# 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 "D:\Program Files\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 "D:\Program Files\nodejs\node\_cache" //cache的路径

1. 安装模块，

npm install express -g，安装express到global目录下

npm install；在项目.json文件处使用，自动安装

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

5，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。