正则的扩展

---

1. RegExp构造函数

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，参数是一个正则表示式，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
```

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

上面代码中，`\uD83D\uDC2A`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 = '𠮷';

/^.$/.test(s) // false
/^.$/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。

```
/\u{61}/.test('a') // false
/\u{61}/u.test('a') // true
/\u{20BB7}/u.test('𠮷') // 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
/𠮷{2}/.test('𠮷𠮷') // false
/𠮷{2}/u.test('𠮷𠮷') // true
```

另外，只有在使用`u`修饰符的情况下，Unicode表达式当中的大括号才会被正确解读，否则会被解读为量词。

```
/^\u{3}$/.test('uuu') // true
```

上面代码中，由于正则表达式没有`u`修饰符，所以大括号被解读为量词。加上`u`修饰符，就会被解读为Unicode表达式。

（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$/.test('𠮷') // false
/^\S$/u.test('𠮷') // 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 = '𠮷𠮷';

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
```

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
```

进一步说，`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. RegExp.escape()

字符串必须转义，才能作为正则模式。

```
function escapeRegExp(str) {
  return str.replace(/[\-\[\]\/\{\}\(\)\*\+\?\.\\\^\$\|]/g, '\\$&');
}

let str = '/path/to/resource.html?search=query';
escapeRegExp(str)
// "\/path\/to\/resource\.html\?search=query"
```

上面代码中，`str`是一个正常字符串，必须使用反斜杠对其中的特殊字符转义，才能用来作为一个正则匹配的模式。

已经有提议将这个需求标准化，作为RegExp对象的静态方法RegExp.escape()，放入ES7。2015年7月31日，TC39认为，这个方法有安全风险，又不愿这个方法变得过于复杂，没有同意将其列入ES7，但这不失为一个真实的需求。

```
RegExp.escape('The Quick Brown Fox');
// "The Quick Brown Fox"

RegExp.escape('Buy it. use it. break it. fix it.');
// "Buy it\. use it\. break it\. fix it\."

RegExp.escape('(*.*)');
// "\(\*\.\*\)"
```

字符串转义以后，可以使用RegExp构造函数生成正则模式。

```
var str = 'hello. how are you?';
var regex = new RegExp(RegExp.escape(str), 'g');
assert.equal(String(regex), '/hello\. how are you\?/g');
```

目前，该方法可以用上文的`escapeRegExp`函数或者垫片模块regexp.escape实现。

```
var escape = require('regexp.escape');
escape('hi. how are you?');
// "hi\\. how are you\\?"
```

---

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，在ES7加入后行断言。V8引擎4.9版已经支持，Chrome浏览器49版打开"experimental JavaScript features"开关（地址栏键入`about:flags`），就可以使用这项功能。

"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`只有不在`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，就会匹配失败。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。

"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。

留言