[第一章 ECMAScript 6 简介 5](#_Toc14129)

[Node 5](#_Toc23447)

[Babel 转码器 5](#_Toc13718)

[配置文件.babelrc 5](#_Toc18676)

[命令行转码babel-cli 5](#_Toc1524)

[Win7安装babel-cli 6](#_Toc6171)

[babel-node 7](#_Toc15076)

[babel-register 7](#_Toc26809)

[babel-core 8](#_Toc30425)

[babel-polyfill 8](#_Toc14097)

[第二章 let 和 const 命令 9](#_Toc25170)

[Let命令 9](#_Toc22977)

[块级作用域 9](#_Toc4371)

[ES6的作用域 9](#_Toc14001)

[块级作用域与函数声明 9](#_Toc13767)

[Const命令 9](#_Toc12450)

[ES6 声明变量的六种方法 10](#_Toc16365)

[顶层对象的属性 10](#_Toc6634)

[global 对象获取 10](#_Toc7687)

[变量的解构赋值 10](#_Toc7781)

[数组的解构赋值 10](#_Toc28254)

[基本用法 11](#_Toc26022)

[默认值 11](#_Toc31457)

[对象的解构赋值 11](#_Toc30695)

[解构嵌套结构对象 11](#_Toc222)

[对象解构的默认值 12](#_Toc847)

[字符串的解构赋值 12](#_Toc2987)

[数值和布尔值的解构赋值 12](#_Toc19584)

[函数参数的解构赋值 12](#_Toc8569)

[圆括号问题 13](#_Toc11410)

[变量解构赋值的用途 13](#_Toc15339)

[字符串的扩展 15](#_Toc20451)

[字符的 Unicode 表示法 15](#_Toc4642)

[codePointAt() 15](#_Toc17507)

[String.fromCodePoint() 16](#_Toc10327)

[字符串的遍历器接口 16](#_Toc6981)

[normalize() 16](#_Toc31049)

[includes(), startsWith(), endsWith() 16](#_Toc16705)

[repeat() 16](#_Toc28672)

[padStart()，padEnd() 17](#_Toc2498)

[matchAll() 17](#_Toc23728)

[模板字符串 17](#_Toc3259)

[实例：模板编译 18](#_Toc9674)

[标签模板 18](#_Toc13308)

[String.raw() 18](#_Toc15994)

[模板字符串的限制 19](#_Toc31795)

[正则的扩展 19](#_Toc30239)

[RegExp 构造函数 19](#_Toc20229)

[字符串的正则方法 19](#_Toc4229)

[u 修饰符 19](#_Toc18146)

[y 修饰符 20](#_Toc21732)

[sticky 属性 20](#_Toc29311)

[flags 属性 21](#_Toc29177)

[s 修饰符：dotAll 模式 21](#_Toc7452)

[数值的扩展 21](#_Toc18348)

[二进制和八进制表示法 21](#_Toc2561)

[Number.isFinite(), Number.isNaN() 21](#_Toc27425)

[Number.parseInt(), Number.parseFloat() 21](#_Toc17621)

[Number.isInteger() 22](#_Toc8236)

[Number.EPSILON 22](#_Toc30141)

[安全整数和 Number.isSafeInteger() 22](#_Toc7034)

[Math 对象的扩展 22](#_Toc24421)

[指数运算符（\*\*） 22](#_Toc15088)

[函数的扩展 23](#_Toc7776)

[函数参数的默认值 23](#_Toc3594)

[基本用法 23](#_Toc15496)

[与解构赋值默认值结合使用 23](#_Toc5708)

[函数的 length 属性 23](#_Toc58)

[作用域 23](#_Toc10227)

[应用 23](#_Toc24233)

[rest 参数 24](#_Toc21908)

[严格模式 24](#_Toc29087)

[name 属性 24](#_Toc25928)

[箭头函数（=>） 24](#_Toc5216)

[基本用法 24](#_Toc1167)

[双冒号运算符 25](#_Toc10381)

[尾调用 25](#_Toc22559)

[函数参数的尾逗号 25](#_Toc19257)

[数组的扩展 25](#_Toc12074)

[扩展运算符 25](#_Toc17294)

[含义 25](#_Toc7656)

[扩展运算符的应用 26](#_Toc32695)

[Array.from() 26](#_Toc8554)

[Array.of() 26](#_Toc23008)

[数组实例的 copyWithin() 26](#_Toc16125)

[数组实例的 find() 和 findIndex() 26](#_Toc28312)

[数组实例的 fill() 27](#_Toc25508)

[数组实例的 entries()，keys() 和 values() 27](#_Toc22228)

[数组实例的 includes() 27](#_Toc11386)

[对象的扩展 27](#_Toc27483)

[属性的简洁表示法 27](#_Toc18160)

[属性名表达式 28](#_Toc9332)

[方法的 name 属性 28](#_Toc20558)

[Object.is() 28](#_Toc0)

[Object.assign() 28](#_Toc5595)

[属性的遍历 29](#_Toc26984)

[Object.getOwnPropertyDescriptors() 29](#_Toc25490)

[\_\_proto\_\_属性，Object.setPrototypeOf()，Object.getPrototypeOf() 29](#_Toc14863)

[\_\_proto\_\_属性 29](#_Toc15708)

[Object.setPrototypeOf() 30](#_Toc24091)

[Object.getPrototypeOf() 30](#_Toc32538)

[super 关键字 30](#_Toc18651)

[Object.keys()，Object.values()，Object.entries() 30](#_Toc522)

[Object.keys() 30](#_Toc32025)

[Object.values() 30](#_Toc6730)

[Object.entries 31](#_Toc20744)

[对象的扩展运算符(...) 31](#_Toc5515)

[Symbol 31](#_Toc4796)

[概述 31](#_Toc27691)

[作为属性名的 Symbol 32](#_Toc6475)

[属性名的遍历 32](#_Toc8272)

[Symbol.for()，Symbol.keyFor() 32](#_Toc28679)

[Set 和 Map 数据结构 33](#_Toc25278)

[Set 33](#_Toc29809)

[基本用法 33](#_Toc4991)

[Set 实例的属性和方法 33](#_Toc17618)

[WeakSet 34](#_Toc9361)

[含义 34](#_Toc3235)

[语法 34](#_Toc22321)

[WeakSet方法 34](#_Toc25895)

[Map 34](#_Toc20623)

[含义和基本用法 34](#_Toc29672)

[实例的属性和操作方法 35](#_Toc10462)

[遍历方法 35](#_Toc32237)

[WeakMap 35](#_Toc28700)

[Proxy代理 35](#_Toc7659)

[Reflect 35](#_Toc15893)

[Promise 对象 35](#_Toc31939)

[Promise 的含义 36](#_Toc20322)

[基本用法 36](#_Toc9983)

[Promise.prototype.then() 36](#_Toc32625)

[其他方法 36](#_Toc12079)

[Iterator 和 for...of 循环 37](#_Toc12261)

[默认 Iterator 接口 37](#_Toc3168)

[遍历器对象的 return()，throw() 37](#_Toc28616)

[for...of 循环 37](#_Toc23800)

[Generator 函数的语法 37](#_Toc2398)

[yield 表达式 38](#_Toc14779)

[与 Iterator 接口的关系 38](#_Toc5029)

[next 方法的参数 38](#_Toc25040)

[Generator.prototype.throw() 38](#_Toc19654)

[Generator.prototype.return() 38](#_Toc4081)

[yield\* 表达式 38](#_Toc27897)

[作为对象属性的 Generator 函数 39](#_Toc2614)

[Generator 函数的this 39](#_Toc23985)

[异步操作的同步化表达 39](#_Toc18713)

# 第一章 ECMAScript 6 简介

ECMAScript 和 JavaScript 的关系：前者是后者的规格，后者是前者的一种实现。

## Node

JavaScript 的服务器运行环境（runtime）。

node --v8-options | grep harmony，查看 Node 已经实现的 ES6 特性。

## Babel 转码器

可以将 ES6 代码转为 ES5 代码，从而在现有环境执行。

转码前：

input.map(item => item+1); //类似lambda表达式

转码后：

input.map( function(item){

return item+1;

});

Babel在线转换 <https://babeljs.io/repl/>

### 配置文件.babelrc

Babel 的配置文件是.babelrc，存放在项目的根目录下。该文件用来设置转码规则和插件。

基本格式：

{

“presets”: [ ],

“plugins”: [ ]

}

presets字段设定转码规则，按需安装官方提供的规则集。

--save-dev babel-preset-latest；安装latest版，其他版本：react、stage-0~4

安装转码规则后，添加规则：

{

“presets”: [

“latest ”,

“react”],

“plugins”: [ ]

}

### 命令行转码babel-cli

#### 安装在全局环境

安装命令：npm install --global babel-cli

基本用法：

# 转码结果输出到标准输出

babel example.js

# 转码结果写入一个文件

# --out-file 或 -o 参数指定输出文件

babel example.js --out-file compiled.js

# 或者

babel example.js -o compiled.js

# 整个目录转码

# --out-dir 或 -d 参数指定输出目录

babel src --out-dir lib

# 或者

babel src -d lib

# -s 参数生成source map文件

babel src -d lib -s

#### 安装在项目中

1. 安装babel-cli

npm install --save-dev babel-cli

1. 改写package.json

{

// ...

"devDependencies": {

"babel-cli": "^6.0.0"

},

"scripts": {

"build": "babel src -d lib"

},

}

转码命令：

npm run build

### Win7安装babel-cli

npm install babel-cli -g

#### [NodeJS、NPM安装配置步骤(windows7 64位)](http://blog.csdn.net/huanghanqian/article/details/51440592)

<http://blog.csdn.net/huanghanqian/article/details/51440592>

**NodeJS/NPM安装**

1. 登陆官网（<http://nodejs.org/>），点击自动下载安装。
2. 安装过程基本直接“NEXT”；安装过程中会自动添加path的系统变量，变量值是你的安装路径，例如“C:\Program Files\nodejs”）。
3. 安装完成后使用node -v，查看安装版本。
4. npm的安装。新版NodeJS集成npm；使用npm -v查看安装版本。

**npm配置**

1，配置npm的全局模块的存放路径以及cache的路径，例如我希望将以上两个文件夹放在NodeJS的主目录下，便在NodeJs下建立"node\_global"及"node\_cache"两个文件夹。

2，启动cmd，输入

npm config set prefix "E:\javasoftware\nodejs\node\_global" //全局模块的存放路径

npm config get prefix //获取prefix路径

npm config set cache "E:\javasoftware\nodejs\node\_cache" //cache的路径

1. 安装模块，

npm install express -g，安装express到global目录npm install；在项目.json文件处使用，自动安装下

4，进入环境变量对话框，在系统变量下新建"NODE\_PATH"，输入”E:\javasoftware\nodejs\node\_global\node\_modules“。不改path变量。

5，重新开启cmd命令行，进入node，输入“require('express')”来测试下node的模块全局路径是否配置正确。

### babel-node

babel-cli工具自带一个babel-node命令，提供一个支持 ES6 的 REPL 环境。它支持 Node 的 REPL 环境的所有功能，而且可以直接运行 ES6 代码。它不用单独安装，而是随babel-cli一起安装。然后，执行babel-node就进入 REPL 环境。

使用babel-node替代node，这样script.js本身就不用做任何转码处理。

babel-node 进入babel-node模式，进入 REPL 环境

babel-node es.js 执行es.js

**babel-node安装于项目**：

1. npm install --save-dev babel-cli
2. 改写package.json

{

"scripts": {

"script-name": "babel-node script.js"

}

}

### babel-register

babel-register模块改写require命令，为它加上一个钩子。此后，每当使用require加载.js、.jsx、.es和.es6后缀名的文件，就会先用 Babel 进行转码。

npm install --save-dev babel-register 安装

使用时，必须首先加载babel-register。

require("babel-register");

require("./index.js");

babel-register只会对require命令加载的文件转码，而不会对当前文件转码。另外，由于它是实时转码，所以只适合在开发环境使用。

### babel-core

如果某些代码需要调用 Babel 的 API 进行转码，就要使用babel-core模块。

npm install babel-core --save 安装

转码：

var babel = require('babel-core');

// 字符串转码

babel.transform('code();', options);

// => { code, map, ast }

// 文件转码（异步）

babel.transformFile('filename.js', options, function(err, result) {

result; // => { code, map, ast }

});

// 文件转码（同步）

babel.transformFileSync('filename.js', options);

// => { code, map, ast }

// Babel AST转码

babel.transformFromAst(ast, code, options);

// => { code, map, ast }

### babel-polyfill

Babel 默认只转换新的 JavaScript 句法（syntax），而不转换新的 API。想让API运行，必须使用babel-polyfill，为当前环境提供一个垫片。

npm install --save babel-polyfill

然后，在脚本头部，加入如下任一行代码。

import 'babel-polyfill';

require('babel-polyfill');

# 第二章 let 和 const 命令

## Let命令

Es6新增；它的用法类似于var，但是所声明的变量，只在let命令所在的代码块内有效。

特性：

不存在变量提升；必须先声明后使用。

暂时性死区，声明前使用则报错。

不允许重复声明，不允许作用域中存在同名的let变量、var变量。

个人理解：var--对象、引用，let--值；在for循环中/子块中，以const形式传入循环体？且传入量的改变不会对let变量的值造成影响。

## 块级作用域

ES5 只有全局作用域和函数作用域，没有块级作用域，会产生不合理情况：

内层变量覆盖外层变量。var变量声明以最后声明为准？

计数的循环变量泄露为全局变量。

### ES6的作用域

ES6 允许块级作用域的任意嵌套；

外层代码块不受内层代码块的影响。

外层作用域无法读取内层作用域的变量。

内层作用域可以定义外层作用域的同名变量。

### 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。

考虑到环境导致的行为差异太大，应该避免在块级作用域内声明函数。如果确实需要，也应该写成函数表达式(a = function(){})，而不是函数声明语句。

### Const命令

#### 基本用法

const声明一个只读的常量。一旦声明，常量的值就不能改变。类似final。

const一旦声明变量，就必须立即初始化。

与let命令相同：只在声明所在的块级作用域内有效，不提升，存在暂时性死区，只能在声明的位置后面使用，不可重复声明。

const实际上保证的，是变量指向的那个内存地址不得改动；故指向对象时值可以改变。

如果真的想将对象冻结，应该使用Object.freeze方法。

const foo = Object.freeze({});

将对象彻底冻结

var constantize = (obj) => {

Object.freeze(obj);

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

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

constantize( obj[key] );

}

});

};

### ES6 声明变量的六种方法

ES5 两种声明变量的方法：var、function。

ES6 添加：let、const、import、class。

## 顶层对象的属性

顶层对象，在浏览器环境指的是window对象，在 Node 指的是global对象。

ES5 之中，顶层对象的属性(window.a)与全局变量(var a)是等价的。

var a = 1;

window.a; //1

ES6规定，为了保持兼容性：

var命令和function命令声明的全局变量，是顶层对象的属性；

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。

## global 对象获取

CommonJS 的写法

var global = require('system.global')();

ES6 模块的写法

import getGlobal from 'system.global';

const global = getGlobal();

# 变量的解构赋值

## 数组的解构赋值

### 基本用法

ES6 允许从数组和对象(可遍历)中提取值，对变量进行赋值，这被称为解构（Destructuring）。

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

let [a,...arr] = [1]; //a=1,arr=[]

如果解构不成功，变量的值就等于undefined。

let [foo] = []; //foo=udefined

let [bar,foo] = [1]; //bar=1,foo=undefined

不完全解构，即等号左边的模式，只匹配一部分的等号右边的数组。解构依然成功。

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

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，那么将会报错。

### 默认值

解构赋值允许指定默认值。

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

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

let [x = 1] = [null]; //x = null

如果默认值是一个表达式，那么这个表达式是惰性求值的，即只有在用到的时候，才会求值。

function f() { console.log('aaa'); }

let [x = f()] = [1]; //f()不执行

默认值可以引用解构赋值的其他变量，但该变量必须已经声明。

let [x = 1, y = x] = [1, 2]; // x=1; y=2

let [x = y, y = 1] = []; // ReferenceError: y is not defined

## 对象的解构赋值

对象的解构赋值时变量必须与属性同名(顺序不要求)，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'

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;

### 解构嵌套结构对象

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 { p, p: [x, { y }] } = obj; //x = "Hello",y = "World",p = ["Hello", {y: "World"}]

### 对象解构的默认值

默认值生效的条件：对象的属性值严格等于undefined。

如果解构模式是嵌套的对象，而且子对象所在的父属性不存在，那么将会报错。

对undefined取子属性会报错。

### 字符串的解构赋值

字符串可被转换成了一个类似数组的对象，有length属性。

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

let {length : len} = 'hello'; //len=5

### 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 = [Function: toString]

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。

### 函数参数的解构赋值

[[1, 2], [3, 4]].map(([a, b]) => a + b); // [ 3, 7 ]；map()函数，逐一传值？

函数参数的解构也可以使用默认值，条件为undefined。

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]

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

### 圆括号问题

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。不要在模式中放置圆括号。

个人：圆括号括起，为单独的运算元。

#### 不能使用圆括号的情况

（1）变量声明语句

let [(a)] = [1]; // 报错

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

（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。

function f([(z)]) { return z; } // 报错

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

（3）赋值语句的模式

({ p: a }) = { p: 42 }; // 报错

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

### 变量解构赋值的用途

（1）交换变量的值

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

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

1. 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 example() {

return [1, 2, 3];

}

let [a, b, c] = example();

// 返回一个对象

function example() {

return {

foo: 1,

bar: 2

};

}

let { foo, bar } = example();

（3）函数参数的定义

// 参数是一组有次序的值

function f([x, y, z]) { ... }

f([1, 2, 3]);

// 参数是一组无次序的值

function f({x, y, z}) { ... }

f({z: 3, y: 2, x: 1});

（4）提取 JSON 数据

let jsonData = {

id: 42,

status: "OK",

data: [867, 5309]

};

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

（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

};

（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

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

# 字符串的扩展

ES6 加强了对 Unicode 的支持，并且扩展了字符串对象。

## 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"

超出这个范围的字符，必须用两个双字节的形式表示。

"\uD842\uDFB7" // "吉"

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}" // "吉"，字符无法显示，替代字

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

## 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，charAt()不能正常工作。

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 = '吉a';

s.codePointAt(0) // 134071

s.codePointAt(1) // 57271，仍不正常

s.codePointAt(2) // 97

使用for...of循环，识别 32 位的 UTF-16 字符。

let s = '吉a';

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

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

## 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）。

ES6 提供了String.fromCodePoint方法，可以识别大于0xFFFF的字符。

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

## 字符串的遍历器接口

ES6 为字符串添加了遍历器接口，使得字符串可以被for...of循环遍历；可以识别大于0xFFFF的码点。

for (let codePoint of 'foo') {

console.log(codePoint)

}

## normalize()

将字符的不同表示方法统一为同样的形式，这称为 Unicode 正规化。

'\u01D1'.normalize() === '\u004F\u030C'.normalize() // true

normalize方法目前不能识别三个或三个以上字符的合成。这种情况下，还是只能使用正则表达式，通过 Unicode 编号区间判断。

## includes(), startsWith(), endsWith()

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。

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，对前5个字符

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

## 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) // ""

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

**特殊参数**

小数，取整；正数向下取整，负数向上取整。

负数或者Infinity，报错。

NaN等同于 0。

字符串，则会先转换成数字。

## padStart()，padEnd()

字符串补全长度。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'

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'

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'

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。

'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"

## matchAll()

返回一个正则表达式在当前字符串的所有匹配。

## 模板字符串

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。

$('#result').append(

'There are <b>' + basket.count + '</b> ' +

'items in your basket, ' +

'<em>' + basket.onSale +

'</em> are on sale!'

);

在模板字符串中使用反引号，用反斜杠转义。

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。可以使用trim方法消除。

模板字符串中嵌入变量，需要将变量名写在${}之中；变量可为：对象、属性、函数、模板字符串等。变量没有声明则报错

如果大括号中的值不是字符串，将按照一般的规则转为字符串。

## 实例：模板编译

通过正则表达式，转换为echo输入；并封装。

## 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)

如果模板字符里面有变量，会将模板字符串先处理成多个参数，再调用函数。

let [a,b] = [5,10];

tag`Hello ${ a + b } world ${ a \* b }`; // 等同于 tag(['Hello ', ' world ', ''], 15, 50);

“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;

}

## String.raw()

模板字符串的处理函数，返回一个斜杠都被转义（即斜杠前面再加一个斜杠）的字符串，对应于替换变量后的模板字符串。

String.raw`Hi\n${2+3}!`; // 返回 "Hi\\n5!"

如果原字符串的斜杠已经转义，那么String.raw会进行再次转义。

String.raw`Hi\\n` // 返回 "Hi\\\\n"

String.raw方法作为正常的函数;第一个参数，应该是一个具有raw属性的对象，且raw属性的值应该是一个数组。

String.raw({ raw: 'test' }, 0, 1, 2); // 't0e1s2t';

等同于String.raw({ raw: ['t','e','s','t'] }, 0, 1, 2);

## 模板字符串的限制

前面提到标签模板里面，可以内嵌其他语言。但是，模板字符串默认会将字符串转义，导致无法嵌入其他语言。

# 正则的扩展

## RegExp 构造函数

在 ES5 中，RegExp构造函数的参数有两种情况。

var regex = new RegExp('xyz', 'i'); // 等价于 var regex = /xyz/i;

var regex = new RegExp(/xyz/i); // 等价于 var regex = /xyz/i;

参数为"/xyz/i"时，ES5 不允许使用第二个参数添加修饰符。ES6 可以，第二个参数指定修饰符覆盖原有正则对象的修饰符。

new RegExp(/abc/ig, 'i').flags; //i覆盖ig

## 字符串的正则方法

字符串对象共有 4 个方法，可以使用正则表达式：match()、replace()、search()和split()。

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

用来正确处理大于\uFFFF的 Unicode 字符；即正确处理四个字节的 UTF-16 编码。

/^\uD83D/u.test('\uD83D\uDC2A') // false

/^\uD83D/.test('\uD83D\uDC2A') // true

一旦加上u修饰符号，就会修改下面这些正则表达式的行为。

**（1）点字符**

点（.）字符在正则表达式中，是除了换行符以外的任意单个字符。对于码点大于0xFFFF的Unicode 字符，点字符不能识别，必须加上u修饰符。

var s = '吉';

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

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

**（2）Unicode 字符表示法**

ES6 新增了使用大括号表示 Unicode 字符，这种表示法在正则表达式中必须加上u修饰符，才能识别当中的大括号，否则会被解读为量词。

/\u{61}/.test('a') // false，认为匹配 61 个连续的u。

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

**（3）量词**

使用u修饰符后，所有量词都会正确识别码点大于0xFFFF的 Unicode 字符。

/a{2}/.test('aa') // true

/吉{2}/u.test('吉吉') // true

**（4）预定义模式**

u修饰符也影响到预定义模式，能否正确识别码点大于0xFFFF的 Unicode 字符。

/^\S$/.test('吉') // false，认为有两个字符

/^\S$/u.test('吉') // true

**（5）i 修饰符**

有些 Unicode 字符的编码不同，但是字型很相近，比如，\u004B与\u212A都是大写的K。需加u修饰符，识别非规范的K字符。

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

/[a-z]/iu.test('\u212A') // true

## y 修饰符

“粘连”（sticky）修饰符；确保匹配必须从剩余的第一个位置开始。

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

## sticky 属性

表示是否设置了y修饰符。

var r = /hello\d/y;

r.sticky // true

## flags 属性

返回正则表达式的修饰符。

/abc/ig.source; // "abc"；ES5 的 source 属性，返回正则表达式的正文

/abc/ig.flags; // 'gi'；ES6 的 flags 属性，返回正则表达式的修饰符

## s 修饰符：dotAll 模式

点（.）代表任意的单个字符：例外：四个字节的 UTF-16 字符，、行终止符（换行符，回车符，行分隔符，段分隔符）。s修饰符使得.可以匹配任意单个字符。

/foo.bar/.test('foo\nbar') // false

/foo.bar/s.test('foo\nbar') // true

dotAll属性，返回一个布尔值，表示该正则表达式是否处在dotAll模式。

/s修饰符和多行修饰符/m不冲突，两者一起使用的情况下，.匹配所有字符，而^和$匹配每一行的行首和行尾。

# 数值的扩展

## 二进制和八进制表示法

二进制：0b、0B；八进制：0o、0O

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.isFinite(), Number.isNaN()

Number.isFinite()；检查数值是否为有限（finite）。如果参数类型不是数值，返回false。

Number.isNaN()；检查值是否为NaN。如果参数类型不是数值，返回false。

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

## Number.parseInt(), 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。可逐步减少全局性方法，使得语言逐步模块化。

Number.parseInt('12.34') // 12

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

## Number.isInteger()

判断一个数值是否为整数。

Number.isInteger(25) // true

Number.isInteger(25.1) // false

Number.isInteger(25.0) // true；js整数和浮点数储存方法相同，25和25.0被视为同一个值。

如果参数不是数值，Number.isInteger返回false。

误判：js中误差小于Number.EPSILON，则认为没有误差。

## Number.EPSILON

表示 1 与大于 1 的最小浮点数之间的差，2 的 -52 次方。实际上是js能够表示的最小精度，误差如果小于这个值，就可以认为没有意误差。

用来设置“能够接受的误差范围”。

## 安全整数和 Number.isSafeInteger()

js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。

Number.MAX\_SAFE\_INTEGER；最大值，2^53-1

Number.MIN\_SAFE\_INTEGER；最小值，-2^53+1

Number.isSafeInteger()；判断一个整数是否在安全范围之内。验证运算结果是否落在安全整数的范围内，不要只验证运算结果，而要同时验证参与运算的每个值。

-2^53

## Math 对象的扩展

Math.trunc()；去除一个数的小数部分，返回整数部分。

Math.sign()；判断一个数到底是正数1、负数-1、零+-0、非数NaN。

Math.cbrt()；计算一个数的立方根。

Math.clz32()；返回一个数的 32 位无符号整数形式有多少个前导 0。

Math.imul()；返回两个数以 32 位带符号整数形式相乘的结果( 32 位的带符号整数)。

Math.fround()；返回一个数的32位单精度浮点数形式。

Math.hypot()；返回所有参数的平方和的平方根。

## 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; //2.25

# 函数的扩展

## 函数参数的默认值

### 基本用法

直接写在参数定义的后面。

function log(x, y = 'World') {

console.log(x, y);

}

参数默认值不是传值的，而是每次都重新计算默认值表达式的值。也就是说，参数默认值是惰性求值的。

### 与解构赋值默认值结合使用

见解构。

### 函数的 length 属性

返回没有指定默认值的参数个数。

(function (a, b, c = 5) {}).length // 2

如果设置了默认值的参数不是尾参数，length属性不计入后面的参数。

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

### 作用域

设置参数的默认值后，函数进行声明初始化时，参数会形成一个单独的作用域（context）。等到初始化结束，这个作用域就会消失。

### 应用

利用参数默认值设定参数默认为异常函数，可以指定某一个参数不得省略，如果省略就抛出一个错误。

## rest 参数

形式：...变量名，用于获取函数的多余参数。用数组形式获取。

函数的length属性，不包括 rest 参数。

## 严格模式

函数体中声明：'use strict'。

只要函数参数使用了默认值、解构赋值、或者扩展运算符，则函数内部就不能显式设定为严格模式，否则会报错。可设定全局性的严格模式、把函数包在一个无参数的立即执行函数中。

## name 属性

返回该函数的函数名。

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 () {};

f.name // ""，ES5

f.name // "f"，ES6

Function构造函数返回的函数实例，name属性的值为anonymous。

(new Function).name // "anonymous"

bind返回的函数，name属性值会加上bound前缀。

function foo() {};

foo.bind({}).name // "bound foo"

## 箭头函数（=>）

### 基本用法

类似lambda表达式；使用箭头函数

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

numbers(1, 2, 3, 4, 5); // [1,2,3,4,5]，圆括号传参

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 f = () => 5;

var sum = (num1, num2) => num1 + num2;

代码块部分多于一条语句，使用大括号括起，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圆括号。

let getTempItem = id => ({ id: id, name: "Temp" });

如果箭头函数只有一行语句，且不需要返回值，可以采用下面的写法。

let fn = () => void doesNotReturn();

箭头函数内部，还可以再使用箭头函数。

## 双冒号运算符

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);

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.foo; // 等同于var method = obj::obj.foo;

## 尾调用

尾调用（Tail Call），函数的最后一步是调用另一个函数。

function f(x){

return g(x); //直接调用g(x)不属于尾调用，会返回undefined。

}

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。

## 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）。使得函数参数与数组和对象的尾逗号规则保持一致。

# 数组的扩展

## 扩展运算符

### 含义

扩展运算符（spread）是三个点（...）；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

该运算符主要用于函数调用。

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

const args = [0, 1];

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

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]

### 扩展运算符的应用

扩展运算符用于数组赋值，只能放在参数的最后一位。

（1）复制数组，复制值，而不是指针

const a1 = [1, 2];

const a2 = [...a1]; // 写法一

const [...a2] = a1; // 写法二

（2）合并数组

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

（3）与解构赋值结合

const [first, ...rest] = [1, 2, 3, 4, 5];

first // 1

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

## Array.from()

将两类对象转为数组：类似数组的对象（array-like object）、可遍历（iterable）的对象。

Array.from('hello'); // ['h', 'e', 'l', 'l', 'o']

## Array.of()

将一组值，转换为数组。

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

## 数组实例的 copyWithin()

在当前数组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。

格式：copyWithin(target, start = 0, end = this.length)

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

## 数组实例的 find() 和 findIndex()

find()，用于找出第一个符合条件的数组成员,然后返回;如果没有则返回undefined。

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

findIndex()，返回第一个符合条件的数组成员的位置，如果没有则返回-1。

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

return value > 9;

}) // 2

## 数组实例的 fill()

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]

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']

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。

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

arr[0].name = "Ben";

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

## 数组实例的 entries()，keys() 和 values()

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。可以用for...of循环进行遍历。

for (let index of ['a', 'b'].keys()) {

console.log(index); // 0，1

}

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']

## 数组实例的 includes()

返回一个布尔值，表示是否包含给定的值。

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

# 对象的扩展

## 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。

属性简写：

const foo = 'bar';

const baz = {foo};

baz // {foo: "bar"}，等同于const baz = {foo: foo};

方法简写：

const o = {

method() {

return "Hello!";

}

};

// 等同于

const o = {

method: function() {

return "Hello!";

}

};

## 属性名表达式

JavaScript 定义对象的属性，有两种方法。

obj.foo = true; // 方法一

obj['a' + 'bc'] = 123; // 方法二；

用表达式作为属性名/方法名，需将表达式放在方括号之内。

属性名表达式与简洁表示法，不能同时使用。

## 方法的 name 属性

普通方法a()返回方法名a。

getter/setter方法返回 get a

bind方法创造的函数返回 bound a；

Function构造函数创造的函数返回anonymous。

方法是Symbol 值返回Symbol 值的描述

## Object.is()

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

不同之处：+0不等于-0，NaN等于自身。

## Object.assign()

基本用法

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}

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。

## 属性的遍历

（1）for...in

循环遍历对象自身的和继承的可枚举属性（不含 Symbol 属性）。

（2）Object.keys(obj)

返回一个数组，包括对象自身的（不含继承的）所有可枚举属性（不含 Symbol 属性）的键名。

（3）Object.getOwnPropertyNames(obj)

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)

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。

## Object.getOwnPropertyDescriptors()

Object.getOwnPropertyDescriptor，返回某个对象属性的描述对象（descriptor）。

Object.getOwnPropertyDescriptors，返回指定对象所有自身属性（非继承属性）的描述对象。

## \_\_proto\_\_属性，Object.setPrototypeOf()，Object.getPrototypeOf()

### \_\_proto\_\_属性

读取或设置当前对象的prototype对象(内部属性)。一般使用函数。

实现上，\_\_proto\_\_调用的是Object.prototype.\_\_proto\_\_。

### Object.setPrototypeOf()

设置一个对象的prototype对象，返回参数对象本身。

Object.setPrototypeOf(obj, proto);

//将proto设为obj对象的原型，obj可以读取proto对象的属性；返回obj。

如果第一个参数不是对象，会自动转为对象。但返回的还是第一个参数；第一个参数为undefined、null，报错。

Object.setPrototypeOf(1, {}) === 1 // true

### Object.getPrototypeOf()

读取一个对象的原型对象，一般与Object.setPrototypeOf()联用。

Object.getPrototypeOf(obj);

如果参数不是对象，会被自动转为对象。参数为undefined、null，报错。

## super 关键字

指向当前对象的原型对象。

var say = 'hi';

Object.setPrototypeOf(say,'hello');

say.find(); //'hello'

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。

## Object.keys()，Object.values()，Object.entries()

### Object.keys()

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"]

### Object.values()

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]

如果参数不是对象，Object.values会先将其转为对象。

### Object.entries

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] ]

如果原对象的属性名是一个 Symbol 值，该属性会被忽略。

## 对象的扩展运算符(...)

**解构赋值**

等号右边是undefined或null时报错。

解构赋值必须是最后一个参数，否则报错。

let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 }; x // x=1,y=2,z={ a: 3, b: 4 }

扩展运算符的解构赋值，不能复制继承自原型对象的属性。

# Symbol

## 概述

一种新的原始数据类型(不是对象，不能new)，表示独一无二的值。

JavaScript的数据类型：undefined、null、布尔值（Boolean）、字符串（String）、数值（Number）、对象（Object）、Symbol。

Symbol 值通过Symbol函数生成。对象属性名的两种类型：字符串、Symbol 类型。

let s = Symbol();

typeof s // "symbol"

注意：Symbol函数前不能使用new命令，否则会报错。

Symbol函数可以接受一个字符串作为参数(参数为对象时转为字符串)，表示对 Symbol 实例的描述，用以区分。

let s1 = Symbol('foo');

s1 // Symbol(foo)

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

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(Symbol唯一性)。

// 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

Symbol 值可以显式转为字符串，也可以转为布尔值，但是不能转为数值。

let sym = Symbol('My symbol');

String(sym) // 'Symbol(My symbol)'

sym.toString() // 'Symbol(My symbol)'

Boolean(sym) // true

!sym // false

Number(sym) // TypeError

## 作为属性名的 Symbol

由于Symbol 值是唯一的，Symbol值用作对象的属性名时，能保证不会出现同名的属性。

通过方括号结构、Object.defineProperty，将对象的属性名指定为一个 Symbol 值。

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!' });

a[mySymbol] // "Hello!"，以上写法都得到同样结果

## 属性名的遍历

Symbol 作为属性名，只能通过Object.getOwnPropertySymbols获取，而后遍历。

Object.getOwnPropertySymbols();返回一个数组，成员是当前对象的所有用作属性名的 Symbol 值。

## Symbol.for()，Symbol.keyFor()

Symbol.for(str)，搜索，如有str的Symbol值，返回这个值；没有则新建一个，并登记在全局环境中供搜索

Symbol()，不登记。

Symbol.keyFor()，返回一个已登记的 Symbol 类型值的key；Symbol()返回undefined。

let s1 = Symbol.for('foo');

let s2 = Symbol.for('foo');

s1 === s2 // true

Symbol.keyFor(s1); //'foo'

Symbol.keyFor(Symbol()); //undefined

# Set 和 Map 数据结构

## Set

### 基本用法

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()构造函数，用来生成 Set 数据结构。可以接受数组（或者具有 iterable 接口的其他数据结构）作为参数。

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

}

const set = new Set([1, 2, 3, 4, 4]);

[...set] // [1, 2, 3, 4]

Set判断值相等类似于精确相等运算符（===），主要的区别是NaN等于自身。

Array.from()，将 Set 结构转为数组；可用于去除数组重复成员。

const array = Array.from(new Set([1,2,1,34]));

Const unq = [...new Set([1,2,1,34])];

### Set 实例的属性和方法

#### Set 实例的属性

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）。

**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。

**遍历方法**

keys()：返回键名的遍历器

values()：返回键值的遍历器

entries()：返回键值对的遍历器

forEach()：使用回调函数遍历每个成员

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。

entries()、forEach()，返回的键名和键值都是set.value

Set的遍历顺序就是插入顺序。使用 Set 保存一个回调函数列表，调用时按照添加顺序调用。

## WeakSet

### 含义

WeakSet 结构与 Set 类似，也是不重复的值的集合。

与 Set 的区别：

1，WeakSet 的成员只能是对象。

2，WeakSet 中的对象都是弱引用(故ES6 规定 WeakSet 不可遍历)。

### 语法

WeakSet()构造函数，可以使用new命令，创建 WeakSet 数据结构。

const ws = new WeakSet();

WeakSet()可接受任何具有 Iterable 接口的对象a作为参数；a的所有成员自动成为 WeakSet的成员(不是对象a)，故a的成员必须是对象。

### WeakSet方法

WeakSet.prototype.add(value)：向 WeakSet 实例添加一个新成员。

WeakSet.prototype.delete(value)：清除 WeakSet 实例的指定成员。

WeakSet.prototype.has(value)：返回一个布尔值，表示某个值是否在 WeakSet 实例之中。

## Map

### 含义和基本用法

Map()构造函数参数：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。

对同一个键多次赋值，后面的值将覆盖前面的值。

读取一个未知的键，返回undefined。

注意：只有对同一个对象的引用(内存地址相同，内存地址绑定)，Map 结构才将其视为同一个键。

使用扩展运算符（...），可将Map转为数组。

[...new Map().set(“hi”,”li”)];

### 实例的属性和操作方法

size，返回 Map 结构的成员总数。

set(key, value)，设置键名key对应的键值为value，返回整个 Map 结构。

get(key)，读取key对应的键值，如果找不到key，返回undefined。

has(key)，返回一个布尔值，表示某个键是否在当前 Map 对象之中。

delete(key)，删除某个键，返回true。如果删除失败，返回false。

clear()，清除所有成员，没有返回值。

### 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。

注意：Map 的遍历顺序就是插入顺序。

## WeakMap

类似WeakSet；键名必须为对象，弱引用。不可遍历。

# Proxy代理

Proxy 用于修改某些操作的默认行为，等同于在语言层面做出修改。

Proxy 构造函数

var proxy = new Proxy(target, handler); //target为拦截对象，handler为拦截行为对象

# Reflect

反射

# Promise 对象

## Promise 的含义

Promise 是一种异步编程解决方案。

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。

## 基本用法

Promise对象是一个构造函数，用来生成Promise实例。

const promise = new Promise(function(resolve, reject) {

// ... some code

if (/\* 异步操作成功 \*/){

resolve(value);

} else {

reject(error);

}

});

Promise构造函数接受一个函数作为参数，该函数的两个参数分别是resolve和reject。它们是两个函数，由 JavaScript 引擎提供，不用自己部署。

resolve函数将Promise对象状态从 pending 变为 resolved；

reject函数将Promise对象的状态从 pending 变为 rejected。

## Promise.prototype.then()

指定resolved状态和rejected状态的回调函数。回调函数在当前脚本所有同步任务执行完才会执行。

promise.then(function(value) {

// success

}, function(error) { //失败回调函数可省。

// failure

});

## 其他方法

Promise.prototype.catch()，指定发生错误时的回调函数。

Promise.prototype.finally()，指定不管 Promise 对象最后状态如何，都会执行的操作。

Promise.all()，将多个 Promise 实例，包装成一个新的 Promise 实例。

Promise.race()，将多个 Promise 实例，包装成一个新的 Promise 实例。

Promise.resolve()，将现有对象转为 Promise 对象。

Promise.reject()，返回一个新的 Promise 实例，该实例的状态为rejected。

Promise.try()，以Promise方式处理函数f(不管同步异步)，用then方法指定下一步流程，用catch方法处理f抛出的错误

# Iterator 和 for...of 循环

Iterator 接口主要供for...of消费。遍历器对象本质上，就是一个指针对象，返回一个包含value和done两个属性的对象。

## 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。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。

ES6 规定，默认的 Iterator 接口部署在数据结构的Symbol.iterator属性，通过[Symbol.iterator]获取。

原生具备 Iterator 接口的数据结构：Array，Map，Set，String，TypedArray，函数的 arguments对象，NodeList 对象。

yield，iterator中next返回值。

## 遍历器对象的 return()，throw()

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throw，或者有break语句或continue语句）。

return方法必须返回一个对象，这是 Generator 规格决定的。

## for...of 循环

遍历部署了Symbol.iterator属性的数据结构，内部调用的是数据结构的Symbol.iterator方法。

for...in循环，只能获得对象的键名，不能直接获取键值。

数组提供内置的forEach方法。

对类似数组的对象，通过Array.from()转换为数组遍历；对普通对象，通过Object.keys()获取key数组，而后遍历获得value，或使用Generator函数重新包装对象。

# Generator 函数的语法

Generator 函数是 ES6 提供的一种异步编程解决方案。

特征：一，function关键字与函数名之间有一个星号(function\* hello(){})；二，函数体内部使用yield表达式，定义不同的内部状态。

调用Generator函数，返回iterator遍历器，而后通过next()方法获取状态。

## yield 表达式

暂停标志。遍历器执行时，返回yield表达式值、或return值。

“惰性求值”，执行到对应的yield表达式时，才计算表达式的值。

yield表达式只能用在 Generator 函数里面。

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；获取对象后调用next()方法时执行。

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。

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。

## 与 Iterator 接口的关系

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。

## 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。

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，可在 Generator 函数开始运行之后，从而调整函数行为。

## 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。

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。

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。

## Generator.prototype.return()

返回给定的值，并且终结遍历 Generator 函数。

如果 Generator 函数内部有try...finally代码块，return方法会推迟到finally代码块执行完再执行。

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

## 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\* () {

// ···

}

};

## 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。用无法获取函数内的this属性。

## 异步操作的同步化表达

在yield表达式中调用异步函数，在异步函数中调用next()方法；如需将异步函数参数传入Generator函数，需在next()方法中传递。

# Generator 函数的异步应用

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。连续的执行就叫做同步。

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）。

# Class 的基本语法

## 简介

通过class关键字，可以定义类。

类的数据类型就是函数，类本身就指向构造函数。

类的所有方法都定义在类的prototype属性上面。Object.assign方法可以很方便地一次向类添加多个方法。在类的实例上面调用方法，其实就是调用原型上的方法。

类的内部所有定义的方法，都是不可枚举的（non-enumerable）。

## 类的实例对象

实例的属性除非显式定义在其本身（即定义在this对象上），否则都是定义在原型上（即定义在class上）。类的所有实例共享一个原型对象。

## Class 的静态方法

static关键字，该方法不会被实例继承，而是直接通过类来调用，这就称为“静态方法”。

如果静态方法包含this关键字，这个this指的是类，而不是实例。

静态方法可以与非静态方法重名。

父类的静态方法，可以被子类继承。

静态方法也是可以从super对象上调用的。

## Class 的静态属性和实例属性

静态属性指的是 Class 本身的属性，即Class.propName，而不是定义在实例对象（this）上的属性。

Foo.prop = 1;

## new.target 属性

new是从构造函数生成实例对象的命令。new.target属性用在构造函数之中，返回new命令作用于的那个构造函数。如果构造函数不是通过new命令调用的，new.target会返回undefined

子类继承父类时，new.target会返回子类。

# Class 的继承

Class 可以通过extends关键字实现继承。

子类必须在constructor方法中调用super方法，否则新建实例时会报错。这是因为子类没有自己的this对象，而是继承父类的this对象，然后对其进行加工。如果不调用super方法，子类就得不到this对象。

## Object.getPrototypeOf()

Object.getPrototypeOf方法可以用来从子类上获取父类。可以使用这个方法判断，一个类是否继承了另一个类。

## super 关键字

可以当作函数使用，也可以当作对象使用。

super作为函数调用时，代表父类的构造函数。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。super()只能用在子类的构造函数之中，用在其他地方就会报错。

super作为对象时，在普通方法中，指向父类的原型对象；在静态方法中，指向父类。

## 类的 prototype 属性和\_\_proto\_\_属性

\_\_proto\_\_属性，指向对应的构造函数的prototype属性。