

一些零散知识点总结

正则表达式

- 正则表达式(regular expression)描述了一种字符串匹配的模式 (pattern)
- 我们可以使用 `*` , `?` , `+` , `$` 对字符串内的字符进行匹配
- `*` : 匹配多个字符
- `?` : 匹配单个字符 (0个或1个)
- `+` : 匹配数组的多个数字
 - 例如:

Plain Text

1

[0-9]匹配字符串内0-9的单个数字

2

[0-9]+匹配字符串内的0-9的多个数字

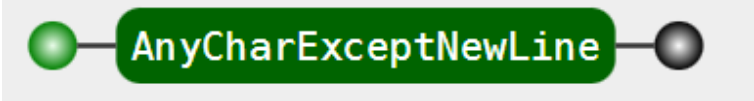
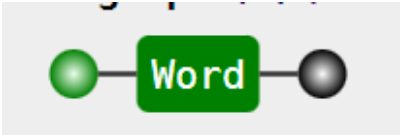
- `$` : 匹配字符串的结尾部分

正则表达式 – 语法 | 菜鸟教程 (runoob.com)

工具: [Regulex: JavaScript Regular Expression Visualizer \(jex.im\)](#)

普通字符

[ABC]	<div>匹配 [...] 中的所有字符，例如 [aeiou] 匹配字符串 "google runoob taobao" 中所有的 e o u a 字母。</div> <div><div>One of:</div><div><div></div><div>ABC</div><div></div></div></div>
[^ABC]	<div>匹配除了 [...] 中字符的所有字符，例如 [^aeiou] 匹配字符串 "google runoob taobao" 中除了 e o u a 字母的所有字母。</div> <div><div>None of:</div><div><div></div><div>ABC</div><div></div></div></div>
[A-Z]	<div>[A-Z] 表示一个区间，匹配所有大写字母，[a-z] 表示所有小写字母。</div> <div><div>One of:</div><div></div></div>

	
.	匹配除换行符（\n、\r）之外的任何单个字符，相等于是 <code>^[^n\r]</code> 。 
<code>[\s\S]</code>	匹配所有。 \s 是匹配所有空白符，包括换行， \S 非空白符，不包括换行。 
<code>\w</code>	匹配字母、数字、下划线。等价于 <code>[A-Za-z0-9_]</code> 

非打印字符

<code>\cx</code>	匹配由x指明的控制字符。例如， <code>\cM</code> 匹配一个 Control-M 或回车符。 x 的值必须为 A-Z 或 a-z 之一。否则，将 c 视为一个原义的 'c' 字符。
<code>\f</code>	匹配一个换页符。等价于 <code>\x0c</code> 和 <code>\cL</code> 。
<code>\n</code>	匹配一个换行符。等价于 <code>\x0a</code> 和 <code>\cJ</code> 。
<code>\r</code>	匹配一个回车符。等价于 <code>\x0d</code> 和 <code>\cM</code> 。
<code>\s</code>	匹配任何空白字符，包括空格、制表符、换页符等等。等价于 <code>[\f\n\r\t\v]</code> 。注意 Unicode 正则表达式会匹配全角空格符。
<code>\S</code>	匹配任何非空白字符。等价于 <code>^[^f\n\r\t\v]</code> 。
<code>\t</code>	匹配一个制表符。等价于 <code>\x09</code> 和 <code>\cI</code> 。
<code>\v</code>	匹配一个垂直制表符。等价于 <code>\x0b</code> 和 <code>\cK</code> 。

特殊字符



所谓特殊字符，就是一些有特殊含义的字符，如上面说的 **runoo*b** 中的 *****，简单的说就是表示任何字符串的意思。如果要查找字符串中的 ***** 符号，则需要对 ***** 进行转义，即在其前加一个 ****，**runo*ob** 匹配字符串 **runo*ob**。

许多元字符要求在试图匹配它们时特别对待。若要匹配这些特殊字符，必须首先使字符"转义"，即，将反斜杠字符 **** 放在它们前面。下表列出了正则表达式中的特殊字符：

\$	匹配输入字符串的结尾位置。如果设置了 RegExp 对象的 Multiline 属性，则 \$ 也匹配 '\n' 或 '\r'。要匹配 \$ 字符本身，请使用 \\$。
()	标记一个子表达式的开始和结束位置。子表达式可以获取供以后使用。要匹配这些字符，请使用 \(和 \)。
*	匹配前面的子表达式零次或多次。要匹配 * 字符，请使用 *。
+	匹配前面的子表达式一次或多次。要匹配 + 字符，请使用 \+。
.	匹配除换行符 \n 之外的任何单字符。要匹配 .，请使用 \.。
[标记一个中括号表达式的开始。要匹配 [，请使用 \[。
?	匹配前面的子表达式零次或一次，或指明一个非贪婪限定符。要匹配 ? 字符，请使用 \?。
\	将下一个字符标记为或特殊字符、或原义字符、或向后引用、或八进制转义符。例如，'n' 匹配字符 'n'。'\n' 匹配换行符。序列 '\\' 匹配 "\"，而 \"(' 则匹配 "("。
^	匹配输入字符串的开始位置，除非在方括号表达式中使用，当该符号在方括号表达式中使用，表示不接受该方括号表达式中的字符集合。要匹配 ^ 字符本身，请使用 \^。
{	标记限定符表达式的开始。要匹配 {，请使用 \{。
	指明两项之间的一个选择。要匹配 ，请使用 \ 。

限定符



限定符用来指定正则表达式的一个给定组件必须要出现多少次才能满足匹配。有 ***** 或 **+** 或 **?** 或 **{n}** 或 **{n,}** 或 **{n,m}** 共6种。

正则表达式的限定符有：

*	匹配前面的子表达式零次或多次。例如，zo* 能匹配 "z" 以及 "zoo"。* 等价于 {0,}。
+	匹配前面的子表达式一次或多次。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。+ 等价于 {1,}。

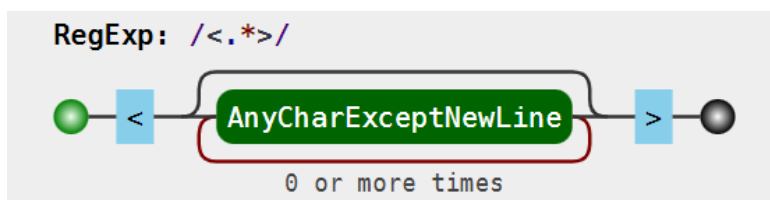
?	匹配前面的子表达式零次或一次。例如, "do(es)?" 可以匹配 "do"、"does" 中的 "does"、"doxy" 中的 "do"。? 等价于 {0,1}。
{n}	n 是一个非负整数。匹配确定的 n 次。例如, 'o{2}' 不能匹配 "Bob" 中的 'o', 但是能匹配 "food" 中的两个 o。
{n,}	n 是一个非负整数。至少匹配n 次。例如, 'o{2,}' 不能匹配 "Bob" 中的 'o', 但能匹配 "fooooood" 中的所有 o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o*'。
{n,m}	m 和 n 均为非负整数, 其中n <= m。最少匹配 n 次且最多匹配 m 次。例如, "o{1,3}" 将匹配 "fooooood" 中的前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格。

注意事项

*** 和 + 限定符都是贪婪的, 因为它们会尽可能多的匹配文字, 只有在它们的后面加上一个 ? 就可以实现非贪婪或最小匹配。**

例如, 您可能搜索 HTML 文档, 以查找在 **h1** 标签内的内容。HTML 代码如下:

```
HTML
1 <h1>RUNOOB-菜鸟教程</h1>
```



结果:

Expression

`/<.*>/`

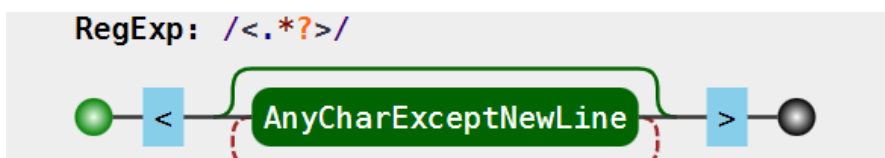
Text

匹配整个字符串

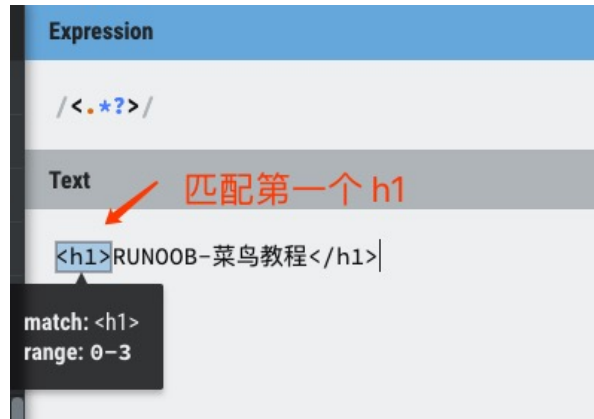
`<h1>RUNOOB-菜鸟教程</h1>`

match: `<h1>RUNOOB-菜鸟教程</h1>`

range: 0-19



结果：



定位符

<code>^</code>	匹配输入字符串开始的位置。如果设置了 RegExp 对象的 Multiline 属性， <code>^</code> 还会与 <code>\n</code> 或 <code>\r</code> 之后的位置匹配。
<code>\$</code>	匹配输入字符串结尾的位置。如果设置了 RegExp 对象的 Multiline 属性， <code>\$</code> 还会与 <code>\n</code> 或 <code>\r</code> 之前的位置匹配。
<code>\b</code>	匹配一个单词边界，即字与空格间的位置。
<code>\B</code>	非单词边界匹配。



注意：不能将限定符与定位符一起使用。由于在紧靠换行或者单词边界的前面或后面不能有一个以上位置，因此不允许诸如 `^*` 之类的表达式。

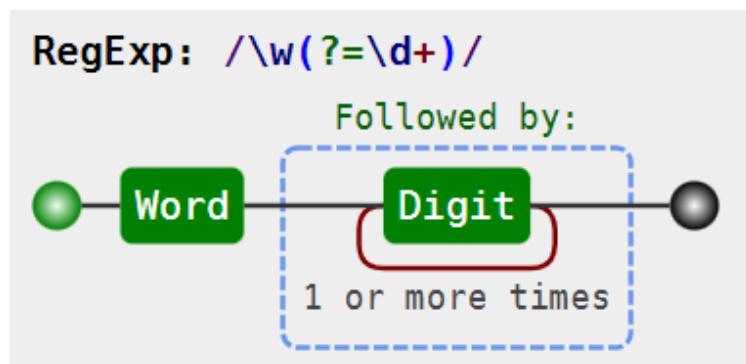
若要匹配一行文本开始处的文本，请在正则表达式的开始使用 `^` 字符。不要将 `^` 的这种用法与中括号表达式内的用法混淆。

选择(Group)与非捕获符

- 用圆括号 `()` 将所有选择项括起来，相邻的选择项之间用 `|` 分隔。
- `()` 表示捕获分组，`()` 会把每个分组里的匹配的值保存起来，多个匹配值可以通过数字 `n` 来查看(`n` 是一个数字，表示第 `n` 个捕获组的内容)。
- 非捕获元(`?<! ?= ?! ?<=`)

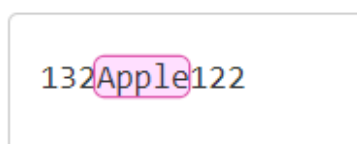
示例：

a. `?=` 图示: `\w(?\=\d+)`



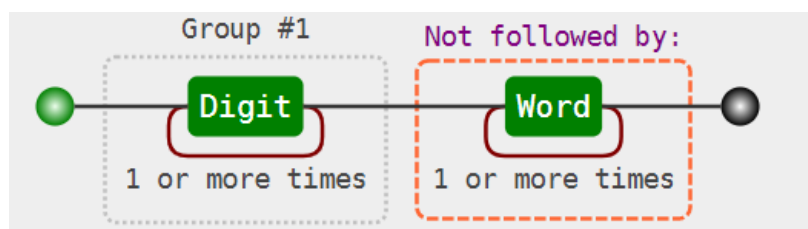
说明: 数字之前的word

b. `?<=` 图示: `(?<=[\d]+)([A-Za-z]+)`



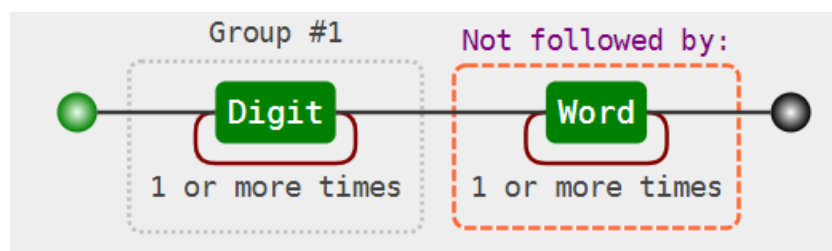
说明: 匹配数字之后的word

c. `?!` 图示: `([\d]+)(?![\w]+)`



说明: 匹配不在word后的数字

d. `?<!` 图示: `(?<![\w]+)([\d]+)`



说明: 匹配不在word前的数字

反向引用_codegarble的博客

- 捕获组-----捕获组捕获到的内容，不仅可以在正则表达式外部通过程序进行引用，也可以在正则表达式内部进行引用，这种引用方式就是反向引用。要了解反向引用，首先要了解捕获组，关于捕获组，参考 [正则基础之——捕获组 \(capture group\)](#)。
- 反向引用的作用通常是用来查找或限定重复、查找或限定指定标识配对出现等等。

- 原理：捕获组(Expression)在匹配成功时，会将子表达式匹配到的内容，保存到内存中一个以数字编号的组里，可以简单的认为是对一个局部变量进行了赋值，这时就可以通过反向引用方式，引用这个局部变量的值。一个捕获组(Expression)在匹配成功之前，它的内容可以是不确定的，一旦匹配成功，它的内容就确定了，反向引用的内容也就是确定的了

2.1 从一个简单例子说起

源字符串：abcdebccde

正则表达式：([ab])\1

对于正则表达式 “([ab])\1”，捕获组中的子表达式 “[ab]” 虽然可以匹配 “a” 或者 “b”，但是捕获组一旦匹配成功，反向引用的内容也就确定了。如果捕获组匹配到 “a”，那么反向引用也就只能匹配 “a”，同理，如果捕获组匹配到的是 “b”，那么反向引用也就只能匹配 “b”。由于后面反向引用 “\1” 的限制，要求必须是两个相同的字符，在这里也就是 “aa” 或者 “bb” 才能匹配成功。

考察一下这个正则表达式的匹配过程，在位置0处，由 “([ab])” 匹配 “a” 成功，将捕获的内容保存在编号为1的组中，然后把控制权交给 “\1”，由于此时捕获组已记录了捕获内容为 “a”，“\1” 也就确定只有匹配到 “a” 才能匹配成功，这里显然不满足，“\1” 匹配失败，由于没有可供回溯的状态，整个表达式在位置0处匹配失败。

正则引擎向前传动，在位置5之前，“([ab])” 一直匹配失败。传动到位置5处时，“([ab])” 匹配到 “b”，匹配成功，将捕获的内容保存在编号为1的组中，然后把控制权交给 “\1”，由于此时捕获组已记录了捕获内容为 “b”，“\1” 也就确定只有匹配到 “b” 才能匹配成功，满足条件，“\1” 匹配成功，整个表达式匹配成功，匹配结果为 “bb”，匹配开始位置为5，结束位置为7。

扩展一下，正则表达式 “([a-z])\1{2}” 也就表达连续三个相同的小写字母。

Java隐式类型转换

(参考：[java方法中返回值的理解_正确理解Java方法的返回值_彼岸枫桥的博客](#))

在Java的方法(method)中，往往会出现返回值与方法声明的返回值类型不同的情况，例如：

Java

```
1 //可以通过编译
2 double Main(){
3     int r=0;
4     return r;
5 }
```

Java

```
1 //不能通过编译，会报错
2 int Main(){
3     double r=0;
4     return r;
5 }
```

Java中，对返回值的类型要求不是很严格，但是需要满足一点：**返回值的实际数据类型必须能够转换成方法所声明的数据类型**，且在接受返回值时，必须使用与方法声明的数据类型相同的数据类型的变量来接受，否则不能通过编译。

Java中包含有8大基本数据类型，其中整数类型有4种，分别是long、int、short和byte。这4种整数类型有着明显的“**向下兼容**”的特性，也就是说，较小的数据类型都可以自动转换为较大的数据类型，比如int类型的数据在赋值给long类型的变量时，可以自动完成从int到long的转换。因此，**声明返回值为较大的数据类型，实际返回较小的数据类型，是肯定没有问题**。比如，声明返回值为long类型，但实际返回int型数据是完全可以的。在这种情况下，int型的数据返回到主调方法中，在赋值给一个long类型的变量时，可以自动完成类型转换。同样，两种浮点数double和float也能够做到“向下兼容”，声明double型，返回float型也完全可以。

表示字符的**char**类型，在实际存储数据的过程中，存储的也是一个占两个字节的“整数”，这个“整数”其实就是字符的编码值。那么char类型的数据能否与Java的4种整型完成自动类型转换呢？实际情况是：**char可以自动转换为long和int，但不能自动转换为short和byte**。反过来，short和byte也不能自动转换为char。说的直白一点，就是如果我们把方法的返回值声明为char类型，方法实际返回的是long、int、short和byte型的数据都不行。

任意整型和字符型都可以自动转换为浮点型，但反之不成立。也就是说，long、int、short、byte以及char都可以自动转换为double和float。但反过来，double和float都无法自动转换成long、int、short、byte以及char。这里可能有小伙伴会有一些疑惑：8个字节的long类型数据真的能自动转换为4个字节的float型吗？从语法的角度是没有问题的，我们把一个long类型的数据赋值给一个float类型的变量不需要做强制类型转换，编译器也不会报错。但实际存储过程中，有可能会导致数据损失精度。注意，这里所说的是“有可能”损失精度，而不是“一定”损失精度。