第20讲 正则表达式

学习目标

- [ ] 掌握正则表达式
- [ ] 能在开发中自定义正则表达式

概念

介绍：正则表达式(regular expression)是一个描述字符模式的对象，是一种表达文本模式（即字符串结构）的方法，有点像字符串的模板，常常用来按照“给定模式”匹配文本。

正则表达式能够进行强大的'模式匹配'和'文本检索与替换'功能,前端往往有大量的表单数据校检的工作,采用正则表达式会使得数据校检的工作量大大减轻。比如，正则表达式给出一个 Email 地址的模式，然后用它来确定一个字符串是否为 Email 地址。

创建正则表达式

使用RegExp构造函数

语法：

```
var reg = new RegExp(pattern[, flags])
```

参数：

- pattern 正则表达式的文本。
- flags 模式修饰符，可包含下列任何字符的组合:
  - i:case-insensitive,忽略大小写
  - g:global,表示全局匹配
  - m:multiline 表示多行匹配（将开始和结束字符(^ and $)视为在多行上工作。换句话说，匹配每一行的开头或结尾each line (由\n或者\r 分隔)，而不仅仅是整个输入字符串的开头或结尾。）

例子：

```
var reg = new RegExp('study');
var reg = new RegExp('\\d\\w+');
var reg = new RegExp('xyz', 'i');
```

注意：如果正则模式是一个空字符串，则匹配所有字符串。

使用字面量（推荐）

语法：不使用引号以斜杠表示开始和结束。并在后面设置对应的修饰符

```
var reg= /xyz/;
var reg = /xyz/gi;
```

注意：上面两种创建方法结果是等价的，都新建了一个新的正则表达式对象。它们的主要区别是：字面量方法在引擎编译代码时，就会新建正则表达式，构造函数方法在运行时新建正则表达式，所以字面量方法的效率较高。并且比较便利和直观，所以实际应用中，基本上都采用字面量定义正则表达式。

实例属性

介绍：正则对象的实例属性分成两类。

一类是修饰符相关，返回一个布尔值，表示对应的修饰符是否设置。

- `RegExp.prototype.ignoreCase` (只读)：返回一个布尔值，表示是否设置了i修饰符。
- `RegExp.prototype.global` (只读)：返回一个布尔值，表示是否设置了g修饰符。
- `RegExp.prototype.multiline` (只读)：返回一个布尔值，表示是否设置了m修饰符。

另一类是与修饰符无关的属性，主要是下面两个。

- `RegExp.prototype.lastIndex`：返回一个数值，表示下一次开始搜索的位置。该属性可读写，但是只在表达式带有g修饰符并进行连续搜索时有意义。

> 正则实例对象的lastIndex属性不仅可读，还可写。设置了g修饰符的时候，只要手动设置了lastIndex的值，就会从指定位置开始匹配。

- `RegExp.prototype.source` (只读)：返回正则表达式的字符串形式（不包括反斜杠）

```
var r = /abc/igm;

r.lastIndex // 0
r.source // "abc"
```

实例方法

- `RegExp.prototype.test()`：正则实例对象的test方法返回一个布尔值，表示当前模式是否能匹配参数字符串。

```
// 验证参数字符串之中是否包含cat，结果返回true。
/cat/.test('cats and dogs') // true
```

> 如果正则表达式带有g修饰符，则每一次test方法都从上一次结束的位置开始向后匹配。

```
// 带有g修饰符时，可以通过正则对象的lastIndex属性指定开始搜索的位置。
var r = /x/g;
var s = '_x_x';

r.lastIndex // 0
r.test(s) // true

r.lastIndex // 2
r.test(s) // true

r.lastIndex // 4
r.test(s) // false
```

> 上面代码的正则表达式使用了g修饰符，表示是全局搜索，会有多个结果。接着，三次使用test方法，每一次开始搜索的位置都是上一次匹配的后一个位置。

例子：

```
var r = /x/g;
var s = '_x_x';
// 通过lastIndex属性指定从字符串的第五个位置开始搜索，这个位置是没有字符的，所以返回false。
r.lastIndex = 4;
r.test(s) // false
```

- `RegExp.prototype.exec()`：正则实例对象的exec方法，用来返回匹配结果。如果发现匹配，就返回一个数组，成员是匹配成功的子字符串，否则返回null。

```
var s = '_x_x';
var r1 = /x/;
var r2 = /y/;
// 正则对象r1匹配成功，返回一个数组，成员是匹配结果；正则对象r2匹配失败，返回null。
r1.exec(s) // ["x"]
r2.exec(s) // null
```

注意：如果正则表示式包含圆括号（即含有“组匹配”），则返回的数组会包括多个成员。第一个成员是整个匹配成功的结果，后面的成员就是圆括号对应的匹配成功的组。也就是说，第二个成员对应第一个括号，第三个成员对应第二个括号，以此类推。整个数组的length属性等于组匹配的数量再加1。

```
var s = '_x_x';
var r = /_(x)/;
// 第一个成员是整个匹配的结果，第二个成员是圆括号匹配的结果。
r.exec(s) // ["_x", "x"]
```

补充：exec方法的返回数组还包含以下两个属性：

- input：整个原字符串。
- index：整个模式匹配成功的开始位置（从0开始计数）。

```
var r = /a(b+)a/;
var arr = r.exec('_abbba_aba_');

arr // ["abbba", "bbb"]

// index属性等于1，是因为从原字符串的第二个位置开始匹配成功。
arr.index // 1
arr.input // "_abbba_aba_"
```

如果正则表达式加上g修饰符，则可以使用多次exec方法，下一次搜索的位置从上一次匹配成功结束的位置开始。

```
var reg = /a/g;
var str = 'abc_abc_abc'

var r1 = reg.exec(str);
r1 // ["a"]
r1.index // 0
reg.lastIndex // 1

var r2 = reg.exec(str);
r2 // ["a"]
r2.index // 4
reg.lastIndex // 5

var r3 = reg.exec(str);
r3 // ["a"]
r3.index // 8
reg.lastIndex // 9

var r4 = reg.exec(str);
r4 // null
reg.lastIndex // 0
```

> 上面代码连续用了四次exec方法，前三次都是从上一次匹配结束的位置向后匹配。当第三次匹配结束以后，整个字符串已经到达尾部，匹配结果返回null，正则实例对象的lastIndex属性也重置为0，意味着第四次匹配将从头开始。

利用g修饰符允许多次匹配的特点，可以用一个循环完成全部匹配。

```
var reg = /a/g;
var str = 'abc_abc_abc'

while(true) {
  var match = reg.exec(str);
  // 只要exec方法不返回null，就会一直循环下去，每次输出匹配的位置和匹配的文本。
  if (!match) break;
  console.log('#' + match.index + ':' + match[0]);
}
// #0:a
// #4:a
// #8:a
```

匹配规则

介绍： 正则表达式的规则很复杂，一般常用规则如下：

字面量字符和元字符

概念：大部分字符在正则表达式中，就是字面的含义，比如 `/a/` 匹配a， `/b/` 匹配b。如果在正则表达式之中，某个字符只表示它字面的含义（就像前面的a和b），那么它们就叫做“字面量字符”。

例子：

```
// 正则表达式的dog，就是字面量字符，所以/dog/匹配old dog，因为它就表示d、o、g三个字母连在一起。
/dog/.test('old dog') // true
```

注意：除了字面量字符以外，还有一部分字符有特殊含义，不代表字面的意思。它们叫做“元字符”（metacharacters），主要有以下几个。

1. 点字符 `.`：匹配除回车（\r）、换行(\n)、行分隔符（\u2028）和段分隔符（\u2029）以外的所有字符。注意，对于码点大于0xFFFF的 Unicode 字符，点字符不能正确匹配，会认为这是两个字符。

```
//c.t匹配c和t之间包含任意一个字符的情况，只要这三个字符在同一行，比如cat、c2t、c-t等等，但是不匹配coot。
/c.t/
```

1. 位置字符：用来提示字符所处的位置，主要有两个字符。

- `^` 表示字符串的开始位置
- `$` 表示字符串的结束位置

```
// test必须出现在开始位置
/^test/.test('test123') // true

// test必须出现在结束位置
/test$/.test('new test') // true

// 从开始位置到结束位置只有test
/^test$/.test('test') // true
/^test$/.test('test test') // false
```

1. 选择符 `|`：在正则表达式中表示“或关系”（OR），即 `cat|dog` 表示匹配 `cat或dog`。

```
// 正则表达式指定必须匹配11或22。
/11|22/.test('911') // true
```

多个选择符可以联合使用。

```
// 匹配fred、barney、betty之中的一个
/fred|barney|betty/

// 选择符会包括它前后的多个字符，比如/ab|cd/指的是匹配ab或者cd，而不是指匹配b或者c。如果想修改这个行为，可以使用圆括号。
// a和b之间有一个空格或者一个制表符。
/a( |\t)b/.test('a\tb') // true
```

> 其他的元字符还包括\、*、+、?、()、[]、{}等，将在下文解释。

1. 转义符：`\` 正则表达式中那些有特殊含义的元字符，如果要匹配它们本身，就需要在它们前面要加上反斜杠。比如要匹配 `+`，就要写成 `\+`。

```
// 因为加号是元字符，不代表自身
/1+1/.test('1+1')// false
// 使用反斜杠对加号转义，就能匹配成功
/1\+1/.test('1+1')// true
```

注意：

- 正则表达式中，需要反斜杠转义的，一共有12个字符：`^`、`.`、`[`、`$`、`(`、`)`、`|`、`*`、`+`、`?`、`{`和`\\`。

- 需要特别注意的是，如果使用RegExp方法生成正则对象，转义需要使用两个斜杠，因为字符串内部会先转义一次。

```
(new RegExp('1\+1')).test('1+1')// false
// RegExp作为构造函数，参数是一个字符串。但是，在字符串内部，反斜杠也是转义字符，所以它会先被反斜杠转义一次，然后再被正则表达式转义一次，因此需要两个反斜杠转义。
(new RegExp('1\\+1')).test('1+1')// true
```

特殊字符

介绍：正则表达式对一些不能打印的特殊字符，提供了表达方法。

- `\n` 匹配换行键。
- `\r` 匹配回车键。
- `\t` 匹配制表符 tab（U+0009）。
- `\0` 匹配null字符（U+0000）。
- `\xhh` 匹配一个以两位十六进制数（\x00-\xFF）表示的字符。
- `\uhhhh` 匹配一个以四位十六进制数（\u0000-\uFFFF）表示的 Unicode 字符。

字符类

介绍：字符类（class）表示有一系列字符可供选择，只要匹配其中一个就可以了。所有可供选择的字符都放在方括号内，比如 `[xyz]` 表示x、y、z之中任选一个匹配。

```
// 字符串hello world不包含a、b、c这三个字母中的任一个，所以返回false
/[abc]/.test('hello world') // false
// 字符串apple包含字母a
/[abc]/.test('apple') // true
```

有两个字符在字符类中有特殊含义。

1. 脱字符 `^`：如果方括号内的第一个字符是[^]，则表示除了字符类之中的字符，其他字符都可以匹配。比如，`[^xyz]` 表示除了x、y、z之外都可以匹配。

```
// 字符串hello world不包含字母a、b、c中的任一个，所以返回true
/[^abc]/.test('hello world') // true
// 字符串bbc不包含a、b、c以外的字母，
/[^abc]/.test('bbc') // false
```

注意：

- 如果方括号内没有其他字符，即只有[^]，就表示匹配一切字符，其中包括换行符。相比之下，点号作为元字符（.）是不包括换行符的。

```
var s = 'Please yes\nmake my day!';
// 字符串s含有一个换行符，点号不包括换行符，所以第一个正则表达式匹配失败
s.match(/yes.*day/) // null
// 第二个正则表达式[^]包含一切字符，所以匹配成功。
s.match(/yes[^]*day/) // [ 'yes\nmake my day']
```

- 脱字符只有在字符类的第一个位置才有特殊含义，否则就是字面含义。

1. 连字符 `-`：某些情况下，对于连续序列的字符，连字符 `-` 用来提供简写形式，表示字符的连续范围。比如，`[abc]` 可以写成 `[a-c]`，`[0123456789]` 可以写成 `[0-9]`，同理 `[A-Z]` 表示26个大写字母。

```
// 当连字号（dash）不出现在方括号之中，就不具备简写的作用
/a-z/.test('b') // false
// 只有当连字号用在方括号之中，才表示连续的字符序列
/[a-z]/.test('b') // true
```

以下都是合法的字符类简写形式。

```
[0-9.,]
[0-9a-fA-F]
[a-zA-Z0-9-]
// [1-31]，不代表1到31，只代表1到3。
[1-31]
```

连字符还可以用来指定 Unicode 字符的范围。

```
// \u0128-\uFFFF表示匹配码点在0128到FFFF之间的所有字符。
var str = "\u0130\u0131\u0132";
/[\u0128-\uFFFF]/.test(str)
// true
```

注意：不要过分使用连字符，设定一个很大的范围，否则很可能选中意料之外的字符。最典型的例子就是[A-z]，表面上它是选中从大写的A到小写的z之间52个字母，但是由于在 ASCII 编码之中，大写字母与小写字母之间还有其他字符，结果就会出现意料之外的结果。

```
// 由于反斜杠（\）的ASCII码在大写字母与小写字母之间，结果会被选中
/[A-z]/.test('\\') // true
```

预定义模式

介绍：预定义模式指的是某些常见模式的简写方式。

- `\d` 匹配0-9之间的任一数字，相当于 `[0-9]`。
- `\D` 匹配所有0-9以外的字符，相当于 `[^0-9]`。
- `\w` 匹配任意的字母、数字和下划线，相当于 `[A-Za-z0-9_]`。
- `\W` 除所有字母、数字和下划线以外的字符，相当于 `[^A-Za-z0-9_]`。
- `\s` 匹配空格（包括换行符、制表符、空格符等），相等于 `[ \t\r\n\v\f]`。
- `\S` 匹配非空格的字符，相当于 `[^ \t\r\n\v\f]`。
- `\b` 匹配词的边界。
- `\B` 匹配非词边界，即在词的内部。

例子：

```
// \s  表示空格
/\s\w*/.exec('hello world') // [" world"]

// \b  表示词的边界,world的词首必须独立（词尾是否独立未指定）
/\bworld/.test('hello world') // true
/\bworld/.test('hello-world') // true
/\bworld/.test('helloworld') // false

// \B 表示非词的边界，只有world的词首不独立，才会匹配
/\Bworld/.test('hello-world') // false
/\Bworld/.test('helloworld') // true
```

通常，正则表达式遇到换行符（\n）就会停止匹配。

```
var html = "<b>Hello</b>\n<i>world!</i>";

/.*/.exec(html)[0] // "<b>Hello</b>"

var html = "<b>Hello</b>\n<i>world!</i>";
// 字符串html包含一个换行符，结果点字符（.）不匹配换行符，导致匹配结果可能不符合原意。这时使用\s字符类，就能包括换行符。
/[\S\s]*/.exec(html)[0]
// "<b>Hello</b>\n<i>world!</i>" [\S\s]指代一切字符。
```

重复类

介绍：模式的精确匹配次数，使用大括号 `{}` 表示。 `{n}` 表示恰好重复n次， `{n,}` 表示至少重复n次，`{n,m}` 表示重复不少于n次，不多于m次。

```
/lo{2}k/.test('look') // true
/lo{2,5}k/.test('looook') // true
上面代码中，第一个模式指定o连续出现2次，第二个模式指定o连续出现2次到5次之间。
```

量词符

介绍：量词符用来设定某个模式出现的次数。

- `?` 问号表示某个模式出现0次或1次，等同于{0, 1}。
- `*` 星号表示某个模式出现0次或多次，等同于{0,}。
- `+` 加号表示某个模式出现1次或多次，等同于{1,}。

```
// t 出现0次或1次
/t?est/.test('test') // true
/t?est/.test('est') // true

// t 出现1次或多次
/t+est/.test('test') // true
/t+est/.test('ttest') // true
/t+est/.test('est') // false

// t 出现0次或多次
/t*est/.test('test') // true
/t*est/.test('ttest') // true
/t*est/.test('tttest') // true
/t*est/.test('est') // true
```

例子：
默认情况下都是最大可能匹配，即匹配直到下一个字符不满足匹配规则为止。这被称为贪婪模式。

```
var s = 'aaa';
s.match(/a+/) // ["aaa"]
// /a+/，表示匹配1个a或多个a，因为默认是贪婪模式，会一直匹配到字符a不出现为止，所以匹配结果是3个a。
```

如果想将贪婪模式改为非贪婪模式，可以在量词符后面加一个问号。

```
// 模式结尾添加了一个问号/a+?/，这时就改为非贪婪模式，一旦条件满足，就不再往下匹配。
var s = 'aaa';
s.match(/a+?/) // ["a"]
```

除了非贪婪模式的加号，还有非贪婪模式的星号（*）。

- `*?`：表示某个模式出现0次或多次，匹配时采用非贪婪模式。
- `+?`：表示某个模式出现1次或多次，匹配时采用非贪婪模式。

组匹配

介绍：正则表达式的括号表示分组匹配，括号中的模式可以用来匹配分组的内容。

```
// 模式没有括号，结果+只表示重复字母d
/fred+/.test('fredd') // true

// 模式有括号+就表示匹配fred这个词。
/(fred)+/.test('fredfred') // true
```

例子：分组捕获。

```
// 正则表达式/(.)b(.)/一共使用两个括号，第一个括号捕获a，第二个括号捕获c。
var m = 'abcabc'.match(/(.)b(.)/);

// m ['abc', 'a', 'c']
```

注意：

1. 使用组匹配时，不宜同时使用g修饰符，否则match方法不会捕获分组的内容。

```
 // g修饰符的正则表达式，结果match方法只捕获了匹配整个表达式的部分。
 var m = 'abcabc'.match(/(.)b(.)/g);
 m // ['abc', 'abc']

 // 这时必须使用正则表达式的exec方法，配合循环，才能读到每一轮匹配的组捕获。
 var str = 'abcabc';
 var reg = /(.)b(.)/g;
 while (true) {
   var result = reg.exec(str);
   if (!result) break;
   console.log(result);
 }
 // ["abc", "a", "c"]
 // ["abc", "a", "c"]
```

1. 组匹配也可以配合 `RegExp.$n` 获取分组匹配到的字符串值,其中n为正整数表示第几个组匹配到的字符串

```
/(.)b(.)/.test('abcabc');

RegExp.$1 // "a"
RegExp.$2 // "c"
RegExp.$3 //  ""
```

1. 正则表达式内部，还可以用\n引用括号匹配的内容，n是从1开始的自然数，表示对应顺序的括号。

```
// \1表示第一个括号匹配的内容（即a），\2表示第二个括号匹配的内容（即c）。
/(.)b(.)\1b\2/.test("abcabc")// true

/y(..)(.)\2\1/.test('yabccab') // true

// 括号还可以嵌套。\1指向外层括号，\2指向内层括号。
/y((..)\2)\1/.test('yabababab') // true
```

例子：组匹配非常有用，下面是一个匹配网页标签的例子。

```
// 圆括号匹配尖括号之中的标签，而\1就表示对应的闭合标签。
var tagName = /<([^>]+)>[^<]*<\/\1>/;

tagName.exec("<b>bold</b>")[1] // 'b'
```

上面代码略加修改，就能捕获带有属性的标签。

```
var html = '<b class="hello">Hello</b><i>world</i>';
var tag = /<(\w+)([^>]*)>(.*?)<\/\1>/g;

var match = tag.exec(html);

match[1] // "b"
match[2] // " class="hello""
match[3] // "Hello"

match = tag.exec(html);

match[1] // "i"
match[2] // ""
match[3] // "world"
```

非捕获组

介绍：`(?:x)`称为非捕获组（Non-capturing group），表示不返回该组匹配的内容，即匹配的结果中不计入这个括号。

概念：非捕获组的作用请考虑这样一个场景，假定需要匹配foo或者foofoo，正则表达式就应该写成`/(foo){1, 2}/`，但是这样会占用一个组匹配。这时，就可以使用非捕获组，将正则表达式改为`/(?:foo){1, 2}/`，它的作用与前一个正则是一样的，但是不会单独输出括号内部的内容。

例子：

```
// 两个括号中第一个括号是非捕获组，所以最后返回的结果中没有第一个括号，只有第二个括号匹配的内容
var m = 'abc'.match(/(?:.)b(.)/);
m // ["abc", "c"]


// 分解网址的正则表达式。第一个正则表达式是正常匹配，第一个括号返回网络协议

// 正常匹配
var url = /(http|ftp):\/\/([^/\r\n]+)(\/[^\r\n]*)?/;

url.exec('http://google.com/');
// ["http://google.com/", "http", "google.com", "/"]

// 第二 个正则表达式是非捕获匹配，返回结果中不包括网络协议。
// 非捕获组匹配
var url = /(?:http|ftp):\/\/([^/\r\n]+)(\/[^\r\n]*)?/;

url.exec('http://google.com/');
// ["http://google.com/", "google.com", "/"]
```

先行断言

介绍：`x(?=y)`称为先行断言，x只有在y前面才匹配，y不会被计入返回结果。比如，要匹配后面跟着百分号的数字，可以写成`/\d+(?=%)/`。“先行断言”中，括号里的部分是不会返回的。

```
// 下面的代码使用了先行断言，b在c前面所以被匹配，但是括号对应的c不会被返回。
var m = 'abc'.match(/b(?=c)/);
m // ["b"]
```

先行否定断言

介绍：`x(?!y)`称为先行否定断言，x只有不在y前面才匹配，y不会被计入返回结果。比如，要匹配后面跟的不是百分号的数字，就要写成`/\d+(?!%)/`。“先行否定断言”中，括号里的部分是不会返回的

```
// 下面代码中，正则表达式指定，只有不在小数点前面的数字才会被匹配，因此返回的结果就是14。
/\d+(?!\.)/.exec('3.14')
// ["14"]

// b不在c前面所以被匹配，而且括号对应的d不会被返回。
var m = 'abd'.match(/b(?!c)/);
m // ['b']
```

字符串的实例方法

介绍：字符串的实例方法之中，有4种与正则表达式有关。

- `String.prototype.match()`：返回一个数组，成员是所有匹配的子字符串。
- `String.prototype.search()`：按照给定的正则表达式进行搜索，返回一个整数，表示匹配开始的位置。
- `String.prototype.replace()`：按照给定的正则表达式进行替换，返回替换后的字符串。
- `String.prototype.split()`：按照给定规则进行字符串分割，返回一个数组，包含分割后的各个成员。

`String.prototype.match()`

语法：字符串实例对象的match方法对字符串进行正则匹配，返回匹配结果。

```
var s = '_x_x';
var r1 = /x/;
var r2 = /y/;
// 字符串的match方法与正则对象的exec方法非常类似：匹配成功返回一个数组，匹配失败返回null。
s.match(r1) // ["x"]
s.match(r2) // null
```

如果正则表达式带有g修饰符，则该方法与正则对象的exec方法行为不同，会一次性返回所有匹配成功的结果。

```
var s = 'abba';
var r = /a/g;

s.match(r) // ["a", "a"]
r.exec(s) // ["a"]
```

设置正则表达式的lastIndex属性，对match方法无效，匹配总是从字符串的第一个字符开始。

```
var r = /a|b/g;
r.lastIndex = 7;
'xaxb'.match(r) // ['a', 'b']
r.lastIndex // 0
```

String.prototype.search()

语法：字符串对象的search方法，返回第一个满足条件的匹配结果在整个字符串中的位置。如果没有任何匹配，则返回-1。

```
'_x_x'.search(/x/) // 1
```

String.prototype.replace()

语法：字符串对象的replace方法可以替换匹配的值。它接受两个参数，第一个是正则表达式，表示搜索模式，第二个是替换的内容。

```
str.replace(search, replacement)
```

正则表达式如果不加g修饰符，就替换第一个匹配成功的值，否则替换所有匹配成功的值。
例子：

```
'aaa'.replace('a', 'b') // "baa"
'aaa'.replace(/a/, 'b') // "baa"
'aaa'.replace(/a/g, 'b') // "bbb"
// 最后一个正则表达式使用了g修饰符，导致所有的b都被替换掉了。
```

消除字符串首尾两端的空格。

```
var str = '  #id div.class  ';

str.replace(/^\s+|\s+$/g, '')
// "#id div.class"
```

补充：

1. replace方法的第二个参数可以使用美元符号$，用来指代所替换的内容。
    - **$&**：匹配的子字符串。
    - **$`**：匹配结果前面的文本。
    - **$’**：匹配结果后面的文本。
    - **$n**：匹配成功的第n组内容，n是从1开始的自然数。
    - **$$**：指代美元符号$。

```
// 将匹配的组互换位置
'hello world'.replace(/(\w+)\s(\w+)/, '$2 $1')
// "world hello"

// 改写匹配的值
'abc'.replace('b', '[$`-$&-$\']')
// "a[a-b-c]c"
```

1. replace方法的第二个参数还可以是一个函数，将每一个匹配内容替换为函数返回值。

```
'3 and 5'.replace(/[0-9]+/g, function (match) {
  return 2 * match;
})
// "6 and 10"

var a = 'The quick brown fox jumped over the lazy dog.';
var pattern = /quick|brown|lazy/ig;

a.replace(pattern, function replacer(match) {
  return match.toUpperCase();
});
// The QUICK BROWN fox jumped over the LAZY dog.
```

1. 作为replace方法第二个参数的替换函数，可以接受多个参数。其中，第一个参数是捕捉到的内容，第二个参数是捕捉到的组匹配（有多少个组匹配，就有多少个对应的参数）。此外，最后还可以添加两个参数，倒数第二个参数是捕捉到的内容在整个字符串中的位置（比如从第五个位置开始），最后一个参数是原字符串。下面是一个网页模板替换的例子。

```
var prices = {
  'p1': '$1.99',
  'p2': '$9.99',
  'p3': '$5.00'
};

var template = '<span id="p1"></span>'
  + '<span id="p2"></span>'
  + '<span id="p3"></span>';
// 下面代码的捕捉模式中，有四个括号，所以会产生四个组匹配，在匹配函数中用$1到$4表示。匹配函数的作用是将价格插入模板中。
template.replace(
  /(<span id=")(.*?)(">)(<\/span>)/g,
  function(match, $1, $2, $3, $4){
    return $1 + $2 + $3 + prices[$2] + $4;
  }
);
// "<span id="p1">$1.99</span><span id="p2">$9.99</span><span id="p3">$5.00</span>"
```

「课堂练习」

使用所学知识将日期转化为指定模板格式

要求：

1. 创建一个

   ```
   getTemplateTime
   ```

   函数接受两个参数

   - templateStr：指定输出日期格式字符串模板其中 `y` 表示年、 `M` 表示月、 `d` 表示日、 `h` 表示时、 `m` 表示分、 `s` 表示秒。如： `yyyy-MM-DD hh:mm` 、 `yyyy年MM月hh:mm:ss` 等
   - date： 传入的日期数据类型
2. 函数会将传入的日期按照接受的指定格式输出

部分代码

```
var date = new Date();

var template1 = 'yyyy-MM-dd hh:mm:ss'

var result1 = getTemplateTime(template1, date);
console.log(result1) //2021-08-28 09:12:46
        //
 var template2 = 'hh:mm / ss 秒  yyyy年MM月dd日'

 var result2 = getTemplateTime(template2,date)

 console.log(result2) //09:12 / 46 秒 2021年08月28日
```

示例代码

```
// 解析：
// 本题考察的正则表达式的组匹配,以及配合replace方法中n 或 使用RegExp.n或使用RegExp.n
// 方法一 replace $n
function getTemplateTime(tStr, date) {
    if (/y+/.test(tStr)) {
        tStr = tStr.replace(/y+/g, function (match) {
            return (date.getFullYear() + '').substr(4 - match.length)
        })
    }
    var o = {
        'M+': date.getMonth() + 1,
        'd+': date.getDate(),
        'h+': date.getHours(),
        'm+': date.getMinutes(),
        's+': date.getSeconds()
    }
    for (var k in o) {
        var reg = new RegExp(k, 'g');
        if (reg.test(tStr)) {
            var time = o[k] + '';
            tStr = tStr.replace(reg, function (match) {
                return (match.length === 1) ? time : ('00' +
time).substr(time.length);
            })
        }
    }
    return tStr
}

// 方法二 使用RegExp.$n
function getTemplateTimeBy$n(tStr, date) {
    if (/(y+)/.test(tStr)) {
        // RegExp.$1 = yyyy
        tStr = tStr.replace(RegExp.$1, (date.getFullYear() + '').substr(4 -
RegExp.$1.length))
    }
    var o = {
        'M+': date.getMonth() + 1,
        'd+': date.getDate(),
        'h+': date.getHours(),
```

```
            'm+': date.getMinutes(),
            's+': date.getSeconds()
        }

        for (var k in o) {
            if (new RegExp('(' + k + ')').test(tStr)){
                var str = o[k]+'';
                tStr = tStr.replace(RegExp.$1,(RegExp.$1.length === 1)?str:
    ('00'+str).substr(str.length))
            }
        }

        return tStr
    }


    var date = new Date();

    var template1 = 'yyyy-MM-dd hh:mm:ss'

    var result1 = getTemplateTime(template1, date);
    console.log(result1) //2021-08-28 09:12:46
            //
     var template2 = 'hh:mm / ss 秒  yyyy年MM月dd日'

     var result2 = getTemplateTime(template2,date)

     console.log(result2) //09:12 / 46 秒 2021年08月28日
```

String.prototype.split()

语法：字符串对象的split方法按照正则规则分割字符串，返回一个由分割后的各个部分组成的数组。

```
str.split(separator, [limit])
```

参数:

- separator：正则表达式，表示分隔规则，
- limit：返回数组的最大成员数。

例子:

```
// 非正则分隔
'a,  b,c, d'.split(',')
// [ 'a', '  b', 'c', ' d' ]

// 正则分隔，去除多余的空格
'a,  b,c, d'.split(/, */)
// [ 'a', 'b', 'c', 'd' ]

// 指定返回数组的最大成员
'a,  b,c, d'.split(/, */, 2)
// [ 'a', 'b' ]

'aaa*a*'.split(/a*/)
// [ '', '*', '*' ]

```

```
16
17  'aaa**a*'.split(/a*/)
18  // ["", "*", "*", "*"]
```

注意：如果正则表达式带有括号，则括号匹配的部分也会作为数组成员返回。

```
1  // 正则表达式使用了括号，第一个组匹配是aaa，第二个组匹配是a，它们都作为数组成员返回。
2  'aaa*a*'.split(/(a*)/)
3  // [ '', 'aaa', '*', 'a', '*' ]
```

课后作业

(字符串处理、正则表达式、事件、组织浏览器默认行为)

实验：表单的验证

要求：

1. 表单中有以下输入框，并包含一定的输入要求
2. 账号：不能为空 不能使用特殊字符 只能使用(数字,字母,下划线,-) 6-18位
3. 昵称：昵称只能是中文,长度不能超过7
4. 电子邮件：邮箱验证支持以下邮箱：`x@.qq.com`、`x@163.com`、`x.@a-r.com.cn`、`@vip.163.com`
5. 身份证： 19位最后一位可能是x如：`445655199907072165`、`44565519990707216x`
6. 手机号码: 1开头的11位数字
7. 生日： 支持验证`1999/05/08`、`1999-05-08`、`19990508`三种格式
8. 密码：长度小于20、不能有空格、不能中文

部分代码：

```
1  <div class="box">
2      <form>
3          账号:<input type="text" id="account"><br>
4          昵称:<input type="text" id="name"><br>
5          邮箱:<input type="text" id="mail"><br>
6          身份证:<input type="text" id="id"><br>
7          生日:<input type="text" id="birthday"><br>
8          密码:<input type="text" id="password"><br>
9          <input type="submit" id="btn">
10     </form>
11 </div>
```

参考代码：

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>Title</title>
6      <style>
7          .box{
8              margin: 0 auto;
9              width: 200px;
10         }
11     </style>
12     <script>
```

```
        // 解析
        // 通过正则表达式与字符串API、字符串相关操作实现在表单提交时实现验证效果
        window.onload= function () {
            var account = document.getElementById("account");
            var name = document.getElementById("name");
            var mail = document.getElementById("mail");
            var id = document.getElementById("id");
            var birthday = document.getElementById("birthday");
            var password = document.getElementById("password");
            var again = document.getElementById("again");
            var btn = document.getElementById("btn");
            var obj={
                "account":/^[a-zA-Z0-9_-]{6,20}$/,
                "name":/^[\u4e00-\u9fa5]{1,}$/,
                "mail":/^[A-Za-z0-9_-]+@[a-zA-Z0-9_-]+(\.[a-zA-Z0-9_-]+)+$/,
                "id":/^(\d{15}|\d{17}[\dxX])$/,

                "birthday":/^(([0-9]{4})[-](0[1-9]|1[0-2])[-](0[1-9]|[1,2]\d|3[0-1]))|(([0-9]{4})[/](0[1-9]|1[0-2])[/](0[1-9]|[1,2]\d|3[0-1]))|(([0-9]{4})(0[1-9]|1[0-2])(0[1-9]|[1,2]\d|3[0-1]))$/,

                "password":/^[a-zA-Z0-9_-]{1,20}$/,

                "telephone":/^1((3[0-9])|(4[5|7|9])|(5[0-3|5-9])|(7[0|1|3|5-8])|(8[0-9]))\d{8}$/,



                "birthday":/^[1|2]\d{3}[-|\/|\.]?((0?[1-9])|(1[0-2]))[-/\.]?((0?[1-9])|([1|2]\d)|(3[0-1]))$/
            };
            btn.onclick = function () {
              if (!obj.account.test(account.value)) {
                    alert("输入账号错误")
                }
                if (!obj.name.test(name.value)) {
                    alert("输入昵称错误")
                }
                if (!obj.mail.test(mail.value)) {
                    alert("输入邮箱格式错误")
                }
                if (!obj.id.test(id.value)) {
                    alert("身份证输入错误")
                }*/
                if (!obj.birthday.test(birthday.value)) {
                    alert("输入生日格式错误")
                }
                else {
                    alert("正确")
                }
                if(!obj.password.test(password.value)){
                    alert("密码输入错误")
                }
            }

        }
    </script>
</head>
```

```
<body>
<!--
表单的元素
账号 (不能为空,只能使用 数字字母下划线横杠-,长度6-20位)
昵称 只能是中文
电子邮件
身份证
生日 1999/05/08  1999-05-08 19990508
密码     长度小于20 不能包含空格
确认密码
-->
<div class="box">
    <form>
        账号:<input type="text" id="account"><br>
        昵称:<input type="text" id="name"><br>
        邮箱:<input type="text" id="mail"><br>
        身份证:<input type="text" id="id"><br>
        生日:<input type="text" id="birthday"><br>
        密码:<input type="text" id="password"><br>
        确认密码:<input type="text" id="again"><br>
        <input type="submit" id="btn">
    </form>
</div>
</body>
</html>
```