

ECMAScript 6 入门

作者：阮一峰

授权：署名-非商用许可证

🔍

目录

- 0.前言
- 1.ECMAScript 6简介
- 2.let 和 const 命令
- 3.变量的解构赋值
- 4.字符串的扩展
- 5.正则的扩展
- 6.数值的扩展
- 7.函数的扩展
- 8.数组的扩展
- 9.对象的扩展
- 10.Symbol
- 11.Set 和 Map 数据结构
- 12.Proxy
- 13.Reflect
- 14.Promise 对象
- 15.Iterator 和 for...of 循环
- 16.Generator 函数的语法
- 17.Generator 函数的异步应用
- 18.async 函数
- 19.Class 的基本语法
- 20.Class 的继承
- 21.Decorator
- 22.Module 的语法
- 23.Module 的加载实现
- 24.编程风格
- 25.读懂规格
- 26.ArrayBuffer
- 27.参考链接

其他

- 源码
- 修订历史
- 反馈意见

正则的扩展

- 1.RegExp 构造函数
- 2.字符串的正则方法
- 3.u 修饰符
- 4.y 修饰符
- 5.sticky 属性
- 6.flags 属性
- 7.s 修饰符：dotAll 模式
- 8.后行断言
- 9.Unicode 属性类
- 10.具名组匹配

在 ES5 中，`RegExp` 构造函数的参数有两种情况。

第一种情况是，参数是字符串，这时第二个参数表示正则表达式的修饰符（flag）。

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

第二种情况是，参数是一个正则表达式，这时会返回一个原有正则表达式的拷贝。

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

但是，ES5 不允许此时使用第二个参数添加修饰符，否则会报错。

```
var regex = new RegExp(/xyz/, 'i');  
// Uncaught TypeError: Cannot supply flags when constructing one RegExp from another
```

ES6 改变了这种行为。如果 `RegExp` 构造函数第一个参数是一个正则对象，那么可以使用第二个参数指定修饰符。而且，返回的正则表达式会忽略原有的正则表达式的修饰符，只使用新指定的修饰符。

```
new RegExp(/abc/ig, 'i').flags  
// "i"
```

上面代码中，原有正则对象的修饰符是 `ig`，它会被第二个参数 `i` 覆盖。

2. 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]`

3. u 修饰符

ES6 对正则表达式添加了 `u` 修饰符，含义为“Unicode 模式”，用来正确处理大于 `\uFFFF` 的 Unicode 字符。也就是说，会正确处理四个字节的 UTF-16 编码。

```
/^uD83D/u.test('\uD83D\uD83D') // false  
/^uD83D/.test('\uD83D\uD83D') // true
```

上面代码中，`\uD83D\uD83D` 是一个四个字节的 UTF-16 编码，代表一个字符。但是，ES5 不支持四个字节的 UTF-16 编码，会将其识别为两个字符，导致第二行代码结果为 `true`。加了 `u` 修饰符以后，ES6 就会识别其为一个字符，所以第一行代码结果为 `false`。

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，含义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
```

上面代码表示，如果不添加 **u** 修饰符，正则表达式就会认为字符串为两个字符，从而匹配失败。

（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
/\u{61}/u.test('a') // true
/\u{20BB7}/u.test('吉') // true
```

上面代码表示，如果不加 **u** 修饰符，正则表达式无法识别 `\u{61}` 这种表示法，只会认为这匹配 61 个连续的 **u**。

（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
/a{2}/u.test('aa') // true
/吉{2}/.test('吉吉') // false
/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$/.test('吉') // false
(/^\S$/u).test('吉') // true
```

上面代码的 **\S** 是预定义模式，匹配所有不是空格的字符。只有加了 **u** 修饰符，它才能正确匹配码点大于 **0xFFFF** 的 Unicode 字符。

利用这一点，可以写出一个正确返回字符串长度的函数。

```
function codePointLength(text) {
  var result = text.match(/[\s\S]/gu);
  return result ? result.length : 0;
}

var s = '吉吉';

s.length // 4
codePointLength(s) // 2
```

（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**。

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

上面代码中，不加 **u** 修饰符，就无法识别非规范的 **K** 字符。

4. 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

```
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
```

上面代码有两个正则表达式，一个使用 **g** 修饰符，另一个使用 **y** 修饰符。这两个正则表达式各执行了两次，第一次执行的时候，两者行为相同，剩余字符串都是 **_aa_a**。由于 **g** 修饰没有位置要求，所以第二次执行会返回结果，而 **y** 修饰符要求匹配必须从头部开始，所以返回 **null**。

如果改一下正则表达式，保证每次都能头部匹配，**y** 修饰符就会返回结果了。

```
var s = 'aaa_aa_a';
var r = /a_/_y;

r.exec(s) // ["aaa_"]
r.exec(s) // ["aa_"]
```

上面代码每次匹配，都是从剩余字符串的头部开始。

使用 **lastIndex** 属性，可以更好地说明 **y** 修饰符。

```
const REGEX = /a/g;

// 指定从2号位置（y）开始匹配
REGEX.lastIndex = 2;

// 匹配成功
const match = REGEX.exec('xaya');

// 在3号位置匹配成功
match.index // 3

// 下一次匹配从4号位开始
REGEX.lastIndex // 4

// 4号位开始匹配失败
REGEX.exec('xaxa') // null
```

上面代码中，**lastIndex** 属性指定每次搜索的开始位置，**g** 修饰符从这个位置开始向后搜索，直到发现匹配为止。

y 修饰符同样遵守 **lastIndex** 属性，但是要求必须在 **lastIndex** 指定的位置发现匹配。

```
const REGEX = /a/y;

// 指定从2号位置开始匹配
REGEX.lastIndex = 2;

// 不是粘连，匹配失败
REGEX.exec('xaya') // null

// 指定从3号位置开始匹配
REGEX.lastIndex = 3;

// 3号位置是粘连，匹配成功
const match = REGEX.exec('xaxa');
match.index // 3
REGEX.lastIndex // 4
```

实际上，**y** 修饰符号隐含了头部匹配的标志 **^**。

```
/b/y.exec('aba')
// null
```

上面代码由于不能保证头部匹配，所以返回 **null**。**y** 修饰符的设计本意，就是让头部匹配的标志 **^** 在全局匹配中都有效。

在 `split` 方法中使用 `y` 修饰符，原字符串必须以分隔符开头。这也意味着，只要匹配成功，数组的第一个成员肯定是空字符串。

```
// 没有找到匹配
'x##'.split(/#/y)
// [ 'x##' ]

// 找到两个匹配
'##x'.split(/#/y)
// [ '', '', 'x' ]
```

后续的分隔符只有紧跟前面的分隔符，才会被识别。

```
'#x#'.split(/#/y)
// [ '', 'x#' ]

'##'.split(/#/y)
// [ '', '', '' ]
```

下面是字符串对象的 `replace` 方法的例子。

```
const REGEX = /a/gy;
'aaxa'.replace(REGEX, '-') // '--xa'
```

上面代码中，最后一个 `a` 因为不是出现在下一次匹配的头部，所以不会被替换。

单单一个 `y` 修饰符对 `match` 方法，只能返回第一个匹配，必须与 `g` 修饰符联用，才能返回所有匹配。

```
'a1a2a3'.match(/a\d/y) // ["a1"]
'a1a2a3'.match(/a\d/gy) // ["a1", "a2", "a3"]
```

`y` 修饰符的一个应用，是从字符串提取 `token`（词元），`y` 修饰符确保了匹配之间不会有漏掉的字符。

```
const TOKEN_Y = /\s*(\+|[0-9]+)\s*/y;
const TOKEN_G = /\s*(\+|[0-9]+)\s*/g;

tokenize(TOKEN_Y, '3 + 4')
// [ '3', '+', '4' ]
tokenize(TOKEN_G, '3 + 4')
// [ '3', '+', '4' ]

function tokenize(TOKEN_REGEX, str) {
  let result = [];
  let match;
  while (match = TOKEN_REGEX.exec(str)) {
    result.push(match[1]);
  }
  return result;
}
```

上面代码中，如果字符串里面没有非法字符，`y` 修饰符与 `g` 修饰符的提取结果是一样的。但是，一旦出现非法字符，两者的行为就不一样了。

```
tokenize(TOKEN_Y, '3x + 4')
// [ '3' ]
tokenize(TOKEN_G, '3x + 4')
// [ '3', '+', '4' ]
```

上面代码中，`g` 修饰符会忽略非法字符，而 `y` 修饰符不会，这样就很容易发现错误。

5. sticky 属性

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

```
var r = /hello\d/y;  
r.sticky // true
```

6. 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'
```

7. s 修饰符：dotAll 模式

正则表达式中，点（.）是一个特殊字符，代表任意的单个字符，但是行终止符（line terminator character）除外。

以下四个字符属于“行终止符”。

- U+000A 换行符（\n）
- U+000D 回车符（\r）
- U+2028 行分隔符（line separator）
- U+2029 段分隔符（paragraph separator）

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

上面代码中，因为 . 不匹配 \n，所以正则表达式返回 **false**。

但是，很多时候我们希望匹配的是任意单个字符，这时有一种变通的写法。

```
/foo[^]bar/.test('foo\nbar')  
// true
```

这种解决方案毕竟不太符合直觉，所以现在有一个[提案](#)，引入 **/s** 修饰符，使得 . 可以匹配任意单个字符。

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

这被称为 **dotAll** 模式，即点（dot）代表一切字符。所以，正则表达式还引入了一个 **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。

```
const re = /foo.bar/s;  
// 另一种写法  
// const re = new RegExp('foo.bar', 's');  
  
re.test('foo\nbar') // true  
re.dotAll // true  
re.flags // 's'
```

/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。

8. 后行断言

JavaScript 语言的正则表达式，只支持先行断言（lookahead）和先行否定断言（negative lookahead），不支持后行断言（lookbehind）和后行否定断言（negative lookbehind）。目前，有一个[提案](#)，引入后行断言，V8 引擎 4.9 版已经支持。

“先行断言”指的是，`x` 只有在 `y` 前面才匹配，必须写成 `/x(?=y)/`。比如，只匹配百分号之前的数字，要写成 `/\d+(?=%)/`。“先行否定断言”指的是，`x` 只有在 `y` 前面才匹配，必须写成 `/x(?!y)/`。比如，只匹配不在百分号之前的数字，要写成 `/\d+(?!%)/`。

```
/\d+(?=%)/.exec('100% of US presidents have been male') // ["100"]
/\d+(?!%)/.exec('that's all 44 of them') // ["44"]
```

上面两个字符串，如果互换正则表达式，就不会得到相同结果。另外，还可以看到，“先行断言”括号之中的部分（`(?=%)`），是不计入返回结果的。

“后行断言”正好与“先行断言”相反，`x` 只有在 `y` 后面才匹配，必须写成 `/(?<=y)x/`。比如，只匹配美元符号之后的数字，要写成 `/(?<=\$)\d+/`。“后行否定断言”则与“先行否定断言”相反，`x` 只有不在 `y` 后面才匹配，必须写成 `/(?<!y)x/`。比如，只匹配不在美元符号后面的数字，要写成 `/(?<!\$)\d+/`。

```
/(?<=\$)\d+/.exec('Benjamin Franklin is on the $100 bill') // ["100"]
/(?<!\$)\d+/.exec('it's is worth about €90') // ["90"]
```

上面的例子中，“后行断言”的括号之中的部分（`(?<=\$)`），也是不计入返回结果。

下面的例子是使用后行断言进行字符串替换。

```
const RE_DOLLAR_PREFIX = /(?!<=\$)foo/g;
'$foo %foo foo'.replace(RE_DOLLAR_PREFIX, 'bar');
// '$bar %foo foo'
```

上面代码中，只有在美元符号后面的 `foo` 才会被替换。

“后行断言”的实现，需要先匹配 `/(?<=y)x/` 的 `x`，然后再回到左边，匹配 `y` 的部分。这种“先右后左”的执行顺序，与所有其他正则操作相反，导致了一些不符合预期的行为。

首先，“后行断言”的组匹配，与正常情况下结果是不一样的。

```
/(?<=(\d+)(\d+))$/ .exec('1053') // ["", "1", "053"]
/^(?<=(\d+)(\d+))$/ .exec('1053') // ["1053", "105", "3"]
```

上面代码中，需要捕捉两个组匹配。没有“后行断言”时，第一个括号是贪婪模式，第二个括号只能捕获一个字符，所以结果是 `105` 和 `3`。而“后行断言”时，由于执行顺序是从右到左，第二个括号是贪婪模式，第一个括号只能捕获一个字符，所以结果是 `1` 和 `053`。

其次，“后行断言”的反斜杠引用，也与通常的顺序相反，必须放在对应的那个括号之前。

```
/(?<=(o)d\1)r/.exec('hodor') // null
/(?<=\1d(o))r/.exec('hodor') // ["r", "o"]
```

上面代码中，如果后行断言的反斜杠引用（`\1`）放在括号的后面，就不会得到匹配结果，必须放在前面才可以。因为后行断言是先从左到右扫描，发现匹配以后再回过头，从右到左完成反斜杠引用。

9. Unicode 属性类

目前，有一个[提案](#)，引入了一种新的类的写法 `\p{...}` 和 `\P{...}`，允许正则表达式匹配符合 Unicode 某种属性的所有字符。

```
const regexGreekSymbol = /\p{Script=Greek}/u;
regexGreekSymbol.test('π') // true
```

上面代码中，`\p{Script=Greek}` 指定匹配一个希腊文字母，所以匹配 `π` 成功。

对于某些属性，可以只写属性名。

```
\p{UnicodePropertyName}
```

`\P{...}` 是 `\p{...}` 的反向匹配，即匹配不满足条件的字符。

注意，这两种类只对 **Unicode** 有效，所以使用的时候一定要加上 **u** 修饰符。如果不加 **u** 修饰符，正则表达式使用 **\p** 和 **\P** 会报错，ECMAScript 预留了这两个类。

由于 Unicode 的各种属性非常多，所以这种新的类的表达能力非常强。

```
const regex = /^p{Decimal_Number}+$/u;
regex.test('1234567890123456') // true
```

上面代码中，属性类指定匹配所有十进制字符，可以看到各种字型的十进制字符都会匹配成功。

`\p{Number}` 甚至能匹配罗马数字。

```
// 匹配所有数字
const regex = /^p{Number}+$/u;
regex.test('231%%') // true
regex.test('㊟㊿㊿') // true
regex.test(' I II III IV V VI VII VIII IX X XI XII') // true
```

下面是其他一些例子。

```
// 匹配各种文字的所有字母，等同于 Unicode 版的 \w
[\p{Alphabetic}\p{Mark}\p{Decimal_Number}\p{Connector_Punctuation}\p{Join_Control}]

// 匹配各种文字的所有非字母的字符，等同于 Unicode 版的 \W
[^\p{Alphabetic}\p{Mark}\p{Decimal_Number}\p{Connector_Punctuation}\p{Join_Control}]

// 匹配所有的箭头字符
const regexArrows = /^[\p{Block=Arrows}]+$/u;
regexArrows.test('←→↔↕↖↗↘↙↞↠↡↢↣↤↥↦↧↨↩↪↫↬↭↮↯↰↱↲↳↴↵↶↷↸↹↺↻') // true
```

10. 具名组匹配

简介

正则表达式使用圆括号进行组匹配。

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 matchObj = RE_DATE.exec('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引用，要是组的顺序变了，引用的时候就必须修改序号。

现在有一个“具名组匹配”（Named Capture Groups）的提案，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”（`?(<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。

具名组匹配等于为每一组匹配加上了 ID，便于描述匹配的目的。如果组的顺序变了，也不用改变匹配后的处理代码。

如果具名组没有匹配，那么对应的 `groups` 对象属性会是 `undefined`。

```
const RE_OPT_A = /^(?<as>a+)?$/;
const matchObj = RE_OPT_A.exec('');

matchObj.groups.as // undefined
'as' in matchObj.groups // true
```

上面代码中，具名组 `as` 没有找到匹配，那么 `matchObj.groups.as` 属性值就是 `undefined`，并且 `as` 这个键名在 `groups` 是始终存在的。

解构赋值和替换

有了具名组匹配以后，可以使用解构赋值直接从匹配结果上为变量赋值。

```
let {groups: {one, two}} = /^(?<one>.*):(?<two>.*$)/u.exec('foo:bar');
one // foo
two // bar
```

字符串替换时，使用 `$(组名)` 引用具名组。

```
let re = /(?(<year>\d{4})-(?(<month>\d{2})-(?(<day>\d{2}))/u;

'2015-01-02'.replace(re, '$<day>/${<month>}/${<year>}')
// '02/01/2015'
```

上面代码中，`replace` 方法的第二个参数是一个字符串，而不是正则表达式。

`replace` 方法的第二个参数也可以是函数，该函数的参数序列如下。

```
'2015-01-02'.replace(re, (
  matched, // 整个匹配结果 2015-01-02
  capture1, // 第一个组匹配 2015
  capture2, // 第二个组匹配 01
  capture3, // 第三个组匹配 02
  position, // 匹配开始的位置 0
  S, // 原字符串 2015-01-02
  groups // 具名组构成的一个对象 {year, month, day}
) => {
  let {day, month, year} = args[args.length - 1];
  return `${day}/${month}/${year}`;
});
```

具名组匹配在原来的基础上，新增了最后一个函数参数：具名组构成的一个对象。函数内部可以直接对这个对象进行解构赋值。

引用

如果要在正则表达式内部引用某个“具名组匹配”，可以使用 `\k<组名>` 的写法。

```
const RE_TWICE = /^(?<word>[a-z]+)!\k<word>$;/;
RE_TWICE.test('abc!abc') // true
RE_TWICE.test('abc!ab') // false
```

数字引用（`\1`）依然有效。

```
const RE_TWICE = /^(?<word>[a-z]+)!\1$/;
RE_TWICE.test('abc!abc') // true
RE_TWICE.test('abc!ab') // false
```

这两种引用语法还可以同时使用。

```
const RE_TWICE = /^(?<word>[a-z]+)!\k<word>!\1$/;
RE_TWICE.test('abc!abc!abc') // true
RE_TWICE.test('abc!abc!ab') // false
```

留言

