# 正则表达式

**元字符**

|  |  |
| --- | --- |
| \d | 数字 |
| \D | 非数字 |
| \w | 字母、数字、下划线 |
| \W | 非字母、数字、下划线 |
| \s | 空白符 |
| \S | 非空白符 |
| . | 除换行符以外的任意单个字符 |
| ^ | 以谁开始 |
| $ | 以谁结束 |

**限定符**

| **限定符** | **重复说明** |
| --- | --- |
| \* | 大于等于0次 |
| + | 大于等于1次 |
| ? | 0次或者1次 |
| {n} | n次 |
| {n,} | 大于等于n次 |
| {n,m} | n到m次 |

其他

| **字符** | **说明** |
| --- | --- |
| [ ] | 匹配中括号中任一字符 |
| [ ^ ] | 匹配除中括号内容以外的字符 |
| \ | 转义符 |
| | | 或者 |
| ( ) | 组，圆括号内的内容表示的是一个表达式 |

**简单使用**

匹配：给定的字符串是否符合正则表达式的过滤逻辑 **reg.test(str)**

var str = "2018-8-8";

var reg = /^\d{4}-\d{1,2}-\d{1,2}$/gi;

console.log(reg.test(str)); // true

提取：使用正则表达式，从字符串中进行提取

// 提取

var str = "西虹市首富: 2391.70, 狄仁杰之四大天王: 623.70, 风语咒: 425.00";

var reg = /[0-9]+([.]{1}[0-9]+){0,1}/g;

var arr = str.match(reg);

console.log(arr); // (3) ["2391.70", "623.70", "425.00"]

// 分组提取

var str = "西虹市首富:2391";

var reg = /([\u4e00-\u9fa5]+):([0-9]+)/g;

if (reg.test(str)) {

console.log(RegExp.$1); // 西虹市首富

console.log(RegExp.$2); // 2391

}

替换：替换字符串 **str.replace(reg, "")**

//替换字符串中的空格

var str = "西 虹 市 首 富:2391";

var reg = /\s/g;

var strNew = str.replace(reg, "");

console.log(strNew); //西虹市首富:2391

**贪婪匹配和惰性匹配**

**贪婪匹配**，就是尽可能多的匹配（默认）

**惰性匹配**，就是尽可能少的匹配（通过在量词后面加个问号就能实现惰性匹配）

所有惰性匹配情形如下：

{m,n}?

{m,}?

??

+?

**锚字符----匹配位置**

在ES5中，共有6个锚字符：

^ $ \b \B (?=p) (?!p)

**^和$**

^（脱字符）匹配开头，在多行匹配中匹配行开头。

$（美元符号）匹配结尾，在多行匹配中匹配行结尾。

**\b和\B**

\b是单词边界，具体就是\w和\W之间的位置，也包括\w和^之间的位置，也包括\w和$之间的位置

**(?=p)和(?!p) 即正向先行断言和负向先行断言**

(?=p)，其中p是一个子模式，即p前面的位置。

比如(?=l)，表示'l'字符前面的位置，例如：

var result = "hello".replace(/(?=l)/g, '#');

console.log(result);

// => "he#l#lo"

而(?!p)就是(?=p)的反面意思，比如：

var result = "hello".replace(/(?!l)/g, '#');

console.log(result);

// => "#h#ell#o#"

二者的学名分别是positive lookahead和negative lookahead。

中文翻译分别是正向先行断言和负向先行断言。

**非捕获分组**

之前文中出现的分组，都会捕获它们匹配到的数据，以便后续引用，因此也称他们是捕获型分组。

如果只想要括号最原始的功能，但不会引用它，即，既不在API里引用，也不在正则里反向引用。

此时可以使用非捕获分组(?:p)，例如：

var regex = /(?:ab)+/g;

var string = "ababa abbb ababab";

console.log( string.match(regex) );

// => ["abab", "ab", "ababab"]

**结构和操作符**

操作符：

1.转义符 \

2.括号和方括号 (...)、(?:...)、(?=...)、(?!...)、[...]

3.量词限定符 {m}、{m,n}、{m,}、?、\*、+

4.位置和序列 ^ 、$、 \元字符、 一般字符

5. 管道符（竖杠）|

上面操作符的优先级从上至下，由高到低。

这里，我们来分析一个正则：

/ab?(c|de\*)+|fg/

由于括号的存在，所以，(c|de\*)是一个整体结构。

在(c|de\*)中，注意其中的量词\*，因此e\*是一个整体结构。

又因为分支结构“|”优先级最低，因此c是一个整体、而de\*是另一个整体。

同理，整个正则分成了 a、b?、(...)+、f、g。而由于分支的原因，又可以分成ab?(c|de\*)+和fg这两部分。

**正则表达式的四种操作**

正则表达式是匹配模式，不管如何使用正则表达式，万变不离其宗，都需要先“匹配”。

有了匹配这一基本操作后，才有其他的操作：验证、切分、提取、替换。

**1.1 验证**

验证是正则表达式最直接的应用，比如表单验证。

在说验证之前，先要说清楚匹配是什么概念。

所谓匹配，就是看目标字符串里是否有满足匹配的子串。因此，“匹配”的本质就是“查找”。

有没有匹配，是不是匹配上，判断是否的操作，即称为“验证”。

这里举一个例子，来看看如何使用相关API进行验证操作的。

比如，判断一个字符串中是否有数字。

使用search

var regex = /\d/;

var string = "abc123";

console.log( !!~string.search(regex) );

// => true

使用test

var regex = /\d/;

var string = "abc123";

console.log( regex.test(string) );

// => true

使用match

var regex = /\d/;

var string = "abc123";

console.log( !!string.match(regex) );

// => true

使用exec

var regex = /\d/;

var string = "abc123";

console.log( !!regex.exec(string) );

// => true

其中，最常用的是test。

**1.2 切分**

匹配上了，我们就可以进行一些操作，比如切分。

所谓“切分”，就是把目标字符串，切成一段一段的。在JS中使用的是split。

比如，目标字符串是"html,css,javascript"，按逗号来切分：

var regex = /,/;

var string = "html,css,javascript";

console.log( string.split(regex) );

// => ["html", "css", "javascript"]

又比如，如下的日期格式：

2017/06/26

2017.06.26

2017-06-26

可以使用split“切出”年月日：

var regex = /\D/;

console.log( "2017/06/26".split(regex) );

console.log( "2017.06.26".split(regex) );

console.log( "2017-06-26".split(regex) );

// => ["2017", "06", "26"]

// => ["2017", "06", "26"]

// => ["2017", "06", "26"]

**1.3 提取**

虽然整体匹配上了，但有时需要提取部分匹配的数据。

此时正则通常要使用分组引用（分组捕获）功能，还需要配合使用相关API。

这里，还是以日期为例，提取出年月日。注意下面正则中的括号：

match

var regex = /^(\d{4})\D(\d{2})\D(\d{2})$/;

var string = "2017-06-26";

console.log( string.match(regex) );

// =>["2017-06-26", "2017", "06", "26", index: 0, input: "2017-06-26"]

exec

var regex = /^(\d{4})\D(\d{2})\D(\d{2})$/;

var string = "2017-06-26";

console.log( regex.exec(string) );

// =>["2017-06-26", "2017", "06", "26", index: 0, input: "2017-06-26"]

test

var regex = /^(\d{4})\D(\d{2})\D(\d{2})$/;

var string = "2017-06-26";

regex.test(string);

console.log( RegExp.$1, RegExp.$2, RegExp.$3 );

// => "2017" "06" "26"

search

var regex = /^(\d{4})\D(\d{2})\D(\d{2})$/;

var string = "2017-06-26";

string.search(regex);

console.log( RegExp.$1, RegExp.$2, RegExp.$3 );

// => "2017" "06" "26"

replace

var regex = /^(\d{4})\D(\d{2})\D(\d{2})$/;

var string = "2017-06-26";

var date = [];

string.replace(regex, function(match, year, month, day) {

date.push(year, month, day);

});

console.log(date);

// => ["2017", "06", "26"]

其中，最常用的是match。

**1.4 替换**

找，往往不是目的，通常下一步是为了替换。在JS中，使用replace进行替换。

比如把日期格式，从yyyy-mm-dd替换成yyyy/mm/dd：

var string = "2017-06-26";

var today = new Date( string.replace(/-/g, "/") );

console.log( today );

// => Mon Jun 26 2017 00:00:00 GMT+0800 (中国标准时间)

这里只是简单地应用了一下replace。但，replace方法是强大的，是需要重点掌握的。

2. 相关API注意要点

从上面可以看出用于正则操作的方法，共有6个，字符串实例4个，正则实例2个：

String#search

String#split

String#match

String#replace

RegExp#test

RegExp#exec

本文不打算详细地讲解它们的方方面面细节，具体可以参考《JavaScript权威指南》的第三部分。本文重点列出一些容易忽视的地方，以飨读者。

**2.1 search和match的参数问题**

我们知道字符串实例的那4个方法参数都支持正则和字符串。

但search和match，会把字符串转换为正则的。

var string = "2017.06.27";

console.log( string.search(".") );

// => 0

//需要修改成下列形式之一

console.log( string.search("\\.") );

console.log( string.search(/\./) );

// => 4

// => 4

console.log( string.match(".") );

// => ["2", index: 0, input: "2017.06.27"]

//需要修改成下列形式之一

console.log( string.match("\\.") );

console.log( string.match(/\./) );

// => [".", index: 4, input: "2017.06.27"]

// => [".", index: 4, input: "2017.06.27"]

console.log( string.split(".") );

// => ["2017", "06", "27"]

console.log( string.replace(".", "/") );

// => "2017/06.27"

**2.2 match返回结果的格式问题**

match返回结果的格式，与正则对象是否有修饰符g有关。

var string = "2017.06.27";

var regex1 = /\b(\d+)\b/;

var regex2 = /\b(\d+)\b/g;

console.log( string.match(regex1) );

console.log( string.match(regex2) );

// => ["2017", "2017", index: 0, input: "2017.06.27"]

// => ["2017", "06", "27"]

没有g，返回的是标准匹配格式，即，数组的第一个元素是整体匹配的内容，接下来是分组捕获的内容，然后是整体匹配的第一个下标，最后是输入的目标字符串。

有g，返回的是所有匹配的内容。

当没有匹配时，不管有无g，都返回null。

**2.3 exec比match更强大**

当正则没有g时，使用match返回的信息比较多。但是有g后，就没有关键的信息index了。

而exec方法就能解决这个问题，它能接着上一次匹配后继续匹配：

var string = "2017.06.27";

var regex2 = /\b(\d+)\b/g;

console.log( regex2.exec(string) );

console.log( regex2.lastIndex);

console.log( regex2.exec(string) );

console.log( regex2.lastIndex);

console.log( regex2.exec(string) );

console.log( regex2.lastIndex);

console.log( regex2.exec(string) );

console.log( regex2.lastIndex);

// => ["2017", "2017", index: 0, input: "2017.06.27"]

// => 4

// => ["06", "06", index: 5, input: "2017.06.27"]

// => 7

// => ["27", "27", index: 8, input: "2017.06.27"]

// => 10

// => null

// => 0

其中正则实例lastIndex属性，表示下一次匹配开始的位置。

比如第一次匹配了“2017”，开始下标是0，共4个字符，因此这次匹配结束的位置是3，下一次开始匹配的位置是4。

从上述代码看出，在使用exec时，经常需要配合使用while循环：

var string = "2017.06.27";

var regex2 = /\b(\d+)\b/g;

var result;

while ( result = regex2.exec(string) ) {

console.log( result, regex2.lastIndex );

}

// => ["2017", "2017", index: 0, input: "2017.06.27"] 4

// => ["06", "06", index: 5, input: "2017.06.27"] 7

// => ["27", "27", index: 8, input: "2017.06.27"] 10

**2.4 修饰符g，对exex和test的影响**

上面提到了正则实例的lastIndex属性，表示尝试匹配时，从字符串的lastIndex位开始去匹配。

字符串的四个方法，每次匹配时，都是从0开始的，即lastIndex属性始终不变。

而正则实例的两个方法exec、test，当正则是全局匹配时，每一次匹配完成后，都会修改lastIndex。下面让我们以test为例，看看你是否会迷糊：

var regex = /a/g;

console.log( regex.test("a"), regex.lastIndex );

console.log( regex.test("aba"), regex.lastIndex );

console.log( regex.test("ababc"), regex.lastIndex );

// => true 1

// => true 3

// => false 0

注意上面代码中的第三次调用test，因为这一次尝试匹配，开始从下标lastIndex即3位置处开始查找，自然就找不到了。

如果没有g，自然都是从字符串第0个字符处开始尝试匹配：

var regex = /a/;

console.log( regex.test("a"), regex.lastIndex );

console.log( regex.test("aba"), regex.lastIndex );

console.log( regex.test("ababc"), regex.lastIndex );

// => true 0

// => true 0

// => true 0

**2.5 test整体匹配时需要使用^和$**

这个相对容易理解，因为test是看目标字符串中是否有子串匹配正则，即有部分匹配即可。

如果，要整体匹配，正则前后需要添加开头和结尾：

console.log( /123/.test("a123b") );

// => true

console.log( /^123$/.test("a123b") );

// => false

console.log( /^123$/.test("123") );

// => true

**2.6 split相关注意事项**

split方法看起来不起眼，但要注意的地方有两个的。

第一，它可以有第二个参数，表示结果数组的最大长度：

var string = "html,css,javascript";

console.log( string.split(/,/, 2) );

// =>["html", "css"]

第二，正则使用分组时，结果数组中是包含分隔符的：

var string = "html,css,javascript";

console.log( string.split(/(,)/) );

// =>["html", ",", "css", ",", "javascript"]

**例子学习**

**匹配任意字符**

使用[\d\D]、[\w\W]、[\s\S]和[^]中任何的一个

**不匹配任何东西的正则**

/.^/ 此正则要求只有一个字符，但该字符后面是开头

**匹配URL**

**<https://www.cnblogs.com/kaisela/p/5085752.html>**

var parse\_url = /^(?:([A-Za-z]+):)?(\/{0,3})([0-9.\-A-Za-z]+)(?::(\d+))?(?:\/([^?#]\*))?(?:\?([^#]\*))?(?:#(.\*))?$/;

url:http://qiji123.kerlai.net:81/GoodsBasic/Operate/12678?q#simen

scheme:http

slash://

host:qiji123.kerlai.net

port:81

path:GoodsBasic/Operate/12678

query:q

hash:simen

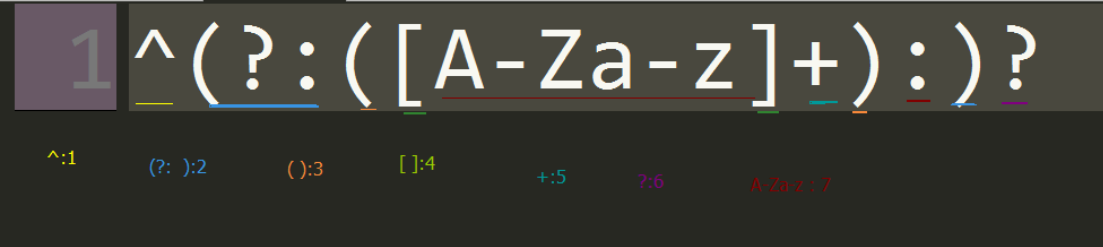
代码中result数组的集合是['http://qiji123.kerlai.net:81/GoodsBasic/Operate/12678?q#simen','http', '//', 'qiji123.kerlai.net', '81', 'GoodsBasic/Operate/12678', 'q', 'simen']

1、捕获型分组：(...)

2、非捕获型分组：(?: .....)

3、向前正向匹配：(?=........)

4、向前负向匹配：(?!.........)



1、^表示字符串的开始

整个正则因子是匹配一个协议名：http

2、(?: )表示一个非捕获型分组：即在这个括号内的，但是不在其子括号内所匹配到的字符将不放入结果数组中。

3、()表示一个捕获型分组，此括号内所匹配到的字符放入结果数组中对应url中的:http字符

4、[]为正则表达式类，表示符合中括号内任一一个字符。

7、A-Za-z表示字母A到字母Z，字母a到字母z。[A-Za-z]表示符合字母A到字母Z，字母a到字母z的任一一个字符

5、+表示匹配1次货多次

6、？表示此组为可选匹配条件

第二个正则因子：(\/{0,3})://

捕获型分组，\/表示一个应该被匹配的/，{0,3}表示\将被匹配0次或者1到3次之间

([0-9.\-A-Za-z]+):qiji123.kerlai.net

捕获型分组，由一个或多个数字 ，“.”，”\-“(转义成”-“)，字母A到Z和字母a到z组成

(?::(\d+))?：81

前置：放在非捕获型分组中将不会出现在返回数组中，\d表示匹配数字。整个因子就是匹配前置为：后面跟随一个或多个数字。此分组因子为可选的

(?:\/([^?#]\*))?：GoodsBasic/Operate/12678

该分组由/开始，^在此处表示非的意思，即除?#之外的所有字符 最后一个？表示此正则因子分组可选

(?:\?([^#]\*))? ：q

该分组表示包含0个或多个非#字符

(?:#(.\*))?：simen

该分组以#开始，(.)将匹配除结束符以外的所有字符。

 $表示这个字符串结束。

**匹配16进制颜色值**

要求匹配：

#ffbbad

#Fc01DF

#FFF

#ffE

正则如下：

var regex = /#([0-9a-fA-F]{6}|[0-9a-fA-F]{3})/g;

var string = "#ffbbad #Fc01DF #FFF #ffE";

console.log( string.match(regex) );

// => ["#ffbbad", "#Fc01DF", "#FFF", "#ffE"]

**匹配时间**

要求匹配：

23:59

02:07

**分析：([01][0-9]|[2][0-3]):[0-5][0-9]**

var regex = /^([01][0-9]|[2][0-3]):[0-5][0-9]$/;

console.log( regex.test("23:59") );

console.log( regex.test("02:07") );

// => true

// => true

如果也要求匹配7:9，也就是说时分前面的0可以省略。

此时正则变成：

var regex = /^(0?[0-9]|1[0-9]|[2][0-3]):(0?[0-9]|[1-5][0-9])$/;

console.log( regex.test("23:59") );

console.log( regex.test("02:07") );

console.log( regex.test("7:9") );

// => true

// => true

// => true

**匹配日期**

比如yyyy-mm-dd格式为例。

要求匹配：

2017-06-10

分析：

年，四位数字即可，可用[0-9]{4}。

月，共12个月，分两种情况01、02、……、09和10、11、12，可用(0[1-9]|1[0-2])。

日，最大31天，可用(0[1-9]|[12][0-9]|3[01])。

正则如下：

var regex = /^[0-9]{4}-(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])$/; //0-9可以用\d代替

console.log( regex.test("2017-06-10") );

// => true

var regex = /^\d{4}-(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])$/;

console.log( regex.test("2017-06-10") );

**把yyyy-mm-dd格式，替换成mm/dd/yyyy**

**正则如下：**

var regex = /(\d{4})-(\d{2})-(\d{2})/;

var string = "2017-06-12";

var result = string.replace(regex, "$2/$3/$1");

console.log(result);

// => "06/12/2017"

其中replace中的，第二个参数里用$1、$2、$3指代相应的分组。**等价于如下的形式**：

var regex = /(\d{4})-(\d{2})-(\d{2})/;

var string = "2017-06-12";

var result = string.replace(regex, function() {

return RegExp.$2 + "/" + RegExp.$3 + "/" + RegExp.$1;

});

console.log(result);

// => "06/12/2017"

**也等价于：**

var regex = /(\d{4})-(\d{2})-(\d{2})/;

var string = "2017-06-12";

var result = string.replace(regex, function(match, year, month, day) {

return month + "/" + day + "/" + year;

});

console.log(result);

// => "06/12/2017"

**window操作系统文件路径**

要求匹配：

F:\study\javascript\regex\regular expression.pdf

F:\study\javascript\regex\

F:\study\javascript

F:\

分析：

整体模式是: 盘符:\文件夹\文件夹\文件夹\

其中匹配F:\，需要使用[a-zA-Z]:\\，其中盘符不区分大小写，注意\字符需要转义。

文件名或者文件夹名，不能包含一些特殊字符，此时我们需要排除字符组[^\\:\*<>|"?\r\n/]来表示合法字符。另外不能为空名，至少有一个字符，也就是要使用量词+。因此匹配“文件夹\”，可用[^\\:\*<>|"?\r\n/]+\\。

另外“文件夹\”，可以出现任意次。也就是([^\\:\*<>|"?\r\n/]+\\)\*。其中括号提供子表达式。

路径的最后一部分可以是“文件夹”，没有\，因此需要添加([^\\:\*<>|"?\r\n/]+)?。

最后拼接成了一个看起来比较复杂的正则：

var regex = /^[a-zA-Z]:\\([^\\:\*<>|"?\r\n/]+\\)\*([^\\:\*<>|"?\r\n/]+)?$/;

console.log( regex.test("F:\\study\\javascript\\regex\\regular expression.pdf") );

console.log( regex.test("F:\\study\\javascript\\regex\\") );

console.log( regex.test("F:\\study\\javascript") );

console.log( regex.test("F:\\") );

// => true

// => true

// => true

// => true

其中，**JS中字符串表示\时，也要转义**

**匹配位置不是开头**

匹配开头可以使用^，但要求这个位置不是开头怎么办？

(?!^)

**数字的千位分隔符表示法**

比如把"12345678"，变成"12,345,678"。

可见是需要把相应的位置替换成",

let reg =/(?!^)(?=(\d{3})+$)/g

**支持其他形式**

如果要把"12345678 123456789"替换成"12,345,678 123,456,789"。

此时我们需要修改正则，把里面的开头^和结尾$，替换成\b：

var string = "12345678 123456789",

reg = /(?!\b)(?=(\d{3})+\b)/g;

var result = string.replace(reg, ',')

console.log(result);

// => "12,345,678 123,456,789"

其中(?!\b)怎么理解呢？

要求当前是一个位置，但不是\b前面的位置，其实(?!\b)说的就是\B。

因此最终正则变成了：/\B(?=(\d{3})+\b)/g。

**判断是否包含有某一种字符**

假设，要求的必须包含数字，怎么办？此时我们可以使用(?=.\*[0-9])来做。

**同时包含数字和小写字母**

(?=.\*[0-9])(?=.\*[a-z])

**详解：**

(?=.\*[0-9])^

分开来看就是(?=.\*[0-9])和^。

表示开头前面还有个位置（当然也是开头，即同一个位置，想想之前的空字符类比）。

(?=.\*[0-9])表示该位置后面的字符匹配.\*[0-9]，即，有任何多个任意字符，后面再跟个数字。

翻译成大白话，就是接下来的字符，必须包含个数字。

**不能全部都是数字**，怎么做呢？(?!p)出马！

对应的正则是：

var reg = /(?!^[0-9]{6,12}$)^[0-9A-Za-z]{6,12}$/;

**验证密码问题**

密码长度6-12位，由数字、小写字符和大写字母组成，但必须至少包括2种字符。

解答

我们可以把原题变成下列几种情况之一：

同时包含数字和小写字母

同时包含数字和大写字母

同时包含小写字母和大写字母

同时包含数字、小写字母和大写字母

最终答案是：

var reg = /((?=.\*[0-9])(?=.\*[a-z])|(?=.\*[0-9])(?=.\*[A-Z])|(?=.\*[a-z])(?=.\*[A-Z]))^[0-9A-Za-z]{6,12}$/;

console.log( reg.test("1234567") ); // false 全是数字

console.log( reg.test("abcdef") ); // false 全是小写字母

console.log( reg.test("ABCDEFGH") ); // false 全是大写字母

console.log( reg.test("ab23C") ); // false 不足6位

console.log( reg.test("ABCDEF234") ); // true 大写字母和数字

console.log( reg.test("abcdEF234") ); // true 三者都有

另外一种解法

“至少包含两种字符”的意思就是说，不能全部都是数字，也不能全部都是小写字母，也不能全部都是大写字母。

var reg = /(?!^[0-9]{6,12}$)(?!^[a-z]{6,12}$)(?!^[A-Z]{6,12}$)^[0-9A-Za-z]{6,12}$/;

console.log( reg.test("1234567") ); // false 全是数字

console.log( reg.test("abcdef") ); // false 全是小写字母

console.log( reg.test("ABCDEFGH") ); // false 全是大写字母

console.log( reg.test("ab23C") ); // false 不足6位

console.log( reg.test("ABCDEF234") ); // true 大写字母和数字

console.log( reg.test("abcdEF234") ); // true 三者都有

**字符串trim方法模拟**

trim方法是去掉字符串的开头和结尾的空白符。有两种思路去做

第一种，匹配到开头和结尾的空白符，然后替换成空字符。如：

function trim(str) {

return str.replace(/^\s+|\s+$/g, '');

}

console.log( trim(" foobar ") );

// => "foobar"

第二种，匹配整个字符串，然后用引用来提取出相应的数据：

function trim(str) {

return str.replace(/^\s\*(.\*?)\s\*$/g, "$1");

}

console.log( trim(" foobar ") );

// => "foobar"

这里使用了惰性匹配\*?，不然也会匹配最后一个空格之前的所有空格的。

当然，前者效率高。

**将每个单词的首字母转换为大写**

function titleize(str) {

return str.toLowerCase().replace(/(?:^|\s)\w/g, function(c) {

return c.toUpperCase();

});

}

console.log( titleize('my name is epeli') );

// => "My Name Is Epeli"

这里不使用非捕获匹配也是可以的。

**驼峰化**

function camelize(str) {

return str.replace(/[-\_\s]+(.)?/g, function(match, c) {

return c ? c.toUpperCase() : '';

});

}

console.log( camelize('-moz-transform') );

// => "MozTransform"

其中分组(.)表示首字母。单词的界定是，前面的字符可以是多个连字符、下划线以及空白符。正则后面的?的目的，是为了应对str尾部的字符可能不是单词字符，比如str是'-moz-transform    '。

**匹配身份证**

正则表达式是：

/^(\d{15}|\d{17}[\dxX])$/

因为竖杠“|”,的优先级最低，所以正则分成了两部分\d{15}和\d{17}[\dxX]。

\d{15}表示15位连续数字。

\d{17}[\dxX]表示17位连续数字，最后一位可以是数字可以大小写字母"x"。

**匹配固定电话**

比如要匹配如下格式的固定电话号码：

055188888888

0551-88888888

(0551)88888888

第一步，了解各部分的模式规则。

上面的电话，总体上分为区号和号码两部分（不考虑分机号和+86的情形）。

区号是0开头的3到4位数字，对应的正则是：0\d{2,3}

号码是非0开头的7到8位数字，对应的正则是：[1-9]\d{6,7}

因此，匹配055188888888的正则是：/^0\d{2,3}[1-9]\d{6,7}$/

匹配0551-88888888的正则是：/^0\d{2,3}-[1-9]\d{6,7}$/

匹配(0551)88888888的正则是：/^\(0\d{2,3}\)[1-9]\d{6,7}$/

第二步，明确形式关系。

这三者情形是或的关系，可以构建分支：

/^0\d{2,3}[1-9]\d{6,7}$|^0\d{2,3}-[1-9]\d{6,7}$|^\(0\d{2,3}\)[1-9]\d{6,7}$/

提取公共部分：

/^(0\d{2,3}|0\d{2,3}-|\(0\d{2,3}\))[1-9]\d{6,7}$/

进一步简写：

/^(0\d{2,3}-?|\(0\d{2,3}\))[1-9]\d{6,7}$/

**判断一个字符串中是否有数字**

使用search

var regex = /\d/;

var string = "abc123";

console.log( !!~string.search(regex) );

// => true

**使用test**

var regex = /\d/;

var string = "abc123";

console.log( regex.test(string) );

// => true

使用match

var regex = /\d/;

var string = "abc123";

console.log( !!string.match(regex) );

// => true

使用exec

var regex = /\d/;

var string = "abc123";

console.log( !!regex.exec(string) );

// => true

其中，最常用的是test。

**匹配成对标签**

要求匹配：

<title>regular expression</title>

不匹配：

<title>wrong!</p>

**但是要求匹配成对标签，那就需要使用反向引用**，如：

但是要求匹配成对标签，那就需要使用**反向引用**，如：

var regex = /<([^>]+)>[\d\D]\*<\/\1>/;

var string1 = "<title>regular expression</title>";

var string2 = "<p>laoyao bye bye</p>";

var string3 = "<title>wrong!</p>";

console.log( regex.test(string1) ); // true

console.log( regex.test(string2) ); // true

console.log( regex.test(string3) ); // false

其中开标签<[^>]+>改成<([^>]+)>，使用括号的目的是为了后面使用反向引用，而提供分组。闭标签使用了反向引用，<\/\1>。

另外[\d\D]的意思是，这个字符是数字或者不是数字，因此，也就是匹配任意字符的意思。

**html转义和反转义**

// 将HTML特殊字符转换成等值的实体

function escapeHTML(str) {

var escapeChars = {

'¢' : 'cent',

'£' : 'pound',

'¥' : 'yen',

'€': 'euro',

'©' :'copy',

'®' : 'reg',

'<' : 'lt',

'>' : 'gt',

'"' : 'quot',

'&' : 'amp',

'\'' : '#39'

};

return str.replace(new RegExp('[' + Object.keys(escapeChars).join('') +']', 'g'), function(match) {

return '&' + escapeChars[match] + ';';

});

}

console.log( escapeHTML('

Blah blah blah

') );

// => "<div>Blah blah blah</div>";

其中使用了用构造函数生成的正则，然后替换相应的格式就行了，这个跟本章没多大关系。

倒是它的逆过程，使用了括号，以便提供引用，也很简单，如下：

// 实体字符转换为等值的HTML。

function unescapeHTML(str) {

var htmlEntities = {

nbsp: ' ',

cent: '¢',

pound: '£',

yen: '¥',

euro: '€',

copy: '©',

reg: '®',

lt: '<',

gt: '>',

quot: '"',

amp: '&',

apos: '\''

};

return str.replace(/\&([^;]+);/g, function(match, key) {

if (key in htmlEntities) {

return htmlEntities[key];

}

return match;

});

}

console.log( unescapeHTML('<div>Blah blah blah</div>') );

// => "

Blah blah blah

"

通过key获取相应的分组引用，然后作为对象的键。

**简单实用的正则测试器**

具体效果如下：

代码，直接贴了，相信你能看得懂：

<section>

<div id="err"></div>

<input id="regex" placeholder="请输入正则表达式">

<input id="text" placeholder="请输入测试文本">

<button id="run">测试一下</button>

<div id="result"></div>

</section>

<style>

section{

display:flex;

flex-direction:column;

justify-content:space-around;

height:300px;

padding:0 200px;

}

section \*{

min-height:30px;

}

#err {

color:red;

}

#result{

line-height:30px;

}

.info {

background:#00c5ff;

padding:2px;

margin:2px;

display:inline-block;

}

</style>

<script>

(function() {

// 获取相应dom元素

var regexInput = document.getElementById("regex");

var textInput = document.getElementById("text");

var runBtn = document.getElementById("run");

var errBox = document.getElementById("err");

var resultBox = document.getElementById("result");

// 绑定点击事件

runBtn.onclick = function() {

// 清除错误和结果

errBox.innerHTML = "";

resultBox.innerHTML = "";

// 获取正则和文本

var text = textInput.value;

var regex = regexInput.value;

if (regex == "") {

errBox.innerHTML = "请输入正则表达式";

} else if (text == "") {

errBox.innerHTML = "请输入测试文本";

} else {

regex = createRegex(regex);

if (!regex) return;

var result, results = [];

// 没有修饰符g的话，会死循环

if (regex.global) {

while(result = regex.exec(text)) {

results.push(result);

}

} else {

results.push(regex.exec(text));

}

if (results[0] == null) {

resultBox.innerHTML = "匹配到0个结果";

return;

}

// 倒序是有必要的

for (var i = results.length - 1; i >= 0; i--) {

var result = results[i];

var match = result[0];

var prefix = text.substr(0, result.index);

var suffix = text.substr(result.index + match.length);

text = prefix

+ '<span class="info">'

+ match

+ '</span>'

+ suffix;

}

resultBox.innerHTML = "匹配到" + results.length + "个结果:<br>" + text;

}

};

// 生成正则表达式，核心函数

function createRegex(regex) {

try {

if (regex[0] == "/") {

regex = regex.split("/");

regex.shift();

var flags = regex.pop();

regex = regex.join("/");

regex = new RegExp(regex, flags);

} else {

regex = new RegExp(regex, "g");

}

return regex;

} catch(e) {

errBox.innerHTML = "无效的正则表达式";

return false;

}

}

})();

</script>