# Let关键字与const关键字

[ES6 入门教程 - ECMAScript 6入门 (ruanyifeng.com)](https://es6.ruanyifeng.com/)

这个人讲的很清楚，流批

## let关键字

1.可以重复声明 var a=10;var a=11;**可以**。let a=10,let a=11;**报错**

2.let拥有块级作用域，作用范围仅限于声明的那个代码块也就是{}之中，内外层可定义相同名称的变量，互不冲突。

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

// IIFE 写法

(function () {

var tmp = ...;

...

}());

// 块级作用域写法

{

let tmp = ...;

...

}

3.不存在变量提升（实际上应该是有的，但是因为暂时性死区显得没有变量提升，变量提升就是在代码编译前提前去收集变量函数等等，例如我在var a=10**之前**输出a并不会报错，输出的是undefined，而这样在let声明的变量中行不通）

4。并不影响作用域链，我在函数外let声明a变量，函数体没有声明，函数体使用a变量时会向外寻找a变量，并不会拘于函数作用域

5.暂时性死区，在一个作用域如果有let声明了该变量，该变量就会被“绑定”在这个区域，在该作用域，在声明语句前任何使用改变量都会报错，这一段区域称为暂时性死区（TDZ）。

## const关键字

const和let特性几乎一样，唯一的区别就是const声明的变量无法修改，也就是说在声明开始就必须赋值，像const a;这种就是绝对错误的

//在es5中只有var 和function两种方式声明变量，而在es6中新增了let和const方式，之后还有import和class命令

# 变量的解构赋值

ES6 允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构（Destructuring）

## 数组解构赋值基本用法

let [foo,a,b]=[1,2,3]

foo //1

a //2

b //3

如果个数多了，可以匹配（称为不完全解构），赋值的数少了匹配不上，结果为undefined

赋值前可以有一个默认值

let[a=2,b]=[1] //a = 1 b = undefined a有值可以赋，默认值取消，b没有，undefined

如果变量默认值等于的是个表达式，那么将惰性求值（用到的时候才回去执行）

function f() {

console.log('aaa');

}

let [x = f()] = [1];

上面代码中，因为x能取到值，所以函数f根本不会执行。

//[0][1]===0 [1][0]===1

默认值还可以是解构赋值的数，前提是必须提前定义

let [a=2,b=a]=[1]; //a=1 b=1;

let[a=b,b=1]; // b没有定义就赋给a，直接报错

## 对象解构赋值基本用法

对象的解构赋值和数组的解构赋值不一样，数组的只需要次序对上即可，而对象的解构需要名字对上才可以

let{foo}={a:1,b:2} //foo为undefined

let{foo,a}={a:100,c=200} //foo为undefined,a为100

利用这点可以把很多已经存在的方法赋值给变量

例如let{sin,cos}=Math;

然后可以直接sin（xxx）这么写了

匹配模式：let {p：foo}={p：123} //foo=123；

这里的P:是一个匹配模式，真正被赋了值的是foo

所以有以下代码，请仔细研究，虽然这看起来有点怪

const node = {

loc: {

start: {

line: 1,

column: 5

}

}

};

let { loc, loc: { start }, loc: { start: { line }} } = node;

第一个loc，node中有和他同样名称的属性，所以loc等于{start:{…}}这一大串，loc是一个对象，其中有一个属性为strat，这个属性的值是一个对象

第二个，（loc逗号之后）loc:是匹配模式，之后，重点来了，左边loc:之后的所有东西等于右边loc:之后的所有东西，也就是说{start:{…}}={start}，这里不是对象赋值给对象，而又是一个解构赋值，类似于let {foo}={foo:123}所以foo=123start={…}，所以整个start是一个对象，他有两个属性，line和colum

第三个，这里主要是匹配模式，loc:和loc:匹配之后，根据上文的“左边loc:之后的所有东西等于右边loc:之后的所有东西”所以{start:{…}}={start:{…}}，所以这里又是一个模式匹配，化简之后便是{line:1,colum:5}={line},然后根据最基本对象解构赋值，line=1；和之前的两个不同的是line是一个变量，值为1

对象解构赋值也有默认值，和数组解构赋值一样，只有严格===undefined时，默认值才不会被取消，第二个为null，不等于undefined，所以默认值失效

var {x = 3} = {x: undefined};x // 3

var {x = 3} = {x: null}; x // null

注意点

1:使用已经声明了的变量时要小心

// 错误的写法

let x;

{x} = {x: 1};

//正确写法

let {x}={x:1};

OR

let x;

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

这里js会把{x}理解为代码块而不是对象,要么用小括号，要么不要这么写

2:即使是空对象（无任何变量名）也可以被赋值，即使没什么意义

({}={x:1}) ({}={b:2}) ({}=[a:2,b:3])

3:数组的本质也是一个特殊的对象，那么数组也可以被拿来对象解构赋值

let a=[1,2,3]

let{0:first,2:last}=a; //first=1 last=3;

## 字符串解构赋值

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。

而数字和布尔值可以转换为对象，因此可以进行解构赋值，例如把他们的tostring方法赋给s(我也不知道有什么用)

let {toString: s} = 123;

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

let {toString: s} = true;

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

## 字符串模板

字符串模板

以往js写字符串都是使用的是”” ‘’现在ES6有了一种全新的声明方式``（反引号，esc下面那个点）

``和‘’ “”不同的是，``里可以包含换行，

模板字符串里

${}里可以写任意js表达式，也可以写对象的属性，当然也包括函数

标签模板

例如alert`Hello`

``之中的东西就是前面那个函数的参数，如果里面含有变量的的时候，将会进行解析，变量单独解析出来，看个例子

let a = 5;

let b = 10;

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

// 等同于

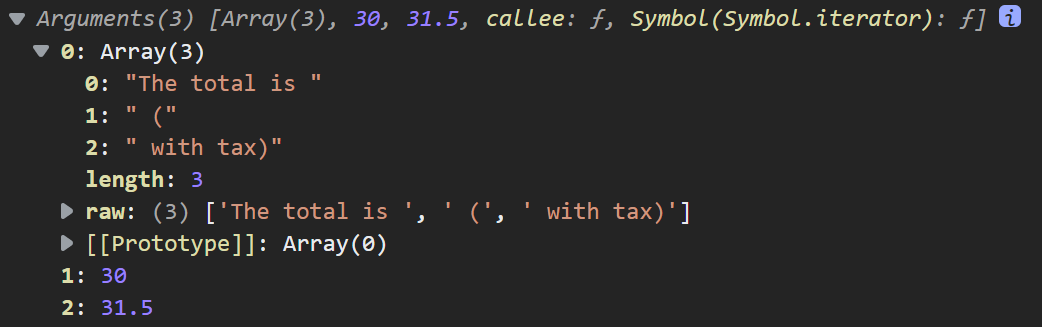
tag(['Hello ', ' world ', ''], 15, 50);

**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。**

关于数组的第三个是空的问题我有一个这样的理解，在字符串与字符串之中如果有一个${}这样的表达式，将会把两旁的字符串变成数组的两个成员，${}为分隔标志，以此类推。而当${}连续出现时，就有${}连续出现的次数-1个 ’ ‘ 算到数组之中，这些’’在数组中的顺序和他们出现的顺序一样，并不会因为是’ ’而全部排到数组后面，但是${}就不一样了，arguments数组怎么排它还是一个数组，${}要按出现顺序排（arguments数组）配合下图更好理解）

最后是${}里的如果解析出来是一个变量就按照顺序排好

还是这个例子，假如tag的参数列表只有一个形参来接收，那么多余的部分（15，50这些）将会被舍弃，但是实际上传过去的参数arguments里不止一个，解析完成之后的的参数列表将会放在arguments里传过去，但是函数你收不收就是另外一回事了



## 字符串新增实例方法

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。都是返回布尔值

**includes()是否包含 startsWith（）是否由参数开始 endsWith（）是否由参数结束**

let s = 'Hello world!';

s.startsWith('Hello') // true

s.endsWith('!') // true

s.includes('o') // true

**repeat（）实例方法**

这个方法会返回一个新的被重复多次的字符串

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

参数会被取整，-1到0之间的负小数会被视作零，若是小于-1的复数或infinity会报错

若参数为字符串则会被转为数字，‘3‘会被转为3，其他转不成数字的则会返回‘ ’（空的）

**padStart（），padEnd（）实例方法**

ES2017这些方法用于补全字符串，他们接受两个参数，参数的意义如下

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

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

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

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

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

// '0123456abc'多余的会被截取不要

'x'.padStart(4) // ' x'不写要补全的字符参数默认空格

'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"

**trim（），trimStart（），trimEnd（）实例方法**

ES2019他们都是用来消除空格（包括tab符，换行）的方法，trim（）是消除整个字符串的空格，trimStart（）是消除字符串前的，后面不管，trimEnd（）同理

他们都是返回一个结果，并不会修改原字符串

浏览器还有两个方法和trimStart（），trimEnd（）一样，叫trimLeft（）与trimRight（）

const s = ' abc ';

s.trim() // "abc"

s.trimStart() // "abc "

s.trimEnd() // " abc"

**replaceAll（）实例方法**

ES2021（ES12）传统的replace（）方法只能替换第一个匹配到的，若要全部替换则需要正则表达式，而replaceAll（）方法可以全部替换

replaceAll（）不会改变原字符串，只返回一个新的字符串

但是如果给replaceAll（）使用正则表达式，如果不带匹配模式则会报错

// 不报错

'aabbcc'.replace(/b/, '\_')

// 报错，没带匹配模式

'aabbcc'.replaceAll(/b/, '\_')

replaceAll（）的第二个参数还可以为函数，替换的值为这个函数的返回值

replaceAll()的第二个参数replacement是一个字符串，表示替换的文本，其中可以使用一些特殊字符串。

$&：匹配的字符串。

$` ：匹配结果前面的文本。

$'：匹配结果后面的文本。

$n：匹配成功的第n组内容，n是从1开始的自然数。这个参数生效的前提是，第一个参数必须是正则表达式。

$$：指代美元符号$。

下面是一些例子。

// $& 表示匹配的字符串，即`b`本身

// 所以返回结果与原字符串一致

'abbc'.replaceAll('b', '$&')

// 'abbc'

// $` 表示匹配结果之前的字符串

// 对于第一个`b`，$` 指代`a`

// 对于第二个`b`，$` 指代`ab`（这里不会匹配到就立即替换的）

'abbc'.replaceAll('b', '$`')

// 'aaabc'

// $' 表示匹配结果之后的字符串

// 对于第一个`b`，$' 指代`bc`

// 对于第二个`b`，$' 指代`c`

'abbc'.replaceAll('b', `$'`)

// 'abccc'

// $1 表示正则表达式的第一个组匹配，指代`ab`

// $2 表示正则表达式的第二个组匹配，指代`bc`

'abbc'.replaceAll(/(ab)(bc)/g, '$2$1')

// 'bcab'

// $$ 指代 $

'abc'.replaceAll('b', '$$')

// 'a$c'

**at（）实例方法**

at()方法接受一个整数作为参数，返回参数指定位置的字符，支持负索引（即倒数的位置）。

const str = 'hello';

str.at(1) // "e"

str.at(-1) // "o"

如果参数位置超出了字符串范围，at()返回undefined。

## 正则表达式的扩展

在ES5中，构建正则表达式两种方法

1. var regex = new RegExp (‘xyz’,’i’)

2. var regex = /xyz/i

这两种写法都是合法的，但是如果将两种写法混合起来则会出现错误，例如

var regex = new RegExp (/xyz/,’i’) //Error

var regex = new RegExp (‘/xyz/’,’i’) //这样写匹配的东西就是/xyz/了 ：）

但是在ES6中这是合法的，例如

var regex = new RegExp(/xyz/ig,’i’)

这样不会报错，而且匹配模式将会被后面的替换

**字符串的四个正则方法**

match replace split search

ES6 将这 4 个方法，在语言内部全部调用RegExp的实例方法，从而做到所有与正则相关的方法，全都定义在RegExp对象上。

String.prototype.match 调用 RegExp.prototype[Symbol.match]

String.prototype.replace 调用 RegExp.prototype[Symbol.replace]

String.prototype.search 调用 RegExp.prototype[Symbol.search]

String.prototype.split 调用 RegExp.prototype[Symbol.split]

**u修饰符**

在ES5中，无法识别超过0xFFFF的字符，如果超过了，会把超过的那部分截取下来当作第二个字符接在后面

在ES6中有了u修饰符后解决了这个问题，例如

var s = '𠮷';

/^.$/.test(s) // false

/^.$/u.test(s) // true

（这里解释一下上面的代码，/^表示首字符匹配， $表示末字符匹配， . 代表除了换行字符任意字符，结合起来就是，任意一个字符开头结尾，通俗的讲就是判断这个是不是任意一个字符，第一个例子没有加\u所以.点字符识别不了UTF-16的超过0xFFFF的字符，变为两个字符了，当然和一个字符的正则表达式匹配不上，而第二个加上了\u所以可以识别出来这是一个字符，结果当然就是true了

这种写法并没有多余，如果不是这么写就无法分出区别了，因为如果是两个字符，通过“ . ”都是true，这样写是为了锁定只有一个任意的数）

**配合{}表示unicode字符**

例如

/\u{61}/.test('a') // false

/\u{61}/u.test('a') // true

第二个可以识别出来这个是‘a’，第一个直接无法识别，匹配结果为false会认为是61个‘u’

（{123}这里面的数字默认是重复的次数，例如a{2}表示两个a）

**\u影响预定义模式**

|  |  |
| --- | --- |
| 字符 | 描述 |
| \cx | 匹配由x指明的控制字符。例如， \cM 匹配一个 Control-M 或回车符。  x 的值必须为 A-Z 或 a-z 之一。否则，将 c 视为一个原义的 'c' 字符。 |
| \f | 匹配一个换页符。等价于 \x0c 和 \cL。 |
| \n | 匹配一个换行符。等价于 \x0a 和 \cJ。 |
| \r | 匹配一个回车符。等价于 \x0d 和 \cM。 |
| \s | 匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [ \f\n\r\t\v]。注意 Unicode 正则表达式会匹配全角空格符。 |
| \S | 匹配任何非空白字符。等价于 [^ \f\n\r\t\v]。 |
| \t | 匹配一个制表符。等价于 \x09 和 \cI。 |
| \v | 匹配一个垂直制表符。等价于 \x0b 和 \cK。 |

所谓预定义模式也就是/d /s /S这种以及定义好的匹配模式也称之为元字符

**元字符，即用来传达非自身含义的字符（这个解释准确一点）**

/^\S$/.test('𠮷') // false，没有\u无法识别超过0xFFFF的字符

/^\S$/u.test('𠮷') // true，可以识别

**i修饰符的特殊情况**

在一些情况下，有些字形相同但是字符编码不同，例如大写K

/[a-z]/i.test('\u212A') // false

/[a-z]/iu.test('\u212A') // true非规范的大写K，需要加上u,/i是不够的，所以/iu组合起来才能识别

/[a-z]/i.test('\u004B') // true规范的大写K

**转义**

没有u修饰符的情况下，正则中没有定义的转义（如逗号的转义\,）无效，而在u模式会报错。

/\,/ // /\,/

/\,/u // 报错

上面代码中，没有u修饰符时，逗号前面的反斜杠是无效的，加了u修饰符就报错。

**y 修饰符**

除了u修饰符，ES6 还为正则表达式添加了y修饰符，叫做“粘连”（sticky）修饰符。

y修饰符的作用与g修饰符类似，也是全局匹配，后一次匹配都从上一次匹配成功的下一个位置开始。不同之处在于，g修饰符只要剩余位置中存在匹配就可，而y修饰符确保匹配必须从剩余的第一个位置开始，这也就是“粘连”的涵义。

var s = 'aaa\_aa\_a';

var r1 = /a+/g;

var r2 = /a+/y;

r1.exec(s) // ["aaa"] r2.exec(s) // ["aaa"]

r1.exec(s) // ["aa"] r2.exec(s) // null

这里要解释一下exec这个方法，这个是用来匹配字符串的，例如

var reg = new RegExp（/a+/）

var str=‘aaa’

那么reg.exec(str)就为’aaa’

和test方法不同的是，当正则表达式是全局正则表达式时，第一次匹配成功时，位置会保留并结束这次匹配，第二次从那位置继续，/g是剩下的有匹配就像,/y是第二次开始那个位置就必须匹配，否则返回false，详情看这篇文章

（详见[JavaScript exec() 方法 (w3school.com.cn)](https://www.w3school.com.cn/jsref/jsref_exec_regexp.asp)）

**m修饰符**

[JS正则表达式修饰符中multiline(/m)用法分析\_javascript技巧\_脚本之家 (jb51.net)](https://www.jb51.net/article/101399.htm)

简单来说就是正则表达式中要有^ $ /n或者/r这三者，/m才有效果，这个时候如果加上了/m，那么“^”与字符串开始位置以及“\n”或“\r”之后的位置相匹配，而“$”与字符串结束位置以及“\n”或“\r”之前的位置相匹配。

**具名组匹配**

正则表达式使用圆括号进行组匹配。

const RE\_DATE = /(\d{4})-(\d{2})-(\d{2})/;

上面代码中，正则表达式里面有三组圆括号。使用exec方法，就可以将这三组匹配结果提取出来。

const RE\_DATE = /(\d{4})-(\d{2})-(\d{2})/;

const matchObj = RE\_DATE.exec('1999-12-31');

const year = matchObj[1]; // 1999-12-31

const year = matchObj[1]; // 1999

const month = matchObj[2]; // 12

const day = matchObj[3]; // 31

组匹配的一个问题是，每一组的匹配含义不容易看出来，而且只能用数字序号（比如matchObj[1]）引用，要是组的顺序变了，引用的时候就必须修改序号。

ES2018 引入了[具名组匹配](https://github.com/tc39/proposal-regexp-named-groups)（Named Capture Groups），允许为每一个组匹配指定一个名字，既便于阅读代码，又便于引用。

const RE\_DATE = /(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/;

const matchObj = RE\_DATE.exec('1999-12-31');

const year = matchObj.groups.year; // "1999"

const month = matchObj.groups.month; // "12"

const day = matchObj.groups.day; // "31"

上面代码中，“具名组匹配”在圆括号内部，模式的头部添加“问号 + 尖括号 + 组名+ 尖括号”（?<year>），然后就可以在exec方法返回结果的groups属性上引用该组名。同时，数字序号（matchObj[1]）依然有效。

**RegExp.prototype.unicode 属性**

正则实例对象新增unicode属性，表示是否设置了u修饰符。

const r1 = /hello/;

const r2 = /hello/u;

r1.unicode // false

r2.unicode // true

上面代码中，正则表达式是否设置了u修饰符，可以从unicode属性看出来。

**RegExp.prototype.sticky 属性**

与y修饰符相匹配，ES6 的正则实例对象多了sticky属性，表示是否设置了y修饰符。

var r = /hello\d/y;

r.sticky // true

**RegExp.prototype.flags 属性**

ES6 为正则表达式新增了flags属性，会返回正则表达式的修饰符。

// ES5 的 source 属性

// 返回正则表达式的正文

/abc/ig.source

// "abc"

// ES6 的 flags 属性

// 返回正则表达式的修饰符

/abc/ig.flags

// 'gi'

**先行断言、先行否定断言、后行断言、后行否定断言**

感觉不是很重要，暂略

**Unicode属性类**

这个可以用来匹配一些特殊字符，比如数希腊字母，十进制字符，罗马数字

感觉不重要，需要时再查

# 数值扩展

## 二进制和八进制表示

二进制:0b或者0B

八进制:0o或者0O

## 数值分隔符合

因为欧美国家的书写习惯，可以在数值中使用分隔符，例如1200写成1,200

或者1\_200

// 二进制

0b1010\_0001\_1000\_0101

// 十六进制

0xA0\_B0\_C0

这时分隔符显着格外有用

关于语法，分隔符

不可以写在0b或者0o这些符号的前后

不可写在小数点前后

不可写在数的前后

不可连续写

分隔符不影响存储和使用

但是在一些函数中，不支持使用分隔符，因为分隔符只是为了看起来直观，书写方便，并不是为了参与运算（处理外部输入数据）

## Number.isFinite(),与Number.isNaN()

第一个是用来判断一个数是不是有限即判断是不是非无限（infinity）

Number.isFinite(5) //true

Number.isFinite(infinity) //false

若参数不是数值类型一律返回false

第二个是用来判断一个数是不是NaN

如果参数不是NaN，一律返回false

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

和ES5用法一模一样，只是从全局移植到了Number之下

## Number.isInteger()

判断参数是否是整数，如果参数不是数值类型，返回false

如果对精度有很高的要求的话最好不要用这个来判断，在以下情况下会误判

Number.isInteger(3.0000000000000002) // true达到了小数点16位，视为0

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

Number.isInteger(5E-325) // true 太小了，视为0

## Number.EPSILON

他表示一个很小的常量，官方定义是，大于1的最小的浮点数，在js允许的范围内，这个数是2\*10^-52次方

Number.EPSILON

// 2.220446049250313e-16

若一个数小于他，则认为没有意义

这个数的意义，我暂时也没觉得有什么意义，阮一峰这么写的



## 安全整数与Number.isSafeInteger()

JavaScript能表示的数在-2^53到2^53之间，不包括端点，ES6引入了俩数用来表示这个端点，他们分别是Number.MAX\_SAFE\_INTEGER和Number.MIN\_SAFE\_INTEGER

如果超过这个范围则无法正确的使用,例如

9007199254740993 === 9007199254740992 //true

9007199254740994 === 9007199254740992 //false

Number.IsSafeInteger()就是用来判断一个是不是在这个范围内的，如果参数不是数值类型的话依然返回的是false

注意如果里面写的是表达式，那么表达式里的数也要是一个安全的整数，例如

9007199254740993 - 990

// 返回结果 9007199254740002

// 正确答案应该是 9007199254740003

## Math对象扩展

ES6在Math对象上新增了17个数学相关方法，他们都是静态方法，只能在Math对象上调用

Math.trunc方法用于去除一个数的小数部分，返回整数部分。

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。

Math.cbrt()方法用于计算一个数的立方根。

Math.clz32()方法将参数转为 32 位无符号整数的形式，然后返回这个 32 位值里面有多少个前导 0。

Math.imul方法返回两个数以 32 位带符号整数形式相乘的结果，返回的也是一个 32 位的带符号整数。

Math.fround方法返回一个数的32位单精度浮点数形式。

Math.hypot方法返回所有参数的平方和的平方根。

还有一些对数，双曲线的就不做了解了

## BigInt数据类型

ES5有五种基本数据类型，他们分别是 到ES6又多了两种基本数据类型

布尔Boolean BigInt

数值Number Symbol

Undefined

Null

字符串String

一种引用数据类型Object

这里讲述BigInt

BigInt是为了解决Js无法储存大于十进制16六位，二进制53位的问题，BigInt是用来存储整数的，无论多大都可以并且可以保持精度，这一点在普通的数值类型中不可以

这种数据类型的写法要在数字的后面加上n来区分

42!=42n

42==42n //值相同true

42===42n //数据类型不同false

他们只是值相同，数据类型并不同

## BigInt函数

和Number一样，BigInt也有自己的函数，例如转为BigInt的函数

BigInt(‘123’) 123n

BigInt(123) 123n

BigInt(true) 1n

BigInt(false) 0n

传其他的参数都会报错，像‘123n’这种也会报错

// Number.parseInt() 与 BigInt.parseInt() 的对比

Number.parseInt('9007199254740993', 10)

// 9007199254740992

BigInt.parseInt('9007199254740993', 10)

// 9007199254740993n

**转换规则**

可以使用Boolean()、Number()和String()这三个方法，将 BigInt 可以转为布尔值、数值和字符串类型。

Boolean(0n) // false

Boolean(1n) // true

Number(1n) // 1

String(1n) // "1"

上面代码中，注意最后一个例子，转为字符串时后缀n会消失。

另外，取反运算符（!）也可以将 BigInt 转为布尔值。

!0n // true

!1n // false

**数学运算**

加减乘除还有\*\*(x的n次方)这个运算符都和Number一样，就是除法会忽略小数

BinInt不能和Number类型进行运算，会报错（精度丢失）

+1n会报错

-1n不报错

因为BigInt总是带着符号的，所以一些符号运算会报错

几乎所有的数值运算符都可以用在 BigInt，但是有两个例外。

* 不带符号的右移位运算符>>>
* 一元的求正运算符+

BigInt与Sting进行运算时，会将BigInt先转为字符串（不带n）

其他的就是和Number一样

# 函数扩展

## 函数参数的默认值

在ES5中是不允许函数的参数默认传参的，在ES6新增了这个功能，写法与之前的一样

function Point(x = 0, y = 0) {

this.x = x;

this.y = y;

}

const p = new Point();

p // { x: 0, y: 0 }

参数列表的默认赋值是默认声明，函数体不能再次声明，否则会报错

let x = 99;

function foo(p = x + 1) {

console.log(p);

}

foo() // 100

x = 100;

foo() // 101

参数列表每次都会重新计算值再赋给默认参数

**参数默认值的解构赋值**

请区分以下代码

function foo({x, y = 5}) {

console.log(x, y);

}

foo({}) // undefined 5

function foo({x, y = 5} = {}) {

console.log(x, y);

}

foo() // undefined 5 这里y是一个变量，而不是对象的属性

区分

// 写法一

function m1({x = 0, y = 0} = {}) {

return [x, y];

}

// 写法二

function m2({x, y} = { x: 0, y: 0 }) {

return [x, y];

}

原文：

上面两种写法都对函数的参数设定了默认值，区别是写法一函数参数的默认值是空对象，但是设置了对象解构赋值的默认值；写法二函数参数的默认值是一个有具体属性的对象，但是没有设置对象解构赋值的默认值

*写法一使用到了参数的默认赋值，但是没有设置解构赋值的默认赋值*

*写法二使用了解构赋值的默认赋值，但是没有为参数列表赋默认值，只是声明了x，y这俩个变量*

*写法一参数的默认赋值是一个空对象，但是设置了解构赋值的默认赋值*

*写法二没有使用解构赋值的默认赋值，但是参数列表有赋默认值，默认赋值的是一个有具体属性的对象*

*以上红色字有部分错误*

## 默认参数的位置

如果有默认值得参数不是末尾得参数，则无法省略中间的参数来启用他的默认值，例如

function f(x = 1, y) {

return [x, y];

}

f() // [1, undefined]

f(2) // [2, undefined]

f(, 1) // 报错

f(undefined, 1) // [1, 1]

这里第三个，省略了第一个，给第二个参数传了值，想实现x=1，y=1的效果，但是实际上这是不行的，如果不是末尾，想启用默认值，必须严格写undefined，写null都没用，如果写了null，那个参数的值就会变成null而不是默认值

## 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;

}

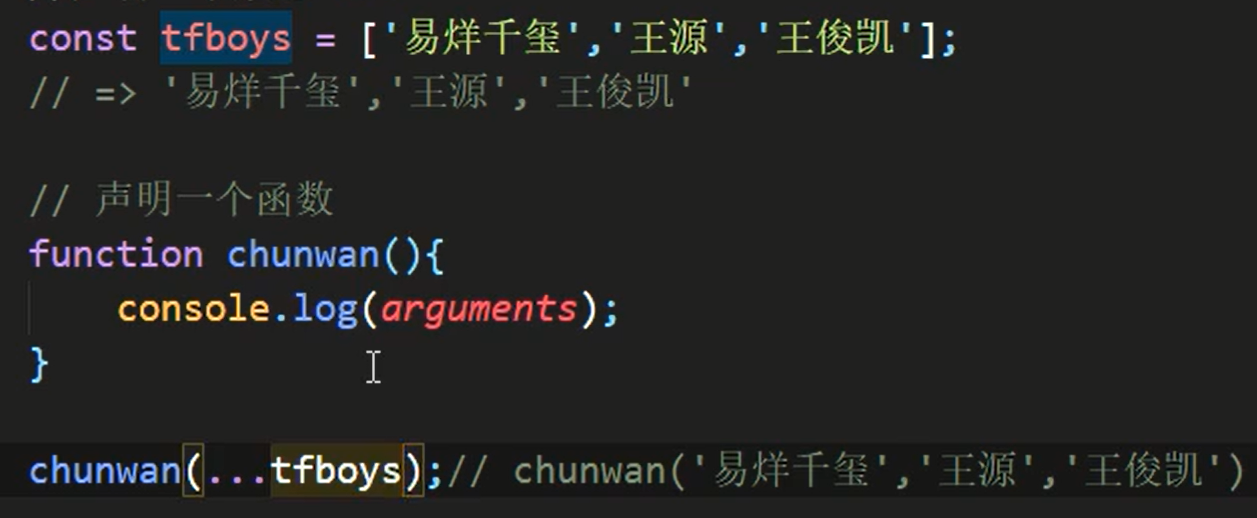
return sum;

}

add(2, 5, 3) // 10

rest参数之后不能有任何参数，也就是rest参数一定是最后一个参数

### 拓展运算符



和rest参数很像，但是不一样，rest参数是在函数形参那里，而这个扩展运算符是写在函数实参那里，rest是把所有的形参组成一个数组，

而扩展运算符可以将一个数组（…tfboys）的变成一个参数序列，用逗号分隔于是可以这样写，视作传过去了数组那么多个实参

let a =[…ary1,…ary2],这样a就等于ary1，ary2的参数列表之和

## 严格模式

## name属性

f.name会返回函数名

function foo() {}

foo.name // "foo"

这个属性早就被浏览器广泛支持，但是直到 ES6，才将其写入了标准。

需要注意的是，ES6 对这个属性的行为做出了一些修改。如果将一个匿名函数赋值给一个变量，ES5 的name属性，会返回空字符串，而 ES6 的name属性会返回实际的函数名。

var f = function () {};

// ES5

f.name // ""

// ES6

f.name // "f"

上面代码中，变量f等于一个匿名函数，ES5 和 ES6 的name属性返回的值不一样。

如果将一个具名函数赋值给一个变量，则 ES5 和 ES6 的name属性都返回这个具名函数原本的名字。

const bar = function baz() {};

// ES5

bar.name // "baz"

// ES6

bar.name // "baz"

## 箭头函数

箭头函数有几个使用注意点。

（1）箭头函数没有自己的this对象（详见下文）。

（2）不可以当作构造函数，也就是说，不可以对箭头函数使用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。

以上最重要的是第一点。对于普通函数来说，内部的this指向函数运行时所在的对象，但是这一点对箭头函数不成立。它没有自己的this对象，内部的this始终指向作用域下的 this的值。也就是说，箭头函数内部的this指向是固定的，相比之下，普通函数的this指向是可变的。

如何定义箭头函数？

var f =()=>{code something}

箭头函数的一个用处是简化回调函数。

// 普通函数写法

[1,2,3].map(function (x) {

return x \* x;

});

// 箭头函数写法

[1,2,3].map(x => x \* x);

另一个例子是

// 普通函数写法

var result = values.sort(function (a, b) {

return a - b;

});

// 箭头函数写法

var result = values.sort((a, b) => a - b);

某种情况下还可以简写

当只有一个参数时，可以省略小括号

var f = n =>{return n+n}

当代码题只有一条语句的时候可以省略花括号,return语句也必须省略

var f =n=n\*n

语句的执行结果就是函数的返回值

## foreach方法

这个其实不是ES6的，但是之前没用过，这里学函数rest参数的时候看到了就了解了一下

foreach是一个数组的方法，它的基本用法为

Array.foreach(function(item,index,arr))

foreach里一定要放一个函数，回调函数也好，其他写好的函数也罢。这个函数可以有三个参数，但至少需要一个函数，这个item参数代表的就是这个数组的每个元素的值，数组有多长，这里面的函数就执行多少次，item依次变换成数组的元素值，这也就是foreach的含义，for循环数组的each元素

index代表的是当次循环的数组元素的索引，arr代表的数组

forEach() 方法对数组的每个元素执行一次提供的函数。总是返回undefined；

var arr = [1,2,3,4];

arr.forEach(alert);

// 等价于：

var arr = [1, 2, 3, 4];

for (var k = 0, length = arr.length; k < length; k++) {

alert(array[k]);

}

forEach方法中的function回调有三个参数：

第一个参数是遍历的数组内容，

第二个参数是对应的数组索引，

第三个参数是数组本身

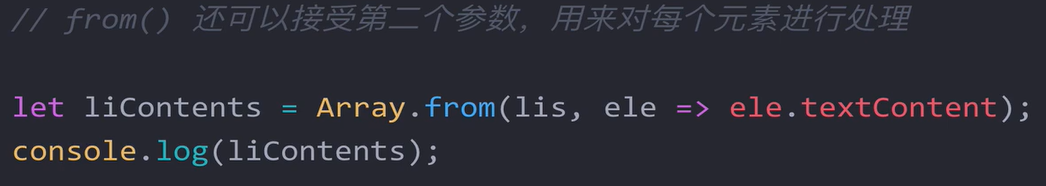
原文链接：<https://blog.csdn.net/fanfan_h/article/details/91406310>

# 数组扩展

## Array.from方法

转换为真正的数组，例如arguments

let a =Array.from(arguments)

这个from还可以跟第二个参数，可以加一个回调函数

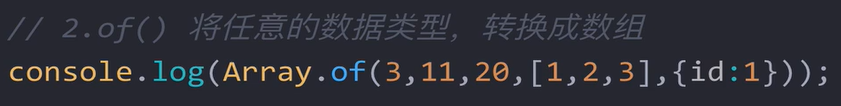
上图将lis的每一个元素变成字符串再返回一个数组给lisContents

## 拓展运算符

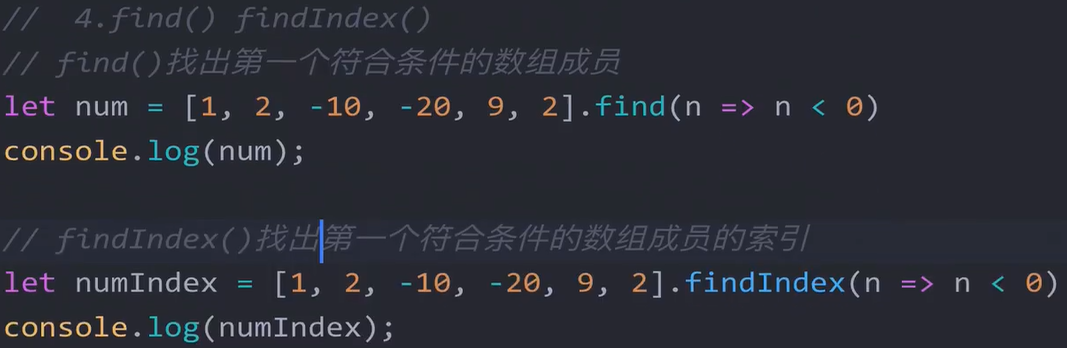
就是那三个点…也可以转换为真正的数组

## Array.of()

将任意数据类型转换成数组



## find和findindex



## values,keys,entries



这些返回的就是Iterator的东西了

## includes

[1,2,3,4].includes(2) //如果数组里有这个值则返回true，没有则flase

## indexOf

[1,2,3,4].indexOf(2) //如果数组里有这个值则返回它的下标，没有则返回-1

# 对象扩展

## 对象的简洁写法

ES6允许在大括号里直接写变量以及函数



在对象里写方法也可以省去=function

improve(){…}而不是improve:function(){…}

# Symbol

## 概述

symbol是一种新的数据类型，为了解决对象属性重名问题而在es6中提出的。创建方式为

let f=Symbol()；这个是Symbol函数，可以创建一个Symbol类型的值，每一个Symbol类型的值都不一样，这样就可以保证每个值都是独一无二的了。

Symbol函数不可以使用new，因为Symbol值不是一个对象，无法添加属性，是有点类似String类型的数据类型。Symbol值不能与任何值继续运算

Symbol的Boolean值是true Symbol可显示转换为字符串类型

其实可以理解为Symbol类型是一串随机密钥作为属性，这样就是独一无二的了

cons MySymbol=Symbol（）其实是生成了一个Symbol类型的密钥赋给MySymbol，这个MySymbol是一个Symbol类型值的变量

## Symbol函数使用

Symbol()在直接使用的时候很难以区分，为了区分各种Symbol的值，可一个Symbol（）函数传一个参数，用于描述这个生成的Symbol值，这样可以在控制台以及转为字符时可以区分

let s1 = Symbol('foo');

let s2 = Symbol('bar');

s1 // Symbol(foo)

s2 // Symbol(bar)

s1.toString() // "Symbol(foo)"

s2.toString() // "Symbol(bar)"

如果传过来的参数是一个对象的话，那么则会调用其同string方法转换为字符串

即便你传参传的是同一个字符串他们的Symbol值都是不一样的

## Symbol.prototype.description

Symbol可以使用tostring将自己的描述转换为字符串显示出来，但是调用tostring方法太过繁琐，Symbol有一个description属性，一样可以实现这个功能

const sym = Symbol('foo');

sym.description // "foo"

## 作为属性名的Symbol

还记得如何使用对象的属性吗？一般是a.index但是还有一种永远都不会错的方法那便是

对象[‘属性名’]

Symbol作为属性时便是一个这样使用的属性，它不能使用.来点出来，那样会报错，必须使用中括号加俩‘’引号的方式，其余方式一律报错

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!"

这里注意，mySymbol是一个Symbol类型的变量，在a.mySymbol=’Hello!’这里，因为用了点运算符，mySymbol变成了一串字符串，而不再是Symbol值了，但是你如果是用第二行那个方法去赋值，那么就是Symbol类型了，这样一来你用点运算符反而是无法正确显示出那个Hello了，得用括号运算符才可以正确使用

原文：

上面代码中，因为点运算符后面总是字符串，所以不会读取mySymbol作为标识名所指代的那个值，导致a的属性名实际上是一个字符串，而不是一个 Symbol 值。

## 实例：消除魔术字符串

[Symbol - ECMAScript 6入门 (ruanyifeng.com)](https://es6.ruanyifeng.com/#docs/symbol#Symbol-prototype-description)

## Symbol的内置属性

# Set和Map数据结构

## Set

set是一个集合，表示一个无重复值的集合列表

set的键值和value是一样的

let set =new Set();

这样一个set集合就创建好了，这个set有很多自带的属性，例如重要的

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。

以下是操作方法

Set.prototype.add(value)：添加某个值，返回 Set 结构本身。

Set.prototype.delete(value)：删除某个值，返回一个布尔值，表示删除是否成功。

Set.prototype.has(value)：返回一个布尔值，表示该值是否为Set的成员。

Set.prototype.clear()：清除所有成员，没有返回值。

将set转换为数组

let set2 = new Set([1,2,3,4,4])

let arry = […set2] //使用了扩展运算符

// arry [1,2,3,4]

//以下的不太重要

set中的对象无法被释放

let set3 = new Set(),obj={}

set3.add(obj)

obj=null

但是这个obj其实还在set3里面，若要释放则可以使用WeakSet

weakset和set差不多，可以释放其中被引用的对象，但是缺少了很多方法

只保留了add constructor has delete其他方法都无了

WeakSet

无法传入非对象类型参数

不可迭代

没有forEach

没有size属性

## Map

Map是一个键值对的有序列表，且键和值可以是任意类型

let map = new Map()

map.set(‘name’,‘张三’)

map.get(‘name’) //张三

map.delete(‘name’)

map.clear() //清除所有

map.set([1,2,3],’张三’) //键值为一个Array 值为张三

直接初始化

new Map（[‘a’,1],[‘b’,2]）

Map也和Set不能释放其中的对象的引用，Map也有一个WeakMap

不过多介绍了，少用

# Promise

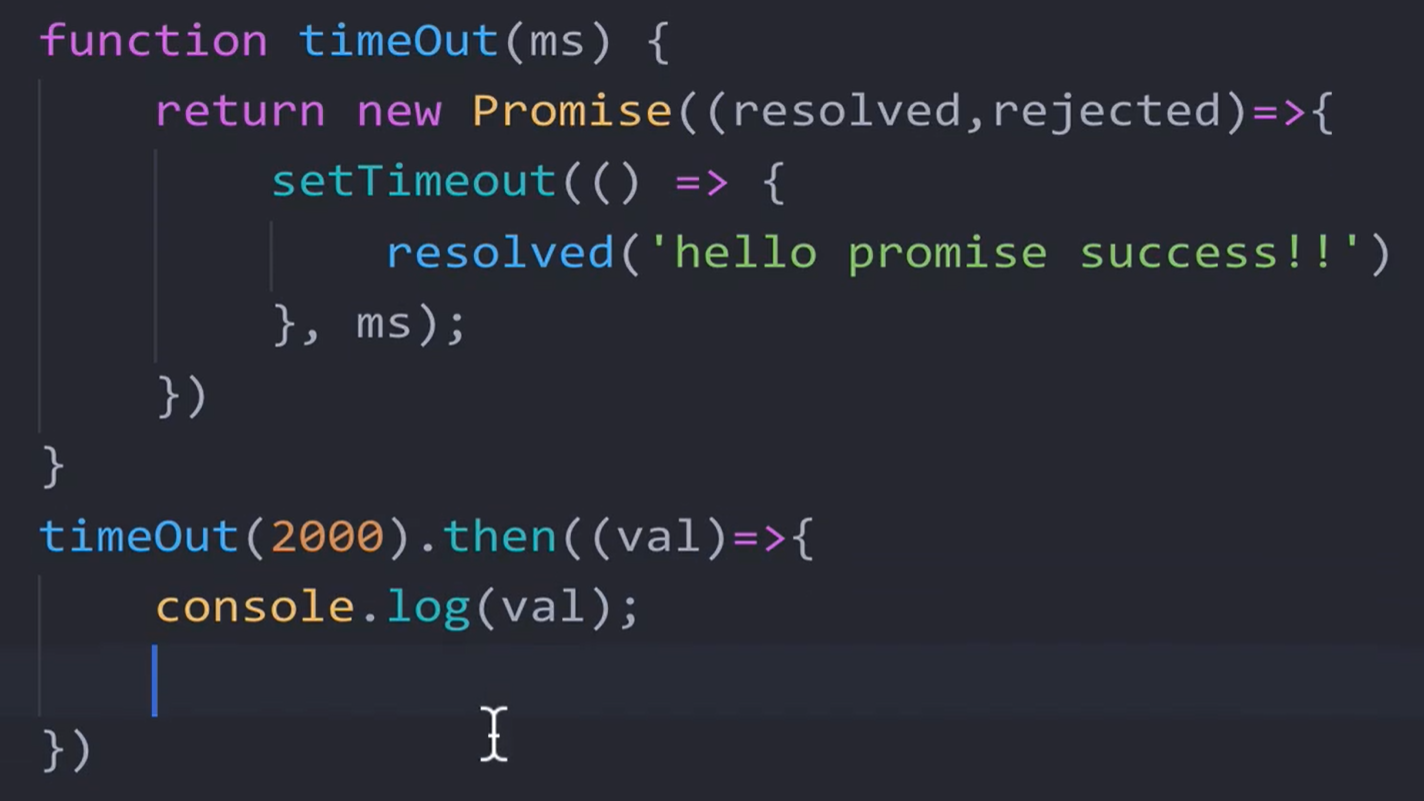
所谓Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。Promise 提供统一的 API，各种异步操作都可以用同样的方法进行处理。

有三种状态：pending（进行中）、fulfilled（已成功）和rejected（已失败）

Promise对象有以下两个特点。

1.Promise不受外界影响

2.Promise的状态一旦改变就确定了，无法改变（进行中——成功）、（进行中——失败）



timeOut传入2000ms,然后返回一个Promise对象，Promise后面跟着的回调函数有俩参数，他们分别是resolved和rejected（都是系统配置好的函数，代表失败和成功），然后执行ms秒的计时器，这里写的成功，返回resolved函数。而外部的timeOut(2000).then的这个then就是用来接收返回（then好像是接收成功的结果，错误的由catch接收，好像是）结果的，then里面有一个参数，其value值就是Promise的成功函数返回的值，这里将其输出了，正确两秒后输出了

# Iterator和for of

## 概述

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};

}

};

}

# Generator函数的语法

## 概述

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。

### 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。并不是yield的返回值。yield表达式本身没有返回值，或者说总是返回undefined

（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。

yield如果在普通函数里使用了的话会报错

### 与Iterator的关系

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

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

我的理解是：有些数据结构有Symbol.Iterator属性，而这个属性是一个方法（Genertaor），这个方法返回的是指向内部状态的指针对象，而Genertor有俩特征，声明的时候function后跟着一个\*号（位置随意）

函数内部是一步一步执行的，以yield为标志。调用时不执行，返回的是指向内部状态的指针对象，而且只有next（）方法可以执行下一步，next返回一个对象，这个对象的values值是yield语句的返回值，done值一般是flase，直到return语句，values值是return的返回值，而done值变成true。

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。

### next方法的参数

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\* foo(x) {

var y = 2 \* (yield (x + 1));

var z = yield (y / 3);

return (x + y + z);

}

var a = foo(5);

a.next() // Object{value:6, done:false}

a.next() // Object{value:NaN, done:false}

a.next() // Object{value:NaN, done:true}

var b = foo(5);

b.next() // { value:6, done:false }

b.next(12) // { value:8, done:false }

b.next(13) // { value:42, done:true }

a的next因为没有带参数，导致从第二次开始，上一次yield表达式的值是undefined，带入运算导致都是NaN

而b的next从第二个开始，有参数12，即上一次yield的表达式是12，即可以正确计算

# 类的继承

通过extends关键字来继承父类，例如

class Point {

}

class ColorPoint extends Point {

}

和java类似

不过这里放一个我自己理解的概念constructor里的是实例

而在父类里，不写在constructor构造函数里的方法和和属性是prototype的

class Foo {

constructor() {

console.log(1);

}

}

class Bar extends Foo {

constructor() {

super();

console.log(2);

}

}

const bar = new Bar();

// 1

// 2

子类如果写了构造函数没写super（）的话，新建实例会报错，super（）必须写在构造函数的第一行，不然不能使用this关键字。如果你不写构造函数，系统自己帮你写而且还有super（），另外写了super（）函数的话，父类会先执行一遍他的构造函数，所以上图有两个输出。

super（）只能在子类构造函数中写

# Super关键字

class A {}

class B extends A {

constructor() {

super();

}

}

super虽然代表了父类A的构造函数，但是返回的是子类B的实例，即super内部的this指的是B的实例

不过这里放一个我自己理解的概念constructor里的是实例的方法和属性

而在父类里，不写在constructor构造函数里的方法和和属性是prototype的

super当作一般的对象时，代表的是 父类.prototype，只能调用原型的属性

且super里的this指向的是子类的原型

super当作对象在静态方法中就不是指向父类原型了，而是父类了，可以调用父类的静态方法，且this指向的是子类的实例对象而不是子类实例了