# 正则表达式

## 基本概念

正则表达式是对字符串操作的一种**逻辑公式**，就是用事先定义好的一些特定字符、及这些特定字符的组合，组成一个“规则字符串”，这个“规则字符串”用来表达对字符串的一种过滤逻辑。

正则表达式（Regular Expression）是一种文本模式，包括普通字符和特殊字符。

正则表达式使用单个字符串来描述、匹配一系列某个句法规则的字符串。

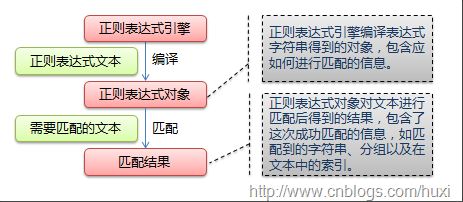
## 简介

### 为什么使用正则表达式

* 测试字符串内的模式
* 替换文本
* 基于模式匹配从字符串中提取子字符串

## 语法

正则表达式（regular expression）描述了一种**字符串匹配的模式**（pattern），可以用来检查一个串是否含有某些子串、将匹配的子串替换或者从某个串中取出符合某个条件的子串等。



### 普通字符

普通字符包括没有显示指定为元字符（特殊字符和符号）的所有可打印和不可打印字符，这包括所有大写和小写字母、所有数字、所有标点符号和一些其他字符。

### 非打印字符



特别注意：\s和\S

### 特殊字符

所谓特殊字符，就是一些含有**特殊含义**的字符。许多元字符要求在试图匹配它们时特别对待。若要匹配这些特殊字符，必须首先使用字符“转义”，即，将反斜杠“\”放在它们前面。



### 限定符

限定符用来指定正则表达式的一个给定组件必须要出现多少次才能满足匹配。



* “\*”匹配前面的子表达式零次或多次，例如/zo\*/，匹配“\*”前的字符“o”零次或多次，匹配“z”时匹配了“\*”前的子表达式“o”是零次。
* \*、+和?限定符都是贪婪的，因为它们会尽可能多的匹配文字，只有在它们的后面加上?就可以实现非贪婪或最小匹配。例如：<H1>Chapter 1 - 介绍正则表达式</H1>

贪婪模式：

下面的表达式匹配从开始小于符号 (<) 到关闭 H1 标记的大于符号 (>) 之间的所有内容。/<.\*>/ ，贪婪模式匹配结果：<H1>Chapter 1 - 介绍正则表达式</H1>

import re

pattern = *r'<.\*>'*

p = re.compile(pattern)

match = p.match(*'<H1>Chapter 1 - 介绍正则表达式</H1>'*)

if match:

print match.group() # <H1>Chapter 1 - 介绍正则表达式</H1>

非贪婪：

如果您只需要匹配开始和介绍 H1 标记，下面的非贪婪表达式只匹配 <H1>。

/<.\*?>/，非贪婪只会匹配：<H1>

import re

pattern = *r'<.\*?>'*

p = re.compile(pattern)

match = p.match(*'<H1>Chapter 1 - 介绍正则表达式</H1>'*)

if match:

print match.group() # <H1>

### 定位符

定位符能够将正则表达式固定到行首或行尾。



注意：

* 不能将限定符与定位点一起使用。不允许诸如^\*之类的表达式。
* 若要匹配一行文本开始处的文本，在正则表达式的开始使用^字符。
* 若要匹配一行文本的结束处的文本，在正则表达式的结束处使用$字符。

例如：

/^Chapter [1-9][0-9]{0,1}$/

字边界是单词和空格之间的位置。非字边界是任何其他位置。下面的表达式匹配单词Chapter的开头三个字符，因为这三个字符出现在字边界的后面：

/\bCha/

\b字符的位置是非常重要的。如果它位于匹配字符串的开始，它在单词开始处查找匹配项。如果位于字符串的结尾，他在单词的结尾处查找匹配项。

\ter/b\

下面的表达式匹配Chapter中的字符串apt，但不匹配aptitude中的字符串apt：

/\Bapt/

1、\b只能匹配字母、数字、汉字、下划线

2、\b就近匹配，比如\bAB 匹配A，AB\b 匹配B；但如果只写一个\b或者在两个字母、数字、汉字、下划线之间有\b时就在所有字符或者两个字母、数字、汉字、下划线之间所有字符去逐个匹配

>>> p1 = r'\bchina\b'

>>> m = re.match(p1, 'china come!') # 匹配字词边界

>>> m

<\_sre.SRE\_Match object at 0x00000000030F84A8>

>>> m = re.match(p1, 'chinas come!') # chinas不符合a\b字词边界，不能匹配。

>>> print m

None>>>

>>> m = re.match(r'\bchina\B', 'chinas come!') # a\B是非字词边界，可以匹配。

>>> m

<\_sre.SRE\_Match object at 0x00000000030F84A8>

>>> m = re.search(p1, 'My china come!') # 查找匹配字词边界

>>> m

<\_sre.SRE\_Match object at 0x00000000030F8440>

### 选择

正则表达式中的小括号的作用是对字符进行分组，并保存匹配的文本。与位于小括号之间的模式匹配的内容都会被捕获。

import re

pattern = *r'(\d)(\d)'*

p = re.compile(pattern)

match = p.match(*'92/01/2009Description'*)

if match:

print match.group() # 92

print match.groups() # ('9', '2')

用圆括号将所有选择项括起来，相邻的选择项之间用|分隔。但圆括号会有一个副作用，是相关的匹配会被缓存，此时可用?:放在第一个选项前来消除这种副作用。

import re

pattern = *r'(\d)(?:\d)'*

p = re.compile(pattern)

match = p.match(*'92/01/2009Description'*)

if match:

print match.group() # 92

print match.groups() # ('9', )不捕获

其中 ?: 是非捕获元之一，还有两个非捕获元是 ?= 和 ?!，这两个还有更多的含义，前者为正向预查，在任何开始匹配圆括号内的正则表达式模式的位置来匹配搜索字符串，后者为负向预查，在任何开始不匹配该正则表达式模式的位置来匹配搜索字符串。

import re

pattern = *r'(\d)(?=\d)'*

p = re.compile(pattern)

match = p.match(*'92/01/2009Description'*)

if match:

print match.group() # 9

print match.groups() # ('9', )不捕获

### 组

(?P<name>) ?P代表的是Python的语法扩展，<name>代表自定义命名，...代表命名的语法规则。比如：(?P<num>\d+)，命名之后可以在后面调用。比如：re.findall(r'(?P<num>\d+)\[a-z]+(?P=num)',s) # 找出中间夹有字母的数字。

**?P<name>为每个子表达式定义名称：**

import re

pattern = *r'(?P<first>\d)(?P<second>\d)'*

p = re.compile(pattern)

print p.pattern # (?P<first>\d)(?P<second>\d)

print p.flags # 0

print p.groups # 2

print p.groupindex # {'second': 2, 'first': 1}

match = p.match(*'92/01/2009Description'*)

if match:

print match.group() # 92

print match.groups() # ('9', '2')

match = p.findall('92/01/2009Description')

print match # [('9', '2'), ('0', '1'), ('2', '0'), ('0', '9')]

### 反向引用

对一个正则表达式模式或部分模式两边添加圆括号将导致相关匹配存储到一个临时缓冲区中，所捕获的每个子匹配都按照在正则表达式模式中从左到右出现的顺序存储。缓冲区编号从 1 开始，最多可存储 99 个捕获的子表达式。每个缓冲区都可以使用 \n访问，其中 n 为一个标识特定缓冲区的一位或两位十进制数。

可以使用非捕获元字符 ?:、?= 或 ?! 来重写捕获，忽略对相关匹配的保存。

例如：

var str = "Is is the cost of of gasoline going up up";

var patt1 = /\b([a-z]+) \1\b/;

document.write(str.match(patt1));

捕获的表达式，正如 [a-z]+ 指定的，包括一个或多个字母。正则表达式的第二部分是对以前捕获的子匹配项的引用，即，单词的第二个匹配项正好由括号表达式匹配。\1 指定第一个子匹配项。

字边界元字符确保只检测整个单词。否则，诸如 "is issued" 或 "this is" 之类的词组将不能正确地被此表达式识别。

正则表达式后面的全局标记 g 指定将该表达式应用到输入字符串中能够查找到的尽可能多的匹配。

表达式的结尾处的不区分大小写 i 标记指定不区分大小写。

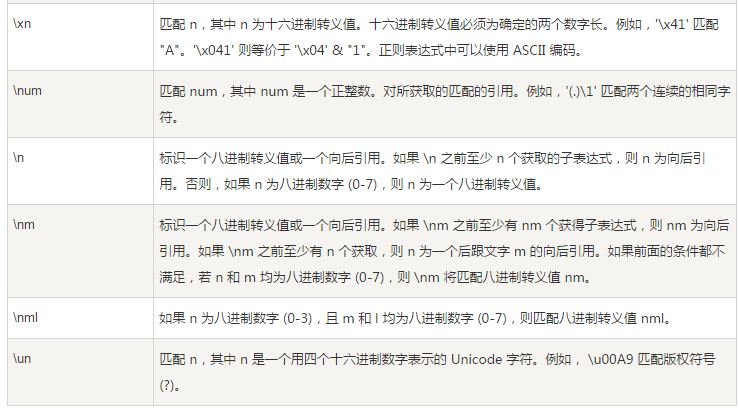
多行标记指定换行符的两边可能出现潜在的匹配。

## 元字符









## 运算符优先级



## 匹配规则

### 基本模式匹配

模式是正则表达式最基本的元素，它们是一组描述字符串特征的字符。

### 字符簇

[a-z] //匹配所有的小写字母

[A-Z] //匹配所有的大写字母

[a-zA-Z] //匹配所有的字母

[0-9] //匹配所有的数字

[0-9\.\-] //匹配所有的数字，句号和减号

[ \f\r\t\n] //匹配所有的白字符

[^a-z] //除了小写字母以外的所有字符

[^\\\/\^] //除了(\)(/)(^)之外的所有字符

[^\"\'] //除了双引号(")和单引号(')之外的所有字符



### 确定重复出现



^[a-zA-Z0-9\_]{1,}$ // 所有包含一个以上的字母、数字或下划线的字符串

^[1-9][0-9]{0,}$ // 所有的正整数

^\-{0,1}[0-9]{1,}$ // 所有的整数

^[-]?[0-9]+\.?[0-9]+$ // 所有的浮点数

^[a-zA-Z0-9\_]+$ // 所有包含一个以上的字母、数字或下划线的字符串

^[1-9][0-9]\*$ // 所有的正整数

^\-?[0-9]+$ // 所有的整数

^\-?[0-9]+\.?[0-9]\*$ // 所有的浮点数

\number Matches the contents of the group of the same number.

\A Matches only at the start of the string.

\Z Matches only at the end of the string.

\b Matches the empty string, but only at the start or end of a word.

\B Matches the empty string, but not at the start or end of a word.

\d Matches any decimal digit; equivalent to the set [0-9].

\D Matches any non-digit character; equivalent to the set [^0-9].

\s Matches any whitespace character; equivalent to [ \t\n\r\f\v].

\S Matches any non-whitespace character; equiv. to [^ \t\n\r\f\v].

\w Matches any alphanumeric character; equivalent to [a-zA-Z0-9\_].

With LOCALE, it will match the set [0-9\_] plus characters defined

as letters for the current locale.

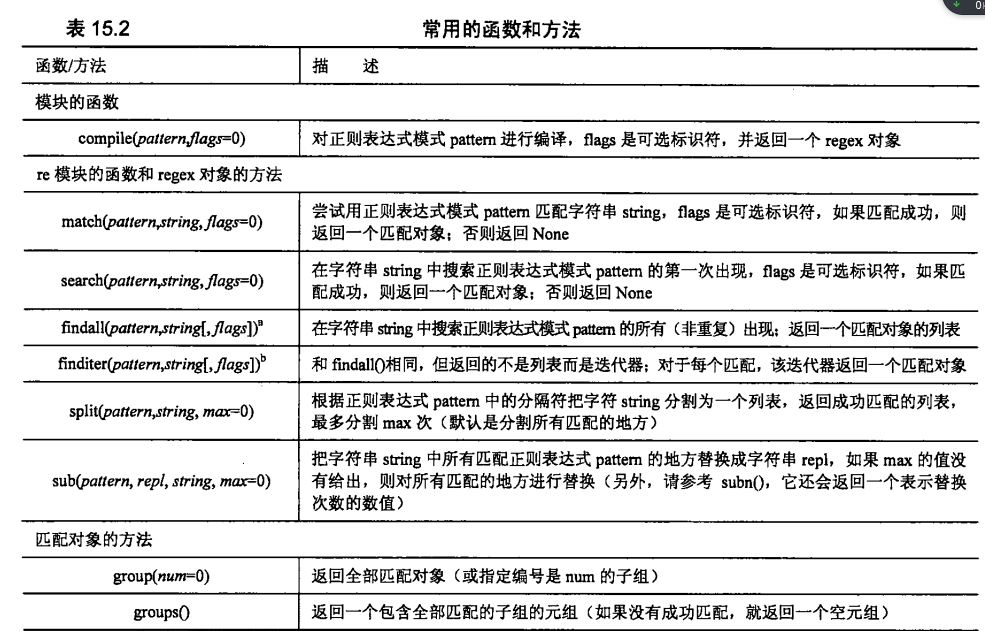
\W Matches the complement of \w.

\\ Matches a literal backslash.

## 示例



## 常用函数和方法



### Pattern

**re.compile(strPattern[, flag]):**

这个方法是Pattern类的工厂方法，**用于将字符串形式的正则表达式编译为Pattern对象**。 第二个参数flag是匹配模式，取值可以使用按位或运算符'|'表示同时生效，比如re.I | re.M。另外，你也可以在regex字符串中指定模式，比如re.compile('pattern', re.I | re.M)与re.compile('(?im)pattern')是等价的。   
可选值有：

* re.**I**(re.IGNORECASE): 忽略大小写（括号内是完整写法，下同）
* **M**(MULTILINE): 多行模式，改变'^'和'$'的行为（参见上图）
* **S**(DOTALL): 点任意匹配模式，改变'.'的行为
* **L**(LOCALE): 使预定字符类 \w \W \b \B \s \S 取决于当前区域设定
* **U**(UNICODE): 使预定字符类 \w \W \b \B \s \S \d \D 取决于unicode定义的字符属性
* **X**(VERBOSE): 详细模式。这个模式下正则表达式可以是多行，忽略空白字符，并可以加入注释。以下两个正则表达式是等价的：

a = re.compile(r"""\d +  # the integral part

                   \.    # the decimal point

                   \d \*  # some fractional digits""", re.X)

>>> m = a.match('3.45')

>>> m.group()

'3.45'

>>>

Pattern对象是一个编译好的正则表达式，通过Pattern提供的一系列方法可以对文本进行匹配查找。

class SRE\_Pattern(\_\_builtin\_\_.object)

| Compiled regular expression objects

|

| Methods defined here:

|

| \_\_copy\_\_(...)

|

| \_\_deepcopy\_\_(...)

|

| findall(...)

| findall(string[, pos[, endpos]]) --> list.

| Return a list of all non-overlapping matches of pattern in string.

|

| finditer(...)

| finditer(string[, pos[, endpos]]) --> iterator.

| Return an iterator over all non-overlapping matches for the

| RE pattern in string. For each match, the iterator returns a

| match object.

|

| match(...)

| match(string[, pos[, endpos]]) --> match object or None.

| Matches zero or more characters at the beginning of the string

|

| scanner(...)

|

| search(...)

| search(string[, pos[, endpos]]) --> match object or None.

| Scan through string looking for a match, and return a corresponding

| match object instance. Return None if no position in the string matches.

|

| split(...)

| split(string[, maxsplit = 0]) --> list.

| Split string by the occurrences of pattern.

|

| sub(...)

| sub(repl, string[, count = 0]) --> newstring

| Return the string obtained by replacing the leftmost non-overlapping

| occurrences of pattern in string by the replacement repl.

|

| subn(...)

| subn(repl, string[, count = 0]) --> (newstring, number of subs)

| Return the tuple (new\_string, number\_of\_subs\_made) found by replacing

| the leftmost non-overlapping occurrences of pattern with the

| replacement repl.

|

| ----------------------------------------------------------------------

| Data descriptors defined here:

|

| flags

|

| groupindex

|

| groups

|

| pattern

Pattern不能直接实例化，必须使用re.compile()进行构造。

Pattern提供了几个可读属性用于获取表达式的相关信息：

1.pattern: 编译时用的表达式字符串。

2.flags: 编译时用的匹配模式。数字形式。

3.groups: 表达式中分组的数量。

4.groupindex: 以表达式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。

import re

p = re.compile(r'(\w+) (\w+)(?P<sign>.\*)', re.DOTALL)

print "p.pattern:", p.pattern

print "p.flags:", p.flags

print "p.groups:", p.groups

print "p.groupindex:", p.groupindex

### output ###

# p.pattern: (\w+) (\w+)(?P<sign>.\*)

# p.flags: 16

# p.groups: 3

# p.groupindex: {'sign': 3}

### re模块Pattern实例方法

* **match(string[, pos[, endpos]]) | re.match(pattern, string[, flags]):**这个方法将从string的pos下标处起尝试匹配pattern；如果pattern结束时仍可匹配，则返回一个Match对象；如果匹配过程中pattern无法匹配，或者匹配未结束就已到达endpos，则返回None。  
  pos和endpos的默认值分别为0和len(string)；re.match()无法指定这两个参数，参数flags用于编译pattern时指定匹配模式。   
  注意：这个方法并不是完全匹配。当pattern结束时若string还有剩余字符，仍然视为成功。想要完全匹配，可以在表达式末尾加上边界匹配符'$'。



* **search(string[, pos[, endpos]]) | re.search(pattern, string[, flags]):**这个方法用于查找字符串中可以匹配成功的子串。从string的pos下标处起尝试匹配pattern，如果pattern结束时仍可匹配，则返回一个Match对象；若无法匹配，则将pos加1后重新尝试匹配；直到pos=endpos时仍无法匹配则返回None。   
  pos和endpos的默认值分别为0和len(string))；re.search()无法指定这两个参数，参数flags用于编译pattern时指定匹配模式。



**re.match与re.search的区别**

re.match只匹配字符串的开始，如果字符串开始不符合正则表达式，则匹配失败，函数返回None；而re.search匹配整个字符串，直到找到一个匹配。

* **split(string[, maxsplit]) | re.split(pattern, string[, maxsplit]):**按照能够匹配的子串将string分割后返回列表。maxsplit用于指定最大分割次数，不指定将全部分割。
* **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子串。

正则表达式仅有一个子组时，findall()返回子组匹配的字符串组成的列表；如果表达式有多个子组，返回的结果是一个元组的列表，元组中每个元素都是一个子组的匹配内容，像这样的元组（每一个成功匹配对应一个元组）构成了返回列表中的元素。

1. **finditer(string[, pos[, endpos]]) | re.finditer(pattern, string[, flags]):**搜索string，返回一个顺序访问每一个匹配结果（Match对象）的迭代器。

import re

p = re.compile(r'\d+')

for m in p.finditer('one1two2three3four4'):

    print m.group(),

### output ###

# 1 2 3 4

**6.sub(repl, string[, count]) | re.sub(pattern, repl, string[, count]):**使用repl替换string中每一个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是一个方法时，这个方法应当只接受一个参数（Match对象），并返回一个字符串用于替换（返回的字符串中不能再引用分组）。   
count用于指定最多替换次数，不指定时全部替换。

7.**subn(repl, string[, count]) |re.sub(pattern, repl, string[, count]):**返回 (sub(repl, string[, count]), 替换次数)。

## 常见的正则表达式

### IP地址

正则表达式：

((2[0-4]\d|25[0-5]|[01]?\d\d?)\.){3}(2[0-4]\d|25[0-5]|[01]?\d\d?)

红色块代表：第一个字符是2，第二个字符是0到4，第三个字符是任意一位数字。表示200~249。

绿色块代表：第一个字符是2，第二个字符是5，第三个字符是0到5。表示250~255。

蓝色块代表：第一个字符是0，或者1，或者可以没有这个字符，第二个字符是任意一位数字，第三个字符是任意一位数字，可以没有这个字符。表示1~199，可以有前导零。

“|”代表“或者”的意思，只要满足三块中的任意一块的意思。

前后的“(”和“)”代表这是一组

((2[0-4]\d|25[0-5]|[01]?\d\d?)\.){3}(2[0-4]\d|25[0-5]|[01]?\d\d?)

{3}代表重复三次。比如“255.255.255.”。

((2[0-4]\d|25[0-5]|[01]?\d\d?)\.){3}(2[0-4]\d|25[0-5]|[01]?\d\d?)

最后这一组的意思同上，就是在“.”后面加上相同的检测。

### 校验中文

china\_pattern = *u'([\u4e00-\u9fa5])+'*

### Email地址

[r'^[a-zA-Z0-9\_-]+@[a-zA-Z0-9\_-]+(\.[a-zA-Z0-9\_-]+)+$'](mailto:r'^[a-zA-Z0-9_-]+@[a-zA-Z0-9_-]+(/.[a-zA-Z0-9_-]+)+$')

### URL地址

r'(https?|ftp|file)://[-A-Za-z0-9+&@#/%?=~\_|!:,.;]+[-A-Za-z0-9+&@#/%=~\_|]'