

# 目录

## 1 文档简介

- 1.1 作者简介
- 1.2 阅读建议
- 1.3 版权声明

## 2 Python 正则表达式

- 2.1 基本概念
- 2.2 正则匹配规则表
  - 2.2.1 基本字符规则：
  - 2.2.2 预定义字符集：
  - 2.2.3 常用字符集：
  - 2.2.4 数量词：
  - 2.2.5 边界匹配器：
  - 2.2.6 逻辑、分组：
  - 2.2.7 非捕获组与环视：
  - 2.2.8 匹配模式和注释：
  - 2.2.9 基本字符集匹配测试
- 2.3 贪婪模式、非贪婪模式和独占模式
  - 2.3.1 贪婪模式 ( Greedy )
  - 2.3.2 非贪婪模式 ( Lazy )
  - 2.3.3 贪婪与非贪婪模式对比示例
  - 2.3.4 回溯算法
  - 2.3.5 独占模式 ( Possessive )
- 2.4 原生字符串简化反斜杠 \ 转义问题
- 2.5 分组
  - 2.5.1 分组编号的计算规则
  - 2.5.2 命名分组
- 2.6 断言 ( Assertion )
  - 2.6.1 单词边界 ( Word Boundary )
  - 2.6.2 行的开始或结束
  - 2.6.3 环视 ( Look Around )
- 2.7 Python的re模块
- 2.8 flags标志位
  - 2.8.1 ASCII和UNICODE模式
  - 2.8.2 IGNORECASE模式
  - 2.8.3 MULTILINE模式
  - 2.8.4 DOTALL模式
  - 2.8.5 VERBOSE模式
  - 2.8.6 TEMPLATE模式
  - 2.8.7 DEBUG模式
- 2.9 正则匹配
  - 2.9.1 基本函数
  - 2.9.2 `re.MatchObject` 对象
  - 2.9.3 匹配手机号码
  - 2.9.4 匹配邮箱地址
  - 2.9.5 匹配时引用分组
- 2.10 正则查找
- 2.11 正则替换
  - 2.11.1 基本替换
  - 2.11.2 环视替换
  - 2.11.3 repl替换表达式引用分组
  - 2.11.4 repl替换表达式使用函数

- 2.12 正则切割
- 2.13 compile编译正则表达式
- 2.14 其他

### 3 补充资料

- 3.1 正则表达式的历史与流派
  - 3.1.1 正则表达式简史
  - 3.1.2 正则表达式流派
    - 3.1.2.1 POSIX 流派
    - 3.1.2.2 PCRE 流派
  - 3.1.3 在Linux中使用正则
- 3.2 正则的匹配原理以及优化原则
  - 3.2.1 有穷状态自动机
  - 3.2.2 正则的匹配过程
  - 3.2.3 DFA& NFA 工作机制
  - 3.2.4 POSIX NFA 与 传统 NFA 区别
  - 3.2.5 回溯详解

# 1 文档简介

---

## 1.1 阅读建议

---

本文档本身可能并不适合初学者学习，但非常适合对正则有一定基础了解的朋友系统性学习。

对于初学者建议在B站找两部正则入门教程入门之后再学习本文档。

初学者直接阅读本文档，可以先跳过正则匹配规则表部分，不要纠结，等学完后续部分之后再回来  
看。正则匹配规则表主要针对已经掌握正则的朋友，随时回来查询规则。

---

---

## 2 Python 正则表达式

---

### 2.1 基本概念

---

正则表达式在每种编程语言中都具有相同的概念，整体规则都大致一致，只是部分语言没有实现少部分规则。

正则表达式的本质就是用一些特定字符的组合，组成一个“规则字符串”表达对字符串的一种过滤逻辑，可以很方便的从指定的字符串中提取出我们想要的内容。

python正则表达式的官方文档是：<https://docs.python.org/zh-cn/3.7/library/re.html>

一个优秀的正则测试网站：<https://regex101.com/>

本地正则测试软件（依赖.Net 4.8）：<https://deerchao.cn/tools/regester/index.htm>

下面我们看一下正则规则表，可以在以后需要的时候，随时回来查询相应的规则：

### 2.2 正则匹配规则表

---

#### 2.2.1 基本字符规则：

模式	描述	实例	完整字符串
一般字符	匹配自身	abc	abc
.	默认匹配除了换行符以外的任意字符，匹配模式指定 re.DOTALL 标记时，也可以匹配包括换行符的任意字符。	a.c	abc
\	转义模式字符作为被匹配的字符，例如字符 * 需要匹配，可以使用 \* 或 [*]	a\.c a\\c	a.c a\c
[...]	用来表示一组字符，例如 [amk] 匹配 'a', 'm' 或 'k'	a[bcd]e	abe ace ade
[^...]	匹配不在 [] 中的字符：例如 [^abc] 匹配除了 a,b,c 之外的字符。	a[^bcd]e	aae afe age
\t	制表符 ('\\u0009')		
\n	新行（换行）符 ('\\u000A')		
\r	回车符 ('\\u000D')		
\f	换页符 ('\\u000C')		
\a	报警 (bell) 符 ('\\u0007')		
\e	转义符 ('\\u001B')		

## 2.2.2 预定义字符集：

（可以写在字符集 [...] 中）

模式	描述	实例	完整字符串
\d	数字：[0-9]	a\dc	a1c
\D	非数字：[^0-9]	a\Dc	abc
\s	空白字符：[<空格>\t\r\n\v\f]	a\sc	a c
\S	非空白字符：[^\s]	a\Sc	abc
\w	单词字符：[a-zA-Z_0-9]	a\wc	abc
\W	非单词字符：[^\w]	a\Wc	a c

## 2.2.3 常用字符集：

模式	描述
[\u4e00-\u9fa5]	中文字符
[A-Za-z0-9]	英文和数字
[A-Za-z]	26个英文字母
[A-Z]	26个大写英文字母
[a-z]	26个小写英文字母
\w	数字、26 个英文字母或者下划线
[\u4E00-\u9FA5A-Za-z0-9_]	中文、英文、数字包括下划线
[\u4E00-\u9FA5A-Za-z0-9]	中文、英文、数字 但不包括下划线等符号
[^%&',;=?\$\\x22]	^%&',;=?\$"等字符
[^\x00-\xff]	双字节字符，可以用来计算字符串的长度(一个双字节字符长度计 2，ASCII 字符计1)

## 2.2.4 数量词：

模式	描述	实例	完整字符串
*	贪婪模式，匹配前1个字符0次或多次， <code>*?</code> 表示非贪婪模式， <code>*+</code> 表示独占模式（re模块不支持）	abc*	ab abcccc
+	贪婪模式，匹配前1个字符1次或多次， <code>+?</code> 表示非贪婪模式， <code>++</code> 表示独占模式（re模块不支持）	abc+	abc abcccc
?	匹配前一个字符0次或1次，同样分为贪婪模式(默认)，非贪婪模式(加?)和独占模式(加+)	abc?	ab abc
{m}	匹配前一个字符m次	ab{2}c	abbc
{m,n}	{m,n}匹配前一个字符m至n次 {,n}匹配前一个字符0至n次 {m,}匹配前一个字符m至多次 同样分为贪婪模式(默认)，非贪婪模式(加?)和独占模式(加+)	ab{1,2}c	abc abbc

## 2.2.5 边界匹配器：

模式	描述	实例	完整字符串
<code>^</code>	行的开头，多行模式中每一行的开头	<code>^abc</code>	abc
<code>\$</code>	行的结尾，多行模式中每一行的结尾	<code>abc\$</code>	abc
<code>\b</code>	单词边界，即匹配 <code>\w</code> 和 <code>\W</code> 之间的内容	<code>a\b!bc</code>	a!bc
<code>\B</code>	非单词边界： <code>[\^\b]</code>	<code>a\Bbc</code>	abc
<code>\A</code>	仅匹配整个字符串的开头	<code>\Aabc</code>	abc
<code>\Z</code>	仅匹配整个字符串的结尾	<code>abc\Z</code>	abc

## 2.2.6 逻辑、分组：

模式	描述	实例	完整字符串
<code> </code>	<code> </code> 表示左右表达式任意匹配一个，没有被包括在 <code>()</code> 中时，则它的范围是整个正则表达式	<code>abc def</code>	abc def
<code>(...)</code>	被括起来的表达式将作为分组，每遇到一个左括号 <code>(</code> ，分组编号+1。 分组表达式可接数量词， <code> </code> 仅对当前分组有效。	<code>(abc){2}</code> <code>a(123 456)c</code>	abcabc a123c 和 a456c
<code>(?P&lt;name&gt;...)</code>	给分组指定一个额外的别名	<code>(?P&lt;id&gt;abc){2}</code>	abcabc
<code>\&lt;number&gt;</code>	引用编号为 <code>\&lt;number&gt;</code> 的分组匹配到的字符串	<code>(\d)abc\1</code>	1abc1 5abc5
<code>(?P=name)</code>	引用别名为 <code>&lt;name&gt;</code> 的分组匹配到的字符串	<code>(? P&lt;id&gt;\d)abc(? P=id)</code>	1abc1 5abc5
<code>(? (id/name)Y N)</code>	如果编号为 <code>id</code> 或别名为 <code>name</code> 的组匹配到字符，则需要匹配 <code>Y</code> ，否则需要匹配 <code>N</code>	<code>(\d)abc(? (1)\d efg)</code>	1abc2 abcefg

## 2.2.7 非捕获组与环视：

模式	描述	实例	完整字符串
<code>(?:...)</code>	非捕获组，表示这个括号内的内容不作为分组	<code>(?:abc){2}</code>	abcabc
<code>(?=...)</code>	肯定环视，表示右边是指定内容的位置	<code>a(?\d)</code>	右边是数字的a
<code>(?!...)</code>	否定环视，表示右边不是指定内容的位置	<code>a(?!\d)</code>	右边不是数字的a
<code>(?&lt;=...)</code>	肯定逆序环视，表示左边是指定内容的位置	<code>(?&lt;=\d)a</code>	左边是数字的a
<code>(?&lt;!=...)</code>	否定逆序环视，表示左边不是指定内容的位置	<code>(?&lt;!\d)a</code>	左边不是数字的a

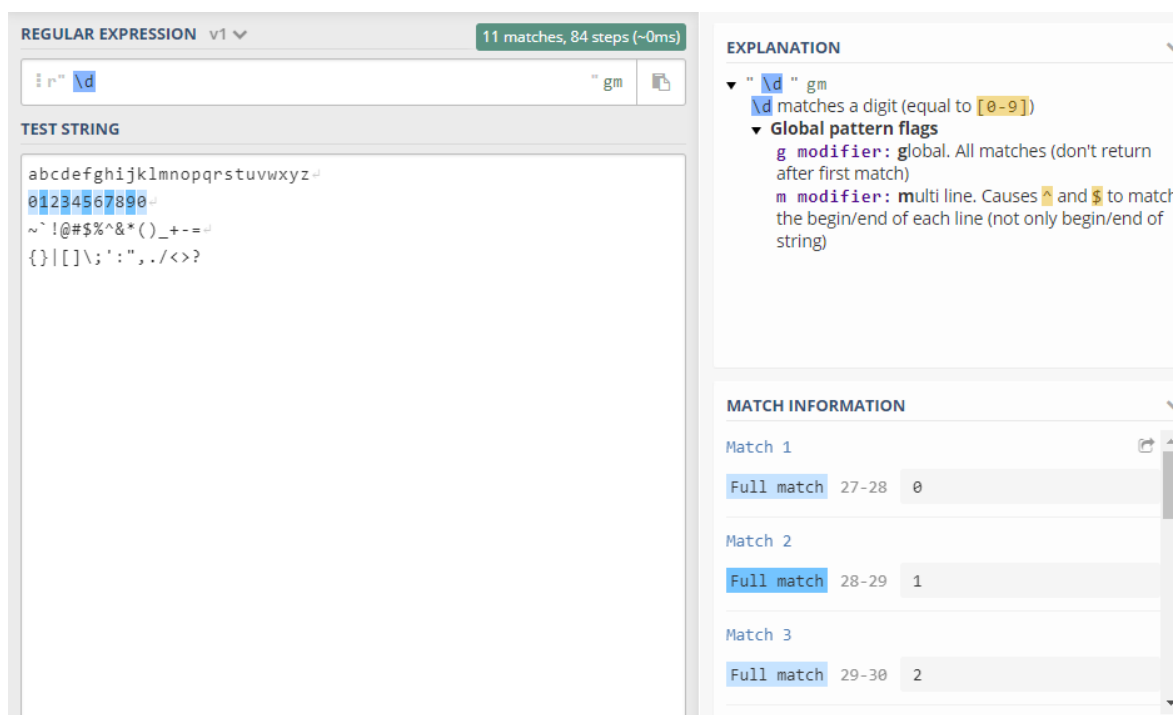
## 2.2.8 匹配模式和注释：

模式	描述	实例	完整字符串
(? aiLmsux)	aiLmsux的每个字符代表一个匹配模式（例如i代表忽略大小写），只能在字符串开头使用	(?i)abc	AbC
(?#...)	表示注释，将被直接忽略	abc(? #comment)123	abc123

(`'a'`, `'i'`, `'L'`, `'m'`, `'s'`, `'u'`, `'x'` 中的一个或多个) 这个组合匹配一个空字符串；这些字符对正则表达式设置以下标记 `re.A` (只匹配ASCII字符), `re.I` (忽略大小写), `re.L` (语言依赖), `re.M` (多行模式), `re.S` (点dot匹配全部字符), `re.U` (Unicode匹配), and `re.X` (冗长模式)。如果你需要将这些标记包含在正则表达式中，这个方法就很有用，免去了在 `re.compile()` 中传递 **flag** 参数。

## 2.2.9 基本字符集匹配测试

打开我之前保存的正则：<https://regex101.com/r/GN99Cs/1>



REGULAR EXPRESSION v1 11 matches, 84 steps (~0ms)

REGEX: `\d` gm

TEST STRING

abcdefghijklmnopqrstuvwxyz  
0123456789  
~`!@#\$%^&\*()\_+-=  
{ } | [ ] \ ; ' : " , . / < > ?

EXPLANATION

▼ `"\d" gm`  
`\d` matches a digit (equal to `[0-9]`)  
▼ Global pattern flags  
`g` modifier: global. All matches (don't return after first match)  
`m` modifier: multi line. Causes `^` and `$` to match the begin/end of each line (not only begin/end of string)

MATCH INFORMATION

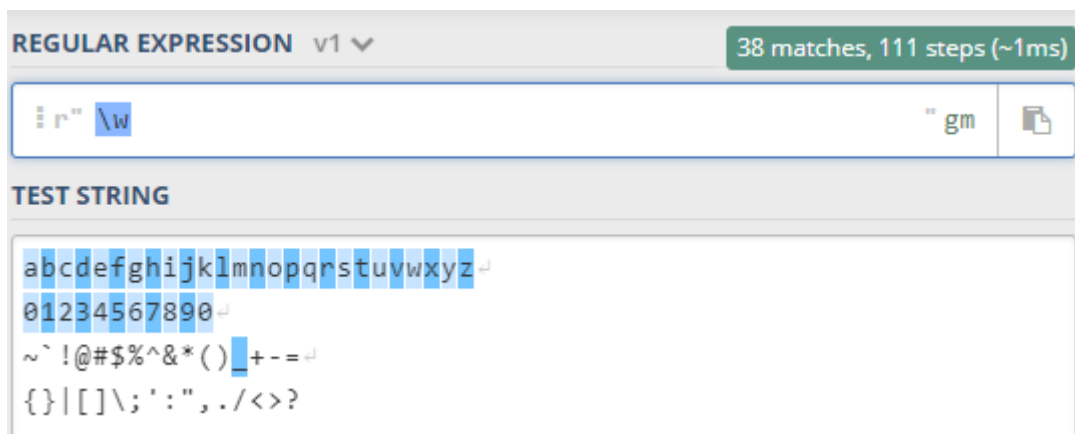
Match 1  
Full match 27-28 0

Match 2  
Full match 28-29 1

Match 3  
Full match 29-30 2

可以很清晰的看到数字字符集 `\d` 成功的匹配了所有的单个数字。

假如改成 `\w`：



顺利的匹配所有的数字、字母和下划线。

你还可以自己测试上述正则匹配规则表中的各类字符集。

## 2.3 贪婪模式、非贪婪模式和独占模式

为什么会有贪婪与非贪婪模式呢？我们先来回顾一下正则中表示数量词的规则（前面的正则规则表中有）：

- `*`：0次到多次
- `+`：1次到多次
- `?`：0次到1次
- `{m}`：m次
- `{m,n}`：m至n次
- `{,n}`：0至n次
- `{m,}`：m至多次

`{m,n}` 可以等价表示 `*`, `+`, `?` 这 3 种数量词：

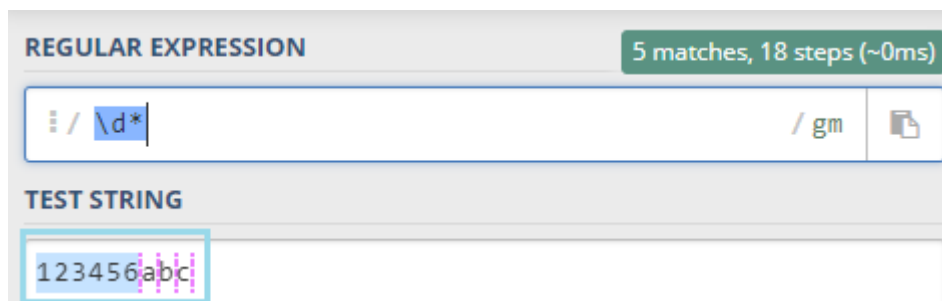
- `*`：`{0,}`
- `+`：`{1,}`
- `?`：`{1}`或`{0,1}`(部分编程语言只支持)

数量词 `+` 和 `*` 并没有我们想象的那么简单，测试一下：

```
1 >>> import re
2 >>> re.findall('\d+', '123456abc')
3 ['123456']
4 >>> re.findall('\d*', '123456abc')
5 ['123456', '', '', '', '']
```

可以看到 `*` 号额外匹配了4个空字符串，通过<https://regex101.com/>可以清楚的看到这4个空字符串（粉红色竖线）所在的位置。





\* 表示0次到多次，由于可以匹配0次所以匹配到了空字符串。结果是否令你有些难以理解，那么我们接下来看看贪婪模式和非贪婪模式具体的概念。

**贪婪模式**就是尽可能匹配**更多**的字符，**非贪婪模式**则是尽可能匹配**最少**的字符，这两种匹配模式会产生不同的匹配结果。

### 2.3.1 贪婪模式 ( Greedy )

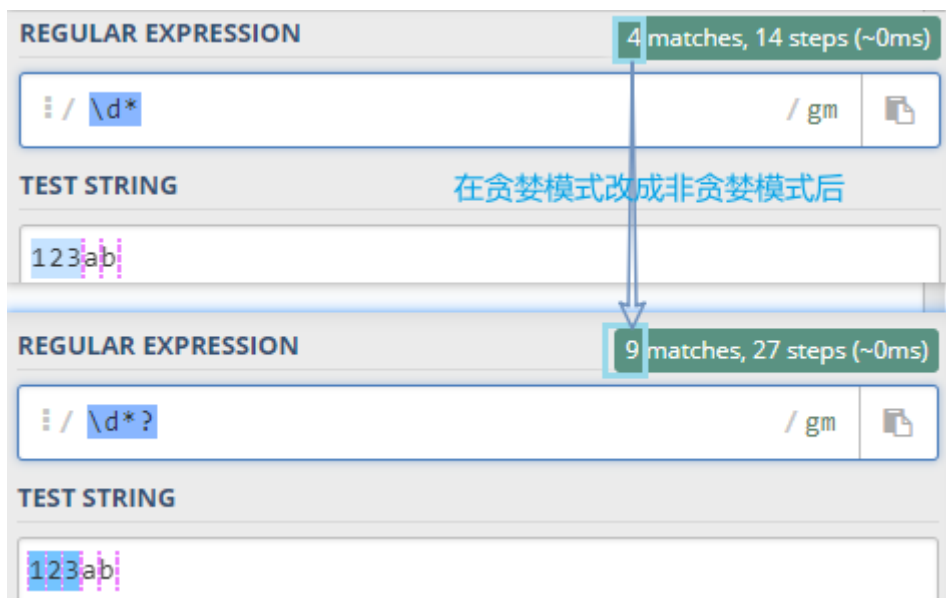
我们先看一下贪婪匹配下，字符串 123456ab 中使用正则 \d\* 的匹配过程：

字符串	1	2	3	4	5	6	a	b
下标	0	1	2	3	4	5	6	7
第1次匹配到字母a时发现不满足 输出123456							↑	
第2次匹配剩下的ab发现匹配不上							↑	
第3次匹配剩下的b发现匹配不上 输出空字符串								↑
第4次匹配剩下的空字符串 输出空字符串								

\d\* 在匹配开头的数字时，会尝试尽量匹配更多的数字，直到出现第一个字母a 不满足要求为止，匹配全部数字后每次匹配都得到了空字符串。

### 2.3.2 非贪婪模式 ( Lazy )

在数量词后面加上英文的问号 (?)，正则就变成了 \d\*? 就是非贪婪模式：



可以看到非贪婪模式下，原本的4项匹配成功变成了9项：

```
1 >>> import re
2 >>> re.findall('\d*', '123ab')
3 ['123', '', '', '']
4 >>> re.findall('\d*?', '123ab')
5 ['', '1', '', '2', '', '3', '', '', '']
```

非贪婪模式下匹配到的结果都是单个的数字，就连每个数字左边的空字符串也匹配上了。

### 2.3.3 贪婪与非贪婪模式对比示例

以下代码首先需执行：

```
1 import re
```

#### 示例1

匹配出数字后面的0：

```
1 >>> re.fullmatch('(\d+)(0*)', '102300').groups()
2 ('102300', '')
```

由于`\d+`采用贪婪匹配，直接把后面的0全部匹配了，`0*`就只能匹配空字符串了。

加个`?`就让`\d+`采用非贪婪匹配，把后面的0匹配出来：

```
1 >>> re.fullmatch('(\d+?)(0*)', '102300').groups()
2 ('1023', '00')
```

#### 示例2

比如有一批需要提取出用户名的邮件地址：

```
1 addrs = ['<Tom Paris> tom@voyager.org',
2          'tom@voyager.org', 'bill.gates@microsoft.com']
```

我们希望提取出其中的用户名：`Tom Paris`、`tom`和`bill.gates`。

假如采用默认的贪婪模式，为了匹配开头的<之类的字符，则必须使用非单词字符\W：

```
1 for addr in addrs:
2     print(re.match('\W*([\w\s\.]*)', addr).group(1))
```

但采用非贪婪匹配，开头我们就可以直接使用任意字符 .：

```
1 for addr in addrs:
2     print(re.match('.*?([\w\s\.]*)', addr).group(1))
```

返回的结果都是：

```
1 Tom Paris
2 tom
3 bill.gates
```

## 2.3.4 回溯算法

不管是贪婪模式，还是非贪婪模式，都需要发生回溯才能完成相应的功能。回溯算法是正则表达式里最重要的一种算法思想，依次考察正则表达式中的每个字符，非通配符时就直接跟文本的字符进行匹配，相同则继续往下处理；不同则回溯。

比如遇到遇到正则表达式的 "xy{1,3}z" 有多种匹配方案，就先随意的选择一种匹配方案，然后继续考察剩下的字符。如果中途发现无法继续匹配下去了，就回到这个岔路口，重新选择一种匹配方案，然后再继续匹配剩下的字符。

默认贪婪模式下：

```
1 regex = "xy{1,3}z"
2 text = "xyyz"
```

y{1,3}会尽可能长地去匹配，当匹配完 xyy 后，由于 y 要尽可能匹配最长即三个，但字符串中后面是个 z 就会导致匹配不上，这时候正则就会**向前回溯**，吐出当前字符 z，接着用正则中的 z 去匹配。

regex = "xy{1,3}z" text = "xyyz"	x	y	y	z	
	x	y	y	y	z
使用最长的yyy去匹配，从匹配失败的位置吐出，下次匹配继续					z

非贪婪模式下：

```
1 regex = "xy{1,3}?z"
2 text = "xyyz"
```

由于 y{1,3}? 代表匹配 1 到 3 个 y，尽可能少地匹配。匹配上一个 y 之后，也就是在匹配上 text 中的 xy 后，正则会使用 z 和 text 中的 xy 后面的 y 比较，发现正则 z 和 y 不匹配，这时正则就会**向前回溯**，重新查看 y 匹配两个的情况，匹配上正则中的 xyy，然后再用 z 去匹配 text 中的 z，匹配成功。

<div> <div>regex = "xy{1,3}?z"</div> <div>text = "xyyz"</div> </div>		x	y	y	z
		x	y	z	
<div>使用最短的y去匹配，从匹配失败的位置吐出，下次匹配继续</div>			y	z	

### 2.3.5 独占模式 ( Possessive )

在一些场景下，我们不需要回溯，匹配不上返回失败就好了，因此正则中还有另外一种模式，独占模式，它类似贪婪匹配会尽可能多地去匹配，但匹配失败就结束，不会进行回溯，因此在一些场合下性能会更好，具体的方法就是在数量词后面加上加号+。

独占模式下：

```
1 regex = "xy{1,3}+yz"
2 text = "xyyz"
```

y{1,2}+尽可能多的匹配前两个y，不回溯导致正则z前面的y匹配不上：

<div> <div>regex = "xy{1,2}+yz"</div> <div>text = "xyyz"</div> </div>		x	y	y	z
		x	y	y	
<div>y{1,2}+尽可能多的匹配前两个y 不回溯导致正则z前面的y匹配不上</div>			y	z	

python的标准库re并不支持独占模式，会报错：

```
1 >>> import re
2 >>> re.findall('xy{1,3}+z', 'xyyz')
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5   File "D:\Anaconda3\lib\re.py", line 223, in findall
6     return _compile(pattern, flags).findall(string)
7   File "D:\Anaconda3\lib\re.py", line 286, in _compile
8     p = sre_compile.compile(pattern, flags)
9   File "D:\Anaconda3\lib\sre_compile.py", line 764, in compile
10    p = sre_parse.parse(p, flags)
11   File "D:\Anaconda3\lib\sre_parse.py", line 930, in parse
12    p = _parse_sub(source, pattern, flags & SRE_FLAG_VERBOSE, 0)
13   File "D:\Anaconda3\lib\sre_parse.py", line 426, in _parse_sub
14     not nested and not items))
15   File "D:\Anaconda3\lib\sre_parse.py", line 654, in _parse
16     source.tell() - here + len(this))
17 re.error: multiple repeat at position 7
```

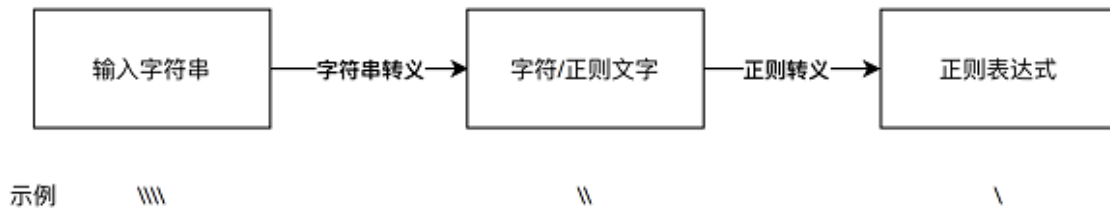
python上要使用独占模式需要安装regex 模块：

```
1 pip install regex
```

再次测试：

在几乎所有的编程语言中 \ 都被作为转义字符，\<某个字母> 在正则表达式也表达了某种预定义字符集或边界，这导致我们需要匹配 \ 时我们需要写4个反斜杠。

在程序使用过程中，从输入的字符串到正则表达式，其实有两步转换过程，分别是字符串转义和正则转义。：



例如，对于字符串 `"F:\\video"`，我们需要提取左边的盘符和右边的路径时，需要这样写：

```
1 s = "F:\\video"
2 re_match = re.match('(.*?)\\\\\\\\(\\.+)', s)
3 re_match.group(1), re_match.group(2)
```

但由于Python支持原生字符串，我们就可以这样写：

```
1 s = r"F:\\video"
2 re_match = re.match(r'(.*?)\\\\(\\.+)', s)
3 re_match.group(1), re_match.group(2)
```

结果均为：

```
1 ('F:', 'video')
```

有了原生字符串，写出来的表达式会更加直观。

### 括号的转义：

在正则中方括号 `[]` 和花括号 `{}` 只需转义开括号，但圆括号 `()` 两个都要转义。我在下面给了你一个比较详细的例子。

```
1 >>> import re
2 >>> re.findall('\\(\\)\\[\\]\\{\\}', 'O[]{}')
3 ['O[]{}']
4 >>> re.findall('\\(\\)\\[\\]\\{\\}', 'O[]{}') # 方括号和花括号都转义也可以
5 ['O[]{}']
```

### 使用函数消除元字符特殊含义：

`re`模块自带的转义函数`escape`也可以实现转义：

```

1 >>> import re
2 >>> re.escape('\d') # 反斜杠和字母d转义
3 '\\d'
4 >>> re.findall(re.escape('\d'), '\d')
5 ['\d']
6 >>> re.escape('[+]') # 中括号和加号
7 '\\[\\+\\]'
8 >>> re.findall(re.escape('[+]'), '[+]')
9 ['[+]']

```

## 2.5 分组

### 2.5.1 分组编号的计算规则

前面的正则匹配规则表中说过：

(...) 被括起来的表达式将作为分组，每遇到一个左括号(，分组编号+1。

表示正则表达式中的分组通过从左到右计算其开括号来编号。

例如表达式 ((A)(B(C))) 存在四个分组：

```

1 ((A)(B(C)))
2 (A)
3 (B(C))
4 (C)

```

而组0始终代表整个表达式 ((A)(B(C)))。

```

1 match_obj = re.match("((\\w)(\\w(\\w)))", "ABC")
2 print(match_obj.groups())
3 print(match_obj.group(0))

```

结果：

```

1 ('ABC', 'A', 'BC', 'C')
2 ABC

```

分组表达式可接数量词，| 仅对当前分组有效

又是什么含义呢？

假如我们有一批字符串：

```

1 brands = [
2     "联想/LENOVO",
3     "狮乐/SHILE",
4     "美的/Midea",
5     "联想/LENOVO",
6     "松下/Panasonic",
7     "红叶/RedLeaf",
8     "纳米亚",
9     "富士施乐/FujiXerox",
10    "佳印",
11    "佳能/CANON",
12    "TCL"
13 ]

```

我们希望提取每个品牌的中文名称和英文名称，但是可能有些品牌只有中文名称，或者只有英文名称。这时我们可以在对应的分组加上数量词`?`表示可能出现一次也可能不出现：

```

1 for brand in brands:
2     print(brand, re.match(r"([\w/]+)?/?(\w+)?", brand, re.A).groups())

```

结果：

```

1 联想/LENOVO ('联想', 'LENOVO')
2 狮乐/SHILE ('狮乐', 'SHILE')
3 美的/Midea ('美的', 'Midea')
4 联想/LENOVO ('联想', 'LENOVO')
5 松下/Panasonic ('松下', 'Panasonic')
6 红叶/RedLeaf ('红叶', 'RedLeaf')
7 纳米亚 ('纳米亚', None)
8 富士施乐/FujiXerox ('富士施乐', 'FujiXerox')
9 佳印 ('佳印', None)
10 佳能/CANON ('佳能', 'CANON')
11 TCL (None, 'TCL')

```

这样就顺利的匹配到每个品牌对应的中文名称和英文名称。

## 2.5.2 命名分组

上面讲了分组编号的计算规则，但由于编号得数在第几个位置，后续如果发现正则有问题改动了括号的个数就会导致编号发生变化。因此正则表达式提供了命名分组（named grouping）的规范，命名分组的格式为`(?P<分组名>正则)`。

比如在 Django 的路由中，命名分组示例如下：

```

1 urlpatterns = [
2     re_path(r'^bio/(?P<username>\w+)/$', views.bio, name='bio'),
3     re_path("^index/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/$", views.index),
4 ]

```

示例：



```
1 s = 'A23G4HFD567'
2 match_obj = re.search("(?P<value>\d+)", s)
3 # 可以通过传入对应命名获取对应分组
4 print(match_obj.group("value"))
```

结果：

```
1 23
```

## 2.6 断言 ( Assertion )

断言是指对匹配到的文本位置有要求，对应于上面正则匹配规则表中的边界匹配器和环视。

### 2.6.1 单词边界 ( Word Boundary )

假如我们想要把下面文本中的 tom 替换成 jerry。注意一下，在文本中出现了 tomorrow 这个单词，tomorrow 也是以 tom 开头的。

```
tom asked me if I would go fishing with him tomorrow.
```

这个时候就要求只匹配单词tom而不匹配包含tom的单词，实现这个效果最佳的办法就是使用单词边界，可以看看效果：

The image displays three sequential screenshots of a regular expression testing interface, demonstrating how word boundaries affect matching results.

- First Screenshot:**
  - REGULAR EXPRESSION:** `/ tom` (4 matches, 18 steps (~1ms))
  - TEST STRING:** `tom asked me if I would go fishing with him tomorrow.atom and atomic.`
  - Matches:** Four matches are shown, highlighting the 'tom' in 'tom', 'tomorrow', 'atom', and 'atomic'.
- Second Screenshot:**
  - REGULAR EXPRESSION:** `/ \btom` (2 matches, 13 steps (~0ms))
  - TEST STRING:** Same as the first screenshot.
  - Matches:** Two matches are shown, highlighting 'tom' in 'tom' and 'atom'.
- Third Screenshot:**
  - REGULAR EXPRESSION:** `/ \btom\b` (1 match, 14 steps (~0ms))
  - TEST STRING:** Same as the first screenshot.
  - Matches:** One match is shown, highlighting 'tom' in 'tom'.

于是我们的正则就可以编写为：

```
1 test_str = "tom asked me if I would go fishing with him tomorrow.atom and atomic."
2 re.sub('\btom\b', 'jerry', test_str)
```

结果：

```
1 'tom asked me if I would go fishing with him tomorrow.atom and atomic.'
```

## 2.6.2 行的开始或结束

一行文本的开头或结尾可以使用 `^` 和 `$` 来进行位置界定。在Windows、Linux、macOS 平台上换行的表示方式略有不同，其中Windows上是`\r\n`，Linux上是`\n`，Mac上是`\r`。

在多行模式下，`^` 和 `$` 符号可以匹配每一行的开头或结尾，但Python的re模块实现的正则默认不是多行匹配模式。

我们还可以使用 `\A` 和 `\Z`（其他语言中使用`\z`）来匹配整个文本的开头或结尾。

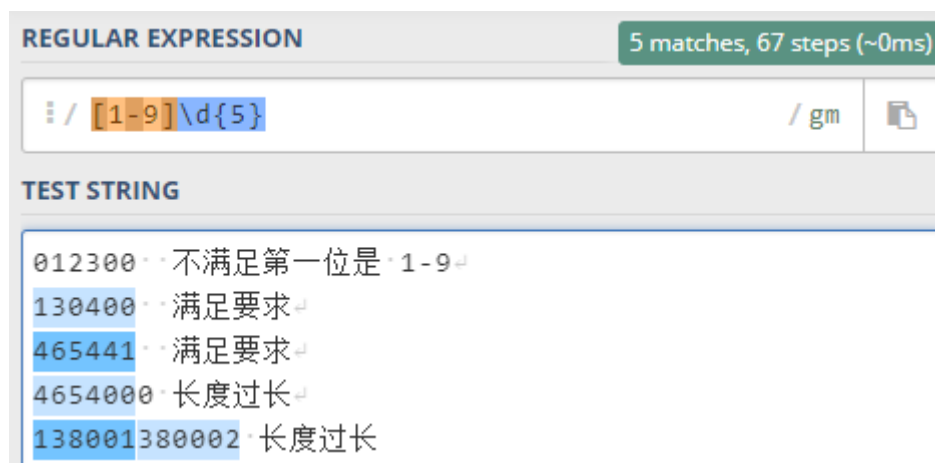
## 2.6.3 环视（Look Around）

环视就是要求匹配部分的前面或后面要满足（或不满足）某种规则，有些地方（例如java）也称环视为**零宽断言**。

举个会用到环视例子，邮政编码的规则是第一位是 1-9，一共有 6 位数字组成。现在要求提取文本中的邮政编码。

根据规则，我们很容易就可以写出邮编的组成 `[1-9]\d{5}`。然后测试下面的文本：

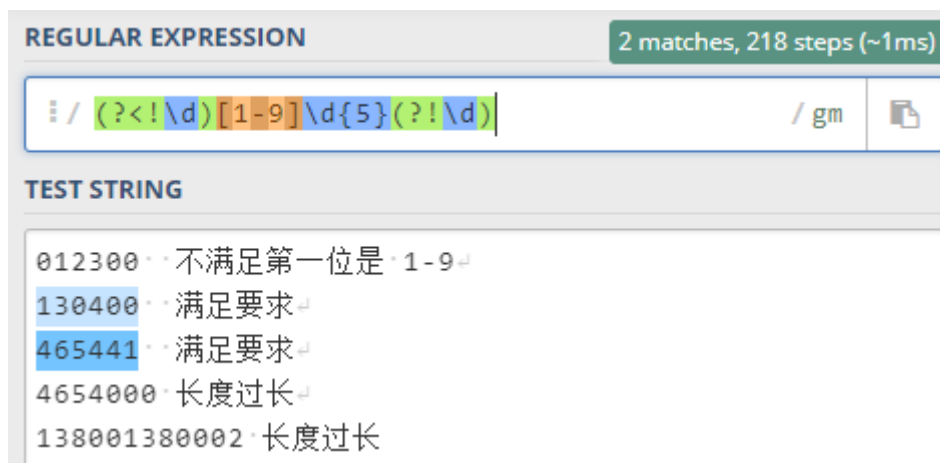
```
1 012300 不满足第一位是 1-9
2 130400 满足要求
3 465441 满足要求
4 4654000 长度过长
5 138001380002 长度过长
```



结果，7 位数的前 6 位也能匹配上，12 位数匹配上了两次，这显然是不符合要求的。

环视的规则可能难以记住，我们可以总结一下：**左尖括号代表看左边，没有尖括号看右边，感叹号是非的意思。**

我们可以通过环视限定左边不是数字，右边也不是数字的开头不是0的 6 位数：`(?<!\d)[1-9]\d{5}(?!0\d)`



环视也可以实现单词边界一样的作用：`\b\w+\b` 也可以写成 `(?<!\w)\w+(?!\\w)`。

表示左右两边都不能是单词。

注意：环视虽然有括号但与非捕获组一样，不会产生子组。

## 2.7 Python的Re模块

要使用python的正则表达式，则需要使用re模板，下面我们简单看看re模块所具备的方法，然后再详解。

正则匹配：

```
1 re.match(pattern, string, flags=0)
2 或
3 re.search(pattern, string, flags=0)
4 或
5 re.fullmatch(pattern, string, flags=0)
```

正则替换：

```
1 re.sub(pattern, repl, string, count=0, flags=0)
2 或
3 re.subn(pattern, repl, string, count=0, flags=0)
```

正则查找：

```
1 re.findall(pattern, string, flags=0)
2 或
3 re.finditer(pattern, string, flags=0)
```

正则切割：

```
1 re.split(pattern, string, maxsplit=0, flags=0)
```

re模块的4种使用方法中都有3个共同的参数：

参数	描述
pattern	匹配的正则表达式
string	被匹配的字符串
flags	标志位，对应于正则规则表中的iLmsux匹配模式

而正则替换re.sub的特有参数有repl和count，分别表示被替换的表达式和替换的总次数。

正则切割re.split的特有参数是maxsplit，最大切割次数。

这些将在后面详解。下面首先详解flags标志位：

## 2.8 flags标志位

简写	全称	含义
A	ASCII	让 <code>\w</code> , <code>\W</code> , <code>\b</code> , <code>\B</code> , <code>\d</code> , <code>\D</code> , <code>\s</code> 和 <code>\S</code> 只匹配ASCII，而不是Unicode。
U	UNICODE	与ASCII模式相反，让 <code>\w</code> , <code>\W</code> , <code>\b</code> , <code>\B</code> , <code>\d</code> , <code>\D</code> , <code>\s</code> 和 <code>\S</code> 匹配Unicode 例如 <code>\w</code> 字符集同时会包含英文字符和中文字符
I	IGNORECASE	忽略大小写
L	LOCALE	由当前语言区域决定是ASCII还是UNICODE，以及是否大小写敏感，这个标记只能对8位字节的byte数据有效。 由于语言区域机制很不可靠， <b>这个标记不推荐使用</b> ，本文也不作演示。
M	MULTILINE	开启多行模式，当某字符串中有换行符 <code>\n</code> 时，让 <code>^</code> 和 <code>\$</code> 分别能匹配行的开头和行的结尾，而不是整个字符串的开头和结尾
S	DOTALL	DOT表示 <code>.</code> ，ALL表示所有， <code>.</code> 默认匹配除了换行符以外的任意字符。而这个标志位让 <code>.</code> 匹配包含换行符 <code>\n</code> 的任意字符
X	VERBOSE	开启详细模式，可以在正则表达式中加python语法的#注释
T	TEMPLATE	关闭回溯，只使用模板匹配，提高正则的性能
	DEBUG	显示编译时的debug信息。

在re模块库的源码中：<https://github.com/python/cpython/blob/3.7/Lib/re.py>

所有的flags标志位都定义在**RegexFlag枚举类**中：

```

1 class RegexFlag(enum.IntFlag):
2     ASCII = A = sre_compile.SRE_FLAG_ASCII # assume ascii "locale"
3     IGNORECASE = I = sre_compile.SRE_FLAG_IGNORECASE # ignore case
4     LOCALE = L = sre_compile.SRE_FLAG_LOCALE # assume current 8-bit locale
5     UNICODE = U = sre_compile.SRE_FLAG_UNICODE # assume unicode "locale"
6     MULTILINE = M = sre_compile.SRE_FLAG_MULTILINE # make anchors look for
    newline
7     DOTALL = S = sre_compile.SRE_FLAG_DOTALL # make dot match newline
8     VERBOSE = X = sre_compile.SRE_FLAG_VERBOSE # ignore whitespace and
    comments
9     # sre extensions (experimental, don't rely on these)
10    TEMPLATE = T = sre_compile.SRE_FLAG_TEMPLATE # disable backtracking
11    DEBUG = sre_compile.SRE_FLAG_DEBUG # dump pattern after compilation

```

使用方式：例如要忽略大小写就传入`re.IGNORECASE` 或简写的 `re.I`，要只匹配英文字符就传入`re.ASCII` 或简写的 `re.A`等等。如果需要同时使用多个模式，可以相加，例如我既要只匹配英文字符，还要开启多行模式，可以传入`re.A+re.M`。

我们可以看一个每个标志位对应的二进制位：

```

1 for flag in re.RegexFlag:
2     print(f"{flag.value:0>9b}", flag, flag.value)

```

结果：

```

1 100000000 RegexFlag.ASCII 256
2 000000010 RegexFlag.IGNORECASE 2
3 000000100 RegexFlag.LOCALE 4
4 000100000 RegexFlag.UNICODE 32
5 000001000 RegexFlag.MULTILINE 8
6 000010000 RegexFlag.DOTALL 16
7 001000000 RegexFlag.VERBOSE 64
8 000000001 RegexFlag.TEMPLATE 1
9 010000000 RegexFlag.DEBUG 128

```

可以看到每个模式数值对应的二进制位只占用1位。所以我们在使用多个模式时使用 `|` 运算符性能更佳，例如`re.A|re.M`。

下面我们看下各种标示位的示例：

## 2.8.1 ASCII和UNICODE模式

简写与首字母一致。

对于字符串：

```

1 s = "Midea美的"

```

我们只希望匹配其中的英文，就需要使用ASCII模式：

```

1 re.search('\w+', s, re.ASCII).group(0)

```

结果：

```
1 | 'Midea'
```

注意：`re.ASCII` 可简写为 `re.A`

假如不设置ASCII模式，默认模式是UNICODE模式：

```
1 | re.search('\w+', s).group(0)
```

结果：

```
1 | 'Midea美的'
```

## 2.8.2 IGNORECASE模式

简写与首字母一致。

对于下面的字符串，希望能找出所有的Python字符串，可以采用使用忽略大小写模式：

```
1 | s = "Python1 python2 PYTHON3"  
2 | re.findall('python', s, re.I)
```

由于I模式存在与(?i)中，还可以直接在正则里面写：

```
1 | s = "Python1 python2 PYTHON3"  
2 | re.findall('(?i)python', s)
```

结果均为：

```
1 | ['Python', 'python', 'PYTHON']
```

如果不设置忽略大小写模式：

```
1 | s = "Python1 python2 PYTHON3"  
2 | re.findall('python', s)
```

结果则为：

```
1 | ['python']
```

## 2.8.3 MULTILINE模式

简写与首字母一致。

例如对于下面这段字符串：

```

1 | s = """数据分析
2 | 软件工程
3 | 数据分析师
4 | 开发工程师
5 | 数据分析工程师
6 | 数据开发工程师
7 | """

```

我们希望取出每行以数据分析开头的文本就可以开启多行模式：

```

1 | re.findall('^数据分析.*', s, re.M)

```

也可以直接通过正则本身(iLmsux范围内的字符都支持)开启多行模式：

```

1 | re.findall('(m)^数据分析.*', s)

```

结果均为：

```

1 | ['数据分析', '数据分析师', '数据分析工程师']

```

当然，如果没有设置多行模式，^和\$的作用与 \A 和 \Z 的作用一致，仅匹配整个字符串的开头和字符串的结尾。

```

1 | re.findall('^数据分析.*', s)

```

和

```

1 | re.findall('\A数据分析.*', s, re.M)

```

结果均为：

```

1 | ['数据分析']

```

## 2.8.4 DOTALL模式

简写为S，并不是首字母。

例如我们有一段很长的sql脚本，我们希望能找到其中所有的以select（不区分大小写）开头的查询sql语句：

```

1 | s="""CREATE TABLE GIRL AS
2 | SELECT
3 |     SNO,
4 |     SNAME,
5 |     AGE
6 | FROM
7 |     STUDENTS
8 | WHERE SEX = '女';
9 |
10 | SELECT

```

```

11      CNO,
12      CNAME
13  FROM
14      COURSES
15  WHERE      CREDIT = 3 ;
16
17  -- 例 查询年龄大于 22 岁的学生情况。
18  SELECT
19      *
20  FROM
21      STUDENTS
22  WHERE AGE > 22 ;
23
24  -- 例 找出籍贯为河北的男生的姓名和年龄。
25  SELECT
26      SNAME,
27      AGE
28  FROM
29      STUDENTS
30  WHERE      BPLACE = ' 河北 '
31      AND SEX = ' 男 ' ;""

```

如果我们不开启DOTALL模式会相对麻烦一点，需要使用 `[\w\W]` 来表示包含换行符的所有字符：

```
1 | re.findall('^select [\w\W]+?;', s, re.I|re.M)
```

或：

```
1 | re.findall('(?:im)^select [\w\W]+?;', s)
```

结果：

```

1 | ['SELECT CNO,CNAME FROM COURSES WHERE CREDIT=3;',
2 | 'SELECT \n * \nFROM\n STUDENTS \nWHERE AGE > 22;',
3 | "SELECT \n SNAME,\n AGE \nFROM\n STUDENTS \nWHERE BPLACE = '河北' \n
   AND SEX = '男';"]

```

但开启DOTALL模式，就可以直接使用 `.` 来匹配包含换行符的所有字符，即：

```
1 | re.findall('^select .+?;', s, re.I|re.M|re.S)
```

或

```
1 | re.findall('(?:ims)^select .+?;', s)
```

## 2.8.5 VERBOSE模式

简写为X，并不是首字母。

该模式的作用就是可以在正则中写#号的注释，对于很复杂的正则，或许我们使用注释会更加清晰易懂。



例如我们有段字符串：

```
1 s = ""中楼层(共9层)|2007年建|1室1厅|24.78平米|北
2 地下室|2014年建|1室0厅|39.52平米|东
3 底层(共2层)5室3厅|326.56平米|东南西北""
```

我们需要提取出每行数据的 层、楼层数、建筑年份、户型、大小和方向。

如果直接写，或许这个正则阅读起来比较费劲：

```
1 re.findall("^(^|(|+))?(?:\\(共\\d+\\)层\\)?(?:\\|\\d{4}\\)年建\\|)?(\\d室\\d厅)\\|
  ([\\d.]+)平米\\|([东南西北]+)", s, re.M)
```

于是我们可以开启VERBOSE模式加个注释：

```
1 pattern = ""^(^|(|+)? # 层
2             (?:\\(共\\d+\\)层\\)? # 楼层数
3             (?:\\|\\d{4}\\)年建\\|)? # 建筑年份
4             (\\d室\\d厅) # 户型
5             \\|
6             ([\\d.]+)平米 # 大小
7             \\|
8             ([东南西北]+) # 方向
9             ""
10 re.findall(pattern, s, re.M|re.X)
```

也可以直接使用正则字符串本身来开启VERBOSE(简写X)模式：

```
1 pattern = ""(?:mx)^(^|(|+)? # 层
2             (?:\\(共\\d+\\)层\\)? # 楼层数
3             (?:\\|\\d{4}\\)年建\\|)? # 建筑年份
4             (\\d室\\d厅) # 户型
5             \\|
6             ([\\d.]+)平米 # 大小
7             \\|
8             ([东南西北]+) # 方向
9             ""
10 re.findall(pattern, s)
```

结果均为：

```
1 [('中楼层', '9', '2007', '1室1厅', '24.78', '北'),
2  ('地下室', '', '2014', '1室0厅', '39.52', '东'),
3  ('底层', '2', '', '5室3厅', '326.56', '东南西北')]
```

再来一个简单的示例：

```

1 pattern = r'''(?mx)
2 ^          # 开头
3 (\d{4})    # 年
4 [ ]        # 空格
5 (\d{2})    # 月
6 $          # 结尾
7 '''
8 re.findall(pattern, '2020 06\n2020 07')

```

结果：

```

1 [ ('2020', '06'), ('2020', '07') ]

```

当然，正则本身也支持通过 `(?#ABC)` 增加注释 `ABC`，例如：

```

1 "(\w+)(?#word) \1(?#word repeat again)"

```

## 2.8.6 TEMPLATE模式

简写与首字母一致。

该模式的作用是关闭回溯，在前面的贪婪模式、非贪婪模式和独占模式一节中，已经讲解过回溯的过程，贪婪模式和非贪婪模式都需要发生回溯才能完成相应的功能。

TEMPLATE表示模板的意思，开启该模式意味着只能使用正则的模板匹配，而不能使用回溯算法，意味着不能再使用任意数量词，包括`*`、`?`、`+`、`{m,n}`等，哪怕固定数量的`{4}`也不允许。

例如我们想要获取一个时间字符串的年月日：

```

1 s = "1980-02-12"
2 re_match = re.match(r'(\d{4})-(\d{2})-(\d{2})', s)
3 re_match.group(1), re_match.group(2), re_match.group(3)

```

结果：

```

1 ('1980', '02', '12')

```

很明显这个时间字符串的每个部分，长度都是确定而且固定的，那我们就完全可以开启TEMPLATE模式，关闭回溯算法。

但直接关闭会报错：

```

1 s = "1980-02-12"
2 re_match = re.match(r'(?t)(\d{4})-(\d{2})-(\d{2})', s)
3 re_match.group(1), re_match.group(2), re_match.group(3)

```

```

1 error: internal: unsupported template operator MAX_REPEAT

```

```

D:\Anaconda3\lib\sre_compile.py in _code(p, flags)
    605
    606     # compile the pattern
--> 607     _compile(code, p.data, flags)
    608
    609     code.append(SUCCESS)

D:\Anaconda3\lib\sre_compile.py in _compile(code, pattern, flags)
    166         emit((group-1)*2)
    167         # _compile_info(code, p, _combine_flags(flags, add_flags, del_flags))
--> 168         _compile(code, p, _combine_flags(flags, add_flags, del_flags))
    169         if group:
    170             emit(MARK)

D:\Anaconda3\lib\sre_compile.py in _compile(code, pattern, flags)
    137     elif op in REPEATING_CODES:
    138         if flags & SRE_FLAG_TEMPLATE:
--> 139             raise error("internal: unsupported template operator %r" % (op,))
    140         if _simple(av[2]):
    141             if op is MAX_REPEAT:

error: internal: unsupported template operator MAX_REPEAT

```

必须这样写：

```

1 | s = "1980-02-12"
2 | re_match = re.match('(\d\d\d\d)-(\d\d)-(\d\d)', s, re.T)
3 | re_match.group(1), re_match.group(2), re_match.group(3)

```

或

```

1 | s = "1980-02-12"
2 | re_match = re.match('(?t)(\d\d\d\d)-(\d\d)-(\d\d)', s)
3 | re_match.group(1), re_match.group(2), re_match.group(3)

```

这个模式目前连官网文档没有任何说明，开启这个模式能否相对直接执行不会发生回溯的正则是否能提升效率还未知。

## 2.8.7 DEBUG模式

这个模式没有简写，对于一般的用户用不上。开启这个模式之后，编译正则表达式时会打印出正则的解释树信息，这些信息并不是任何一种编程语言，而是正则表达式特有的解析树，就类似于hive sql语句编译的解析树一样。通过解析树可以让高级开发人员更清楚了解这个正则执行的性能和效率。

我们随便查看一个正则的编译信息：

```

1 | re.compile('(\d{2}) ?-12', re.DEBUG)

```

结果：

```

1 | SUBPATTERN 1 0 0
2 |   MAX_REPEAT 2 2
3 |     IN
4 |       CATEGORY CATEGORY_DIGIT
5 | MAX_REPEAT 0 1
6 |   LITERAL 32
7 | LITERAL 45
8 | LITERAL 49

```

```

9  LITERAL 50
10
11  0. INFO 4 0b0 5 6 (to 5)
12  5: MARK 0
13  7. REPEAT_ONE 9 2 2 (to 17)
14  11. IN 4 (to 16)
15  13. CATEGORY UNI_DIGIT
16  15. FAILURE
17  16: SUCCESS
18  17: MARK 1
19  19. REPEAT_ONE 6 0 1 (to 26)
20  23. LITERAL 0x20 (' ')
21  25. SUCCESS
22  26: LITERAL 0x2d ('-')
23  28. LITERAL 0x31 ('1')
24  30. LITERAL 0x32 ('2')
25  32. SUCCESS

```

## 2.9 正则匹配

正则匹配即查找并返回一个匹配项。

### 2.9.1 基本函数

re模块完成正则匹配功能的函数有3个：

1. **search**：从字符串任意位置开始匹配，返回第一个匹配成功的对象，匹配失败函数返回None
2. **match**：从字符串开头开始匹配，匹配失败函数返回None
3. **fullmatch**：整个字符串与正则完全匹配

它们的参数均为：

```
1 re.xxx(pattern, string, flags=0)
```

**search**方法从字符串任意位置开始查找，适配性最强，可以通过加入 `^` 匹配开头达到跟**match**相同的效果，**match**也可以通过加入 `$` 匹配结尾达到跟**fullmatch**相同的效果。

首先测试一下**search**：

```

1 print(re.search('www', 'www.taobao.com'))
2 print(re.search('com', 'www.taobao.com'))

```

```

1 <re.Match object; span=(0, 3), match='www'>
2 <re.Match object; span=(11, 14), match='com'>

```

测试**match**：

```

1 print(re.match('www', 'www.taobao.com'))
2 print(re.match('com', 'www.taobao.com'))

```

```

1 <re.Match object; span=(0, 3), match='www'>
2 None

```

最后测试**fullmatch**：

```
1 print(re.fullmatch('www', 'www.taobao.com'))
2 print(re.fullmatch('com', 'www.taobao.com'))
3 print(re.fullmatch('www.taobao.com', 'www.taobao.com'))
```

```
1 None
2 None
3 <re.Match object; span=(0, 14), match='www.taobao.com'>
```

从上述结果中，我们可以清晰的看到**search**、**match**和**fullmatch**三者的区别：

由于'com'在字符串'[www.taobao.com](http://www.taobao.com)'的末尾，所以match函数未匹配到任何结果返回None；而fullmatch函数由于是匹配整个字符串，所以'www'匹配'[www.taobao.com](http://www.taobao.com)'时也返回None。

## 2.9.2 re.MatchObject 对象

同时可以看到，它们均返回了一个 **re.Match** 对象，该对象提供了group(num) 和 groups()方法，group(num) 用于返回对应分组编号的数据，groups()方法用于返回所有分组的数据，而lastindex属性可以获取分组的个数。

示例：

```
1 line = "Cats are smarter than dogs"
2 matchObj = re.match(r'(.*) are (.*?) ', line, re.I)
3 if matchObj:
4     print("总分组数: ", matchObj.lastindex)
5     print("所有分组的数据: ", matchObj.groups())
6     print("整个被匹配的字符串 : ", matchObj.group())
7     print("第1个分组的数据 : ", matchObj.group(1))
8     print("第2个分组的数据 : ", matchObj.group(2))
9 else:
10    print("No match!!")
```

结果：

```
1 总分组数: 2
2 所有分组的数据: ('Cats', 'smarter')
3 整个被匹配的字符串 : Cats are smarter
4 第1个分组的数据 : Cats
5 第2个分组的数据 : smarter
```

**re.MatchObject** 对象的其他方法：

- start() 返回匹配开始的位置
- end() 返回匹配结束的位置
- span() 返回一个元组包含匹配 (开始,结束) 的位置

## 2.9.3 匹配手机号码

目前主要的手机号前三位是：

中国信号段：133，153，180，181，189，173，177，149

中国联通号段：130，131，132，155，156，185，186，145，176，185

中国移动号段：134，135，136，137，138，139，150，151，152，157，158，159，182，183，184，147，178

规律是：

第一位：1

第二位：3，4，5，7，8

第三位：根据第二位来确定

3 + 【0-9】

4 + 【5，7，9】

5 + 【0-9】！4

7 + 【0-9】！4和9

8 + 【0-9】

对手机号比较粗略的匹配（11位数字，前2位符合手机号规则）：

```
1 | "1[34578]\d{9}"
```

较为精确的匹配（11位数字，前3位符合手机号规则）：

```
1 | "1(?:[38]\d|4[579]|5[0-35-9]|7[0-35-8])\d{8}"
```

测试较为精确匹配：

```
1 | import re
2 | import random
3 |
4 |
5 | def random_number(nums):
6 |     result = ""
7 |     for x in range(nums):
8 |         result += str(random.randint(0, 9))
9 |     return result
10 |
11 |
12 | nums = [
13 |     133, 153, 180, 181, 189, 173, 177, 149,
14 |     130, 131, 132, 155, 156, 185, 186, 145, 176, 185,
15 |     134, 135, 136, 137, 138, 139, 150, 151, 152, 157, 158, 159, 182, 183,
16 |     184, 147, 178
17 | ]
18 | for num in nums:
19 |     print(re.fullmatch("1([38]\d|4[579]|5[0-35-9]|7[0-35-8])\d{8}",
20 |                        f"{num}{random_number(8)}").group(), end=",")
```

结果：

```
1 13320640138,15352178619,18010467102,18124689139,18975065050,17380280568,17798
2 275371,14994833499,13068873816,13151192893,13289047370,15594125464,1564821694
3 0,18574982445,18643788553,14516397708,17616874062,18559031583,13443533383,135
4 96265766,13629806068,13745249866,13896644123,13954817486,15076523907,15182868
5 824,15229880699,15794102747,15852468936,15938064514,18297190705,18304331736,1
6 8402303981,14751356440,17847872471,
```

全部成功匹配上。

测试一个错误的电话号码：

```
1 pattern = "1([38]\d|4[579]|5[0-35-9]|7[0-35-8])\d{8}"
2 tel_number = "15452468936"
3 print(re.fullmatch(pattern, tel_number))
```

结果：

```
1 None
```

也成功的匹配失败。

## 2.9.4 匹配邮箱地址

邮箱地址的规则是：user@mail.server.name，即名称+@+网站

常见的邮箱地址一般都是@xxx.com，但也还包括一些特殊邮箱地址，能到三级域名甚至四级域名，例如：

```
@SEED.NET.TW @TOPMARKEPLG.COM.TW @wilnetonline.net @cal3.vsnl.net.in
```

当然这只是少部分，大部分都是二级域名，但我们不能因此让这些域名匹配不成功。

例如，我们认为下面的邮箱地址都是合法的邮箱地址：

```
1 emails = [
2     'someone@gmail.com',
3     'bill.gates@microsoft.com',
4     'mr-bob@example.com',
5     'someone@SEED.NET.TW',
6     'chuck.gt@cal3.vsnl.net.in'
7 ]
```

正则匹配规则可以写为：

```
1 r"[a-z.-]+@[a-zA-Z0-9]+(\.[a-zA-Z]+){1,3}"
```

测试：

```
1 for email in emails:
2     match_obj = re.match(r"[a-z.-]+@[a-zA-Z0-9]+(\.[a-zA-Z]+){1,3}", email,
3     re.I)
4     if match_obj:
5         print(match_obj.group(0))
```

结果：

```
1 someone@gmail.com
2 bill.gates@microsoft.com
3 mr-bob@example.com
4 someone@SEED.NET.TW
5 chuck.gt@cal3.vsn1.net.in
```

再顺便测试一个不是邮箱的字符串：

```
1 print(re.match(r"[a-z.-]+@[a-zA-Z0-9]+(\.[a-zA-Z]+){1,3}",
  'bob#example.com', re.I))
```

未通过校验，打印结果为None。

## 2.9.5 匹配时引用分组

前面的正则匹配规则表中说过：

`\<number>` 表示引用编号为 `\<number>` 的分组匹配到的字符串

### 示例

IT后台有一批用户名和密码的字符串，部门希望找出那些将密码设置的跟用户名一样的用户提醒他们修改密码：

```
1 users = [
2     "user1:password",
3     "user2:user2",
4     "user3:password",
5     "user4:password",
6     "user5:password",
7     "user6:user6",
8     "user7:password",
9     "user8:user8",
10    "user9:password",
11    "user10:user10"
12 ]
```

这时，在匹配时引用分组就会非常方便：

```
1 for user in users:
2     match_obj = re.match(r"(\w+):\1", user, re.A)
3     if match_obj:
4         print(match_obj.group(1))
```

结果：

```
1 user2
2 user6
3 user8
4 user10
```

可以看到顺利的提取出了，用户名和密码一致的用户。



## 2.10 正则查找

实现正则查找的函数有：

- `re.findall`：在字符串中找到正则表达式所匹配的所有子串，并返回一个列表，如果没有找到匹配的，则返回空列表。
- `re.finditer`：在字符串中找到正则表达式所匹配的所有子串，并把它们作为一个迭代器返回。

看完了正则匹配，相信正则查找对于你来说已经很简单。下面直接举几个例子。

### 示例1

我们需要找出这段文本中所有的数字：

```
1 s = ' taobao 123 google 456'
2 re.findall("\d+", s)
```

结果：

```
1 ['123', '456']
```

使用`finditer`返回迭代器：

```
1 it = re.finditer("\d+", s)
2 for match_obj in it:
3     print(match_obj.group(), end=" ")
```

结果：

```
1 123 456
```

### 示例2

例如我们希望查找出下面这段英文中所有4个字母的单词：

```
1 s = "Clothes are so significant in our daily life that we can't live without
   them"
2 re.findall(r"\b[a-z]{4}\b", s, re.I)
```

结果：

```
1 ['life', 'that', 'live', 'them']
```

可以看到很顺利的找到了想要的结果。

`\b` 表示单词边界，可以回正则规则匹配表查看

也可以使用`re.finditer`方法返回一个迭代器：

```
1 for match_obj in re.finditer(r"\b[a-z]{4}\b", s, re.I):
2     print(match_obj.group(), end=" ")
```

结果：

```
1 life
2 that
3 live
4 them
```

注意：re.finditer方法返回的迭代器迭代取出的每一个对象都是 `re.Match` 对象

### 示例3

提取出下面文本中所有的单词（被双引号引起来的要作为一个单词，例如the little cat，最终结果无需去重）：

we found "the little cat" is in the hat, we like "the little cat"

```
1 s = 'we found "the little cat" is in the hat, we like "the little cat"'
2 print(re.findall('\w+|".*?"', s))
```

结果：

```
1 ['we', 'found', '"the little cat"', 'is', 'in', 'the', 'hat', 'we', 'like',
  '"the little cat"']
```

### 示例4

提取出下面网页中head 标签的内容：

```
1 <html>
2   <head>
3     <title>学习正则表达式</title>
4   </head>
5   <body></body>
6 </html>
```

参考解法：

```
1 s = """<html>
2   <head>
3     <title>学习正则表达式</title>
4   </head>
5   <body></body>
6 </html>"""
7
8 re.findall("(?si)<head>(.*?)</head>", s)
```

结果：

```
1 ['\n\t\t<title>学习正则表达式</title>\n\t']
```

## 2.11 正则替换

实现正则替换的函数有`re.sub`和`re.subn`，它们参数均是：

```
1 | re.sub*(pattern, repl, string, count=0, flags=0)
```

参数：

- `pattern`：正则中的模式字符串。
- `repl`：替换的表达式，也可为一个函数。
- `string`：要被替换的原始字符串。
- `count`：模式匹配后替换的最大次数，默认 0 表示替换所有的匹配。

`re.subn`相对`re.sub`的区别是会在`re.sub`返回结果的基础上额外返回替换次数。

### 2.11.1 基本替换

有一个电话号码带注释的字符串：

```
1 | phone = "2004-959-559 # 这是一个电话号码"
```

如果我们需要删除注释：

```
1 | num = re.sub(r'#.*$', "", phone)
2 | print("电话号码：", num)
```

结果：

```
1 | 电话号码：2004-959-559
```

删除所有非数字的内容：

```
1 | num = re.sub(r'\D', "", phone)
2 | print("电话号码：", num)
```

结果：

```
1 | 电话号码：2004959559
```

### 2.11.2 环视替换

给金额添加万分符。

有一个金额字符串：

```

1 s = ""5305256725元
2 4220元
3 870元
4 7866369414527元
5 144995元
6 2069993310元
7 354070715448元
8 711元
9 2113046206元""

```

需要给万亿、亿、万位置添加逗号，例如 7866369414527元 被转换为 7,8663,6941,4527元。

这些位置均为距离元4n个字符的位置，所以可以这样写：

```

1 print(re.sub(r"(?<=\d)(?=(?:\d{4})+元)", ",", s, flags=re.M))

```

结果：

```

1 53,0525,6725元
2 4220元
3 870元
4 7,8663,6941,4527元
5 14,4995元
6 20,6999,3310元
7 3540,7071,5448元
8 711元
9 21,1304,6206元

```

(?<=\d) 表示左边必须是一个数字，(?=(?:\d{4})+元) 表示右边必须是紧挨元，而且是4n个数字。

这样在对应的位置替换成,，便添加了万分符。

### 2.11.3 re.sub替换表达式引用分组

\<number> 表示引用编号为 \<number> 的分组匹配到的字符串，这个规则不仅可以在匹配表达式中使用，还可以在替换表达式中使用。

**需求1：**将重叠的字符替换成单个字符(zzzz->z)

例如，将

```
"我我...我我...我要..要要...要要...学学学....学学...编编编...编程..程.程程...程...程"
```

转成：

```
"我要学编程"
```

思路：

1. 先将字符. 去掉。
2. 再将多个重复的内容变成单个内容。

```

1 s = "我我...我我...我要..要要...要要...学学学....学学...编编编...编程..程.程程...程...程"
2 # 去掉所有的字符.
3 s = s.replace(".", "")
4 # 连续重复字符转单个字符
5 s = re.sub(r"(\.){1+}", r"\1", s)
6 s

```

结果：

```

1 '我要学编程'

```

那么如果使用subn方法有什么特别之处呢？

```

1 s = "我我...我我...我要..要要...要要...学学学....学学...编编编...编程..程.程程...程...程"
2 # 去掉所有的字符.
3 s = s.replace(".", "")
4 print("去掉字符.之后: ", s)
5 # 连续重复字符转单个字符
6 s, count = re.subn(r"(\.){1+}", r"\1", s)
7 print("结果: ", s)
8 print("替换次数: ", count)

```

结果：

```

1 去掉字符.之后:  我我我我我要要要要要学学学学学编编编编程程程程程
2 结果:  我要学编程
3 替换次数:  5

```

可以看到subn方法可以通过元组匹配的方式，额外得到替换的次数，但目前我还没有遇到哪个场景需要得到这个替换次数，但或许哪天你真需要知道替换多少次的时候，使用subn能省不少事。

## 需求2：连续单词去重

有一篇英文文章，里面有一些单词连续出现了多次，我们认为连续出现多次的单词应该是一次，比如：

```
the little cat cat is in the hat hat hat2, we like it.
```

其中 cat 和 hat 连接出现多次，要求处理后结果是：

```
the little cat is in the hat hat2, we like it.
```

```

1 s = "the little cat cat is in the hat hat hat2, we like it."
2 re.sub(r"(\b\w+)(?:\s+\1\b)+", r"\1", s)

```

结果：

```

1 'the little cat is in the hat hat2, we like it.'

```

**需求2：**将ip地址进行地址段顺序的排序。

有一个ip字符串：

```
1 | s = "192.68.1.254 102.49.23.013 10.10.10.10 2.2.2.2 8.109.90.30"
```

现在需要让它内部的每个ip排序输出。

思路：

1. 按照每一段需要的最多的0进行补齐，那么每一段就会至少保证有3位。
2. 将每一段只保留3位。这样，所有的ip地址都是每一段3位。
3. 切割排序，并拼接排好序的ip地址字符串
4. 去掉结果字符串每个数字开头的0

```
1 | s = "192.68.1.254 102.49.23.013 10.10.10.10 2.2.2.2 8.109.90.30"
2 | # 1. 按照每一段需要的最多的0进行补齐，那么每一段就会至少保证有3位。
3 | s = re.sub(r"(\d+)", r"00\1", s)
4 | print("每段开头补2个0后：", s)
5 | # 2. 将每一段只保留3位。这样，所有的ip地址都是每一段3位。
6 | s = re.sub(r"0*(\d{3})", r"\1", s)
7 | print("每段仅保留3个数字：", s)
8 | # 3. 切割排序，并拼接排好序的ip地址字符串
9 | s = " ".join(sorted(s.split()))
10 | print("切割排序并拼接后：", s)
11 | # 4. 去掉结果字符串每个数字开头的0
12 | s = re.sub(r"0*(\d+)", r"\1", s)
13 | print("最终结果：", s)
```

结果：

```
1 | 每段开头补2个0后： 00192.0068.001.00254 00102.0049.0023.00013
   0010.0010.0010.0010 002.002.002.002 008.00109.0090.0030
2 | 每段仅保留3个数字： 192.068.001.254 102.049.023.013 010.010.010.010
   002.002.002.002 008.109.090.030
3 | 切割排序并拼接后： 002.002.002.002 008.109.090.030 010.010.010.010
   102.049.023.013 192.068.001.254
4 | 最终结果： 2.2.2.2 8.109.90.30 10.10.10.10 102.49.23.13 192.68.1.254
```

可以看到最终实现了，ip地址的排序。

## 2.11.4 repl替换表达式使用函数

repl 替换表达式，也可传入一个函数，这个函数的参数必须是一个 `re.MatchObject` 对象（参看前面的正则匹配部分）。

在其他编程语言中，正则表达式的替换功能，往往都是不支持传入函数的，导致要实现一些数值计算性的功能代码会变得比较复杂，而python的正则替换由于支持函数，所以可以很简单的实现一些较为复杂的逻辑。

先从简单的例子开始：

### 示例1：数值翻倍

假如我们有一个字符串：

```
1 | s = 'A23G4HFD567'
```

希望将这个字符串所有连续的数字都翻倍，使用正则替换传入函数会非常方便：

```
1 s = 'A23G4HFD567'
2 print(re.sub('\d+', lambda m: str(int(m.group(0))*2), s))
```

假如python的正则替换不支持传入函数就会相对比较复杂：

```
1 buff = list(s)
2 for m in re.finditer("\d+", s):
3     pos = m.span()
4     buff[pos[0]:pos[1]] = list(str(int(m.group(0))*2))
5 s = "".join(buff)
6 s
```

结果均为：

```
1 'A46G8HFD1134'
```

### 示例2：数值隔断

有一个字符串：

```
1 s='AB837D5D4F7G8H7F8H56D4D7G4D3'
```

想将这个字符串所有>=6的单个数字替换成9，<6的单个数字替换为0：

```
1 s = 'AB837D5D4F7G8H7F8H56D4D7G4D3'
2 print(re.sub('\d', lambda m: '9' if int(m.group()) >= 6 else '0', s))
```

结果：

```
1 AB909D0D0F9G9H9F9H09D0D9G0D0
```

### 示例3：顺序编号

有一段字符串 "a,b,c,d,e,f"，我们希望将每一个出现的字母增加一个递增的编号，比如：

```
1 s = "a,b,c,d,e,f"
```

希望得到结果：'1.a,2.b,3.c,4.d,5.e,6.f'

使用编号迭代器+正则替换会变得非常简单：

```
1 s = "a,b,c,d,e,f"
2 numbers = iter(range(1, 10000))
3 re.sub("\w", lambda m: f"{next(numbers)}.{m.group()}", s)
```

结果：

```
1 '1.a,2.b,3.c,4.d,5.e,6.f'
```

我们使用的编号不可能超过1万，所以range函数的最大值传入10000即可，iter获取了range对象的迭代器，从而得到了一个可以不断获取下一个编号的编号迭代器。

顺序编号的比较典型应用场景是**模板数据回传**，举个比较傻的例子：

比如有一个含有很多sql查询语句的大文本，我们需要将其中所有的sql语句提取出来，执行完毕，再将查询结果写回到sql语句原本所在的位置。

要实现这个功能，首先可以先将所有的sql语句都替换成顺序编号的占位符，我们就以下面这个比较简单的文本为例吧：

```
1  s=""
2  -- 一个查询
3  SELECT
4      CNO,
5      CNAME
6  FROM
7      COURSES
8  WHERE CREDIT = 3;
9
10 -- 例 查询年龄大于 22 岁的学生情况。
11 SELECT
12     *
13 FROM
14     STUDENTS
15 WHERE AGE > 22 ;
16
17 -- 例 找出籍贯为河北的男生的姓名和年龄。
18 SELECT
19     SNAME,
20     AGE
21 FROM
22     STUDENTS
23 WHERE     BPLACE = ' 河北 '
24     AND SEX = ' 男 ' ;""
```

只需执行：

```
1  numbers = iter(range(10000))
2  template_text = re.sub("^select .+?;", lambda m: "{%d}" % next(numbers), s,
3  flags=re.I | re.M | re.S)
4  print(template_text)
```

结果：

```
1  -- 一个查询
2  {0}
3
4  -- 例 查询年龄大于 22 岁的学生情况。
5  {1}
6
7  -- 例 找出籍贯为河北的男生的姓名和年龄。
8  {2}
```



## 2.12 正则切割

实现正则切割的函数是`re.split`，它的参数是：

```
1 | re.split(pattern, string, maxsplit=0, flags=0)
```

这个函数相对前面的函数多了`maxsplit`，表示最大切割次数。

该函数最常见的应用场景就是字段分割。

例如，我们需要切割出下面这个字符串的每个连续的字符串（含有不确定数量的空格和tab）：

```
1 | s="FRN2004001      100      VCP772Z              417  BX              417  BX
    181128307          CN10      1220      2000              5,004  EA              2020.03.30
    45408263      L12 L12"
```

可以使用正则进行切割：

```
1 | fields = re.split("\s+", s)
2 | print(fields)
```

结果：

```
1 | ['FRN2004001', '100', 'VCP772Z', '417', 'BX', '417', 'BX', '181128307',
    'CN10', '1220', '2000', '5,004', 'EA', '2020.03.30', '45408263', 'L12',
    'L12']
```

如果我们希望最后两个L12不被切割，可以设置`maxsplit`：

```
1 | print(re.split("\s+", s, maxsplit=15))
```

结果：

```
1 | ['FRN2004001', '100', 'VCP772Z', '417', 'BX', '417', 'BX', '181128307',
    'CN10', '1220', '2000', '5,004', 'EA', '2020.03.30', '45408263', 'L12\tL12']
```

这样最后一个空白字符\t就没有被切割。

当然对于这个例子，直接使用字符串自带的切割就可以实现，不传参数模式就是用连续的空白字符切割：

```
1 | s.split()
```

和

```
1 | s.split(maxsplit=15)
```

结果与上面一致。

下面举一个必须用正则切割才能解决的问题：

环视切割，有一个字符串：

```
1 | s = "北京西北京站北京北北京南站北京东"
```

我们需要取出其中所有的北京xx，即北京西、北京站、北京北、北京南站 和 北京东。

使用正则切割会非常简单：

```
1 | s = "北京西北京站北京北北京南站北京东"
2 | re.split("(?<!(^)(?=北京)", s)
```

结果：

```
1 | ['北京西', '北京站', '北京北', '北京南站', '北京东']
```

在前面的正则匹配规则表中的非捕获组与环视已经说明：

- `(?=...)` 肯定环视，表示右边是指定内容的位置
- `(?<!(...))` 否定逆序环视，表示左边不是指定内容的位置

`(?<!(^))` 表示切割位置的左边不能是行的开头，`(?=北京)` 表示切割位置的右边必须是北京

## 2.13 compile编译正则表达式

`re.compile()` 返回 `re.Pattern` 正则表达式编译对象（跟java语言的Pattern类原理一样）。

语法格式为：

```
1 | re.compile(pattern, flags)
```

前面的每个正则方法：`re.fullmatch`、`re.findall`、`re.sub`、`re.split`等方法，执行过程中都会先编译正则表达式（开启DEBUG模式可以看到解析树），如果有些正则表达式会反复被使用，重复的编译会造成较大的资源浪费。于是我们可以通过`re.compile`方法提前将正则表达式编译好，以后反复使用不会重复编译。

编译一个正则测试一下：

```
1 | pattern = re.compile('(\\d+)')
2 | print(pattern, type(pattern))
```

结果：

```
1 | re.compile('(\\d+)') <class 're.Pattern'>
```

可以看到：

```
1 | print([m for m in dir(pattern) if not m.startswith("__")])
```

```
1 | ['findall', 'finditer', 'flags', 'fullmatch', 'groupindex', 'groups',
    'match', 'pattern', 'scanner', 'search', 'split', 'sub', 'subn']
```

具备上述所有方法。

#### 正则匹配：

```
1 | pattern.search(' taobao 123 google 456')
```

结果：

```
1 | <re.Match object; span=(9, 12), match='123'>
```

#### 正则查找：

```
1 | pattern.findall(' taobao 123 google 456')
```

结果：

```
1 | ['123', '456']
```

#### 正则替换：

```
1 | pattern.sub("", "2004-959-559 # 这是一个电话号码")
```

结果：

```
1 | '-- # 这是一个电话号码'
```

#### 正则切割：

```
1 | pattern.split('AB837D5D4F7G8H7F8H56D4D7G4D3')
```

结果：

```
1 | ['AB', 'D', 'D', 'F', 'G', 'H', 'F', 'H', 'D', 'D', 'G', 'D', '']
```

## 2.14 其他

---

re. **escape**(*pattern*)

转义 *pattern* 中的特殊字符。如果你想对任意可能包含正则表达式元字符的文本字符串进行匹配，它就是有用的。比如

```
>>> print(re.escape('http://www.python.org'))
http://www\.python\.org

>>> legal_chars = string.ascii_lowercase + string.digits + '!#$%&*+,-.^_`|~:/'
>>> print('%s+' % re.escape(legal_chars))
[abcdefghijklmnopqrstuvwxyz0123456789!#$%&*\+,-\.\^_`|~:/>

```

这个函数不能被用于 `sub()` 和 `subn()` 的替换字符串，只有反斜杠应该被转义。例如：

```
>>> digits_re = r'\d+'
>>> sample = '/usr/sbin/sendmail - 0 errors, 12 warnings'
>>> print(re.sub(digits_re, digits_re.replace('\\', r'\\'), sample))
/usr/sbin/sendmail - \d+ errors, \d+ warnings
```

在 3.3 版更改：' ' 不再被转义。

在 3.7 版更改: 只有在正则表达式中具有特殊含义的字符才会被转义。因此, `! !`, `! !`, `! %`, `! "`, `! ,`, `! /`, `! :`, `! ;`, `! <`, `! =`, `! >`, `! @` 和 `! ~` 将不再会被转义。

```
re. purge()
```

清除正则表达式缓存。

来源于python官方文档：<https://docs.python.org/zh-cn/3.7/library/re.html>

### 3 补充资料

本节参考极客时间的课程：《正则表达式入门课》

### 3.1 正则表达式的历史与流派

### 3.1.1 正则表达式简史

正则表达式的起源，可以追溯到，早期神经系统如何工作的研究。在 20 世纪 40 年代，有两位神经生理学家（Warren McCulloch 和 Walter Pitts），研究出了一种用数学方式来描述神经网络的方法。

1956 年，一位数学家（Stephen Kleene）发表了一篇标题为《神经网络事件表示法和有穷自动机》的论文。这篇论文描述了一种叫做“正则集合（Regular Sets）”的符号。

随后，大名鼎鼎的 Unix 之父 Ken Thompson 于 1968 年发表了文章《正则表达式搜索算法》，并且将正则引入了自己开发的编辑器 qed，以及之后的编辑器 ed 中，然后又移植到了大名鼎鼎的文本搜索工具 grep 中。自此，正则表达式被广泛应用到 Unix 系统或类 Unix 系统 (如 macOS、Linux) 的各种工具中。

随后，由于正则功能强大，非常实用，越来越多的语言和工具都开始支持正则。不过遗憾的是，由于没有尽早确立标准，导致各种语言和工具中的正则虽然功能大致类似，但仍然有不少细微差别。

于是，诞生于 1986 年的 POSIX 开始进行标准化的尝试。POSIX 作为一系列规范，定义了 Unix 操作系统应当支持的功能，其中也包括正则表达式的规范。因此，Unix 系统或类 Unix 系统上的大部分工具，如 grep、sed、awk 等，均遵循该标准。我们把这些遵循 POSIX 正则表达式规范的正则表达式，称为 **POSIX 流派**的正则表达式。

在 1987 年 12 月，Larry Wall 发布了 Perl 语言第一版，因其功能强大一票走红，所引入的正则表达式功能大放异彩。之后 Perl 语言中的正则表达式不断改进，影响越来越大。于是在此基础上，1997 年又诞生了 PCRE——**Perl 兼容正则表达式**（Perl Compatible Regular Expressions）。

PCRE 是一个兼容 Perl 语言正则表达式的解析引擎，是由 Philip Hazel 开发的，为很多现代语言和工具所普遍使用。除了 Unix 上的工具遵循 POSIX 标准，PCRE 现已成为其他大部分语言和工具隐然遵循的标准。

之后，正则表达式在各种计算机语言或各种应用领域得到了更为广泛的应用和发展。**POSIX 流派** 与 **PCRE 流派** 是目前正则表达式流派中的两大最主要的流派。

### 3.1.2 正则表达式流派

目前正则表达式主要有两大流派（Flavor）：POSIX 流派与 PCRE 流派。

#### 3.1.2.1 POSIX 流派

POSIX 规范定义了正则表达式的两种标准：

**BRE 标准**（Basic Regular Expression 基本正则表达式）；

**ERE 标准**（Extended Regular Expression 扩展正则表达式）。

这两种标准有什么的异同点呢？

早期 BRE 与 ERE 标准的区别主要在于，BRE 标准不支持数量词问号和加号，也不支持多选分支结构管道符。BRE 标准在使用花括号，圆括号时要转义才能表示特殊含义。ERE 标准则在BRE 标准基础上有了一定改进，在使用花括号，圆括号时不需要转义了，还支持了问号、加号 和 多选分支。

Linux 发行版大多都集成了 GNU 套件。GNU 在实现 POSIX 标准时，做了一定的扩展：

- 1. GNU BRE 支持了 +、?，但转义了才表示特殊含义，即需要用 \+、\? 表示。
- 2. GNU BRE 支持管道符多选分支结构，同样需要转义，即用 \| 表示。
- 3. GNU ERE 也支持使用反引用，和 BRE 一样，使用 \1、\2...\9 表示。

BRE 标准和 ERE 标准的详细区别（浅黄色背景是 BRE 和 ERE 不同的地方，三处天蓝色字体是 GNU 扩展）：

正则表达式特性	BRE标准	ERE标准
点号.、^、\$、[...]、[^...]	✓	✓
"任意数目"量词*	✓	✓
+和?量词	✗（GNU BRE支持）	✓
区间量词	\{min,max\}	{min,max}
圆括号分组	\(...\)	(...)
量词可否限定圆括号分组	✓	✓
捕获文本引用	\1到\9	✗（GNU ERE支持）
多选分支结构	✗（GNU BRE支持）	✓

总之，GNU BRE 和 GNU ERE 它们的功能特性并没有太大区别，区别是在于部分语法层面上，主要是一些字符要不要转义。

**POSIX 字符组：**

POSIX 流派有自己的字符组，叫 POSIX 字符组。这类似于前面正则匹配规则表中预定义字符集中的 \d 表示数字，\s 表示空白符等，POSIX 中也定义了一系列的字符组。具体的清单和解释如下所示：

POSIX字符组	解释	等价表示	备注
<code>[:alnum:]</code>	数字和字母	<code>[0-9A-Za-z]</code>	比 <code>\w</code> 少了下划线
<code>[:alpha:]</code>	字母	<code>[A-Za-z]</code>	
<code>[:ascii:]</code>	ASCII	<code>[\x00-\x7F]</code>	
<code>[:blank:]</code>	空格和制表符	<code>[\t ]</code>	
<code>[:cntrl:]</code>	控制字符	<code>[\x00-\x1F\x7F]</code>	
<code>[:digit:]</code>	数字	<code>[0-9]</code>	<code>\d</code>
<code>[:graph:]</code>	可见字符	<code>[!-~]</code> <code>[A-Za-z0-9!"#\$%&amp;'()*+,-./:;&lt;=&gt;?</code> <code>@[\\\]^_`{ }~]</code>	
<code>[:lower:]</code>	小写字母	<code>[a-z]</code>	
<code>[:upper:]</code>	大写字母	<code>[A-Z]</code>	
<code>[:print:]</code>	可打印字符	<code>[ -~]</code> <code>[ :graph:]</code>	比 <code>graph</code> 多了空格
<code>[:punct:]</code>	标点符号	<code>[!-/:-@[-`{--~]</code>	
<code>[:space:]</code>	空白符号	<code>[\t\n\v\f\r ]</code>	
<code>[:xdigit:]</code>	16进制数字	<code>[0-9A-Fa-f]</code>	

### 3.1.2.2 PCRE 流派

除了 POSIX 标准外，还有一个 Perl 分支，也就是大家现在熟知的 PCRE。随着 Perl 语言的发展，Perl 语言中的正则表达式功能越来越强悍，为了把 Perl 语言中正则的功能移植到其他语言中，PCRE 就诞生了。

目前大部分编程语言的正则表达式都是源于 PCRE 标准，前面的python正则表达式也是基于PCRE 标准实现，这个流派显著特征是有`\d`、`\w`、`\s`这类字符组简记方式。

虽然 PCRE 流派是从 Perl 语言中衍生出来的，但与 Perl 语言中的正则表达式在语法上还是有一些细微差异。

虽然 PCRE 流派是与 Perl 正则表达式相兼容的流派，但这种兼容在各种语言 and 工具中还存在程度上的差别，这包括了直接兼容与间接兼容两种情况。

而且，即便是直接兼容，也并非完全兼容，还是存在部分不兼容的情况。原因也很简单，Perl 语言中的正则表达式在不断改进和升级之中，其他语言和工具不可能完全做到实时跟进与更新。

**直接兼容**，PCRE 流派中与 Perl 正则表达式直接兼容的语言或工具。比如 Perl、PHP preg、PCRE 库等，一般称之为 Perl 系。

**间接兼容**，比如 Java 系（包括 Java、Groovy、Scala 等）、Python 系（包括 Python2 和 Python3）、JavaScript 系（包括原生 JavaScript 和扩展库 XRegExp）、.Net 系（包括 C#、VB.Net 等）等。

### 3.1.3 在Linux中使用正则

在遵循 POSIX 规范的 UNIX/LINUX 系统上，按照 **BRE 标准** 实现的有 `grep`、`sed` 和 `vi/vim` 等，而按照 **ERE 标准** 实现的有 `egrep`、`awk` 等。

在 UNIX/LINUX 系统里 PCRE 流派与 POSIX 流派的对比：

PCRE流派	POSIX流派			
	vim	grep	sed	awk
*	*	*	*	*
+	\+	\+	\+	+
?	\?或\=注1	\?	\?	?
{m,n}	\{m,n}注2	\{m,n}	\{m,n}	{m,n}
\b注3	\< \>	\< \>	\y、 \< \>	\< \>
(... ...)	\(...\ ...\)	\(...\ ...\)	\(...\ ...\)	(... ...)
(...)	\(...\)	\(...\)	\(...\)	(...)
\1 \2	\1 \2	\1 \2	\1 \2	(不支持)

注1：vim中的\?和\=都表示匹配0或1个前面的子表达式，但\?不能在反向查找的“?”命令中使用。

注2：vim中的右花括号}之前加不加反斜杠都可以，比如：\{n,m}。

注3：PCRE中常用\b来表示“单词的起始或结束位置”，但Linux/Unix的工具中，通常用\<来匹配“单词的起始位置”，用\>来匹配“单词的结束位置”，而sed中的\y则与PCRE中的\b一样，可同时匹配这两个位置。

其实上表中的一些linux工具实现同时兼容多种正则标准，比如 grep 和 sed。如果在使用时加上 -E 选项，就是使用 ERE 标准；加上 -P 选项，就是使用 PCRE 标准。

使用 ERE 标准：

```
1 | grep -E '[[[:digit:]]]+' access.log
```

使用 PCRE 标准：

```
1 | grep -P '\d+' access.log
```

在 Linux 系统中可以使用 man 命令查看某个工具所属的流派，例如执行 `man grep`：

```
OPTIONS
Generic Program Information
--help Print a usage message briefly summarizing these command-line options
and the bug-reporting address, then exit.

-v, --version
Print the version number of grep to the standard output stream. This
version number should be included in all bug reports (see below).

Matcher Selection
-E, --extended-regexp
Interpret PATTERN as an extended regular expression (ERE, see below).
(-E is specified by POSIX.)

-F, --fixed-strings, --fixed-regexp
Interpret PATTERN as a list of fixed strings, separated by newlines,
any of which is to be matched. (-F is specified by POSIX,
--fixed-regexp is an obsoleted alias, please do not use it in new
scripts.)

-G, --basic-regexp
Interpret PATTERN as a basic regular expression (BRE, see below).
This is the default.

-P, --perl-regexp
Interpret PATTERN as a Perl regular expression. This is highly
experimental and grep -P may warn of unimplemented features.
```

可以看到选项 -G 是指定使用 BRE 标准（默认），-E 是 ERE 标准，-P 是 PCRE 标准。

```

1 [root@VM_0_9_centos tmp]# cat a.txt
2 abcdf
3 12345
4 [root@VM_0_9_centos tmp]# grep '\d+' a.txt
5 [root@VM_0_9_centos tmp]# grep -E '\d+' a.txt
6 abcdf
7 [root@VM_0_9_centos tmp]# grep -E '[0-9]+' a.txt
8 12345
9 [root@VM_0_9_centos tmp]# grep -P '\d+' a.txt
10 12345
11 [root@VM_0_9_centos tmp]# grep -P '[0-9]+' a.txt
12 12345

```

在 `grep` 中直接使用 `\d+` 查找不到结果是因为默认的 `grep` 属于 BRE 流派，而 `grep -E` 属于 ERE 流派也不支持 `\d`，`\d` 相当于字母 `d`，所以找到了字母那一行。而 `grep -P` 属于 PCRE 标准所以能用 python 几乎相同的正则规则匹配到数字。

当然 `grep -E` 也可以使用 `egrep` 替代，上面的命令可以更换为：

```

1 [root@VM_0_9_centos tmp]# egrep '[0-9]+' a.txt
2 12345
3 [root@VM_0_9_centos tmp]# egrep '\d+' a.txt
4 abcdf

```

下面我们希望分别使用不同的标准（即 BRE、ERE、PCRE）找出下面这段文本中含有 `ftp`、`http` 或 `https` 的行：

```

1 [root@VM_0_9_centos tmp]# cat b.txt
2 https://time.geekbang.org
3 ftp://ftp.ncbi.nlm.nih.gov
4 www.baidu.com
5 www.ncbi.nlm.nih.gov

```

参考答案：

```

1 [root@VM_0_9_centos tmp]# grep 'ftp\|https\?' b.txt
2 https://time.geekbang.org
3 ftp://ftp.ncbi.nlm.nih.gov
4 [root@VM_0_9_centos tmp]# egrep 'ftp|https?' b.txt
5 https://time.geekbang.org
6 ftp://ftp.ncbi.nlm.nih.gov
7 [root@VM_0_9_centos tmp]# grep -P 'ftp|https?' b.txt
8 https://time.geekbang.org
9 ftp://ftp.ncbi.nlm.nih.gov

```

## 3.2 正则的匹配原理以及优化原则

在前面贪婪模式与非贪婪模式一节讲了回溯算法。下面将简单讲解 DFA 和 NFA 引擎的工作方式，即正则匹配过程。

这些原理性的知识，能够帮助我们快速理解为什么有些正则表达式不符合预期，避免一些常见的错误。只有了解正则引擎的工作原理，我们才可以更轻松地写出正确的，性能更好的正则表达式。



### 3.2.1 有穷状态自动机

正则之所以能够处理复杂文本，就是因为采用了**有穷状态自动机**（finite automaton）。

- **有穷状态**是指一个系统具有有穷个状态，不同的状态代表不同的意义。
- **自动机**是指系统可以根据相应的条件，在不同的状态下进行转移。从一个初始状态，根据对应的操作（比如录入的字符集）执行状态转移，最终达到终止状态（可能有一到多个终止状态）。

有穷自动机的具体实现称为正则引擎，主要有 DFA 和 NFA 两种：

- DFA：确定性有穷自动机（Deterministic finite automaton）
- NFA：非确定性有穷自动机（Non-deterministic finite automaton）

而 NFA 又分为**传统的 NFA** 和 **POSIX NFA**。

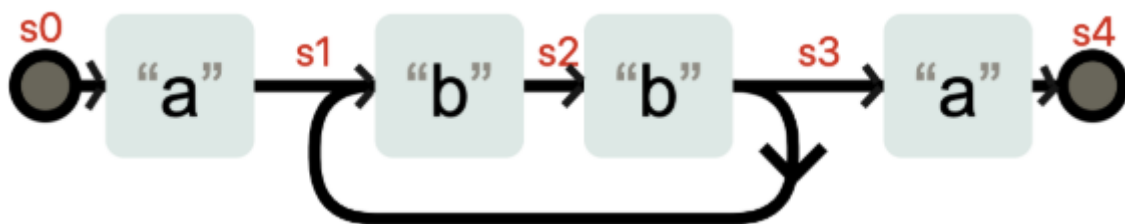
接下来我们通过一些示例，来看下正则表达式的匹配过程：

### 3.2.2 正则的匹配过程

在使用正则表达式时，我们经常会“编译”一下，来提升效率，比如：

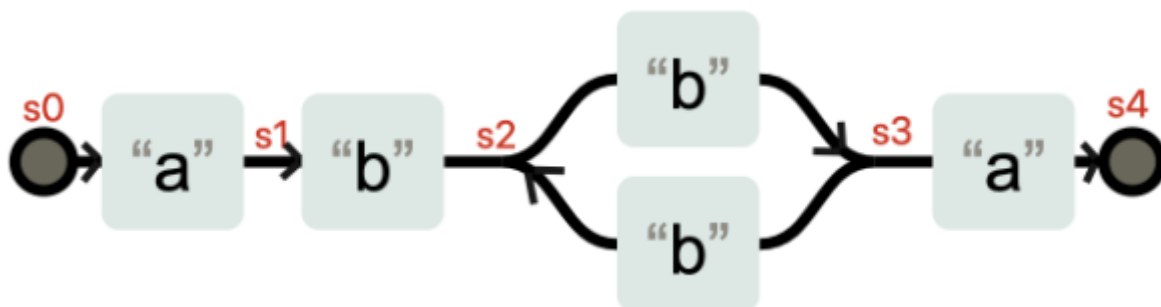
```
1 import re
2 re.compile(r'a(?:bb)+a')
```

这个编译的过程，其实就是生成自动机的过程，正则引擎会拿着这个自动机去和字符串进行匹配。生成的自动机可能是这样的：



在状态 s3 时，不需要输入任何字符，状态也有可能转换成 s1。可以理解成 a(bb)+a 在匹配了字符 abb 之后，到底在 s3 状态，还是在 s1 状态，这是不确定的。这种状态机就是非确定性有穷状态自动机（Non-deterministic finite automaton 简称 NFA）。

**NFA 和 DFA 是可以相互转化的**，当我们把上面的状态表示成下面这样，就是一台 DFA 状态机了，因为在 s0-s4 这几个状态，每个状态都需要特定的输入，才能发生状态变化。



下面看看这两种状态机的工作方式的差异：

### 3.2.3 DFA& NFA 工作机制

假设我们使用的字符串和正则如下：

```
1 | 字符串: we search use baidu engine
2 | 正则: use (sogou|baidu|bing|google)
```

**NFA 引擎**的工作方式是，先看正则，再看文本，而且以正则为主导。正则中的第一个字符是u，NFA 引擎在字符串中查找u，接着匹配其后是否为s，如果是s则继续，这样一直找到 "use "。

```
1 | regex: use (sogou|baidu|bing|google)
2 |           ^
3 | text: we search use baidu engine
4 |           ^
```

再根据正则看文本后面是不是b，发现不是，此时 sogou分支淘汰。

```
1 | regex: use (sogou|baidu|bing|google)
2 |           ^
3 |           淘汰此分支(sogou)
4 | text: we search use baidu engine
5 |           ^
```

我们接着看其它的分支，看文本部分是不是 b，直到 baidu 整个匹配上。当匹配上了 baidu后，整个文本匹配完毕，也不会再看 bing分支和google分支。否则匹配失败会继续尝试匹配 bing分支。

假设这里text文本改一下，把 baidu变成 bing，正则 baidu的a匹配不上时 bing 的 i，会接着使用正则 bing来进行匹配，重新从b开始（NFA 引擎会记住这里）。

第二个分支匹配失败：

```
1 | regex: use (sogou|baidu|bing|google)
2 |           ^
3 |           淘汰此分支(正则a匹配不上文本i)
4 | text: we search use bing engine
5 |           ^
```

再次尝试第三个分支：

```
1 | regex: use (sogou|baidu|bing|google)
2 |           ^
3 | text: we search use bing engine
4 |           ^
```

也就是说，NFA 是以正则为主导，反复测试字符串，这样字符串中同一部分，有可能被反复测试很多次。

而 **DFA**会先看文本，再看正则表达式，是以文本为主导的。在具体匹配过程中，DFA 会从 we 中的 w 开始依次查找u，定位到 u，这个字符后面是s。所以我们接着看正则部分是否有s，如果正则后面是个 s，那就以同样的方式，匹配到后面的"se "。

```
1 | text: we search use bing engine
2 |           ^
3 | regex: use (sogou|baidu|bing|google)
4 |           ^
```

继续进行匹配，空字符串后面的字符是b，DFA接着看正则表达式部分，此时 sogou分支和google分支被淘汰，开头是b的分支 baidu和 bing符合要求。

```
1 text: we search use bing engine
2                               ^
3 regex: use (sogou|baidu|bing|google)
4             ^      ^      ^      ^
5             淘汰   符合   符合   淘汰
```

然后 DFA 依次检查字符串，检测到 bing中的 i 时，只有 bing分支符合，淘汰 baidu，接着看分别文本后面的 ng，和正则比较，匹配成功。

```
1 text: we search use bing engine
2                               ^
3 regex: use (sogou|baidu|bing|google)
4             ^      ^
5             淘汰   符合
```

可以看到，NFA 是以表达式为主导的，先看正则表达式，再看文本。而 DFA 则是以文本为主导，先看文本，再看正则表达式。

一般来说，DFA 引擎会更快一些，因为整个匹配过程中，字符串只看一遍，不会发生回溯，相同的字符不会被测试两次。也就是说 DFA 引擎执行的时间一般是线性的。DFA 引擎可以确保匹配到可能的最长字符串。但由于 DFA 引擎只包含有限的状态，所以它没有反向引用功能；并且因为它不构造显示扩展也不支持捕获子组。

NFA 以表达式为主导，它的引擎是使用贪心匹配回溯算法实现。NFA 通过构造特定扩展，支持子组和反向引用。但由于 NFA 引擎会发生回溯，即它会对字符串中的同一部分，进行很多次对比。因此，在最坏情况下，它的执行速度可能非常慢。

### 3.2.4 POSIX NFA 与传统 NFA 区别

因为传统的 NFA 引擎“急于”报告匹配结果，找到第一个匹配上的就返回了，所以可能会导致还有更长的匹配未被发现。比如使用正则 `pos|posix` 在文本 `posix` 中进行匹配，传统的 NFA 从文本中找到的是 `pos`，而不是 `posix`，而 POSIX NFA 找到的是 `posix`。

REGULAR EXPRESSION

1 match (~0ms)

/ pos|posix

/ gm

TEST STRING

SWITCH TO UNIT TESTS ▶

posix

POSIX NFA 的应用很少，主要是 Unix/Linux 中的某些工具。POSIX NFA 引擎与传统的 NFA 引擎类似，但不同之处在于，POSIX NFA 在找到可能的最长匹配之前会继续回溯，也就是说它会尽可能找最长的，如果分支一样长，以最左边的为准（“The Longest-Leftmost”）。因此，POSIX NFA 引擎的速度要慢于传统的 NFA 引擎。

我们日常面对的，一般都是传统的 NFA，所以通常都是最左侧的分支优先，在书写正则的时候务必要注意这一点。

下面是 DFA、传统 NFA 以及 POSIX NFA 引擎的特点总结：

引擎类型	程序	忽略优先量词 (懒惰)	捕获型 括号	回溯
DFA	Golang、MySQL、awk (大多数版本)、egrep (大多数版本)、flex、lex、Procmail	不支持	不支持	不支持
传统型 NFA	PCRE library、Perl、PHP、Java、Python、Ruby、grep (大多数版本)、GNU Emacs、less、more、.NET 语言、sed (大多数版本)、vi	支持	支持	支持
POSIX NFA	mawk、Mortice Kern Systems' utilities、GNU Emacs (明确指定时使用)	不支持	不支持	支持
DFA/NFA 混合	GNU awk、GNU grep/egrep、Tcl	支持	支持	DFA 支持

### 3.2.5 回溯详解

回溯是 NFA 引擎才有的，并且只有在正则中出现量词或多选分支结构时，才可能会发生回溯。

比如我们使用正则 `a+ab` 来匹配文本 `aab` 的时候，`a+` 是贪婪匹配，会占用掉文本中的两个 `a`，但正则接着又是 `a`，文本部分只剩下 `b`，只能通过回溯，让 `a+` 吐出一个 `a`，再次尝试。

举个极端的例子，使用正则 `. *ab` 去匹配一个比较长的字符串时，`. *` 会吃掉整个字符串（不考虑换行），但正则中还有 `ab` 没匹配到内容只能将 `. *` 已经匹配的字符串吐出一个字符，再尝试，还不行，再吐出一个，不断尝试：

`. *ab` The lab assistant was wearing a white overall.

`. *ab` The lab assistant was wearing a white overall.

`. *ab` The lab assistant was wearing a white overall.

中间过程省略，一直回溯到（吐出I之后的所有匹配上的）

`. *ab` The lab assistant was wearing a white overall.

`. *ab` The lab assistant was wearing a white overall.

`. *ab` The lab assistant was wearing a white overall.

理解了这个过程，我们就能明白，提取引号中的内容时，需要使用 `"[^"]+"` 或者使用非贪婪的方式 `".+?"`，来减少“匹配上的内容不断吐出，再次尝试”的过程。



