (err) {**

**console.log('捕捉错误', v);**

**}**

**console.log('caller done');**

**}**

**log(g());**

**// starting generator**

**// 第一次运行next方法 { value: 1, done: false }**

**// throwing an exception**

**// 捕捉错误 { value: 1, done: false }**

**// 第三次运行next方法 { value: undefined, done: true }**

**// caller done**

**// 上面代码一共三次运行next方法，第二次运行的时候会抛出错误，然后第三次运行的时候，Generator 函数就已经结束了，不再执行下去了**

**}**

## // 5.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。并且，Generator 函数的遍历就终止了，返回值的done属性为true，以后再调用next方法，done属性总是返回true。**

**// 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 }**

**// 如果 Generator 函数内部有try...finally代码块，那么return方法会推迟到finally代码块执行完再执行。**

**function\* numbers () {**

**yield 1;**

**try {**

**yield 2;**

**yield 3;**

**} finally {**

**yield 4;**

**yield 5;**

**}**

**yield 6;**

**}**

**var g = numbers();**

**g.next() // { value: 1, done: false }**

**g.next() // { value: 2, done: false }**

**g.return(7) // { value: 4, done: false }**

**g.next() // { value: 5, done: false }**

**g.next() // { value: 7, done: true }**

**// 上面代码中，调用return方法后，就开始执行finally代码块，然后等到finally代码块执行完，再执行return方法**

**}**

## // 6.next()、throw()、return() 的共同点

**{**

**// next()、throw()、return()这三个方法本质上是同一件事，可以放在一起理解。它们的作用都是让 Generator 函数恢复执行，并且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;**

**// 上面代码中，第二个next(1)方法就相当于将yield表达式替换成一个值1。如果next方法没有参数，就相当于替换成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;**

**}**

## // 7.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。**

**function\* foo() {**

**yield 'a';**

**yield 'b';**

**}**

**function\* bar() {**

**yield 'x';**

**foo();**

**yield 'y';**

**}**

**for (let v of bar()){**

**console.log(v);**

**}**

**// "x"**

**// "y"**

**// 上面代码中，foo和bar都是 Generator 函数，在bar里面调用foo，是不会有效果的。**

**// 这个就需要用到yield\*表达式，用来在一个 Generator 函数里面执行另一个 Generator 函数。**

**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';**

**}**

**for (let v of bar()){**

**console.log(v);**

**}**

**// "x"**

**// "a"**

**// "b"**

**// "y"**

**// 再来看一个对比的例子。**

**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"**

**// 上面例子中，outer2使用了yield\*，outer1没使用。结果就是，outer1返回一个遍历器对象，outer2返回该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。**

**let delegatedIterator = (function\* () {**

**yield 'Hello!';**

**yield 'Bye!';**

**}());**

**let delegatingIterator = (function\* () {**

**yield 'Greetings!';**

**yield\* delegatedIterator;**

**yield 'Ok, bye.';**

**}());**

**for(let value of delegatingIterator) {**

**console.log(value);**

**}**

**// "Greetings!**

**// "Hello!"**

**// "Bye!"**

**// "Ok, bye."**

**// 上面代码中，delegatingIterator是代理者，delegatedIterator是被代理者。由于yield\* delegatedIterator语句得到的值，是一个遍历器，所以要用星号表示。运行结果就是使用一个遍历器，遍历了多个 Generator 函数，有递归的效果。**

**// 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。**

**function\* concat(iter1, iter2) {**

**yield\* iter1;**

**yield\* iter2;**

**}**

**// 等同于**

**function\* concat(iter1, iter2) {**

**for (var value of iter1) {**

**yield value;**

**}**

**for (var value of iter2) {**

**yield value;**

**}**

**}**

**// 上面代码说明，yield\*后面的 Generator 函数（没有return语句时），不过是for...of的一种简写形式，完全可以用后者替代前者。反之，在有return语句时，则需要用var value = yield\* iterator的形式获取return语句的值。**

**// 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 }**

**// 上面代码中，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。**

**let read = (function\* () {**

**yield 'hello';**

**yield\* 'hello';**

**})();**

**read.next().value // "hello"**

**read.next().value // "h"**

**// 上面代码中，yield表达式返回整个字符串，yield\*语句返回单个字符。因为字符串具有 Iterator 接口，所以被yield\*遍历。**

**// 如果被代理的 Generator 函数有return语句，那么就可以向代理它的 Generator 函数返回数据。**

**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}**

**// 上面代码在第四次调用next方法的时候，屏幕上会有输出，这是因为函数foo的return语句，向函数bar提供了返回值。**

**// 再看一个例子。**

**function\* genFuncWithReturn() {**

**yield 'a';**

**yield 'b';**

**return 'The result';**

**}**

**function\* logReturned(genObj) {**

**let result = yield\* genObj;**

**console.log(result);**

**}**

**[...logReturned(genFuncWithReturn())]**

**// The result**

**// 值为 [ 'a', 'b' ]**

**// 上面代码中，存在两次遍历。第一次是扩展运算符遍历函数logReturned返回的遍历器对象，第二次是yield\*语句遍历函数genFuncWithReturn返回的遍历器对象。这两次遍历的效果是叠加的，最终表现为扩展运算符遍历函数genFuncWithReturn返回的遍历器对象。所以，最后的数据表达式得到的值等于[ 'a', 'b' ]。但是，函数genFuncWithReturn的return语句的返回值The result，会返回给函数logReturned内部的result变量，因此会有终端输出。**

**// 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**

**// 下面是一个稍微复杂的例子，使用yield\*语句遍历完全二叉树。**

**// 下面是二叉树的构造函数，**

**// 三个参数分别是左树、当前节点和右树**

**function Tree(left, label, right) {**

**this.left = left;**

**this.label = label;**

**this.right = right;**

**}**

**// 下面是中序（inorder）遍历函数。**

**// 由于返回的是一个遍历器，所以要用generator函数。**

**// 函数体内采用递归算法，所以左树和右树要用yield\*遍历**

**function\* inorder(t) {**

**if (t) {**

**yield\* inorder(t.left);**

**yield t.label;**

**yield\* inorder(t.right);**

**}**

**}**

**// 下面生成二叉树**

**function make(array) {**

**// 判断是否为叶节点**

**if (array.length == 1) return new Tree(null, array[0], null);**

**return new Tree(make(array[0]), array[1], make(array[2]));**

**}**

**let tree = make([[['a'], 'b', ['c']], 'd', [['e'], 'f', ['g']]]);**

**// 遍历二叉树**

**var result = [];**

**for (let node of inorder(tree)) {**

**result.push(node);**

**}**

**result**

**// ['a', 'b', 'c', 'd', 'e', 'f', 'g']**

**}**

## // 8.作为对象属性的 Generator 函数

**{**

**// 如果一个对象的属性是 Generator 函数，可以简写成下面的形式。**

**let obj = {**

**\* myGeneratorMethod() {**

**···**

**}**

**};**

**// 上面代码中，myGeneratorMethod属性前面有一个星号，表示这个属性是一个 Generator 函数。**

**// 它的完整形式如下，与上面的写法是等价的。**

**let obj = {**

**myGeneratorMethod: function\* () {**

**// ···**

**}**

**};**

**}**

## // 9.Generator 函数的this

**{**

**// Generator 函数总是返回一个遍历器，ES6 规定这个遍历器是 Generator 函数的实例，也继承了 Generator 函数的prototype对象上的方法。**

**function\* g() {}**

**g.prototype.hello = function () {**

**return 'hi!';**

**};**

**let obj = g();**

**obj instanceof g // true**

**obj.hello() // 'hi!'**

**// 上面代码表明，Generator 函数g返回的遍历器obj，是g的实例，而且继承了g.prototype。但是，如果把g当作普通的构造函数，并不会生效，因为g返回的总是遍历器对象，而不是this对象。**

**function\* g() {**

**this.a = 11;**

**}**

**let obj = g();**

**obj.next();**

**obj.a // undefined**

**// 上面代码中，Generator 函数g在this对象上面添加了一个属性a，但是obj对象拿不到这个属性。**

**// Generator 函数也不能跟new命令一起用，会报错。**

**function\* F() {**

**yield this.x = 2;**

**yield this.y = 3;**

**}**

**new F()**

**// TypeError: F is not a constructor**

**// 上面代码中，new命令跟构造函数F一起使用，结果报错，因为F不是构造函数。**

**// 那么，有没有办法让 Generator 函数返回一个正常的对象实例，既可以用next方法，又可以获得正常的this？**

**// 下面是一个变通方法。首先，生成一个空对象，使用call方法绑定 Generator 函数内部的this。这样，构造函数调用以后，这个空对象就是 Generator 函数的实例对象了。**

**function\* F() {**

**this.a = 1;**

**yield this.b = 2;**

**yield this.c = 3;**

**}**

**var obj = {};**

**var f = F.call(obj);**

**f.next(); // Object {value: 2, done: false}**

**f.next(); // Object {value: 3, done: false}**

**f.next(); // Object {value: undefined, done: true}**

**obj.a // 1**

**obj.b // 2**

**obj.c // 3**

**// 上面代码中，首先是F内部的this对象绑定obj对象，然后调用它，返回一个 Iterator 对象。这个对象执行三次next方法（因为F内部有两个yield表达式），完成 F 内部所有代码的运行。这时，所有内部属性都绑定在obj对象上了，因此obj对象也就成了F的实例。**

**// 上面代码中，执行的是遍历器对象f，但是生成的对象实例是obj，有没有办法将这两个对象统一呢？**

**// 一个办法就是将obj换成F.prototype。**

**function\* F() {**

**this.a = 1;**

**yield this.b = 2;**

**yield this.c = 3;**

**}**

**var f = F.call(F.prototype);**

**f.next(); // Object {value: 2, done: false}**

**f.next(); // Object {value: 3, done: false}**

**f.next(); // Object {value: undefined, done: true}**

**f.a // 1**

**f.b // 2**

**f.c // 3**

**// 再将F改成构造函数，就可以对它执行new命令了。**

**function\* gen() {**

**this.a = 1;**

**yield this.b = 2;**

**yield this.c = 3;**

**}**

**function F() {**

**return gen.call(gen.prototype);**

**}**

**var f = new F();**

**f.next(); // Object {value: 2, done: false}**

**f.next(); // Object {value: 3, done: false}**

**f.next(); // Object {value: undefined, done: true}**

**f.a // 1**

**f.b // 2**

**f.c // 3**

**}**

## // 10.含义

**{**

**// Generator 与状态机**

**// Generator 是实现状态机的最佳结构。比如，下面的clock函数就是一个状态机。**

**var ticking = true;**

**var clock = function() {**

**if (ticking)**

**console.log('Tick!');**

**else**

**console.log('Tock!');**

**ticking = !ticking;**

**}**

**// 上面代码的clock函数一共有两种状态（Tick和Tock），每运行一次，就改变一次状态。这个函数如果用 Generator 实现，就是下面这样。**

**var clock = function\* () {**

**while (true) {**

**console.log('Tick!');**

**yield;**

**console.log('Tock!');**

**yield;**

**}**

**};**

**/\*上面的 Generator 实现与 ES5 实现对比，可以看到少了用来保存状态的外部变量ticking，这样就更简洁，更安全（状态不会被非法篡改）、更符合函数式编程的思想，在写法上也更优雅。Generator 之所以可以不用外部变量保存状态，是因为它本身就包含了一个状态信息，即目前是否处于暂停态。**

**Generator 与协程**

**协程（coroutine）是一种程序运行的方式，可以理解成“协作的线程”或“协作的函数”。协程既可以用单线程实现，也可以用多线程实现。前者是一种特殊的子例程，后者是一种特殊的线程。**

**（1）协程与子例程的差异**

**传统的“子例程”（subroutine）采用堆栈式“后进先出”的执行方式，只有当调用的子函数完全执行完毕，才会结束执行父函数。协程与其不同，多个线程（单线程情况下，即多个函数）可以并行执行，但是只有一个线程（或函数）处于正在运行的状态，其他线程（或函数）都处于暂停态（suspended），线程（或函数）之间可以交换执行权。也就是说，一个线程（或函数）执行到一半，可以暂停执行，将执行权交给另一个线程（或函数），等到稍后收回执行权的时候，再恢复执行。这种可以并行执行、交换执行权的线程（或函数），就称为协程。**

**从实现上看，在内存中，子例程只使用一个栈（stack），而协程是同时存在多个栈，但只有一个栈是在运行状态，也就是说，协程是以多占用内存为代价，实现多任务的并行。**

**（2）协程与普通线程的差异**

**不难看出，协程适合用于多任务运行的环境。在这个意义上，它与普通的线程很相似，都有自己的执行上下文、可以分享全局变量。它们的不同之处在于，同一时间可以有多个线程处于运行状态，但是运行的协程只能有一个，其他协程都处于暂停状态。此外，普通的线程是抢先式的，到底哪个线程优先得到资源，必须由运行环境决定，但是协程是合作式的，执行权由协程自己分配。**

**由于 JavaScript 是单线程语言，只能保持一个调用栈。引入协程以后，每个任务可以保持自己的调用栈。这样做的最大好处，就是抛出错误的时候，可以找到原始的调用栈。不至于像异步操作的回调函数那样，一旦出错，原始的调用栈早就结束。**

**Generator 函数是 ES6 对协程的实现，但属于不完全实现。Generator 函数被称为“半协程”（semi-coroutine），意思是只有 Generator 函数的调用者，才能将程序的执行权还给 Generator 函数。如果是完全执行的协程，任何函数都可以让暂停的协程继续执行。**

**如果将 Generator 函数当作协程，完全可以将多个需要互相协作的任务写成 Generator 函数，它们之间使用yield表达式交换控制权。**

**Generator 与上下文**

**JavaScript 代码运行时，会产生一个全局的上下文环境（context，又称运行环境），包含了当前所有的变量和对象。然后，执行函数（或块级代码）的时候，又会在当前上下文环境的上层，产生一个函数运行的上下文，变成当前（active）的上下文，由此形成一个上下文环境的堆栈（context stack）。**

**这个堆栈是“后进先出”的数据结构，最后产生的上下文环境首先执行完成，退出堆栈，然后再执行完成它下层的上下文，直至所有代码执行完成，堆栈清空。**

**Generator 函数不是这样，它执行产生的上下文环境，一旦遇到yield命令，就会暂时退出堆栈，但是并不消失，里面的所有变量和对象会冻结在当前状态。等到对它执行next命令时，这个上下文环境又会重新加入调用栈，冻结的变量和对象恢复执行。\*/**

**function\* gen() {**

**yield 1;**

**return 2;**

**}**

**let g = gen();**

**console.log(**

**g.next().value,**

**g.next().value,**

**);**

**// 上面代码中，第一次执行g.next()时，Generator 函数gen的上下文会加入堆栈，即开始运行gen内部的代码。等遇到yield 1时，gen上下文退出堆栈，内部状态冻结。第二次执行g.next()时，gen上下文重新加入堆栈，变成当前的上下文，重新恢复执行。**

**}**

## // 11.应用

**{**

**// Generator 可以暂停函数执行，返回任意表达式的值。这种特点使得 Generator 有多种应用场景。**

**// （1）异步操作的同步化表达**

**// Generator 函数的暂停执行的效果，意味着可以把异步操作写在yield表达式里面，等到调用next方法时再往后执行。这实际上等同于不需要写回调函数了，因为异步操作的后续操作可以放在yield表达式下面，反正要等到调用next方法时再执行。所以，Generator 函数的一个重要实际意义就是用来处理异步操作，改写回调函数。**

**function\* loadUI() {**

**showLoadingScreen();**

**yield loadUIDataAsynchronously();**

**hideLoadingScreen();**

**}**

**var loader = loadUI();**

**// 加载UI**

**loader.next()**

**// 卸载UI**

**loader.next()**

**// 上面代码中，第一次调用loadUI函数时，该函数不会执行，仅返回一个遍历器。下一次对该遍历器调用next方法，则会显示Loading界面（showLoadingScreen），并且异步加载数据（loadUIDataAsynchronously）。等到数据加载完成，再一次使用next方法，则会隐藏Loading界面。可以看到，这种写法的好处是所有Loading界面的逻辑，都被封装在一个函数，按部就班非常清晰。**

**// Ajax 是典型的异步操作，通过 Generator 函数部署 Ajax 操作，可以用同步的方式表达。**

**function\* main() {**

**var result = yield request("http://some.url");**

**var resp = JSON.parse(result);**

**console.log(resp.value);**

**}**

**function request(url) {**

**makeAjaxCall(url, function(response){**

**it.next(response);**

**});**

**}**

**var it = main();**

**it.next();**

**// 上面代码的main函数，就是通过 Ajax 操作获取数据。可以看到，除了多了一个yield，它几乎与同步操作的写法完全一样。注意，makeAjaxCall函数中的next方法，必须加上response参数，因为yield表达式，本身是没有值的，总是等于undefined。**

**// 下面是另一个例子，通过 Generator 函数逐行读取文本文件。**

**function\* numbers() {**

**let file = new FileReader("numbers.txt");**

**try {**

**while(!file.eof) {**

**yield parseInt(file.readLine(), 10);**

**}**

**} finally {**

**file.close();**

**}**

**}**

**// 上面代码打开文本文件，使用yield表达式可以手动逐行读取文件。**

**// （2）控制流管理**

**// 如果有一个多步操作非常耗时，采用回调函数，可能会写成下面这样。**

**step1(function (value1) {**

**step2(value1, function(value2) {**

**step3(value2, function(value3) {**

**step4(value3, function(value4) {**

**// Do something with value4**

**});**

**});**

**});**

**});**

**// 采用 Promise 改写上面的代码。**

**Promise.resolve(step1)**

**.then(step2)**

**.then(step3)**

**.then(step4)**

**.then(function (value4) {**

**// Do something with value4**

**}, function (error) {**

**// Handle any error from step1 through step4**

**})**

**.done();**

**// 上面代码已经把回调函数，改成了直线执行的形式，但是加入了大量 Promise 的语法。Generator 函数可以进一步改善代码运行流程。**

**function\* longRunningTask(value1) {**

**try {**

**var value2 = yield step1(value1);**

**var value3 = yield step2(value2);**

**var value4 = yield step3(value3);**

**var value5 = yield step4(value4);**

**// Do something with value4**

**} catch (e) {**

**// Handle any error from step1 through step4**

**}**

**}**

**// 然后，使用一个函数，按次序自动执行所有步骤。**

**scheduler(longRunningTask(initialValue));**

**function scheduler(task) {**

**var taskObj = task.next(task.value);**

**// 如果Generator函数未结束，就继续调用**

**if (!taskObj.done) {**

**task.value = taskObj.value**

**scheduler(task);**

**}**

**}**

**// 注意，上面这种做法，只适合同步操作，即所有的task都必须是同步的，不能有异步操作。因为这里的代码一得到返回值，就继续往下执行，没有判断异步操作何时完成。如果要控制异步的操作流程，详见后面的《异步操作》一章。**

**// 下面，利用for...of循环会自动依次执行yield命令的特性，提供一种更一般的控制流管理的方法。**

**let steps = [step1Func, step2Func, step3Func];**

**function\* iterateSteps(steps){**

**for (var i=0; i< steps.length; i++){**

**var step = steps[i];**

**yield step();**

**}**

**}**

**// 上面代码中，数组steps封装了一个任务的多个步骤，Generator 函数iterateSteps则是依次为这些步骤加上yield命令。**

**// 将任务分解成步骤之后，还可以将项目分解成多个依次执行的任务。**

**let jobs = [job1, job2, job3];**

**function\* iterateJobs(jobs){**

**for (var i=0; i< jobs.length; i++){**

**var job = jobs[i];**

**yield\* iterateSteps(job.steps);**

**}**

**}**

**// 上面代码中，数组jobs封装了一个项目的多个任务，Generator 函数iterateJobs则是依次为这些任务加上yield\*命令。**

**// 最后，就可以用for...of循环一次性依次执行所有任务的所有步骤。**

**for (var step of iterateJobs(jobs)){**

**console.log(step.id);**

**}**

**// 再次提醒，上面的做法只能用于所有步骤都是同步操作的情况，不能有异步操作的步骤。如果想要依次执行异步的步骤，必须使用后面的《异步操作》一章介绍的方法。**

**// for...of的本质是一个while循环，所以上面的代码实质上执行的是下面的逻辑。**

**var it = iterateJobs(jobs);**

**var res = it.next();**

**while (!res.done){**

**var result = res.value;**

**// ...**

**res = it.next();**

**}**

**// （3）部署 Iterator 接口**

**// 利用 Generator 函数，可以在任意对象上部署 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**

**// 上述代码中，myObj是一个普通对象，通过iterEntries函数，就有了 Iterator 接口。也就是说，可以在任意对象上部署next方法。**

**// 下面是一个对数组部署 Iterator 接口的例子，尽管数组原生具有这个接口。**

**function\* makeSimpleGenerator(array){**

**var nextIndex = 0;**

**while(nextIndex < array.length){**

**yield array[nextIndex++];**

**}**

**}**

**var gen = makeSimpleGenerator(['yo', 'ya']);**

**gen.next().value // 'yo'**

**gen.next().value // 'ya'**

**gen.next().done // true**

**// （4）作为数据结构**

**// Generator 可以看作是数据结构，更确切地说，可以看作是一个数组结构，因为 Generator 函数可以返回一系列的值，这意味着它可以对任意表达式，提供类似数组的接口。**

**function\* doStuff() {**

**yield fs.readFile.bind(null, 'hello.txt');**

**yield fs.readFile.bind(null, 'world.txt');**

**yield fs.readFile.bind(null, 'and-such.txt');**

**}**

**// 上面代码就是依次返回三个函数，但是由于使用了 Generator 函数，导致可以像处理数组那样，处理这三个返回的函数。**

**for (task of doStuff()) {**

**// task是一个函数，可以像回调函数那样使用它**

**}**

**// 实际上，如果用 ES5 表达，完全可以用数组模拟 Generator 的这种用法。**

**function doStuff() {**

**return [**

**fs.readFile.bind(null, 'hello.txt'),**

**fs.readFile.bind(null, 'world.txt'),**

**fs.readFile.bind(null, 'and-such.txt')**

**];**

**}**

**// 上面的函数，可以用一模一样的for...of循环处理！两相一比较，就不难看出 Generator 使得数据或者操作，具备了类似数组的接口**

**}**

# Generator 函数的异步应用

## // 1.传统方法

**{**

**/\*ES6 诞生以前，异步编程的方法，大概有下面四种。**

**回调函数**

**事件监听**

**发布/订阅**

**Promise 对象**

**Generator 函数将 JavaScript 异步编程带入了一个全新的阶段。\*/**

**}**

## // 2.基本概念

**{**

**/\*异步**

**所谓"异步"，简单说就是一个任务不是连续完成的，可以理解成该任务被人为分成两段，先执行第一段，然后转而执行其他任务，等做好了准备，再回过头执行第二段。**

**比如，有一个任务是读取文件进行处理，任务的第一段是向操作系统发出请求，要求读取文件。然后，程序执行其他任务，等到操作系统返回文件，再接着执行任务的第二段（处理文件）。这种不连续的执行，就叫做异步。**

**相应地，连续的执行就叫做同步。由于是连续执行，不能插入其他任务，所以操作系统从硬盘读取文件的这段时间，程序只能干等着。**

**回调函数**

**JavaScript 语言对异步编程的实现，就是回调函数。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，就直接调用这个函数。回调函数的英语名字callback，直译过来就是"重新调用"。**

**读取文件进行处理，是这样写的。\*/**

**fs.readFile('/etc/passwd', 'utf-8', function (err, data) {**

**if (err) throw err;**

**console.log(data);**

**});**

**// 上面代码中，readFile函数的第三个参数，就是回调函数，也就是任务的第二段。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。**

**// 一个有趣的问题是，为什么 Node 约定，回调函数的第一个参数，必须是错误对象err（如果没有错误，该参数就是null）？**

**// 原因是执行分成两段，第一段执行完以后，任务所在的上下文环境就已经结束了。在这以后抛出的错误，原来的上下文环境已经无法捕捉，只能当作参数，传入第二段。**

**// Promise**

**// 回调函数本身并没有问题，它的问题出现在多个回调函数嵌套。假定读取A文件之后，再读取B文件，代码如下。**

**fs.readFile(fileA, 'utf-8', function (err, data) {**

**fs.readFile(fileB, 'utf-8', function (err, data) {**

**// ...**

**});**

**});**

**// 不难想象，如果依次读取两个以上的文件，就会出现多重嵌套。代码不是纵向发展，而是横向发展，很快就会乱成一团，无法管理。因为多个异步操作形成了强耦合，只要有一个操作需要修改，它的上层回调函数和下层回调函数，可能都要跟着修改。这种情况就称为"回调函数地狱"（callback hell）。**

**// Promise 对象就是为了解决这个问题而提出的。它不是新的语法功能，而是一种新的写法，允许将回调函数的嵌套，改成链式调用。采用 Promise，连续读取多个文件，写法如下。**

**var readFile = require('fs-readfile-promise');**

**readFile(fileA)**

**.then(function (data) {**

**console.log(data.toString());**

**})**

**.then(function () {**

**return readFile(fileB);**

**})**

**.then(function (data) {**

**console.log(data.toString());**

**})**

**.catch(function (err) {**

**console.log(err);**

**});**

**// 上面代码中，我使用了fs-readfile-promise模块，它的作用就是返回一个 Promise 版本的readFile函数。Promise 提供then方法加载回调函数，catch方法捕捉执行过程中抛出的错误。**

**// 可以看到，Promise 的写法只是回调函数的改进，使用then方法以后，异步任务的两段执行看得更清楚了，除此以外，并无新意。**

**// Promise 的最大问题是代码冗余，原来的任务被 Promise 包装了一下，不管什么操作，一眼看去都是一堆then，原来的语义变得很不清楚。**

**// 那么，有没有更好的写法呢？**

**}**

## // 3.Generator 函数

**{**

**/\*协程**

**传统的编程语言，早有异步编程的解决方案（其实是多任务的解决方案）。其中有一种叫做"协程"（coroutine），意思是多个线程互相协作，完成异步任务。**

**协程有点像函数，又有点像线程。它的运行流程大致如下。**

**第一步，协程A开始执行。**

**第二步，协程A执行到一半，进入暂停，执行权转移到协程B。**

**第三步，（一段时间后）协程B交还执行权。**

**第四步，协程A恢复执行。**

**上面流程的协程A，就是异步任务，因为它分成两段（或多段）执行。**

**举例来说，读取文件的协程写法如下。\*/**

**function\* asyncJob() {**

**// ...其他代码**

**var f = yield readFile(fileA);**

**// ...其他代码**

**}**

**// 上面代码的函数asyncJob是一个协程，它的奥妙就在其中的yield命令。它表示执行到此处，执行权将交给其他协程。也就是说，yield命令是异步两个阶段的分界线。**

**// 协程遇到yield命令就暂停，等到执行权返回，再从暂停的地方继续往后执行。它的最大优点，就是代码的写法非常像同步操作，如果去除yield命令，简直一模一样。**

**// 协程的 Generator 函数实现**

**// Generator 函数是协程在 ES6 的实现，最大特点就是可以交出函数的执行权（即暂停执行）。**

**// 整个 Generator 函数就是一个封装的异步任务，或者说是异步任务的容器。异步操作需要暂停的地方，都用yield语句注明。Generator 函数的执行方法如下。**

**function\* gen(x) {**

**var y = yield x + 2;**

**return y;**

**}**

**var g = gen(1);**

**g.next() // { value: 3, done: false }**

**g.next() // { value: undefined, done: true }**

**// 上面代码中，调用 Generator 函数，会返回一个内部指针（即遍历器）g。这是 Generator 函数不同于普通函数的另一个地方，即执行它不会返回结果，返回的是指针对象。调用指针g的next方法，会移动内部指针（即执行异步任务的第一段），指向第一个遇到的yield语句，上例是执行到x + 2为止。**

**// 换言之，next方法的作用是分阶段执行Generator函数。每次调用next方法，会返回一个对象，表示当前阶段的信息（value属性和done属性）。value属性是yield语句后面表达式的值，表示当前阶段的值；done属性是一个布尔值，表示 Generator 函数是否执行完毕，即是否还有下一个阶段。**

**// Generator 函数的数据交换和错误处理**

**// Generator 函数可以暂停执行和恢复执行，这是它能封装异步任务的根本原因。除此之外，它还有两个特性，使它可以作为异步编程的完整解决方案：函数体内外的数据交换和错误处理机制。**

**// next返回值的 value 属性，是 Generator 函数向外输出数据；next方法还可以接受参数，向 Generator 函数体内输入数据。**

**function\* gen(x){**

**var y = yield x + 2;**

**return y;**

**}**

**var g = gen(1);**

**g.next() // { value: 3, done: false }**

**g.next(2) // { value: 2, done: true }**

**// 上面代码中，第一个next方法的value属性，返回表达式x + 2的值3。第二个next方法带有参数2，这个参数可以传入 Generator 函数，作为上个阶段异步任务的返回结果，被函数体内的变量y接收。因此，这一步的value属性，返回的就是2（变量y的值）。**

**// Generator 函数内部还可以部署错误处理代码，捕获函数体外抛出的错误。**

**function\* gen(x){**

**try {**

**var y = yield x + 2;**

**} catch (e){**

**console.log(e);**

**}**

**return y;**

**}**

**var g = gen(1);**

**g.next();**

**g.throw('出错了');**

**// 出错了**

**// 上面代码的最后一行，Generator 函数体外，使用指针对象的throw方法抛出的错误，可以被函数体内的try...catch代码块捕获。这意味着，出错的代码与处理错误的代码，实现了时间和空间上的分离，这对于异步编程无疑是很重要的。**

**// 异步任务的封装**

**// 下面看看如何使用 Generator 函数，执行一个真实的异步任务。**

**var fetch = require('node-fetch');**

**function\* gen(){**

**var url = 'https://api.github.com/users/github';**

**var result = yield fetch(url);**

**console.log(result.bio);**

**}**

**// 上面代码中，Generator 函数封装了一个异步操作，该操作先读取一个远程接口，然后从 JSON 格式的数据解析信息。就像前面说过的，这段代码非常像同步操作，除了加上了yield命令。**

**// 执行这段代码的方法如下。**

**var g = gen();**

**var result = g.next();**

**result.value.then(function(data){**

**return data.json();**

**}).then(function(data){**

**g.next(data);**

**});**

**// 上面代码中，首先执行 Generator 函数，获取遍历器对象，然后使用next方法（第二行），执行异步任务的第一阶段。由于Fetch模块返回的是一个 Promise 对象，因此要用then方法调用下一个next方法。**

**// 可以看到，虽然 Generator 函数将异步操作表示得很简洁，但是流程管理却不方便（即何时执行第一阶段、何时执行第二阶段）。**

**}**

## // 4.Thunk 函数

**{**

**// hunk 函数是自动执行 Generator 函数的一种方法。**

**// 参数的求值策略**

**// Thunk 函数早在上个世纪 60 年代就诞生了。**

**// 那时，编程语言刚刚起步，计算机学家还在研究，编译器怎么写比较好。一个争论的焦点是"求值策略"，即函数的参数到底应该何时求值。**

**var x = 1;**

**function f(m) {**

**return m \* 2;**

**}**

**f(x + 5)**

**// 上面代码先定义函数f，然后向它传入表达式x + 5。请问，这个表达式应该何时求值？**

**// 一种意见是"传值调用"（call by value），即在进入函数体之前，就计算x + 5的值（等于 6），再将这个值传入函数f。C 语言就采用这种策略。**

**f(x + 5)**

**// 传值调用时，等同于**

**f(6)**

**// 另一种意见是“传名调用”（call by name），即直接将表达式x + 5传入函数体，只在用到它的时候求值。Haskell 语言采用这种策略。**

**f(x + 5)**

**// 传名调用时，等同于**

**(x + 5) \* 2**

**// 传值调用和传名调用，哪一种比较好？**

**// 回答是各有利弊。传值调用比较简单，但是对参数求值的时候，实际上还没用到这个参数，有可能造成性能损失。**

**function f(a, b){**

**return b;**

**}**

**f(3 \* x \* x - 2 \* x - 1, x);**

**// 上面代码中，函数f的第一个参数是一个复杂的表达式，但是函数体内根本没用到。对这个参数求值，实际上是不必要的。因此，有一些计算机学家倾向于"传名调用"，即只在执行时求值。**

**// Thunk 函数的含义**

**// 编译器的“传名调用”实现，往往是将参数放到一个临时函数之中，再将这个临时函数传入函数体。这个临时函数就叫做 Thunk 函数。**

**function f(m) {**

**return m \* 2;**

**}**

**f(x + 5);**

**// 等同于**

**var thunk = function () {**

**return x + 5;**

**};**

**function f(thunk) {**

**return thunk() \* 2;**

**}**

**// 上面代码中，函数 f 的参数x + 5被一个函数替换了。凡是用到原参数的地方，对Thunk函数求值即可。**

**// 这就是 Thunk 函数的定义，它是“传名调用”的一种实现策略，用来替换某个表达式。**

**// JavaScript 语言的 Thunk 函数**

**// JavaScript 语言是传值调用，它的 Thunk 函数含义有所不同。在 JavaScript 语言中，Thunk 函数替换的不是表达式，而是多参数函数，将其替换成一个只接受回调函数作为参数的单参数函数。**

**// 正常版本的readFile（多参数版本）**

**fs.readFile(fileName, callback);**

**// Thunk版本的readFile（单参数版本）**

**var Thunk = function (fileName) {**

**return function (callback) {**

**return fs.readFile(fileName, callback);**

**};**

**};**

**var readFileThunk = Thunk(fileName);**

**readFileThunk(callback);**

**// 上面代码中，fs模块的readFile方法是一个多参数函数，两个参数分别为文件名和回调函数。经过转换器处理，它变成了一个单参数函数，只接受回调函数作为参数。这个单参数版本，就叫做 Thunk 函数。**

**// 任何函数，只要参数有回调函数，就能写成 Thunk 函数的形式。下面是一个简单的 Thunk 函数转换器。**

**// ES5版本**

**var Thunk = function(fn){**

**return function (){**

**var args = Array.prototype.slice.call(arguments);**

**return function (callback){**

**args.push(callback);**

**return fn.apply(this, args);**

**}**

**};**

**};**

**// ES6版本**

**const Thunk = function(fn) {**

**return function (...args) {**

**return function (callback) {**

**return fn.call(this, ...args, callback);**

**}**

**};**

**};**

**// 使用上面的转换器，生成fs.readFile的 Thunk 函数。**

**var readFileThunk = Thunk(fs.readFile);**

**readFileThunk(fileA)(callback);**

**// 下面是另一个完整的例子。**

**function f(a, cb) {**

**cb(a);**

**}**

**const ft = Thunk(f);**

**ft(1)(console.log) // 1**

**// Thunkify 模块**

**// 生产环境的转换器，建议使用 Thunkify 模块。**

**// 首先是安装。**

**$ npm install thunkify**

**// 使用方式如下。**

**var thunkify = require('thunkify');**

**var fs = require('fs');**

**var read = thunkify(fs.readFile);**

**read('package.json')(function(err, str){**

**// ...**

**});**

**// Thunkify 的源码与上一节那个简单的转换器非常像。**

**function thunkify(fn) {**

**return function() {**

**var args = new Array(arguments.length);**

**var ctx = this;**

**for (var i = 0; i < args.length; ++i) {**

**args[i] = arguments[i];**

**}**

**return function (done) {**

**var called;**

**args.push(function () {**

**if (called) return;**

**called = true;**

**done.apply(null, arguments);**

**});**

**try {**

**fn.apply(ctx, args);**

**} catch (err) {**

**done(err);**

**}**

**}**

**}**

**};**

**// 它的源码主要多了一个检查机制，变量called确保回调函数只运行一次。这样的设计与下文的 Generator 函数相关。请看下面的例子。**

**function f(a, b, callback){**

**var sum = a + b;**

**callback(sum);**

**callback(sum);**

**}**

**var ft = thunkify(f);**

**var print = console.log.bind(console);**

**ft(1, 2)(print);**

**// 3**

**// 上面代码中，由于thunkify只允许回调函数执行一次，所以只输出一行结果。**

**// Generator 函数的流程管理**

**// 你可能会问， Thunk 函数有什么用？回答是以前确实没什么用，但是 ES6 有了 Generator 函数，Thunk 函数现在可以用于 Generator 函数的自动流程管理。**

**// Generator 函数可以自动执行。**

**function\* gen() {**

**// ...**

**}**

**var g = gen();**

**var res = g.next();**

**while(!res.done){**

**console.log(res.value);**

**res = g.next();**

**}**

**// 上面代码中，Generator 函数gen会自动执行完所有步骤。**

**// 但是，这不适合异步操作。如果必须保证前一步执行完，才能执行后一步，上面的自动执行就不可行。这时，Thunk 函数就能派上用处。以读取文件为例。下面的 Generator 函数封装了两个异步操作。**

**var fs = require('fs');**

**var thunkify = require('thunkify');**

**var readFileThunk = thunkify(fs.readFile);**

**var gen = function\* (){**

**var r1 = yield readFileThunk('/etc/fstab');**

**console.log(r1.toString());**

**var r2 = yield readFileThunk('/etc/shells');**

**console.log(r2.toString());**

**};**

**// 上面代码中，yield命令用于将程序的执行权移出 Generator 函数，那么就需要一种方法，将执行权再交还给 Generator 函数。**

**// 这种方法就是 Thunk 函数，因为它可以在回调函数里，将执行权交还给 Generator 函数。为了便于理解，我们先看如何手动执行上面这个 Generator 函数。**

**var g = gen();**

**var r1 = g.next();**

**r1.value(function (err, data) {**

**if (err) throw err;**

**var r2 = g.next(data);**

**r2.value(function (err, data) {**

**if (err) throw err;**

**g.next(data);**

**});**

**});**

**// 上面代码中，变量g是 Generator 函数的内部指针，表示目前执行到哪一步。next方法负责将指针移动到下一步，并返回该步的信息（value属性和done属性）。**

**// 仔细查看上面的代码，可以发现 Generator 函数的执行过程，其实是将同一个回调函数，反复传入next方法的value属性。这使得我们可以用递归来自动完成这个过程。**

**// Thunk 函数的自动流程管理**

**// Thunk 函数真正的威力，在于可以自动执行 Generator 函数。下面就是一个基于 Thunk 函数的 Generator 执行器。**

**function run(fn) {**

**var gen = fn();**

**function next(err, data) {**

**var result = gen.next(data);**

**if (result.done) return;**

**result.value(next);**

**}**

**next();**

**}**

**function\* g() {**

**// ...**

**}**

**run(g);**

**// 上面代码的run函数，就是一个 Generator 函数的自动执行器。内部的next函数就是 Thunk 的回调函数。next函数先将指针移到 Generator 函数的下一步（gen.next方法），然后判断 Generator 函数是否结束（result.done属性），如果没结束，就将next函数再传入 Thunk 函数（result.value属性），否则就直接退出。**

**// 有了这个执行器，执行 Generator 函数方便多了。不管内部有多少个异步操作，直接把 Generator 函数传入run函数即可。当然，前提是每一个异步操作，都要是 Thunk 函数，也就是说，跟在yield命令后面的必须是 Thunk 函数。**

**var g = function\* (){**

**var f1 = yield readFileThunk('fileA');**

**var f2 = yield readFileThunk('fileB');**

**// ...**

**var fn = yield readFileThunk('fileN');**

**};**

**run(g);**

**// 上面代码中，函数g封装了n个异步的读取文件操作，只要执行run函数，这些操作就会自动完成。这样一来，异步操作不仅可以写得像同步操作，而且一行代码就可以执行。**

**// Thunk 函数并不是 Generator 函数自动执行的唯一方案。因为自动执行的关键是，必须有一种机制，自动控制 Generator 函数的流程，接收和交还程序的执行权。回调函数可以做到这一点，Promise 对象也可以做到这一点。**

**}**

## // 5.co 模块

**{**

**// 基本用法**

**// co 模块是著名程序员 TJ Holowaychuk 于 2013 年 6 月发布的一个小工具，用于 Generator 函数的自动执行。**

**// 下面是一个 Generator 函数，用于依次读取两个文件。**

**var gen = function\* () {**

**var f1 = yield readFile('/etc/fstab');**

**var f2 = yield readFile('/etc/shells');**

**console.log(f1.toString());**

**console.log(f2.toString());**

**};**

**// co 模块可以让你不用编写 Generator 函数的执行器。**

**var co = require('co');**

**co(gen);**

**// 上面代码中，Generator 函数只要传入co函数，就会自动执行。**

**// co函数返回一个Promise对象，因此可以用then方法添加回调函数。**

**co(gen).then(function (){**

**console.log('Generator 函数执行完成');**

**});**

**/\*上面代码中，等到 Generator 函数执行结束，就会输出一行提示。**

**co 模块的原理**

**为什么 co 可以自动执行 Generator 函数？**

**前面说过，Generator 就是一个异步操作的容器。它的自动执行需要一种机制，当异步操作有了结果，能够自动交回执行权。**

**两种方法可以做到这一点。**

**（1）回调函数。将异步操作包装成 Thunk 函数，在回调函数里面交回执行权。**

**（2）Promise 对象。将异步操作包装成 Promise 对象，用then方法交回执行权。**

**co 模块其实就是将两种自动执行器（Thunk 函数和 Promise 对象），包装成一个模块。使用 co 的前提条件是，Generator 函数的yield命令后面，只能是 Thunk 函数或 Promise 对象。如果数组或对象的成员，全部都是 Promise 对象，也可以使用 co，详见后文的例子。**

**上一节已经介绍了基于 Thunk 函数的自动执行器。下面来看，基于 Promise 对象的自动执行器。这是理解 co 模块必须的。**

**基于 Promise 对象的自动执行**

**还是沿用上面的例子。首先，把fs模块的readFile方法包装成一个 Promise 对象。**

**\*/**

**var fs = require('fs');**

**var readFile = function (fileName){**

**return new Promise(function (resolve, reject){**

**fs.readFile(fileName, function(error, data){**

**if (error) return reject(error);**

**resolve(data);**

**});**

**});**

**};**

**var gen = function\* (){**

**var f1 = yield readFile('/etc/fstab');**

**var f2 = yield readFile('/etc/shells');**

**console.log(f1.toString());**

**console.log(f2.toString());**

**};**

**// 然后，手动执行上面的 Generator 函数。**

**var g = gen();**

**g.next().value.then(function(data){**

**g.next(data).value.then(function(data){**

**g.next(data);**

**});**

**});**

**// 手动执行其实就是用then方法，层层添加回调函数。理解了这一点，就可以写出一个自动执行器。**

**function run(gen){**

**var g = gen();**

**function next(data){**

**var result = g.next(data);**

**if (result.done) return result.value;**

**result.value.then(function(data){**

**next(data);**

**});**

**}**

**next();**

**}**

**run(gen);**

**// 上面代码中，只要 Generator 函数还没执行到最后一步，next函数就调用自身，以此实现自动执行。**

**// co 模块的源码**

**// co 就是上面那个自动执行器的扩展，它的源码只有几十行，非常简单。**

**// 首先，co 函数接受 Generator 函数作为参数，返回一个 Promise 对象。**

**function co(gen) {**

**var ctx = this;**

**return new Promise(function(resolve, reject) {**

**});**

**}**

**// 在返回的 Promise 对象里面，co 先检查参数gen是否为 Generator 函数。如果是，就执行该函数，得到一个内部指针对象；如果不是就返回，并将 Promise 对象的状态改为resolved。**

**function co(gen) {**

**var ctx = this;**

**return new Promise(function(resolve, reject) {**

**if (typeof gen === 'function') gen = gen.call(ctx);**

**if (!gen || typeof gen.next !== 'function') return resolve(gen);**

**});**

**}**

**// 接着，co 将 Generator 函数的内部指针对象的next方法，包装成onFulfilled函数。这主要是为了能够捕捉抛出的错误。**

**function co(gen) {**

**var ctx = this;**

**return new Promise(function(resolve, reject) {**

**if (typeof gen === 'function') gen = gen.call(ctx);**

**if (!gen || typeof gen.next !== 'function') return resolve(gen);**

**onFulfilled();**

**function onFulfilled(res) {**

**var ret;**

**try {**

**ret = gen.next(res);**

**} catch (e) {**

**return reject(e);**

**}**

**next(ret);**

**}**

**});**

**}**

**// 最后，就是关键的next函数，它会反复调用自身。**

**function next(ret) {**

**if (ret.done) return resolve(ret.value);**

**var value = toPromise.call(ctx, ret.value);**

**if (value && isPromise(value)) return value.then(onFulfilled, onRejected);**

**return onRejected(**

**new TypeError(**

**'You may only yield a function, promise, generator, array, or object, '**

**+ 'but the following object was passed: "'**

**+ String(ret.value)**

**+ '"'**

**)**

**);**

**}**

**// 上面代码中，next函数的内部代码，一共只有四行命令。**

**// 第一行，检查当前是否为 Generator 函数的最后一步，如果是就返回。**

**// 第二行，确保每一步的返回值，是 Promise 对象。**

**// 第三行，使用then方法，为返回值加上回调函数，然后通过onFulfilled函数再次调用next函数。**

**// 第四行，在参数不符合要求的情况下（参数非 Thunk 函数和 Promise 对象），将 Promise 对象的状态改为rejected，从而终止执行。**

**// 处理并发的异步操作**

**// co 支持并发的异步操作，即允许某些操作同时进行，等到它们全部完成，才进行下一步。**

**// 这时，要把并发的操作都放在数组或对象里面，跟在yield语句后面。**

**// 数组的写法**

**co(function\* () {**

**var res = yield [**

**Promise.resolve(1),**

**Promise.resolve(2)**

**];**

**console.log(res);**

**}).catch(onerror);**

**// 对象的写法**

**co(function\* () {**

**var res = yield {**

**1: Promise.resolve(1),**

**2: Promise.resolve(2),**

**};**

**console.log(res);**

**}).catch(onerror);**

**// 下面是另一个例子。**

**co(function\* () {**

**var values = [n1, n2, n3];**

**yield values.map(somethingAsync);**

**});**

**function\* somethingAsync(x) {**

**// do something async**

**return y**

**}**

**// 上面的代码允许并发三个somethingAsync异步操作，等到它们全部完成，才会进行下一步。**

**// 实例：处理 Stream**

**// Node 提供 Stream 模式读写数据，特点是一次只处理数据的一部分，数据分成一块块依次处理，就好像“数据流”一样。这对于处理大规模数据非常有利。Stream 模式使用 EventEmitter API，会释放三个事件。**

**// data事件：下一块数据块已经准备好了。**

**// end事件：整个“数据流”处理“完了。**

**// error事件：发生错误。**

**// 使用Promise.race()函数，可以判断这三个事件之中哪一个最先发生，只有当data事件最先发生时，才进入下一个数据块的处理。从而，我们可以通过一个while循环，完成所有数据的读取。**

**const co = require('co');**

**const fs = require('fs');**

**const stream = fs.createReadStream('./les\_miserables.txt');**

**let valjeanCount = 0;**

**co(function\*() {**

**while(true) {**

**const res = yield Promise.race([**

**new Promise(resolve => stream.once('data', resolve)),**

**new Promise(resolve => stream.once('end', resolve)),**

**new Promise((resolve, reject) => stream.once('error', reject))**

**]);**

**if (!res) {**

**break;**

**}**

**stream.removeAllListeners('data');**

**stream.removeAllListeners('end');**

**stream.removeAllListeners('error');**

**valjeanCount += (res.toString().match(/valjean/ig) || []).length;**

**}**

**console.log('count:', valjeanCount); // count: 1120**

**});**

**// 上面代码采用 Stream 模式读取《悲惨世界》的文本文件，对于每个数据块都使用stream.once方法，在data、end、error三个事件上添加一次性回调函数。变量res只有在data事件发生时才有值，然后累加每个数据块之中valjean这个词出现的次数。**

**}**

# async 函数

## // 1.含义

**{**

**// ES2017 标准引入了 async 函数，使得异步操作变得更加方便。**

**// async 函数是什么？一句话，它就是 Generator 函数的语法糖。**

**// 前文有一个 Generator 函数，依次读取两个文件。**

**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);**

**});**

**});**

**};**

**const gen = function\* () {**

**const f1 = yield readFile('/etc/fstab');**

**const f2 = yield readFile('/etc/shells');**

**console.log(f1.toString());**

**console.log(f2.toString());**

**};**

**// 写成async函数，就是下面这样。**

**const asyncReadFile = async function () {**

**const f1 = await readFile('/etc/fstab');**

**const f2 = await readFile('/etc/shells');**

**console.log(f1.toString());**

**console.log(f2.toString());**

**};**

**/\*一比较就会发现，async函数就是将 Generator 函数的星号（\*）替换成async，将yield替换成await，仅此而已。**

**async函数对 Generator 函数的改进，体现在以下四点。**

**（1）内置执行器。**

**Generator 函数的执行必须靠执行器，所以才有了co模块，而async函数自带执行器。也就是说，async函数的执行，与普通函数一模一样，只要一行。**

**asyncReadFile();**

**上面的代码调用了asyncReadFile函数，然后它就会自动执行，输出最后结果。这完全不像 Generator 函数，需要调用next方法，或者用co模块，才能真正执行，得到最后结果。**

**（2）更好的语义。**

**async和await，比起星号和yield，语义更清楚了。async表示函数里有异步操作，await表示紧跟在后面的表达式需要等待结果。**

**（3）更广的适用性。**

**co模块约定，yield命令后面只能是 Thunk 函数或 Promise 对象，而async函数的await命令后面，可以是 Promise 对象和原始类型的值（数值、字符串和布尔值，但这时等同于同步操作）。**

**（4）返回值是 Promise。**

**async函数的返回值是 Promise 对象，这比 Generator 函数的返回值是 Iterator 对象方便多了。你可以用then方法指定下一步的操作。**

**进一步说，async函数完全可以看作多个异步操作，包装成的一个 Promise 对象，而await命令就是内部then命令的语法糖。\*/**

**}**

## // 2.基本用法

**{**

**// async函数返回一个 Promise 对象，可以使用then方法添加回调函数。当函数执行的时候，一旦遇到await就会先返回，等到异步操作完成，再接着执行函数体内后面的语句。**

**// 下面是一个例子。**

**async function getStockPriceByName(name) {**

**const symbol = await getStockSymbol(name);**

**const stockPrice = await getStockPrice(symbol);**

**return stockPrice;**

**}**

**getStockPriceByName('goog').then(function (result) {**

**console.log(result);**

**});**

**// 上面代码是一个获取股票报价的函数，函数前面的async关键字，表明该函数内部有异步操作。调用该函数时，会立即返回一个Promise对象。**

**// 下面是另一个例子，指定多少毫秒后输出一个值。**

**function timeout(ms) {**

**return new Promise((resolve) => {**

**setTimeout(resolve, ms);**

**});**

**}**

**async function asyncPrint(value, ms) {**

**await timeout(ms);**

**console.log(value);**

**}**

**asyncPrint('hello world', 50);**

**// 上面代码指定 50 毫秒以后，输出hello world。**

**// 由于async函数返回的是 Promise 对象，可以作为await命令的参数。所以，上面的例子也可以写成下面的形式。**

**async function timeout(ms) {**

**await new Promise((resolve) => {**

**setTimeout(resolve, ms);**

**});**

**}**

**async function asyncPrint(value, ms) {**

**await timeout(ms);**

**console.log(value);**

**}**

**asyncPrint('hello world', 50);**

**// async 函数有多种使用形式。**

**// 函数声明**

**async function foo() {}**

**// 函数表达式**

**const foo = async function () {};**

**// 对象的方法**

**let obj = { async foo() {} };**

**obj.foo().then(...)**

**// Class 的方法**

**class Storage {**

**constructor() {**

**this.cachePromise = caches.open('avatars');**

**}**

**async getAvatar(name) {**

**const cache = await this.cachePromise;**

**return cache.match(`/avatars/${name}.jpg`);**

**}**

**}**

**const storage = new Storage();**

**storage.getAvatar('jake').then(…);**

**// 箭头函数**

**const foo = async () => {};**

**}**

## // 3.语法

**{**

**// async函数的语法规则总体上比较简单，难点是错误处理机制。**

**// 返回 Promise 对象**

**// async函数返回一个 Promise 对象。**

**// async函数内部return语句返回的值，会成为then方法回调函数的参数。**

**async function f() {**

**return 'hello world';**

**}**

**f().then(v => console.log(v))**

**// "hello world"**

**// 上面代码中，函数f内部return命令返回的值，会被then方法回调函数接收到。**

**// async函数内部抛出错误，会导致返回的 Promise 对象变为reject状态。抛出的错误对象会被catch方法回调函数接收到。**

**async function f() {**

**throw new Error('出错了');**

**}**

**f().then(**

**v => console.log(v),**

**e => console.log(e)**

**)**

**// Error: 出错了**

**// Promise 对象的状态变化**

**// async函数返回的 Promise 对象，必须等到内部所有await命令后面的 Promise 对象执行完，才会发生状态改变，除非遇到return语句或者抛出错误。也就是说，只有async函数内部的异步操作执行完，才会执行then方法指定的回调函数。**

**// 下面是一个例子。**

**async function getTitle(url) {**

**let response = await fetch(url);**

**let html = await response.text();**

**return html.match(/<title>([\s\S]+)<\/title>/i)[1];**

**}**

**getTitle('https://tc39.github.io/ecma262/').then(console.log)**

**// "ECMAScript 2017 Language Specification"**

**// 上面代码中，函数getTitle内部有三个操作：抓取网页、取出文本、匹配页面标题。只有这三个操作全部完成，才会执行then方法里面的console.log。**

**// await 命令**

**// 正常情况下，await命令后面是一个 Promise 对象。如果不是，会被转成一个立即resolve的 Promise 对象。**

**async function f() {**

**return await 123;**

**}**

**f().then(v => console.log(v))**

**// 123**

**// 上面代码中，await命令的参数是数值123，它被转成 Promise 对象，并立即resolve。**

**// await命令后面的 Promise 对象如果变为reject状态，则reject的参数会被catch方法的回调函数接收到。**

**async function f() {**

**await Promise.reject('出错了');**

**}**

**f()**

**.then(v => console.log(v))**

**.catch(e => console.log(e))**

**// 出错了**

**// 注意，上面代码中，await语句前面没有return，但是reject方法的参数依然传入了catch方法的回调函数。这里如果在await前面加上return，效果是一样的。**

**// 只要一个await语句后面的 Promise 变为reject，那么整个async函数都会中断执行。**

**async function f() {**

**await Promise.reject('出错了');**

**await Promise.resolve('hello world'); // 不会执行**

**}**

**// 上面代码中，第二个await语句是不会执行的，因为第一个await语句状态变成了reject。**

**// 有时，我们希望即使前一个异步操作失败，也不要中断后面的异步操作。这时可以将第一个await放在try...catch结构里面，这样不管这个异步操作是否成功，第二个await都会执行。**

**async function f() {**

**try {**

**await Promise.reject('出错了');**

**} catch(e) {**

**}**

**return await Promise.resolve('hello world');**

**}**

**f()**

**.then(v => console.log(v))**

**// hello world**

**// 另一种方法是await后面的 Promise 对象再跟一个catch方法，处理前面可能出现的错误。**

**async function f() {**

**await Promise.reject('出错了')**

**.catch(e => console.log(e));**

**return await Promise.resolve('hello world');**

**}**

**f()**

**.then(v => console.log(v))**

**// 出错了**

**// hello world**

**// 错误处理**

**// 如果await后面的异步操作出错，那么等同于async函数返回的 Promise 对象被reject。**

**async function f() {**

**await new Promise(function (resolve, reject) {**

**throw new Error('出错了');**

**});**

**}**

**f()**

**.then(v => console.log(v))**

**.catch(e => console.log(e))**

**// Error：出错了**

**// 上面代码中，async函数f执行后，await后面的 Promise 对象会抛出一个错误对象，导致catch方法的回调函数被调用，它的参数就是抛出的错误对象。具体的执行机制，可以参考后文的“async 函数的实现原理”。**

**// 防止出错的方法，也是将其放在try...catch代码块之中。**

**async function f() {**

**try {**

**await new Promise(function (resolve, reject) {**

**throw new Error('出错了');**

**});**

**} catch(e) {**

**}**

**return await('hello world');**

**}**

**// 如果有多个await命令，可以统一放在try...catch结构中。**

**async function main() {**

**try {**

**const val1 = await firstStep();**

**const val2 = await secondStep(val1);**

**const val3 = await thirdStep(val1, val2);**

**console.log('Final: ', val3);**

**}**

**catch (err) {**

**console.error(err);**

**}**

**}**

**// 下面的例子使用try...catch结构，实现多次重复尝试。**

**const superagent = require('superagent');**

**const NUM\_RETRIES = 3;**

**async function test() {**

**let i;**

**for (i = 0; i < NUM\_RETRIES; ++i) {**

**try {**

**await superagent.get('http://google.com/this-throws-an-error');**

**break;**

**} catch(err) {}**

**}**

**console.log(i); // 3**

**}**

**test();**

**// 上面代码中，如果await操作成功，就会使用break语句退出循环；如果失败，会被catch语句捕捉，然后进入下一轮循环。**

**// 使用注意点**

**// 第一点，前面已经说过，await命令后面的Promise对象，运行结果可能是rejected，所以最好把await命令放在try...catch代码块中。**

**async function myFunction() {**

**try {**

**await somethingThatReturnsAPromise();**

**} catch (err) {**

**console.log(err);**

**}**

**}**

**// 另一种写法**

**async function myFunction() {**

**await somethingThatReturnsAPromise()**

**.catch(function (err) {**

**console.log(err);**

**});**

**}**

**// 第二点，多个await命令后面的异步操作，如果不存在继发关系，最好让它们同时触发。**

**let foo = await getFoo();**

**let bar = await getBar();**

**// 上面代码中，getFoo和getBar是两个独立的异步操作（即互不依赖），被写成继发关系。这样比较耗时，因为只有getFoo完成以后，才会执行getBar，完全可以让它们同时触发。**

**// 写法一**

**let [foo, bar] = await Promise.all([getFoo(), getBar()]);**

**// 写法二**

**let fooPromise = getFoo();**

**let barPromise = getBar();**

**let foo = await fooPromise;**

**let bar = await barPromise;**

**// 上面两种写法，getFoo和getBar都是同时触发，这样就会缩短程序的执行时间。**

**// 第三点，await命令只能用在async函数之中，如果用在普通函数，就会报错。**

**async function dbFuc(db) {**

**let docs = [{}, {}, {}];**

**// 报错**

**docs.forEach(function (doc) {**

**await db.post(doc);**

**});**

**}**

**// 上面代码会报错，因为await用在普通函数之中了。但是，如果将forEach方法的参数改成async函数，也有问题。**

**function dbFuc(db) { //这里不需要 async**

**let docs = [{}, {}, {}];**

**// 可能得到错误结果**

**docs.forEach(async function (doc) {**

**await db.post(doc);**

**});**

**}**

**// 上面代码可能不会正常工作，原因是这时三个db.post操作将是并发执行，也就是同时执行，而不是继发执行。正确的写法是采用for循环。**

**async function dbFuc(db) {**

**let docs = [{}, {}, {}];**

**for (let doc of docs) {**

**await db.post(doc);**

**}**

**}**

**// 如果确实希望多个请求并发执行，可以使用Promise.all方法。当三个请求都会resolved时，下面两种写法效果相同。**

**async function dbFuc(db) {**

**let docs = [{}, {}, {}];**

**let promises = docs.map((doc) => db.post(doc));**

**let results = await Promise.all(promises);**

**console.log(results);**

**}**

**// 或者使用下面的写法**

**async function dbFuc(db) {**

**let docs = [{}, {}, {}];**

**let promises = docs.map((doc) => db.post(doc));**

**let results = [];**

**for (let promise of promises) {**

**results.push(await promise);**

**}**

**console.log(results);**

**}**

**// 目前，@std/esm模块加载器支持顶层await，即await命令可以不放在 async 函数里面，直接使用。**

**// async 函数的写法**

**const start = async () => {**

**const res = await fetch('google.com');**

**return res.text();**

**};**

**start().then(console.log);**

**// 顶层 await 的写法**

**const res = await fetch('google.com');**

**console.log(await res.text());**

**// 上面代码中，第二种写法的脚本必须使用@std/esm加载器，才会生效。**

**}**

## // 4.async 函数的实现原理

**{**

**// async 函数的实现原理，就是将 Generator 函数和自动执行器，包装在一个函数里。**

**async function fn(args) {**

**// ...**

**}**

**// 等同于**

**function fn(args) {**

**return spawn(function\* () {**

**// ...**

**});**

**}**

**// 所有的async函数都可以写成上面的第二种形式，其中的spawn函数就是自动执行器。**

**// 下面给出spawn函数的实现，基本就是前文自动执行器的翻版。**

**function spawn(genF) {**

**return new Promise(function(resolve, reject) {**

**const gen = genF();**

**function step(nextF) {**

**let next;**

**try {**

**next = nextF();**

**} catch(e) {**

**return reject(e);**

**}**

**if(next.done) {**

**return resolve(next.value);**

**}**

**Promise.resolve(next.value).then(function(v) {**

**step(function() { return gen.next(v); });**

**}, function(e) {**

**step(function() { return gen.throw(e); });**

**});**

**}**

**step(function() { return gen.next(undefined); });**

**});**

**}**

**}**

## // 5.与其他异步处理方法的比较

**{**

**// 我们通过一个例子，来看 async 函数与 Promise、Generator 函数的比较。**

**// 假定某个 DOM 元素上面，部署了一系列的动画，前一个动画结束，才能开始后一个。如果当中有一个动画出错，就不再往下执行，返回上一个成功执行的动画的返回值。**

**// 首先是 Promise 的写法。**

**function chainAnimationsPromise(elem, animations) {**

**// 变量ret用来保存上一个动画的返回值**

**let ret = null;**

**// 新建一个空的Promise**

**let p = Promise.resolve();**

**// 使用then方法，添加所有动画**

**for(let anim of animations) {**

**p = p.then(function(val) {**

**ret = val;**

**return anim(elem);**

**});**

**}**

**// 返回一个部署了错误捕捉机制的Promise**

**return p.catch(function(e) {**

**/\* 忽略错误，继续执行 \*/**

**}).then(function() {**

**return ret;**

**});**

**}**

**// 虽然 Promise 的写法比回调函数的写法大大改进，但是一眼看上去，代码完全都是 Promise 的 API（then、catch等等），操作本身的语义反而不容易看出来。**

**// 接着是 Generator 函数的写法。**

**function chainAnimationsGenerator(elem, animations) {**

**return spawn(function\*() {**

**let ret = null;**

**try {**

**for(let anim of animations) {**

**ret = yield anim(elem);**

**}**

**} catch(e) {**

**/\* 忽略错误，继续执行 \*/**

**}**

**return ret;**

**});**

**}**

**// 上面代码使用 Generator 函数遍历了每个动画，语义比 Promise 写法更清晰，用户定义的操作全部都出现在spawn函数的内部。这个写法的问题在于，必须有一个任务运行器，自动执行 Generator 函数，上面代码的spawn函数就是自动执行器，它返回一个 Promise 对象，而且必须保证yield语句后面的表达式，必须返回一个 Promise。**

**// 最后是 async 函数的写法。**

**async function chainAnimationsAsync(elem, animations) {**

**let ret = null;**

**try {**

**for(let anim of animations) {**

**ret = await anim(elem);**

**}**

**} catch(e) {**

**/\* 忽略错误，继续执行 \*/**

**}**

**return ret;**

**}**

**// 可以看到 Async 函数的实现最简洁，最符合语义，几乎没有语义不相关的代码。它将 Generator 写法中的自动执行器，改在语言层面提供，不暴露给用户，因此代码量最少。如果使用 Generator 写法，自动执行器需要用户自己提供。**

**}**

## // 6.实例：按顺序完成异步操作

**{**

**// 实际开发中，经常遇到一组异步操作，需要按照顺序完成。比如，依次远程读取一组 URL，然后按照读取的顺序输出结果。**

**// Promise 的写法如下。**

**function logInOrder(urls) {**

**// 远程读取所有URL**

**const textPromises = urls.map(url => {**

**return fetch(url).then(response => response.text());**

**});**

**// 按次序输出**

**textPromises.reduce((chain, textPromise) => {**

**return chain.then(() => textPromise)**

**.then(text => console.log(text));**

**}, Promise.resolve());**

**}**

**// 上面代码使用fetch方法，同时远程读取一组 URL。每个fetch操作都返回一个 Promise 对象，放入textPromises数组。然后，reduce方法依次处理每个 Promise 对象，然后使用then，将所有 Promise 对象连起来，因此就可以依次输出结果。**

**// 这种写法不太直观，可读性比较差。下面是 async 函数实现。**

**async function logInOrder(urls) {**

**for (const url of urls) {**

**const response = await fetch(url);**

**console.log(await response.text());**

**}**

**}**

**// 上面代码确实大大简化，问题是所有远程操作都是继发。只有前一个 URL 返回结果，才会去读取下一个 URL，这样做效率很差，非常浪费时间。我们需要的是并发发出远程请求。**

**async function logInOrder(urls) {**

**// 并发读取远程URL**

**const textPromises = urls.map(async url => {**

**const response = await fetch(url);**

**return response.text();**

**});**

**// 按次序输出**

**for (const textPromise of textPromises) {**

**console.log(await textPromise);**

**}**

**}**

**// 上面代码中，虽然map方法的参数是async函数，但它是并发执行的，因为只有async函数内部是继发执行，外部不受影响。后面的for..of循环内部使用了await，因此实现了按顺序输出。**

**}**

## // 7.异步遍历器

**{**

**/\*遍历器》一章说过，Iterator 接口是一种数据遍历的协议，只要调用遍历器对象的next方法，就会得到一个对象，表示当前遍历指针所在的那个位置的信息。next方法返回的对象的结构是{value, done}，其中value表示当前的数据的值，done是一个布尔值，表示遍历是否结束。**

**这里隐含着一个规定，next方法必须是同步的，只要调用就必须立刻返回值。也就是说，一旦执行next方法，就必须同步地得到value和done这两个属性。如果遍历指针正好指向同步操作，当然没有问题，但对于异步操作，就不太合适了。目前的解决方法是，Generator 函数里面的异步操作，返回一个 Thunk 函数或者 Promise 对象，即value属性是一个 Thunk 函数或者 Promise 对象，等待以后返回真正的值，而done属性则还是同步产生的。**

**ES2018 引入了”异步遍历器“（Async Iterator），为异步操作提供原生的遍历器接口，即value和done这两个属性都是异步产生。**

**异步遍历的接口**

**异步遍历器的最大的语法特点，就是调用遍历器的next方法，返回的是一个 Promise 对象。\*/**

**asyncIterator**

**.next()**

**.then(**

**({ value, done }) => /\* ... \*/**

**);**

**/\*上面代码中，asyncIterator是一个异步遍历器，调用next方法以后，返回一个 Promise 对象。因此，可以使用then方法指定，这个 Promise 对象的状态变为resolve以后的回调函数。回调函数的参数，则是一个具有value和done两个属性的对象，这个跟同步遍历器是一样的。**

**我们知道，一个对象的同步遍历器的接口，部署在Symbol.iterator属性上面。同样地，对象的异步遍历器接口，部署在Symbol.asyncIterator属性上面。不管是什么样的对象，只要它的Symbol.asyncIterator属性有值，就表示应该对它进行异步遍历。**

**下面是一个异步遍历器的例子。\*/**

**const asyncIterable = createAsyncIterable(['a', 'b']);**

**const asyncIterator = asyncIterable[Symbol.asyncIterator]();**

**asyncIterator**

**.next()**

**.then(iterResult1 => {**

**console.log(iterResult1); // { value: 'a', done: false }**

**return asyncIterator.next();**

**})**

**.then(iterResult2 => {**

**console.log(iterResult2); // { value: 'b', done: false }**

**return asyncIterator.next();**

**})**

**.then(iterResult3 => {**

**console.log(iterResult3); // { value: undefined, done: true }**

**});**

**// 上面代码中，异步遍历器其实返回了两次值。第一次调用的时候，返回一个 Promise 对象；等到 Promise 对象resolve了，再返回一个表示当前数据成员信息的对象。这就是说，异步遍历器与同步遍历器最终行为是一致的，只是会先返回 Promise 对象，作为中介。**

**// 由于异步遍历器的next方法，返回的是一个 Promise 对象。因此，可以把它放在await命令后面。**

**async function f() {**

**const asyncIterable = createAsyncIterable(['a', 'b']);**

**const asyncIterator = asyncIterable[Symbol.asyncIterator]();**

**console.log(await asyncIterator.next());**

**// { value: 'a', done: false }**

**console.log(await asyncIterator.next());**

**// { value: 'b', done: false }**

**console.log(await asyncIterator.next());**

**// { value: undefined, done: true }**

**}**

**// 上面代码中，next方法用await处理以后，就不必使用then方法了。整个流程已经很接近同步处理了。**

**// 注意，异步遍历器的next方法是可以连续调用的，不必等到上一步产生的 Promise 对象resolve以后再调用。这种情况下，next方法会累积起来，自动按照每一步的顺序运行下去。下面是一个例子，把所有的next方法放在Promise.all方法里面。**

**const asyncGenObj = createAsyncIterable(['a', 'b']);**

**const [{value: v1}, {value: v2}] = await Promise.all([**

**asyncGenObj.next(), asyncGenObj.next()**

**]);**

**console.log(v1, v2); // a b**

**// 另一种用法是一次性调用所有的next方法，然后await最后一步操作。**

**async function runner() {**

**const writer = openFile('someFile.txt');**

**writer.next('hello');**

**writer.next('world');**

**await writer.return();**

**}**

**runner();**

**// for await...of**

**// 前面介绍过，for...of循环用于遍历同步的 Iterator 接口。新引入的for await...of循环，则是用于遍历异步的 Iterator 接口。**

**async function f() {**

**for await (const x of createAsyncIterable(['a', 'b'])) {**

**console.log(x);**

**}**

**}**

**// a**

**// b**

**// 上面代码中，createAsyncIterable()返回一个拥有异步遍历器接口的对象，for...of循环自动调用这个对象的异步遍历器的next方法，会得到一个 Promise 对象。await用来处理这个 Promise 对象，一旦resolve，就把得到的值（x）传入for...of的循环体。**

**// for await...of循环的一个用途，是部署了 asyncIterable 操作的异步接口，可以直接放入这个循环。**

**let body = '';**

**async function f() {**

**for await(const data of req) body += data;**

**const parsed = JSON.parse(body);**

**console.log('got', parsed);**

**}**

**// 上面代码中，req是一个 asyncIterable 对象，用来异步读取数据。可以看到，使用for await...of循环以后，代码会非常简洁。**

**// 如果next方法返回的 Promise 对象被reject，for await...of就会报错，要用try...catch捕捉。**

**async function () {**

**try {**

**for await (const x of createRejectingIterable()) {**

**console.log(x);**

**}**

**} catch (e) {**

**console.error(e);**

**}**

**}**

**// 注意，for await...of循环也可以用于同步遍历器。**

**(async function () {**

**for await (const x of ['a', 'b']) {**

**console.log(x);**

**}**

**})();**

**// a**

**// b**

**// Node v10 支持异步遍历器，Stream 就部署了这个接口。下面是读取文件的传统写法与异步遍历器写法的差异。**

**// 传统写法**

**function main(inputFilePath) {**

**const readStream = fs.createReadStream(**

**inputFilePath,**

**{ encoding: 'utf8', highWaterMark: 1024 }**

**);**

**readStream.on('data', (chunk) => {**

**console.log('>>> '+chunk);**

**});**

**readStream.on('end', () => {**

**console.log('### DONE ###');**

**});**

**}**

**// 异步遍历器写法**

**async function main(inputFilePath) {**

**const readStream = fs.createReadStream(**

**inputFilePath,**

**{ encoding: 'utf8', highWaterMark: 1024 }**

**);**

**for await (const chunk of readStream) {**

**console.log('>>> '+chunk);**

**}**

**console.log('### DONE ###');**

**}**

**// 异步 Generator 函数**

**// 就像 Generator 函数返回一个同步遍历器对象一样，异步 Generator 函数的作用，是返回一个异步遍历器对象。**

**// 在语法上，异步 Generator 函数就是async函数与 Generator 函数的结合。**

**async function\* gen() {**

**yield 'hello';**

**}**

**const genObj = gen();**

**genObj.next().then(x => console.log(x));**

**// { value: 'hello', done: false }**

**// 上面代码中，gen是一个异步 Generator 函数，执行后返回一个异步 Iterator 对象。对该对象调用next方法，返回一个 Promise 对象。**

**// 异步遍历器的设计目的之一，就是 Generator 函数处理同步操作和异步操作时，能够使用同一套接口。**

**// 同步 Generator 函数**

**function\* map(iterable, func) {**

**const iter = iterable[Symbol.iterator]();**

**while (true) {**

**const {value, done} = iter.next();**

**if (done) break;**

**yield func(value);**

**}**

**}**

**// 异步 Generator 函数**

**async function\* map(iterable, func) {**

**const iter = iterable[Symbol.asyncIterator]();**

**while (true) {**

**const {value, done} = await iter.next();**

**if (done) break;**

**yield func(value);**

**}**

**}**

**// 上面代码中，map是一个 Generator 函数，第一个参数是可遍历对象iterable，第二个参数是一个回调函数func。map的作用是将iterable每一步返回的值，使用func进行处理。上面有两个版本的map，前一个处理同步遍历器，后一个处理异步遍历器，可以看到两个版本的写法基本上是一致的。**

**// 下面是另一个异步 Generator 函数的例子。**

**async function\* readLines(path) {**

**let file = await fileOpen(path);**

**try {**

**while (!file.EOF) {**

**yield await file.readLine();**

**}**

**} finally {**

**await file.close();**

**}**

**}**

**// 上面代码中，异步操作前面使用await关键字标明，即await后面的操作，应该返回 Promise 对象。凡是使用yield关键字的地方，就是next方法停下来的地方，它后面的表达式的值（即await file.readLine()的值），会作为next()返回对象的value属性，这一点是与同步 Generator 函数一致的。**

**// 异步 Generator 函数内部，能够同时使用await和yield命令。可以这样理解，await命令用于将外部操作产生的值输入函数内部，yield命令用于将函数内部的值输出。**

**// 上面代码定义的异步 Generator 函数的用法如下。**

**(async function () {**

**for await (const line of readLines(filePath)) {**

**console.log(line);**

**}**

**})()**

**// 异步 Generator 函数可以与for await...of循环结合起来使用。**

**async function\* prefixLines(asyncIterable) {**

**for await (const line of asyncIterable) {**

**yield '> ' + line;**

**}**

**}**

**// 异步 Generator 函数的返回值是一个异步 Iterator，即每次调用它的next方法，会返回一个 Promise 对象，也就是说，跟在yield命令后面的，应该是一个 Promise 对象。如果像上面那个例子那样，yield命令后面是一个字符串，会被自动包装成一个 Promise 对象。**

**function fetchRandom() {**

**const url = 'https://www.random.org/decimal-fractions/'**

**+ '?num=1&dec=10&col=1&format=plain&rnd=new';**

**return fetch(url);**

**}**

**async function\* asyncGenerator() {**

**console.log('Start');**

**const result = await fetchRandom(); // (A)**

**yield 'Result: ' + await result.text(); // (B)**

**console.log('Done');**

**}**

**const ag = asyncGenerator();**

**ag.next().then(({value, done}) => {**

**console.log(value);**

**})**

**// 上面代码中，ag是asyncGenerator函数返回的异步遍历器对象。调用ag.next()以后，上面代码的执行顺序如下。**

**/\*ag.next()立刻返回一个 Promise 对象。**

**asyncGenerator函数开始执行，打印出Start。**

**await命令返回一个 Promise 对象，asyncGenerator函数停在这里。**

**A 处变成 fulfilled 状态，产生的值放入result变量，asyncGenerator函数继续往下执行。**

**函数在 B 处的yield暂停执行，一旦yield命令取到值，ag.next()返回的那个 Promise 对象变成 fulfilled 状态。**

**ag.next()后面的then方法指定的回调函数开始执行。该回调函数的参数是一个对象{value, done}，其中value的值是yield命令后面的那个表达式的值，done的值是false。**

**A 和 B 两行的作用类似于下面的代码。\*/**

**return new Promise((resolve, reject) => {**

**fetchRandom()**

**.then(result => result.text())**

**.then(result => {**

**resolve({**

**value: 'Result: ' + result,**

**done: false,**

**});**

**});**

**});**

**// 如果异步 Generator 函数抛出错误，会导致 Promise 对象的状态变为reject，然后抛出的错误被catch方法捕获。**

**async function\* asyncGenerator() {**

**throw new Error('Problem!');**

**}**

**asyncGenerator()**

**.next()**

**.catch(err => console.log(err)); // Error: Problem!**

**// 注意，普通的 async 函数返回的是一个 Promise 对象，而异步 Generator 函数返回的是一个异步 Iterator 对象。可以这样理解，async 函数和异步 Generator 函数，是封装异步操作的两种方法，都用来达到同一种目的。区别在于，前者自带执行器，后者通过for await...of执行，或者自己编写执行器。下面就是一个异步 Generator 函数的执行器。**

**async function takeAsync(asyncIterable, count = Infinity) {**

**const result = [];**

**const iterator = asyncIterable[Symbol.asyncIterator]();**

**while (result.length < count) {**

**const {value, done} = await iterator.next();**

**if (done) break;**

**result.push(value);**

**}**

**return result;**

**}**

**// 上面代码中，异步 Generator 函数产生的异步遍历器，会通过while循环自动执行，每当await iterator.next()完成，就会进入下一轮循环。一旦done属性变为true，就会跳出循环，异步遍历器执行结束。**

**// 下面是这个自动执行器的一个使用实例。**

**async function f() {**

**async function\* gen() {**

**yield 'a';**

**yield 'b';**

**yield 'c';**

**}**

**return await takeAsync(gen());**

**}**

**f().then(function (result) {**

**console.log(result); // ['a', 'b', 'c']**

**})**

**// 异步 Generator 函数出现以后，JavaScript 就有了四种函数形式：普通函数、async 函数、Generator 函数和异步 Generator 函数。请注意区分每种函数的不同之处。基本上，如果是一系列按照顺序执行的异步操作（比如读取文件，然后写入新内容，再存入硬盘），可以使用 async 函数；如果是一系列产生相同数据结构的异步操作（比如一行一行读取文件），可以使用异步 Generator 函数。**

**// 异步 Generator 函数也可以通过next方法的参数，接收外部传入的数据。**

**const writer = openFile('someFile.txt');**

**writer.next('hello'); // 立即执行**

**writer.next('world'); // 立即执行**

**await writer.return(); // 等待写入结束**

**// 上面代码中，openFile是一个异步 Generator 函数。next方法的参数，向该函数内部的操作传入数据。每次next方法都是同步执行的，最后的await命令用于等待整个写入操作结束。**

**// 最后，同步的数据结构，也可以使用异步 Generator 函数。**

**async function\* createAsyncIterable(syncIterable) {**

**for (const elem of syncIterable) {**

**yield elem;**

**}**

**}**

**// 上面代码中，由于没有异步操作，所以也就没有使用await关键字。**

**// yield\* 语句**

**// yield\*语句也可以跟一个异步遍历器。**

**async function\* gen1() {**

**yield 'a';**

**yield 'b';**

**return 2;**

**}**

**async function\* gen2() {**

**// result 最终会等于 2**

**const result = yield\* gen1();**

**}**

**// 上面代码中，gen2函数里面的result变量，最后的值是2。**

**// 与同步 Generator 函数一样，for await...of循环会展开yield\*。**

**(async function () {**

**for await (const x of gen2()) {**

**console.log(x);**

**}**

**})();**

**// a**

**// b**

**}**