正则表达式(python)

郭　炜

正则表达式

正则表达式是个字符串，规定了一种模式，如：

```
"abc"
"b.?p.*k"
"\d{3}([a-zA-Z]+).(\d{2}|N/A)\s\1"
```

可以用相关函数求给定字符串和正则表达式的匹配情况

字符			
一般字符	匹配自身	abc	abc
.	匹配任意除换行符"\n"外的字符。 在DOTALL模式中也能匹配换行符。	a.c	abc
\	转义字符，使后一个字符改变原来的意思。 如果字符串中有字符*需要匹配，可以使用*或者字符集[*]。	a\.c a\\c	a.c a\c
[...]	字符集（字符类）。对应的位置可以是字符集中任意字符。 字符集中的字符可以逐个列出，也可以给出范围，如[abc]或[a-c]。第一个字符如果是^则表示取反，如[^abc]表示不是abc的其他字符。 所有的特殊字符在字符集中都失去其原有的特殊含义。在字符集中如果要使用]、-或^，可以在前面加上反斜杠，或把]、-放在第一个字符，把^放在非第一个字符。	a[bcd]e	abe ace ade

[^abc]	匹配 **"a","b","c"** 之外的任意一个字符
[^A-F0-3]	匹配 **"A"~"F","0"~"3"** 之外的任意一个字符

预定义字符集 (可以写在字符集[...]中)			
\d	数字 : [0-9]	a\dc	a1c
\D	非数字 : [^\d]	a\Dc	abc
\s	空白字符 : [<空格>\t\r\n\f\v]	a\sc	a c
\S	非空白字符 : [^\s]	a\Sc	abc
\w	单词字符 : [A-Za-z0-9_]	a\wc	abc
\W	非单词字符 : [^\w]	a\Wc	a c

数量词（用在字符或(...)之后）			
*	匹配前一个字符0或无限次。	abc*	ab abccc
+	匹配前一个字符1次或无限次。	abc+	abc abccc
?	匹配前一个字符0次或1次。	abc?	ab abc
{m}	匹配前一个字符m次。	ab{2}c	abbc
{m,n}	匹配前一个字符m至n次。 m和n可以省略：若省略m，则匹配0至n次；若省略n，则匹配m至无限次。	ab{1,2}c	abc abbc

```
'.+'          匹配任意长度不为0 的字符串
'[\dac]+'     '451a'    '451c78ca'    ......
'\w{5}'       匹配任意长度为5的字母或者数字
```

转义字符

\\	表示\
*	表示*
\$	表示 $
\.	表示.
\[	表示[
\]	表示[
\(	表示(
\)	表示(
\?	表示?
\^	表示^
\{	表示{
\}	表示}

re.match函数

- `re.match(pattern, string, flags=0)`

•从字符串的起始位置匹配一个模式

•flags　标志位，用于控制模式串的匹配方式，如：是否区分大小写，多行匹配等等,如 re.M ！| re.I　表示忽略大小写，且多行匹配

•成功则返回一个匹配对象，否则返回None

re.search函数

- **re.search(pattern, string[, flags]):**

- 查找字符串中可以匹配成功的子串。

- 若匹配成功，则返回匹配对象；若无法匹配，则返回None。

正则表达式修饰符 - 可选标志

➢匹配函数如search,match可以通过可选标志来控制匹配的模式。多个标志可以通过按位 OR(|) 来指定。如 re.I | re.M

修饰符	描述
re.I	使匹配对大小写不敏感
re.L	做本地化识别（locale-aware）匹配
re.M	多行匹配，影响 ^ 和 $
re.S	使 . 匹配包括换行在内的所有字符
re.U	根据Unicode字符集解析字符。这个标志影响 \w, \W, \b, \B.
re.X	该标志通过给予你更灵活的格式以便你将正则表达式写得更易于理解。

分组(...)

- 括号中的表达式是一个分组。多个分组按左括号左到右从1开始依次编号

```
# 找出全部 []和 <> 中的数
s = '233[32]88ab<433>'
m = '\[(\d+)\]|<(\d+)>'          # | 表示 '或'
for x in re.finditer(m,s):
    print(x.group())             # 等价于 group(0)
    print(x.groups())
```

```
[32]
('32', None)
<433>
(None, '433')
```

- re.finditer的返回值是一个匹配对象的集合，每个匹配对象表示匹配好的一个子串
- 匹配对象的 group(i)返回第i个分组匹配的字符串，group()等价于group(0),是整个正则表达式所匹配的字符串
- 匹配对象的 groups 函数返回一个元组，元素依次是1号分组、2号分组.....所匹配的字符串

分组(...)

- 括号中的表达式是一个分组。多个分组按左括号左到右从1开始依次编号

```
m = "(((ab*)c)d)e"
r = re.match(m,"abcdefg")
print(r.groups())              # ('abcd', 'abc', 'ab')
print(r.lastindex)             # 1
print(r.group(0))              # abcde
print(r.group(1))              # abcd
print(r.group(2))              # abc
print(r.group(3))              # ab
m = '(\w+) (\w+)'
r = re.match(m,"hello world")
print(r.groups())              # ('hello', 'world')
r = re.findall(m,"hello world, this is very good")
print(r) # [('hello', 'world'), ('this', 'is'), ('very', 'good')]
```

分组(...)

➢在分组的右边可以通过分组的编号引用该分组所匹配的子串

```
import re
m = r'(((ab*)c)d)e\3'
r = re.match(m,"abbbcdeabbbkfg") # 红色部分少一个b则不能匹配
print(r.group(3))                # abbb
print(r.group())                 # abbbcdeabbb
s = 'abc.xyz'
print(re.sub(r'(.*)\.(.*)', r'\2.\1', s))       # xyz.abc
# 用  '(.*)\.(.*)' 匹配 s,并用 '\r.\1' 替换s中被匹配的子串
```

分组(...)

- 分组可以用 ?P<name> 格式指定名字为name，并用名字引用其匹配的子串

```
import re
m = r'(?P<gp1><\w+)\s+(?P=gp1)\s+(?P=gp1)'
#一个分组名为"gp1"
print(re.search(m, 'at<go <go <go').group('gp1')) # <go
```

分组(...)

- 分组作为一个整体，后面可以跟数量词

```
import re
m = "(((ab*)+c)d)e"
r = re.match(m,"ababcdefg")
print(r.groups())          # ('ababcd', 'ababc', 'ab')

m =re.compile(r'\d{1,3}(\.\d{1,3}){3}')
print(m.match("192.168.0.21").groups())   # ('.21',)
# 一个分组匹配多次，只返回最后一次
```

“匹配对象”的属性

- **string**: 匹配时使用的母串。
- **re**: 匹配时使用的Pattern对象(编译后的)。
- **pos**: 文本中模式串开始搜索的位置。值与Pattern.match()和Pattern.seach() 的同名参数相同。
- **endpos**: 文本中模式串结束搜索的位置。值与Pattern.match()和Pattern.seach()方法的同名参数相同。
- **lastindex**: 最后一个被匹配的分组的编号(不是最大编号）。如果没有被匹配的分组，将为None。
- **lastgroup**: 最后一个被匹配的分组的名字。如果这个分组没有名字或者没有被匹配的分组，将为None。

“匹配对象”的方法

- **group([n1,n2, …]):**

  - 获得一个或多个分组匹配的字符串；指定多个参数时将以元组形式返回。n1,n2...可以使用编号也可以使用名字；

  - 编号0代表整个匹配的子串(与模式里面的"()"无关)；
  - group() 等价于 group(0)；
  - 没有匹配字符串的组返回None；
  - 匹配了多次的组返回最后一次匹配的子串。

“匹配对象”的方法

- **groups([default]):**
  以元组形式返回全部分组匹配的字符串。相当于调用group(1,2,…last)。default表示没有匹配字符串的组以这个值替代，默认为None。
- **groupdict([default]):**
  返回以有名字的组的名字为键、以该组匹配的子串为值的字典，没有名字的组不包含在内。default含义同上。
- **start([group]):**
  返回指定的组匹配的子串在string中的起始位置。group默认值为0。
- **end([group]):**
  返回指定的组匹配的子串在string中的结束位置（子串最后一个字符的位置 +1）。group默认值为0。

“匹配对象”的方法

- **span([group]):**
  返回(start(group), end(group))。
  group可以是组编号，也可以是组名字，缺省为0
- **expand(template):**
  将匹配到的分组代入template中然后返回。template中可以使用\id或\g<id>、\g<name>引用分组(id 为数，不可为0)。\id与\g<id>等价；
  \10 表示第10个分组，\g<1>0表示第1个分组后面跟字符0。

“匹配对象”的方法

```
m = re.match(r'(\w+) (\w+)(?P<sign>.)', 'hello world!ss')
print( m.string)      # hello world!ss
print(m.re)    # re.compile('(\\w+) (\\w+)(?P<sign>.)')
print(m.pos)   # 0
print(m.endpos)       # 14
print(m.lastindex) # 3
print(m.lastgroup) # sign
print(m.group(0,1,2,3))  # ('hello world!', 'hello', 'world', '!')
print(m.groups())  # ('hello', 'world', '!')
print(m.groupdict())  # {'sign': '!'}
print(m.start(2))     # 6
print(m.end(2))               # 11
print(m.span("sign"))# (11, 12)
print(m.expand(r'\2 \1\3')) # world hello!
```

“匹配对象”的方法

```
m = re.match(r'(\w+) (\w+)(?P<sign>.$)', 'hello world!ss')
if m == None:
   print("no match")        # no match
```

'$' 表示匹配到此终止，母串后面再有字符则视为不能匹配

Pattern对象

- **Pattern对象由 re.compile(模式串） 得到**

- **Pattern对象属性：**
  - pattern: 编译时用的模式串。
  - flags: 编译时用的匹配模式(形式为整数)。
  - groups:模式串表达式中组的数量。
  - groupindex: 以模式串中有别名的组的别名为键、以该组对应的编号为值的字典，没有别名的组不包含在内。

Pattern对象的属性

```
import re
p = re.compile(r'(\w+) (\w+)(?P<sign>.*)', re.DOTALL|re.M )
print(p.pattern)          # (\w+) (\w+)(?P<sign>.*)
print( p.flags)           # 48
print (p.groups)          # 3
print(p.groupindex)       {'sign': 3}
```

Pattern对象的方法

- **match(string[, pos[, endpos]]) | re.match(pattern, string[, flags]):**

- 从string的pos处起匹配模式串；若匹配未结束就已到达endpos，则返回None。
- 匹配成功则返回匹配对象
- flags用于编译模式串时指定匹配模式。
- 不是完全匹配。当模式串结束时若string还有剩余字符，仍然视为成功。想要完全匹配，可以在模式串末尾加上边界匹配符'$'。

Pattern对象的方法

- **search(string[, pos[, endpos]]) | re.search(pattern, string[, flags]):**

- 查找字符串中可以匹配成功的子串。

- 从string的pos下标处起匹配模式串，如果模式串结束时仍可匹配，则返回匹配对象；若无法匹配，则将pos加1后重新匹配；pos=endpos时仍无法匹配则返回None。

Pattern对象的方法

- **split(string[, maxsplit]) | re.split(pattern, string[, maxsplit]):**

- 用能够匹配模式串的子串将string分割，返回分割后的子串列表
- maxsplit: 最多分割成 maxsplit + 1 个子串。不指定则等于无穷大

```
p = re.compile(r'\d+')
print(p.split('one1two2three3four4'))
   # ['one', 'two', 'three', 'four', '']
print(p.split('one1two2three3four4',2))
   # ['one', 'two', 'three3four4']
```

Pattern对象的方法

- **findall(string[, pos[, endpos]]) | re.findall(pattern, string[, flags]):**

  搜索string，以列表形式返回全部能匹配的不交叠的子串。

```
p = re.compile(r'\d+|bb')
print(p.findall('one1two2abbbthree33four4'))
# ['1', '2', 'bb', '33', '4']
p = re.compile(r'(\w+) (\w+)')
s = 'i say hello world!'
print( p.findall(s))
# [('i', 'say'), ('hello', 'world')]
```

Pattern对象的方法

- **finditer(string[, pos[, endpos]]) | re.finditer(pattern, string[, flags]):**

  搜索string(findall方式），匹配上的每个子串都对应于一个匹配对象。返回一个顺序访问每一个匹配对象的迭代器。

```
import re
p = re.compile(r'\d+')
for m in p.finditer('one1two22three3four44'):
    print (m.group(),end= " ")
# 1 22 3 44
```

Pattern对象的方法

finditer(string[, pos[, endpos]]) | re.finditer(pattern, string[, flags]):

```
import re
m =re.compile(r"(((ab*)+c|12)d)e")
for x in m.finditer('ababcdefgKK12deK'):
    print (x.groups())
'''
('ababcd', 'ababc', 'ab')
('12d', '12', None)
'''
```

Pattern对象的方法

- **sub(repl, string[, count]) | re.sub(pattern, repl, string[, count]):**

- 使用repl替换string中每一个匹配(search匹配)的子串后，返回替换后的字符串

- 当repl是一个字符串时，可以使用\id或\g<id>、\g<name>引用分组，但不能使用编号0。
- 当repl是一个函数时，这个函数只接受一个参数（Match对象），并返回一个字符串用于替换（返回的字符串中不能引用分组）。

- count用于指定最多替换次数，不指定时全部替换。

Pattern对象的方法

- **sub(repl, string[, count]) | re.sub(pattern, repl, string[, count]):**

```
phone = "2004-959-559 # 这是一个电话号码"
# 删除注释
num = re.sub(r'#.*$', "", phone)
print (num)                    # 2004-959-559
# 移除非数字的内容
num = re.sub(r'\D', "", phone)
print (num)                    # 2004959559
```

Pattern对象的方法

- **sub(repl, string[, count]) | re.sub(pattern, repl, string[, count]):**

```
p = re.compile(r'(\w+) (\w+)')
s = '@i say, hello world!'
print(p.sub('ok', s))# @ok, ok!
print(p.sub(r'\2', s))
   # @say, world! 对每个匹配的子串，用2号分组替代整个子串
print(p.sub(r'\2 \1', s))    # @say i, world hello!
# 对每个匹配的子串，用 "2号分组 1号分组" 替代整个子串
def func(m):
    #m是个匹配对象，里面包含匹配上的子串
    return m.group(1).title() + '_' + m.group(2).title()
print( p.sub(func, s))       # @I_Say, Hello_World!
```

Pattern对象的方法

- **subn(repl, string[, count]) |re.sub(pattern, repl, string[, count]):**

与 sub 相比，多返回替换次数

```
import re
p = re.compile(r'(\w+) (\w+)')
s = 'i say, hello world!'
print (p.subn(r'\2 \1', s))
# ('say i, world hello!', 2)
def func(m):
    return m.group(1).title() + ' ' + m.group(2).title()
print (p.subn(func, s))
# I Say, Hello World!
```

量词的贪婪模式

- 量词 +,*,?,{m,n} 默认匹配尽可能长的字符串

```
m = "ab*"
a = re.match(m,"abbbbk")
print(a.group()) # abbbb

m = "<h3>.*</h3>"
a = re.match(m,"<h3>abd</h3><h3>bcd</h3>")
print(a.group()) # <h3>abd</h3><h3>bcd</h3>
```

量词的非贪婪(懒惰)模式

- 在量词 +,*,?,{m,n} 后面加 '?' 则匹配尽可能短的字符串 。

```
import re
m = "a.*?b"
for k in re.finditer(m,"aabab"):
    print(k.group(),end=" ") # aab ab
m = "<h3>.*?</h3>"
a = re.match(m,"<h3>abd</h3><h3>bcd</h3>")
print(a.group())    # <h3>abd</h3>
m = "<h3>.*?[M|K]</h3>"
a = re.match(m,"<h3>abd</h3><h3>bcK</h3>")
print(a.group())    # <h3>abd</h3><h3>bcK</h3>
```

字符边界

\A　与字符串开始处匹配，不消耗任何字符

\Z　与字符串结束的地方匹配，不消耗任何字符

```
import re
m = "\Ahow are"
print(re.search(m,"ahow are you"))      # None
a = re.search(m,"how are you")
print(a.group())     # how are
m = "are you\Z"
print(re.search(m,"how are you?"))  # None
print(re.search(m,"how are you").group()) # are you
```

字符边界

^ 与字符串开始处匹配，不消耗任何字符。在多行模式中，匹配每一行开头

$ 与字符串结束的地方匹配，不消耗任何字符。在多行模式中，匹配每一行末尾

```
import re
m = "^how are"
g = re.findall(m,"how are you\nhow are me",re.M)
print(g)      # ['how are', 'how are']
m = "are you$"
print(re.search(m,"how are you\nThis",re.M ).group())
# are you
```

字符边界

\b　匹配一个单词的开始处和结束处，不消耗任何字符，等效于匹配\w和\W之间

\B　和\b相反,不允许是单词的开始处和结束处，即[^\b]

```
import re
m = r"\bA.*N\b T"
print(re.search(m,"Ass$NC TK"))     # None
print(re.search(m,"this Ass$N TK").group())     # Ass$N T
m = r"\BA.*N\B\w T"
print(re.search(m,"this Ass$N TK"))           # None
print(re.search(m,"thisAss$NM TK").group())  # Ass$NM T
```

"|" 的用法

- 表示“或”，如果没有放在"()"中，则起作用范围是直到整个正则表达式开头或结尾或另一个 "|"
- 短路匹配

```
"\w{4}ce|c\d{3}|p\w"
```

可以匹配：

"c773"

"ab12ce"

"pk"

"|" 的用法

- '|' 也可以用于分组中，起作用范围仅限于分组内

```
import re
m =re.compile(r"(((ab*)+c|12)d)e")
for x in m.finditer('ababcdefgKK12deK'):
    print (x.groups())
'''
('ababcd', 'ababc', 'ab')
('12d', '12', None)
'''
```

断言

➢分组开头为 ?=，?!，?<=，?<!，则分组成为断言。断言必须被满足，但不消耗字符。

Y(?=X)	声明目标串Y(可为空）右侧须满足模式X。X不消耗任何字符。例如，\w+(?=\d) 与后跟数字的单词匹配，但该数字未被消耗。
Y(?!X)	声明目标串Y(可为空）右侧不允许满足模式X。X不消耗任何字符。例如，例如，\w+(?!\d) 与后不跟数字的单词匹配，但不消耗该数字 。
(?<=X)Y	声明目标串Y(可为空）左侧必须满足模式X。X不消耗任何字符。例如，(?<=19)89 与跟在 19 后面的 89 的实例匹配。X须定长。
(?<!X)Y	声明目标串Y(可为空）左侧不允许满足模式X。例如，(?<!19)89 与不跟在 19 后面的 89 的实例匹配。X须定长。

断言

```
import re
s  = "da12bka3434bdca4343bdca234bm"
#提取s中包含在字符a和b之间的数字，但是这个a之前的字符不能是c；b后面的字符必须是d才能提取。

m = '(?<=[^c]a)\d+(?=bd)'
print(re.search(m,s).group())          # 3434
```

断言

- 如何用正则表达式表示“不含 `hello` 的非空字符串”

断言

➢如何用正则表达式表示“不含 hello 的非空字符串”

```
m = '^(?!.*hello).+'      # 正确写法
print(re.search(m,"fashellodf").group())   # error

m = '(?!.*hello).+'
print(re.search(m,"fashellodf").group())   # ?
```

断言

➢如何用正则表达式表示“不含 hello 的非空字符串”

```
m = '^(?!.*hello).+'
print(re.search(m,"fashellodf").group())   # error

m = '(?!.*hello).+'
print(re.search(m,"fashellodf").group())   # ellodf

m = '(?!.*hello).+?'
print(re.search(m,"fashellodf").group())   # ?
```

断言

➢如何用正则表达式表示“不含 hello 的非空字符串”

```
m = '^(?!.*hello).+'
print(re.search(m,"fashellodf").group())   # error

m = '(?!.*hello).+'
print(re.search(m,"fashellodf").group())   # ellodf

m = '(?!.*hello).+?'
print(re.search(m,"fashellodf").group())   # e
```

断言

- 如何找出下面字符串中不含"hello"的单词

```
s = 'this  hello thello hellob ahellob take me'
```

断言

- 如何找出下面字符串中不含"hello"的单词

```
s = 'this  hello   thello hellob back ahellob take me'
m = r'\b(?![^ ]*hello)\w+\b'
k = re.search(m,s)
for s in re.finditer(m,s):
    print(s.group())
```

this
back
take
me

断言

- 不含字符a和b的字符串

断言

- 不含字符a和b的字符串

```
r'^(?!.*a|.*b).*'
```

断言

- 如何用正则表达式表示“由素数个字符a构成的串”

```
r'^(?!(aa+)\1+$)a+$'
```

断言

- 找出不含192的 ip地址

```
s = "192.168.0.1 8102.10.195.193a 102.387.192.8 " + \
      "ab102.10.195.199b34.23.33.192 12.34.34.8"
m = ?
for x in re.finditer(m,s):
    print(x.group(1))
```

102.10.195.199
12.34.34.8

断言

- 找出不含192的 ip地址

```
s = "192.168.0.1 8102.10.195.193a 102.387.192.8 " + \
      "ab102.10.195.199b34.23.33.192 12.34.34.8"
m = '(?<!\d)(\d{1,3}(?<!192)(\.(?!192)\d{1,3}){3})(?!\d)'
for x in re.finditer(m,s):
    print(x.group(1))
```

102.10.195.199
12.34.34.8

断言

- 找出不含38的 ip地址

```
s = "6102.10.195.193a 38.22.2.22 102.387.192.8 " + \
 "ab138.10.195.199 ab132.381.195.199 123.138.33.44" + \
    "123.38.44.55a299.371.23.3"
m = ?
for x in re.finditer(m,s):
    print(x.group(1))
```

299.371.23.3

断言

- 找出不含38的 ip地址

```
s = "6102.10.195.193a 38.22.2.22 102.387.192.8 " + \
 "ab138.10.195.199 ab132.381.195.199 123.138.33.44" + \
    "123.38.44.55a299.371.23.3"
m = \
'(?<!\d)(\d{1,3}(?<!38)(?<!38.)(?<!.38)(\.(?!.?38.?)\d{1
,3}){3})(?!\d)'
for x in re.finditer(m,s):
    print(x.group(1))
```

299.371.23.3

嵌入条件

- 回溯引用条件写法1：

  (?(backreference)true_exp)

backreference为分组编号或分组名。若其代表的分组在前面已被匹配，则需匹配**true_exp**。此时若无法匹配**true_exp** ，则试图取消对前面分组的匹配

```
m = '(<a>)?\[img.*?](?(1)</a>)'
# m = '(?P<G1><a>)?\[img.*?](?(G1)</a>)' #等价于上一行，G1为分组名
print(re.search(m,"[img aaa]").group()) # [img aaa]
print(re.search(m,"<a>[img aaa]</a>").group()) #<a>[img aaa]</a>
print(re.search(m,"<a>[img aaa]").group())  # [img aaa]
```

嵌入条件

- 回溯引用条件写法2：

  (?(backreference)true_exp|false_exp)

backreference为分组编号或分组名。若其代表的分组在前面已被匹配，则须匹配**true_exp**。否侧须**false_exp**。

```
m = '(?P<G1><a>)?\[img.*?](?(G1)</a>|no)'
print(re.search(m,"[img aaa]no").group())        # [img aaa]no
print(re.search(m,"<a>[img aaa]</a>").group())# <a>[img aaa]</a>
print(re.search(m,"<a>[img aaa]").group())       # error
```

嵌入条件

- 回溯引用条件写法2：

  (?(backreference)true_exp|false_exp)

匹配美式电话号码： (123)456-7890 或 123-456-7890

```
m = '(\()?\d{3}(?(1)\)|-)\d{3}-\d{4}'
for x in  \
      re.finditer(m,"123-456-7890 tome jack (123)911-1357 af"):
    print(x.group(0))
```

```
123-456-7890
(123)911-1357
```

查找汉字

```
m= 'a高.?'
s = 'a高达'
print(re.match(m,s).group())   # a高达

m= '^[\u4E00-\u9FA5]+$'        # 汉字编码范围
s = '北京大学'
print(re.match(m,s).group())
```

C++ STL 常用正则表达式模板

basic_regex	正则表达式模板　typedef basic_regex<char> regex ;
regex_match	判断一个字符串是否和正则表达式完全匹配
regex_search	寻找字符串中的子串中与正则表达式匹配的结果,**在找到第一个匹配的结果后就会停止查找**
regex_replace	使用格式化的替换文本，替换正则表达式匹配到字符序列的地方
regex_iterator	迭代器类模版，用来匹配所有 的子串
match_results	容器类模版，保存正则表达式匹配的结果。

C++正则表达式示例

```
#include <iostream>
#include <regex> //使用正则表达式须包含此文件
using namespace std;
int main()
{
        regex reg("b.?p.*k");
        cout << regex_match("bopggk",reg) <<endl;//输出 1, 表示匹配成功
        cout << regex_match("boopgggk",reg) <<endl;//输出 0, 匹配失败
        cout << regex_match("b pk",reg) <<endl;   //输出 1, 表示匹配成功
        regex reg2("\\d{3}([a-zA-Z]+).(\\d{2}|N/A)\\s\\1");
        string correct="123Hello N/A Hello";
        string incorrect="123Hello 12 hello";
        cout << regex_match(correct,reg2) <<endl; //输出 1,匹配成功
        cout << regex_match(incorrect,reg2) << endl; //输出 0, 失败
}
```

C++ 正则表达式示例

```
#include <iostream>
#include <regex>
#include <string>
using namespace std;

string m = "(\\d{3}-|\\(\\d{3}\\))\\d{3}-\\d{4}";
string s = "123-456-7890 tome jack (123)911-1357 af";
regex rgx(m);
cmatch match; //匹配对象
//cmatch即: match_results<const char *> match;
```

```
int main() {
        if( regex_search(s.c_str(),match,rgx)) {
                auto n = match.size(); //返回匹配对象的分组数目
                cout << "n=" << n << endl;
                for( auto i = 0; i < n; ++i) {
                        cout << match[i] <<  endl;  //输出第 i 个分组
                        //分组起始位置
                        cout << match[i].first - s.c_str() << endl;
                        //分组终止位置
                       cout << match[i].second - s.c_str() << endl;
                }
        }
```

```
n=2
123-456-7890
0
12
123-
0
4
```

```
        for ( sregex_iterator it(s.begin(),s.end(),rgx),end;
        //sregex_iterator 即 regex_iterator <string::const_iterator>
        //end是尾后迭代器,
        it != end;
        ++it)
    {
        std::cout << it->str() << std::endl;
        for(auto i = 0;i < it->size(); ++i) {
                smatch mt = * it;
// smatch 即 match_results<string::const_iterator>
                cout << mt[i] << endl;
                cout << mt[i].first - s.begin() << endl;
                cout << mt[i].second - s.begin() << endl;
        }
    }
        return 0;
}
```

```
123-456-7890
123-456-7890
0
12
123-
0
4
(123)911-1357
(123)911-1357
23
36
(123)
23
28
```

正则表达式示例

```
[1-9]\d*          正整数
-[1-9]\d*         负整数
-?[1-9]\d*        整数
[1-9]\d*|0        非负整数
-?([1-9]\d*\.\d*|0\.\d*[1-9]\d*|0?\.0+|0)    小数
\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*      邮箱
```

参考链接

- http://www.cnblogs.com/huxi/archive/2010/07/04/1771073.html
- http://www.runoob.com/python3/python3-reg-expressions.html
- http://www.cnblogs.com/leezhxing/p/4333773.html
- http://blog.csdn.net/keep_vitality/article/details/50277255
- http://www.cnblogs.com/ittinybird/p/4853532.html

```
m  = "<font>((?!</?font>).)*</font>"
# m = "<font>(?!.*</?font>)</font>"
s = '<font>ssf <font>fdfdfs</font> fasdf</font>fff'
print(re.search(m,s).group())
```