

正则表达式

1 用途

正则表达式主要用于搜索和替换.

2 匹配单个字符

2.1 纯文本匹配

虽然这么用据说有些“低级”甚至是浪费, 但是依然可以视作学习正则的起点.

```
1 import re
2 text1 = 'my name is Peng'
3 print(re.search('Peng', text1))
```

显示结果如下:

```
1 <re.Match object; span=(11, 15), match='Peng'>
```

这类文本一般被称作“静态文本”.

2.1.1 多个匹配结果(全局匹配)

一般来说, 正则表达式引擎默认返回第1个匹配结果.

在python中, 初步学习时可以用 `re.findall` 命令. `findall` 的返回结果是全部匹配项的数组, 如果有捕获项就按照顺序捕获字符串组成的元组. 这里由于是纯文本, 所以结果比较简单. 随着匹配模式的加深, 匹配出的结果也会更加多样.

```
1 import re
2 text1 = 'This is for testing and learning'
3 print(re.findall('ing', text1))
```

结果如下:

```
1 ['ing', 'ing']
```

2.1.2 字母大小写

正则表达式区分大小写, 为了忽视这种影响, 可以传入 `re.I` (或者 `re.IGNORECASE`). 参数意义即参数名所示.

```
1 import re
2 text1 = 'This is for testIng and learning'
3 print(re.findall('ing', text1, re.I))
```

结果如下:

```
1 ['Ing', 'ing']
```

2.2 匹配任意字符

正则表达式里可以使用.字符来匹配任何一个单一字符.

假如我将要匹配某目录下满足以下条件的文件名的文件并使用某种方式把他们整合到一个文件夹里, 那么首先我应当匹配出这些文件名:

```
1 import re
2 mt = 'try.\.csv' #类似其他语言,使用反斜杠\来转义字符,这里我要筛选符合条件的文件名
3 files = 'try1.csv\ntry.csv\nlearn1.csv\ndata.csv\ntryp.csv'
4 print(re.findall(mt, files, re.I))
```

结果如下:

```
1 ['try1.csv', 'tryp.csv']
```

需要注意的是, 这里自己给出的例子往往与现实生活中的数据相去甚远, 再具体运用时一定要事先清洗, 记录在此聊以自警.

这里用到的\ (反斜杠) 被叫做metacharacter, 中文译作**元字符**, 表示“该字符具有特殊含义”. 那么类似地, 如果要对反斜杠本身转义以匹配其本身, 就应当使用双反斜杠\\. markdown里也是)

2.3 小结

正则表达式本质上是字符串.

这些字符由普通字符和元字符构成. 特别地, .可以匹配 任意字符. \用于转义\.

3.匹配一组字符

下面讨论字符集合. 它能够匹配**特定的字符和字符区间**.

3.1 匹配多个字符中的一个

进一步缩小我们在2.2中的的要求, 只要求匹配try+数字组合名的csv文件.

我们可以使用元字符[]来定义字符集合. 具体效果如下:

```
1 import re
2 mt = 'try[1-9]\.csv' #类似其他语言,使用反斜杠\来转义字符,这里我要筛选符合条件的文件名
3 files = 'try1.csv\ntry.csv\nlearn1.csv\ndata.csv\ntryp.csv\ntry3.csv'
4 print(re.findall(mt, files, re.I))
```

```
1 ['try1.csv', 'try2.csv']
```

这里由于数据较为简单,所以几乎不需要测试.现实生活中常常需要验证是否会匹配到错误的信息.

3.2 字符集合区间

这里用到的[1-9]就是一个字符集合区间.

字符集合在不需要区分大小写的情况下、或者只需要匹配某个特定部分的搜索操作里比较常见.

例子里面的- (连字符) 也是一个元字符. **-专用于区间**

需要注意的是:

1. 连字符选择的区间是以ASCII字符表为基准的. 因此如果是形如A-z的匹配模式, 根据字符表还会匹配到一些不需要的字符, 如^等.
2. 区间的尾字符不能小于首字符. 这点与数学定义类似.
3. 连字符-是一个特殊的字符, 仅仅在字符集合内生效, 因此**不需要转义**.

如果要匹配所有的字母和数字并且排除其他字符, 可以按照如下的形式匹配:

```
1 [A-Za-z0-9]
```

不同元素之间不需要加逗号、空格等分隔符, 直接写就行.

3.3 取非匹配

有的时候我们可能需要反过来匹配我们不需要的字符, 例如“清洗”一些不需要的匹配结果.

^专用于取非. 类似逻辑运算非.

注意, ^的效果对字符集合里面**所有的字符和字符区间**都有效, 而不是仅仅跟在^后面的一个字符或者区间.

如以下一个例子:

```
1 import re
2 text='pengine1.csv\nengine2.csv\nengines.csv\nengine3.csv\nengine4.csv\nengine
3 q.csv\nlea.csv'
4 mt = 'engine[^a-z]\.csv'
5 print(re.findall(mt, text, re.I))
```

输出结果:

```
1 ['engine1.csv', 'engine2.csv', 'engine3.csv', 'engine4.csv']
```

3.4 小结

元字符[和]可以用来定义字符集合.

字符集合可以列举也可以通过元字符-来给出.

也可以通过^求非来确定区间. 对应集合论上的补集 $\complement_S A$.

4 元字符专题

4.1 匹配空白字符

在文件中, 我们有的时候需要做一些特定的数据清洗.

举一个实际的例子, 在网站上找到的很多文件可能都会夹杂着空行, 如下所示:

```
1 | 北京
2 |
3 | 上海
4 |
5 | 广东
6 |
7 | 大连
8 |
9 | 成都
10 |
11 | 浙江
12 |
13 | 武汉
```

在文本编辑器里看, 如果隐藏换行符, 看似需要筛选的只有一行, 但我们其实需要连续匹配**两次换行符**.

```
1 | import re
2 | text='北京\n\n上海\n\n广东\n\n大连\n\n成都\n\n浙江\n\n武汉 '
3 | mt = '\n\n'
4 | itall = re.finditer(mt, text)
5 | for i in itall:
6 |     print(i.group())
```

这里的运行结果是12个空行.

比较常用的空白字符是\n, \t, \r. 其中\r是文本行的结束标签, 常和\n结合使用匹配连续的行尾标签. 常用的表示空白元字符的转义元字符如下表所示:

元字符	说明
[\\b]	回退并删除一个字符(对应Backspace)
\\f	换页
\\n	换行
\\r	回车
\\t	制表
\\v	垂直制表

4.2 匹配特定的字符类别

很多时候，为了简化匹配的式子，一些常用的字符集合可以使用特殊元字符来代替。

元字符可以匹配某一“特定类别”的字符，术语称之为“字符类”。

这些元字符被称作“字符类”。

4.2.1 匹配数字和字母

元字符	说明
\\d	任何一个数字字符（等价于[0-9]）
\\D	任何一个数非字符（等价于^[0-9]）
\\w	任何一个字母数字字符（大小写均可）或下划线字符（等价于[a-zA-Z0-9_]）
\\W	任何一个非字母数字字符（大小写均可）或非下划线字符（等价于[^a-zA-Z0-9_]）

```
1 #文本
2 123123213
3 ASDS9D7Q98
4 18EQ9Q793898
5 293821903
6 2141212
7 AJIOF1
8 P1P2P3
```

```
1 #需求：匹配字母+数字 如P1, A1等.
2 #正则表达式: [:alpha:]\\d
3 import re
4 text1 =
5 '123123213\\nASDS9D7Q98\\n18EQ9Q793898\\n293821903\\n2141212\\nAJIOF1\\nP1P2P3'
6 print(re.findall("[:alpha:]\\d", text1))
```

```
1 ['S9', 'D7', 'Q9', 'Q9', 'Q7', 'F1', 'P1', 'P2', 'P3']
```

正则表达式很少有对错之分，关注的主要是能否解决问题。更精确的匹配就需要更复杂的正则表达式。

4.2.2 匹配空白字符

元字符	说明
<code>\s</code>	任何一个空白字符（等价于 <code>[\f\n\r\t\v]</code> ）
<code>\S</code>	任何一个非空白字符（等价于 <code>^[^\f\n\r\t\v]</code> ）

那么上一节里面的空白字符表达式就可以被完美平替，例如可以将这些表达式里面的两个回车全部更改成一个回车。就不会出现空两行的情况了。

4.2.3 匹配十六进制与八进制数值

有些数据是用十六进制的字符组成的，比如网址。这种情况下，正则可能是有用的。

不过不常用，等到要用的时候再学，这里先跳过。

4.3 使用POSIX字符类

实践并查阅资料后发现 python 里面re模块不支持该字符类。故略去。

4.4 小结

这一章用于匹配特定字符（制表符、换行符）。简短的元字符可以简化正则表达式的形式。

5 重复匹配

前面几章主要关注单个字符的匹配，这在现实生活中必然不够用。

为了切合现实生活中的运用，这里需要使用“连续匹配”来匹配多个连续重复使用的例子。

5.1 匹配多个字符

例如，我们需要匹配符合特定模式的字符串，但是这些字符串的字符数是不确定的。

比如邮箱的地址符合如下的形式：

whatever@whatever.whatever

为了解决这类问题，需要一种能够匹配多个字符的办法，可以通过几种特殊的元字符来做到：

字符	用途
*	对它前面的正则式匹配0到任意次重复，尽量多的匹配字符串。ab* 会匹配 'a', 'ab', 或者 'a' 后面跟随任意个 'b'.
+	对它前面的正则式匹配1到任意次重复。ab+ 会匹配 'a' 后面跟随1个以上到任意个 'b', 它不会匹配 'a'.
?	对它前面的正则式匹配0到1次重复。ab? 会匹配 'a' 或者 'ab'.

这类字符都是“**贪婪的**” (greedy). 即若不给出具体的限制条件，会尽可能多的匹配字符。如"aaaaa1", 使用表达式"a*"贪婪地匹配就会尽可能多的匹配a，当然可能实际需求只需要匹配"aaa1". 这将交给元字符{}来处理。

一般来说如果不确定文本里是否有需要匹配的东西，就用元字符?来匹配。

不如以具体的例子来说明这三者的区别。

5.1.1 元字符+

元字符+必须至少匹配一个字符。

测试文本：

```
1 Send personal email to youkonwwho@whatever.com or
   .you.know.who1@whatever.what.com. My personal email is balabala@qq.com. Any
   questions about me are welcomed. Excption: .@fool.com
```

```
1 #正则表达式:(测试元字符+)
2 [\w.]+@[ \w. ]+\. \w+
```

这里为了考虑到诸如“you.know.who”这种在用户名中加点的情况，前面使用了元字符[\w.]; 考虑到多层域名的情况，后面也使用了[\w.]; 至于为什么最后一处不使用[\w.]，是因为句子里出现了以句号为结尾放在网址后的现象。结果如下：

```
1 ['youkonwwho@whatever.com', '.you.know.who1@whatever.what.com',
   'balabala@qq.com', '.@fool.com']
```

这符合我们的预期。

5.1.2 元字符*

元字符*用法同+，可以是零个字符也可以是多个字符。

例如，模式H.* nan (H开头+任意字符+空格+nan的模式) 就可以匹配Hu nan、He nan、Herrfsfsaf nan、Hs.fsfsdfs nan等任意具有类似规律的组合。

测试文本同上，我们略微修改模式，使得结果中原有的第二项只截取到who1的位置。

```
1 #正则表达式: (演示元字符*)
2 [\w]*@[\w.]+\.\w+
```

结果如下:

```
1 ['youkonwwho@whatever.com', '.you.know.who1@whatever.what.com',
  'balabala@qq.com', '.@fool.com']
```

如果这里不使用*, 改成+, 就有:

```
1 ['youkonwwho@whatever.com', 'who1@whatever.what.com', 'balabala@qq.com']
```

若要使得截取的邮箱首字符不是.的同时又要保留用户名加点的情况, 应当如下匹配模式:

```
1 #正则表达式: (演示元字符*)
2 \w[\w.]*@[\w.]+\.\w+
```

结果如下:

```
1 ['youkonwwho@whatever.com', 'you.know.who1@whatever.what.com',
  'balabala@qq.com']
```

正则表达式微言精义, 一定要反复测试确定自己有没有遗漏情况, 或者范围过大的情况.

5.1.3 元字符?

? 只能匹配一个字符. 一般用来判断是否存在的情况. 或者是不超过一次的情况,

例如若要匹配网址, 我们只需要http和https这两种情况, httpsss显然是不合法的匹配. 具体模式可以是:

```
1 #匹配模式: (演示元字符?)
2 https?:\/\/[\w./]+
```

注意有的时候为了可读性, 字符集合内可以只有一个字符, 如这个例子也可以写成:

```
1 http[s]?://
```

5.2 匹配的重数字数 (interval)

观察以上三个字符的定义, 我们可以发现:

元字符	用途
+和*	无上限, 无法设定最大值
+,*和?	至少一个或零个, 无法设定最小值
+和*	无法设定精确的数字

个人认为Interval翻译为区间更为妥当。

可以为重复匹配次数设定精确值、区间、至少或至多的次数。语法同切片，此处不再赘述。

5.3 防止过度匹配

有的时候会出现过度匹配的情况，即虽然满足条件，但是匹配的项目“太多了”。

例如Web页面中可能会出现多个HTML标签。

实例文本：

```
1 | ...which benefits both <B>male<B> and <B>female<B>...
```

```
1 | <[Bb]>.*</[Bb]>
```

结果：

```
1 | ['<B>male<B> and <B>female<B>']
```

这不是我们想要的结果，为了得到我们想要的结果，我们需要防止过度匹配。相对“贪婪”，我们应该选择“懒惰”。

```
1 | <[Bb]>.*?</[Bb]>
```

结果：

```
1 | ['<B>male<B>', '<B>female<B>']
```

6 位置匹配

有时我们只需要对某段文本的特定位置进行匹配，这就要用到位置匹配了。

6.1 边界

位置匹配用来解决在什么地方进行字符串匹配操作的问题。

比如我们想要对以下这句话进行匹配：

```
1 | Damn it! The cat scattered her food all over the table!
```

我们想要把主语cat改成其他主语，可是如果直接改，就会把scatter里面的cat也改掉，这就会形成一个无意义的句子。

```
1 | Damn it! The dog sdogtered her food all over the table!
```

因此我们需要一些元字符来界定我们需要匹配操作在什么位置，这样的位置就叫做**边界**。

6.2 单词边界(boundary)

最常用的边界就是**\b**了，用于匹配一个单词的开始和结尾。

这个单词边界是怎么产生的呢？它匹配的实际上是这样的一个位置，这个位置位于一个能用来构成单词的字符（字符、数字和下划线，也就是与**\w**相匹配的字符）和一个不能用来构成单词的字符（也就是**\W**相匹配的字符）之间。

因此利用这个特性，对于以下一段文字：

```
1 | This is just a snippet of testing text for learning Regular Expression, which  
   | is a brilliant tool.
```

如果仅使用前边界，即只要求is开头的单词：

```
1 | \bis
```

```
1 import re
2 text1 = 'This is just a snippet of testing text for learning Regular Expression, which is a brilliant tool.'
3 print(re.findall("\bis\b", text1))
```

如果仅使用后边界，即只要求is结尾的单词：

```
1 | is\b
```

```
1 import re
2 text1 = 'This is just a snippet of testing text for learning Regular Expression, which is a brilliant tool.'
3 print(re.findall("is\b", text1))
```

如果同时使用边界，就会指定为单个的is单词：

```
1 | \bis\b
```

```
1 import re
2 text1 = 'This is just a snippet of testing text for learning Regular Expression, which is a brilliant tool.'
3 print(re.findall("\bis\b", text1))
```

如果不使用边界，就会匹配到上面所有的情况：

```
1 | is
```

```
import re
text1 = 'This is just a snippet of testing text for learning Regular Expression, which is a brilliant tool.'
print(re.findall("is", text1))
```

6.3 字符串边界

单词边界可以用来进行与单词有关的位置匹配；字符串则是进行字符串有关的位置匹配。用来定义字符串边界的元字符有[^]和^{\$}。

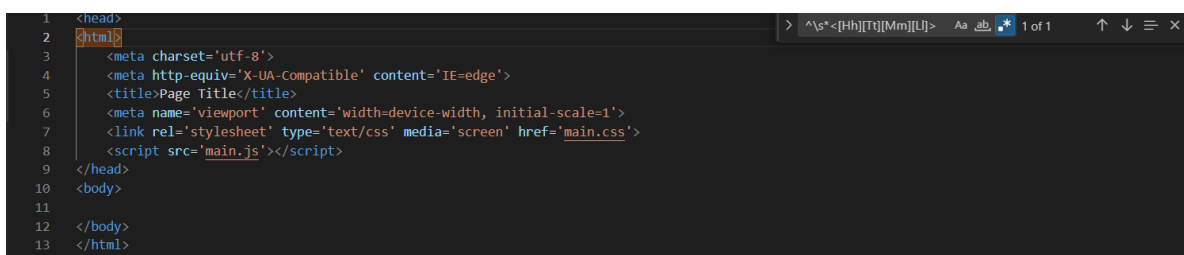
前者用于匹配整个文档的第一个字符，后者用于匹配整个文档的最后一个字符。

例如要检查html文档是否有误，需要检测文档的开头是否是html：

```
1 <head>
2 <html>
3     <meta charset='utf-8'>
4     <meta http-equiv='X-UA-Compatible' content='IE=edge'>
5     <title>Page Title</title>
6     <meta name='viewport' content='width=device-width, initial-scale=1'>
7     <link rel='stylesheet' type='text/css' media='screen' href='main.css'>
8     <script src='main.js'></script>
9 </head>
10 <body>
11
12 </body>
13 </html>
```

可以使用正则表达式如下：

```
1 ^\s*<[Hh][Tt][Mm][Ll]>
```



如果需要检查</html>标签是否在最后一行，就需要使用如下的正则表达式：

```
1 <[/[Hh][Tt][Mm][Ll]]>\s*$
```

这里的s是考虑到了可能存在的空白字符，\$用于整个字符串的匹配。

6.4 分行匹配

有些元字符可以改变另一些元字符的行为。其中之一就是分行匹配模式。

在分行匹配中，^和\$不仅匹配正常的字符串开头和结尾，而且分别匹配行分隔符后面的开始位置和后面的位置。

具体地来说就是每一行的“最前面”和“最后面”。

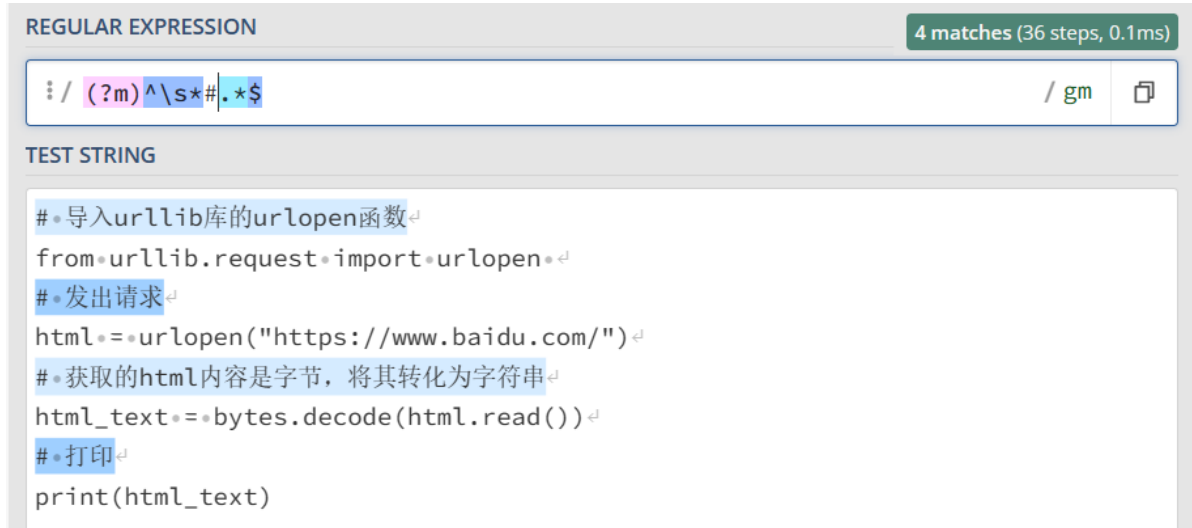
我们需要使用(?m)并把它放到整个模式的最前面，这样就可以查找出所有的内容。

举一个具体的例子，例如我们需要更改一段代码中所有的注释：

```

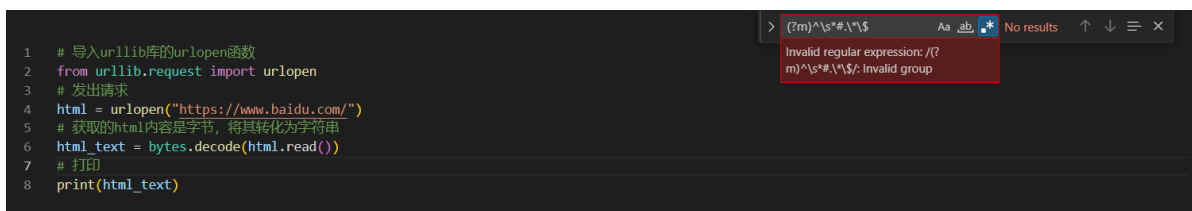
1 # 导入urllib库的urlopen函数
2 from urllib.request import urlopen
3 # 发出请求
4 html = urlopen("https://www.baidu.com/")
5 # 获取的html内容是字节, 将其转化为字符串
6 html_text = bytes.decode(html.read())
7 # 打印
8 print(html_text)

```



匹配: `^\s*#.*$` 首先将匹配一整个字符串的开始, 随后 `\s` 对应任意个空白字符, 然后是 `#`, 对应 python 的注释标记. 但是这个匹配只会进行一次. 而加上 `(?m)` 之后, 这个模式将会将换行符视为一个字符串分隔符, 这样就可以把每一行的注释都匹配出来了.

需要注意的是, 有些正则表达式实现不支持 `(?m)`, 比如 vsc 自带的:



6.5 小结

正则表达式可以匹配任意长度的文本块之外, 还可以匹配出现在特定位置的文本.

`\b` 用来指定一个单词边界 (boundary); 而 `\B` 则与之相反. (事实上, 大写字母和小写字母的作用似乎都是相反的).

`^` 和 `$` 用来指定字符串边界, 具体是指字符串的开头和字符串的结尾.

若和 `(?m)` 配合使用, `^` 和 `$` 可以匹配一个换行符处开头或结束的字符串.

补充: 各个运算符优先级 (来自菜鸟教程)

正则表达式同算术表达式类似, 从左到右, 先高后低.

下表从最高到最低说明了各个运算符的优先级顺序:

运算符	描述
\	转义符
(), (?:), (?:=), []	圆括号和方括号
*, +, ?, {n}, {n,}, {n,m}	限定符
^, \$, \任何 元字符、任 何字符	定位点和序列（即：位置和顺序）
	替换, "或"操作. 字符具有高于替换运算符的优先级, 使得"m food"匹配"m"或"food". 若要匹配"mood"或"food", 请使用括号创建子表达式, 从而产生"(m f)ood".

7 子表达式

7.1 子表达式的优势

之前学过的字符多次重复匹配（第五章），即运用元字符+、*、? 或{}来实现。

但这些字符只能用于紧挨着的前一个字符或元字符。

比如我们可能在一个文档里面把一个单词打了两边重复的，我们希望把这些错误识别出来：

```
1 | ...This is is really a unexpected error...
```

如果我们把匹配模式写成

```
1 | \s\bis\s\b{2, }
```

的话，这个匹配模式实际上匹配不到任何一个字符串，因为\b代表的是单词的边界，相邻的单词只有一个边界，多余的边界是匹配不出来的。因此这个模式即使语法上正确，没有任何意义。

我们应该写成：

```
1 | (\s\bis\b){2}
```

REGULAR EXPRESSION

1 match (14 steps, 0.0ms)

:/ (\s\bis\b){2}

/ gm

📄

TEST STRING

...This is is really a unexpected error...

这里的元字符(和)括起来的表达式(\s\bis\b)就是一个子表达式，就像数学里的括号一样，可以看作运算优先级最高的运算符，因此匹配的时候相当于把这个字符串当成一个“整体”。

7.2 子表达式的嵌套

我们可以随时添加括号来增加可读性. 就像数学里面对于任何可结合的运算,
 $x \circ y \circ z = (x \circ y) \circ z = x \circ (y \circ z)$.

子表达式的嵌套理论上没有限制. 多重嵌套的子表达式很强大, 但可能会让模式难以理解.

把必须匹配的情况考虑周全并写出一个匹配结果符合预期的正则表达式很容易, 但把不需要匹配的情况也考虑周全并确保在匹配结果以外往往要困难得多. 简单的来说, 必要条件容易找, 但是充要条件很难找.

比如我们需要匹配一段ip合法的IP地址, 很常规的必要匹配条件是:

```
1 | (\d{1,3}\.){3}(\d{1,3})
```

REGULAR EXPRESSION

5 matches (70 steps, 0.0ms)

:/ (\d{1,3}\.){3}(\d{1,3}) / gm

TEST STRING

0.0.0.1
0.0.0.256
1.123.434.200
255.255.255.255
78.9.8.1

我们会发现这个条件不够精准, 因为IP地址各组数字是有取值范围的.

一个解决办法是, 把这组数字分类讨论成四种情况:

1. 任何一个1-2位数
2. 任何一个1开头的3位数
3. 任何一个2开头且第2位数字在0-4之间的3位数字
4. 任何一个25开头且第3位的数字在0-5之间的3位数字

具体写成正则表达式, 如下所示:

```
1 | /(?:m)^(((\d{1,2})|(1\d{2})|(2[0-4]\d)|(25[0-5]))\.){3}((\d{1,2})|(1\d{2})|(2[0-4]\d)|(25[0-5]))$/g
```

注意这里要分行匹配, 结果如下:

REGULAR EXPRESSION

3 matches (248 steps, 0.1ms)

:/ (?:m)^(((\d{1,2})|(1\d{2})|(2[0-4]\d)|(25[0-5]))\.){3}((\d{1,2})|(1\d{2})|(2[0-4]\d)|(25[0-5]))\$/g

TEST STRING

0.0.0.1
0.0.0.256
1.123.434.200
255.255.255.255
78.9.8.1

7.3 小结

子表达式的作用是把同一个表达式的各个相关部分组合在一起。

子表达式由(和)组成一个块，对重复次数的元字符的作用对象做出精确的设定。

8 回溯引用：前后一致匹配

8.1 回溯引用

在HTML语言中，常常需要匹配标题标签来定义和排版文字。

现在我们需要匹配任意一对标题标签<H1>与</H1>。

一个比较容易想到的匹配是<[Hh][1-6]>.*?</[Hh][1-6]>，注意这里运用了懒惰型元字符来防止过度匹配。

但是这样真的能解决问题吗？具体看下面这个例子：

The screenshot shows a regular expression testing interface. The top bar indicates 'REGULAR EXPRESSION' and '4 matches (141 steps, 0.0ms)'. The input field contains the regex `<[Hh][1-6]>.*?</[Hh][1-6]>`. Below, the 'TEST STRING' section contains the following HTML code:

```
<BODY>
<H1>This is a home page.</H1>
FOR LEARNING.
<H2>It doesn't make any sence.</H2>
For I just illustrate some specical cases with these codes.
<H2>Notice that this line goes wrong.</H3>
For the label <H2> does not match </H3>.
</BODY>
```

The tool highlights four matches: 1. `<H1>This is a home page.</H1>`, 2. `<H2>It doesn't make any sence.</H2>`, 3. `<H2>Notice that this line goes wrong.</H3>`, and 4. `<H2> does not match </H3>.`

这个例子里面，<H2>实际上和</H3>发生了匹配，这明显的是不合法的标题，但是却被我们匹配到了。

但是如果我们使用回溯引用就不会出现这个问题。

回溯引用就像变量一样，注意很多正则表达式实现里面的语法都不一样，Js运用\来标识回溯引用。注意回溯引用和重复匹配的区别在于，回溯引用针对的是具体的子表达式对应的结果，比如按照前面的规则里，如果拿(和)括住了[1-6]，那么匹配到的[1-6]中的任意一个字符前后都是一个字符，比如<H1>中的1是匹配好的字符，那么后面的一个部分匹配到的就一定是</H1>，而不是</H2>或</H3>这类的；重复匹配则不然，它对匹配的结果限制没有那么“精细”，只要符合了那个对应的模式，就匹配成功。这样的话<H1>匹配到<\H2>的操作就是合理的。

我们接下来使用回溯引用来匹配这个结果：

REGULAR EXPRESSION2 matches (160 steps, 0.2ms)

/ \<[Hh]([1-6])>.*?<\[/[Hh]\1>

TEST STRING

<BODY>
<H1>This is a home page.</H1>
FOR LEARNING.
<H2>It doesn't make any sence.</H2>
For I just illustrate some specical cases with these codes.
<H2>Notice that this line goes wrong.</H3>
For the label <H2> does not match </H3>.
</BODY>

前面在子表达式里面曾经有过匹配 is is 这样的案例，学习了回溯引用之后，就可以匹配满足所有这类格式的字符串部分了：

REGULAR EXPRESSION2 matches (167 steps, 0.1ms)

/ [.]([w+])[.]\1

TEST STRING

<BODY>
<H1>This is a home page.</H1>
FOR LEARNING.
<H2>It doesn't make any any sence.</H2>
For I just illustrate some some specical cases with these codes.
<H2>Notice that this line goes wrong.</H3>
For the label <H2> does not match </H3>.
</BODY>

注意：

由于子表达式是通过相对位置来引用的，所以有的时候如果子表达式的相对位置发生了变化或者是弄错了子表达式的位置，就会出现问题。当然现在有些re实现已经支持命名捕获功能（如.NET），类似python的格式化输出，即给某个子表达式一个唯一的名字，然后用名字来引用这个表达式。这个用到了再学，这里先跳过了。毕竟很多re支持的语法也不一样。

8.2 回溯引用在替换操作中的运用

简单的文本替换操作不需要正则表达式。没有必要做大材小用的事情。事实上这种操作只用普通的字符串处理就可以完成。

在需要使用回溯引用的时候，才用得到正则表达式。

这里实际上涉及到了我将来想实现的snippet功能，即把文本中的一段字符串替换为指定的格式。

替换的时候就需要用到回溯引用。

这里举两个例子，一个是将之前的重复文本替换成一个文本的示例：

REGULAR EXPRESSION

2 matches (167 steps, 0.1ms)

:/[.](\w+)[.]\1

/ gm

TEST STRING

```
<BODY>
<H1>This is a home page.</H1>
FOR LEARNING.
<H2>It doesn't make any any sence.</H2>
For I just illustrate some some specical cases with these codes.
<H2>Notice that this line goes wrong.</H3>
For the label <H2> does not match </H3>.
</BODY>
```

SUBSTITUTION

success (0.2ms)

\.\$1

```
<BODY>
<H1>This is a home page.</H1>
FOR LEARNING.
<H2>It doesn't make any any sence.</H2>
For I just illustrate some some specical cases with these codes.
<H2>Notice that this line goes wrong.</H3>
For the label <H2> does not match </H3>.
</BODY>
```

还有一个是把邮箱转成HTML里面可以点击的链接：

REGULAR EXPRESSION

1 match (242 steps, 0.1ms)

/

(\w+[\w\.]*)@([\w\.]+\.\w+)

/gm

TEST STRING

Hi, dcp.pbb@shu.edu.cn is my personal account(sake one).

SUBSTITUTION

success (0.2ms)

\$1

Hi, dcp.pbb@shu.edu.cn is my personal account(sake one).

所以总的来说替换操作需要用到两个正则表达式，一个是用来搜索的，另一个则是用来匹配文本的替换模式。

需要注意的是，JavaScript里面回溯引用的语法是\$而不是\；ColdFusion则在查找和替换操作里面都需要使用\，而不是\$。

有的re实现可以进行大小写转换，不过只有部分可以做到，所以就略去了。

8.3 小结

子表达式除了可以用在重复匹配操作之外，还可以用于回溯引用。

不同的实现语法不同，这一点需要注意！

回溯引用在文本替换和匹配里面非常有用。

9 前后查找

目前为止，用到的正则表达式都是用来匹配文本的。实际上，正则表达式还可以用来标记匹配文本的位置。这就是所谓前后查找。

9.1 何时使用前后查找

还是以HTML为例子，比如说需要匹配HTML中<TITLE>标签和</TITLE>中间隔的文字，随后进行某种修改或是整体替换；我们当然可以使用子表达式：

```
1 | <TITLE> This is for sure a snippet of text.</TITLE>
```

正则匹配式：

```
1 | <\[Tt]\[Ii]\[Tt][Ll]\[Ee]>(.\*)</\[Tt]\[Ii]\[Tt][Ll]\[Ee]>
```

对应的\$1就是我们想要的结果。

这个例子比较简单，但实际上处理事情的时候，“先检索再手动排除”在工程量大，匹配模式复杂的时候容易出错。这种时候实际上真正需要的是“前后查找”，即包含的匹配本身并不返回，但是是用来确定正确的匹配位置，而不是匹配结果的一部分。

9.2 向前查找

向前查找模式就是一个以?=为开头的子表达式，需要匹配的文本跟在=的后面。

匹配和匹配文本这一操作定义为“消费”。如果被匹配的文本包含在返回结果里，就称为消费；反正，就称为不消费。

比如有一些URL，我们需要将它们的协议名找出来，假设他们合法：

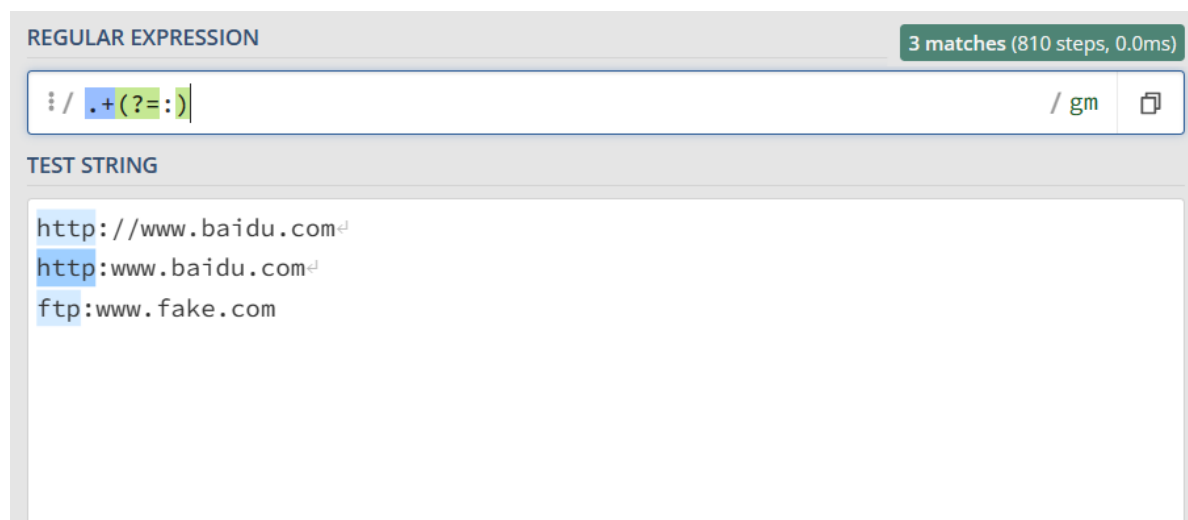
```
1 | http://www.baidu.com
2 | http:www.baidu.com
3 | ftp:www.fake.com
```

如果我们使用正则表达式

```
1 | .+(?:)
```

就会将冒号:也匹配进我们需要的结果里面。

如果我们“不消费”这个冒号:的话，向前匹配的子表达式(?:=:)就可以大显身手了：



The screenshot shows a web-based regular expression testing interface. At the top, it says 'REGULAR EXPRESSION' and '3 matches (810 steps, 0.0ms)'. The input field contains the regex `/ .+(?:=:)/`. Below, under 'TEST STRING', are three lines of text: `http://www.baidu.com`, `http:www.baidu.com`, and `ftp:www.fake.com`. The first two lines are highlighted in blue, indicating they are matches. The third line is not highlighted.

我们可以发现，后者虽然查询到了冒号但并没有显示在匹配结果里面。实际上这个匹配本身是有返回结果的，只是这个结果的字节长度是0而已。因此，前后查找操作有时被称为零宽度匹配操作。

任何一个子表达式都可以转换称为一个向前查找表达式，只需要加上?=前缀。并且一个搜索里面可以出现很多个查找表达式，他们的可以出现在模式里的任意位置。

9.3 向后查找

?=为向前查找操作符；向后查找操作符则是?<=（可以记成“向.....之后查找”）。

他们的用法几乎完全一样，必须用在一个子表达式里，后面是需要匹配的文本。

举一个可以运用向后查找的例子：

```
1 苹果：¥14.0 元/斤
2 香蕉：¥20.0 元/斤
3 香梨：¥16.0 元/斤
```

我们首先提取想要的水果价格：

REGULAR EXPRESSION

v1 3 matches (9 steps, 0.0ms)

:/ [\¥][\.\d]+ / gm

TEST STRING

苹果：¥14.0元/斤
香蕉：¥20.0元/斤
香梨：¥16.0元/斤
项目总数：3

但很多时候我们不需要前面的¥字符，但我们不能简单的把¥给去掉，否则就会把项目总数的3也匹配到。

这里就可以用到向后查找了：

```
1 (?<=¥)[\d\.]+
```

REGULAR EXPRESSION

v1 3 matches (17 steps, 0.1ms)

:/ (?<=¥)[\d\.]+ / gm

TEST STRING

苹果：¥14.0元/斤
香蕉：¥20.0元/斤
香梨：¥16.0元/斤
项目总数：3

这个正则表达式虽然匹配到了¥却没有消费，使得最终的结果只有价格数字。

9.4 向前查找结合向后查找

回到9.1中的例子，如果我们想要搜索标签<TITLE>和</TITLE>里面的文本，就需要结合使用这两者。

REGULAR EXPRESSION

v1 1 match (61 steps, 0.0ms)

:/ (?<=<[Tt][Ii][Tt][Ll][Ee]>).*?(?<=</[Tt][Ii][Tt][Ll][Ee]>) / gm

TEST STRING

<TITLE>This is for sure a snippet of text.</TITLE>

在这个例子里面，只有标题里的文字“被消费了”。另外为了防止歧义，最好在前面的向后查找里对后面一个标签的提示符<进行转义。

9.5 对前后查找取非

我们前面提到的查找文本的用法一般被称为“正向前查找”和“正向后查找”。前后查找还有一种用法叫做“负前后查找”。负向前查找将向前查找与不给定模式相匹配的文本，负向后查找将向后查找不与给定模式相匹配的文本。

注意我们在第三章里用到的取非符号^只能用于对字符集合进行取非处理（当然他还有一个用法是字符串首位匹配，不要忘记），它不能用于前后查找的取非处理。前后查找的区别在于等号和感叹号：

操作符	说明
(?=)	正向前查找
(?!)	负向前查找
(?<=)	正向后查找
(?<!)	负向后查找

一般来说，支持正向查找的re实现一般都支持负向查找。

比如如果只需要查找，且只查找数量，就需要用到取非操作，即只查找不以¥开头的数值：

REGULAR EXPRESSION v1 ▾ 2 matches (23 steps, 0.0ms)

:/ \b(?<![¥])\d+\b / gm 📋

TEST STRING

我在苹果上花费了¥100，在香蕉上花了¥20。苹果买了10个，香蕉买了1串。

注意这里如果没有单词边界\b的话，就会匹配出这样的结果：

REGULAR EXPRESSION v1 ▾ 4 matches (22 steps, 0.1ms)

:/ (?<![¥])\d+ / gm 📋

TEST STRING

我在苹果上花费了¥100，在香蕉上花了¥20。苹果买了10个，香蕉买了1串。

所以即使是很简单的例子，正则表达式也容易出错。

9.6 小结

正向查找和负向查找可以帮助我们进一步缩小搜索范围。但是有的re实现不支持负向查找。需要注意。

10 嵌入条件

嵌入条件是一个非常强大的功能，但是不常用到。这里引入嵌入条件来作为re学习的最后一个部分。

同样的需要注意，不是所有的re实现都支持条件处理。

10.1 嵌入条件

在过滤不合法格式的时候，常常需要用到这种做法。有些复杂一点的问题，只能通过条件处理来完成匹配。

10.2 正则表达式的条件

正则表达式里的条件用`?`来声明。这很符合问号`?`本身常用的情景，就好像在询问：“这个条件适用吗？”的样子。

事实上前面的正向前后搜索里面的`?`本质上也是一个“条件”。

单个的`?`匹配前一个字符，如果该字符存在；

`?=`和`<=`匹配前后文本，如果文本（也可以是子表达式）存在。

嵌入条件只在两种情况下使用。一种是根据一个回溯引用来条件处理；另一种则是根据一个前后查找来条件处理。

10.2.1 基于回溯引用的条件处理

首先这个条件处理只在前面子表达式匹配成功的情况下才可以使用。有点类似于条件控制语句 `if...then...else....` 可以用来进行两个分别在情况满足和不满足的情况下所执行的匹配操作。

用来定义这种语法的方式是：

```
1 | (? (回溯引用) 真情况表达式 | 假情况表达式)
```

比如我们想要解决一个正则表达式的经典运用例子，美国电话号码匹配：

```
1 | 792-113-3211
2 | (123)232-3231
3 | (321)-223-2132
4 | (123-132-2323
5 | 1232323231
6 | 123 131 1232
```

这里面只有第一行和第二行的类型是合理的，如果需要匹配电话号码，只能使用条件处理：

```
1 | (\(\)\d{3}(?(1)\)|-)\d{3}-\d{4})
```

这里首先把`\()`括成一个子表达式`\()`，接着匹配之后的三个数字，随后开始条件处理：若前面没括号，匹配连字符，然后继续后面的匹配操作；若前面有括号，匹配括号，然后继续后面的匹配操作。这里面`?(1)`就是一个回溯引用(back reference)条件，它对应的就是前面半个括号的匹配操作。

10.2.2 基于前后查找的条件处理

嵌入了前后查找的条件模式相当少见，一般来说可以拿更简单的办法来完成同样的目的。

前后查找的条件处理语法和回溯引用几乎差不多，只需要把回溯引用里面的编号替换成完整的前后查找表达式即可。

比如有一串文本（假设相邻的文本之间只有\n分割），假设这些文本里面只由数字和连字符组成。我们要求匹配：连续五个数字字符；或者连续五个数字字符-四个数字字符。

```
1 | 13213
2 | -1232
3 | 145-123
4 | 12344-
5 | 12434-54
6 | 24324-3434
```

如果需要使用前后查找条件处理，可以这么写：

```
1 | (?m)^(?d{5}(?(?=-)-\d{4}))$
```

TEST STRING

```
13213
-1232
145-123
12344-
12434-54
24324-3434
```

当然还可以这么写：

```
1 | (\b\d{5}\b\n)|(\b\d{5}-\d{4}\b\n)
```

注意这里的\b必不可少。虽然看上去后者表达式长了些，但其实比前者更容易理解。两者的效果是一样的。

REGULAR EXPRESSION

v1 v

2 matches (129 steps, 0.1ms)

```
:/ (\b\d{5}\b\n)|(\b\d{5}-\d{4}\b\n)
```

/ gm



TEST STRING

```
13213
-1232
145-123
12344-
12434-54
24324-3434
224324-34341
```

10.3 小结

本节介绍了嵌入条件处理. 在现实运用中, 一定要化繁为简, 尽可能地用简单的表达式. 如果不行, 就分开来单独处理, 然后小心地使用条件处理.