

# 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.字符串的新增方法
- 6.正则的扩展
- 7.数值的扩展
- 8.函数的扩展
- 9.数组的扩展
- 10.对象的扩展
- 11.对象的新增方法
- 12.Symbol
- 13.Set 和 Map 数据结构
- 14.Proxy
- 15.Reflect
- 16.Promise 对象
- 17.Iterator 和 for...of 循环
- 18.Generator 函数的语法
- 19.Generator 函数的异步应用
- 20.async 函数
- 21.Class 的基本语法
- 22.Class 的继承
- 23.Module 的语法
- 24.Module 的加载实现
- 25.编程风格
- 26.读懂规格
- 27.异步遍历器
- 28.ArrayBuffer
- 29.最新提案
- 30.Decorator
- 31.参考链接

## 其他

- 源码
- 修订历史
- 反馈意见

# 字符串的新增方法

- 1.String.fromCodePoint()
- 2.String.raw()
- 3.实例方法：codePointAt()

4.实例方法: `normalize()`

5.实例方法: `includes()`, `startsWith()`, `endsWith()`

6.实例方法: `repeat()`

7.实例方法: `padStart()`, `padEnd()`

8.实例方法: `trimStart()`, `trimEnd()`

9.实例方法: `matchAll()`

《ES6 实战教程》 深入学习一线大厂必备 ES6 技能。VIP 教程限时免费领取。 [← 立即查看](#)

本章介绍字符串对象的新增方法。

---

## 1. String.fromCodePoint()

ES5 提供 `String.fromCharCode()` 方法，用于从 Unicode 码点返回对应字符，但是这个方法不能识别码点大于 `0xFFFF` 的字符。

```
String.fromCharCode(0x20BB7)
// "𐄇"
```

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，正好与下面的 `codePointAt()` 方法相反。

```
String.fromCodePoint(0x20BB7)
// "𐄇"
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。

---

## 2. String.raw()

ES6 还为原生的 `String` 对象，提供了一个 `raw()` 方法。该方法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，往往用于模板字符串的处理方法。

```
String.raw`Hi\n${2+3}!`
// 实际返回 "Hi\\n5!", 显示的是转义后的结果 "Hi\n5!"

String.raw`Hi\u000A!`;
// 实际返回 "Hi\\u000A!", 显示的是转义后的结果 "Hi\u000A!"
```

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`Hi\\n` === "Hi\\\\n" // true
```

`String.raw()` 方法可以作为处理模板字符串的基本方法，它会将所有变量替换，而且对斜杠进行转义，方便下一步作为字符串来使用。

`String.raw()` 本质上是一个正常的函数，只是专用于模板字符串的标签函数。如果写成正常函数的形式，它的第一个参数，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，对应模板字符串解析后的值。

```
// `foo${1 + 2}bar`  
// 等同于  
String.raw({ raw: ['foo', 'bar'] }, 1 + 2) // "foo3bar"
```

上面代码中，`String.raw()` 方法的第一个参数是一个对象，它的 `raw` 属性等同于原始的模板字符串解析后得到的数组。

作为函数，`String.raw()` 的代码实现基本如下。

```
String.raw = function (strings, ...values) {  
  let output = '';  
  let index;  
  for (index = 0; index < values.length; index++) {  
    output += strings.raw[index] + values[index];  
  }  
  
  output += strings.raw[index]  
  return output;  
}
```

---

### 3. 实例方法：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 = "吉";  
  
s.length // 2  
s.charAt(0) // ''  
s.charAt(1) // ''  
s.charCodeAt(0) // 55362  
s.charCodeAt(1) // 57271
```

上面代码中，汉字“吉”（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 = '吉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“吉a”视为三个字符，`codePointAt` 方法在第一个字符上，正确地识别了“吉”，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“吉”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 = '吉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 = '吉a';
for (let ch of s) {
  console.log(ch.codePointAt(0).toString(16));
}
// 20bb7
// 61
```

另一种方法也可以，使用扩展运算符（`...`）进行展开运算。

```
let arr = [...'吉a']; // arr.length === 2
arr.forEach(
  ch => 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("吉") // true
is32Bit("a") // false
```

---

## 4. 实例方法：normalize()

许多欧洲语言有语调符号和重音符号。为了表示它们，Unicode 提供了两种方法。一种是直接提供带重音符号的字符，比如 ö（\u01D1）。另一种是提供合成符号（combining character），即原字符与重音符号的合成，两个字符合成一个字符，比如 o（\u004F）和 ˇ（\u030C）合成 ö（\u004F\u030C）。

这两种表示方法，在视觉和语义上都等价，但是 JavaScript 不能识别。

```
'\u01D1' === '\u004F\u030C' //false
```

```
'\u01D1'.length // 1
'\u004F\u030C'.length // 2
```

上面代码表示，JavaScript 将合成字符视为两个字符，导致两种表示方法不相等。

ES6 提供字符串实例的 `normalize()` 方法，用来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` 方法可以接受一个参数来指定 `normalize` 的方式，参数的四个可选值如下。

- **NFC**，默认参数，表示“标准等价合成”（Normalization Form Canonical Composition），返回多个简单字符的合成字符。所谓“标准等价”指的是视觉和语义上的等价。
- **NFD**，表示“标准等价分解”（Normalization Form Canonical Decomposition），即在标准等价的前提下，返回合成字符分解的多个简单字符。
- **NFKC**，表示“兼容等价合成”（Normalization Form Compatibility Composition），返回合成字符。所谓“兼容等价”指的是语义上存在等价，但视觉上不等价，比如“囍”和“喜喜”。（这只是用来举例，`normalize` 方法不能识别中文。）
- **NFKD**，表示“兼容等价分解”（Normalization Form Compatibility Decomposition），即在兼容等价的前提下，返回合成字符分解的多个简单字符。

```
'\u004F\u030C'.normalize('NFC').length // 1  
'\u004F\u030C'.normalize('NFD').length // 2
```

上面代码表示，**NFC** 参数返回字符的合成形式，**NFD** 参数返回字符的分解形式。

不过，`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。

---

## 5. 实例方法：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。

---

## 6. 实例方法：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"
```

---

## 7. 实例方法：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

如果原字符串的长度，等于或大于最大长度，则字符串补全不生效，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最大长度，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"
```

---

## 8. 实例方法：trimStart(), trimEnd()

ES2019 对字符串实例新增了 `trimStart()` 和 `trimEnd()` 这两个方法。它们的行为与 `trim()` 一致，`trimStart()` 消除字符串头部的空格，`trimEnd()` 消除尾部的空格。它们返回的都是新字符串，不会修改原始字符串。

```
const s = '  abc  ';

s.trim() // "abc"
s.trimStart() // "abc  "
s.trimEnd() // "  abc"
```

上面代码中，`trimStart()` 只消除头部的空格，保留尾部的空格。`trimEnd()` 也是类似行为。

除了空格键，这两个方法对字符串头部（或尾部）的 tab 键、换行符等不可见的空白符号也有效。

浏览器还部署了额外的两个方法，`trimLeft()` 是 `trimStart()` 的别名，`trimRight()` 是 `trimEnd()` 的别名。

---

## 9. 实例方法：matchAll()

`matchAll()` 方法返回一个正则表达式在当前字符串的所有匹配，详见《正则的扩展》的一章。

