java.util.regex是一个用正则表达式定制的模式来对字符串进行匹配的包.

包括两个类：Pattern和Matcher

一、捕获组的概念

捕获组可以通过从左到右计算其开括号来编号，编号从1开始。例如，在表达式 ((A)(B(C)))中，存在四个这样的组：

1 ((A)(B(C)))

2 (A)

3 (B(C))

4 (C)

组零始终代表整个表达式。以 (?) 开头的组是非捕获组，它不捕获文本，也不针对组合计进行计数。

**非打印字符**：

\f 匹配一个换页符

\n 匹配一个换行符

\r 匹配一个回车符

\s 匹配任何空白字符，包括空格、制表符、换页符, 等价于[\f\n\r\t\v]

\S 匹配任何非空白字符

\t 匹配一个制表符

\v 匹配一个垂直制符

**括号的使用**：

( ) : 标记一个子表达式

[ ] : 限定字符集的范围，如[A-Za-z0-9+﹢一二三四五六七八九十.、]

{ } : 表示前面的字符或表达式出现的次数，如{n}匹配n次，{n,}至少匹配n次，｛n,m｝最少匹配n次且最多匹配m次(逗号和两个数之间不能有空格).

**次数限定符匹配**：

. 匹配除换行符(\n \r)之外的任何单个字符

\* 零个或多个

+ 一个或多个

？ 零个或一个

{n} 匹配n次

{n,} 至少匹配n次

{n,m} 最少匹配n次最多匹配m次(逗号与两个数之间不能有空格)

**贪婪匹配**：

限定符\*、+、?、{n}、{n,}、{n,m}都是贪婪的，因此会尽可能多的匹配字符，在它们后面加上一个?可以实现最小匹配，如对于：

<H1>Chapter 1 - 介绍正则表达式</H1>

正则表达式<.\*>会匹配上述所有字符，而<.\*?>则只匹配<H1>和</H1>

---------------------------------------------------------------------------------------------------------------------------------

？的两个作用：

1、作为限定符，匹配前面的字符0次或1次

2、作为非贪婪匹配符，当?跟在任何一个其他限制符 (\*, +, ?, {n}, {n,}, {n,m}) 后面时，匹配模式是非贪婪的。非贪婪模式尽可能少的匹配所搜索的字符串，而默认的贪婪模式则尽可能多的匹配所搜索的字符串。例如，对于字符串 "oooo"，'o+?' 将匹配单个 "o"，而 'o+' 将匹配所有 'o'。

---------------------------------------------------------------------------------------------------------------------------------

**定位符**：

^ 匹配字符串开始位置

$ 匹配字符串结尾位置

\b 匹配一个字边界，字边界是单词和空格之间的位置

\B 匹配非字边界，非字边界是任何其他位置

不能将次数限定符与定位符一起使用。由于在紧靠换行或者字边界的前面或后面不能有一个以上位置，因此不允许诸如 ^\* 之类的表达式.

对于单词Chapter，有：

/\bCha/ 匹配开头的三个字符

/ter\b/ 匹配结尾的三个字符

/\Bapt/ 匹配中间的apt，但不能匹配aptitude开头的apt

**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。

一个捕获组在匹配成功之前，它的内容是不确定的，一旦匹配成功，它的内容就确定了，反向引用的内容也就确定了。反向引用匹配的内容与对应编号的捕获组匹配内容相同.

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。

反向引用的最简单的、最有用的应用之一，是提供查找文本中两个相同的相邻单词的匹配项的能力，如对于字符串：

Is is the cost of of gasoline going up up

表达式\b([a-z]+) \1\b （中间有空格）将匹配：

of of、up up

**非捕获组**：

(?:pattern)

匹配 pattern但不捕获匹配结果，也就是说这是一个非获取匹配，不进行存储供以后使用。正常情况(pattern)中的pattern会被作为新增的一个组序号输出，比如(A)(B)，A的序号1, B的序号2如果(?:A)(B)，A将没有序号不输出, B的序号为1，如:

industries ies

正则表达式industr(y|ies) \1将匹配industries ies，但industr(?:y|ies) \1将不匹配上述字符串，因为没有\1这个捕获组.

(?=pattern)：正向肯定预查（look ahead positive assert）

在任何匹配pattern的字符串开始处匹配查找字符串，可以用来筛选(?=pattern)之前的字符是否符合标准，如对于：

Windows2000

表达式”Windows(?=95|98|NT|2000)”能匹配其中的Windows, 但却不能匹配"Windows3.1"中的"Windows"

(?!pattern): 正向否定预查(negative assert)

在任何不匹配pattern的字符串开始处匹配查找字符串,可以用来筛选(?=pattern)之前的字符是否符合标准, 如对于：  
 Windows3.1

表达式”Windows(?!95|98|NT|2000)”能匹配其中的Windows，但却不能匹配Windows2000中的”Windows”

(?<=pattern):反向(look behind)肯定预查

与正向肯定预查类似，只是方向相反。例如，"(?<=95|98|NT|2000)Windows"能匹配"2000Windows"中的"Windows"，但不能匹配"3.1Windows"中的"Windows"。

(?<!pattern):反向否定预查

与正向否定预查类似，只是方向相反。例如"(?<!95|98|NT|2000)Windows"能匹配"3.1Windows"中的"Windows"，但不能匹配"2000Windows"中的"Windows"。

其他重要匹配：

\d 匹配一个数字字符，等价于[0-9]

\D 匹配一个非数字字符，等价于[^0-9]

\w 匹配字母、数字、下划线，等价于[A-Za-z0-9\_]

\W 相当于[^A-Za-z0-9\_]

/ 在某些语言中作为正则的定界符, 匹配/需要使用\/

一些例外的匹配规则：

在中括号表达式中匹配-:

1、用反斜杠将它转义：[\-]

2、将连字符放在中括号列表的开始或结尾[a-z-]、[-a-z]

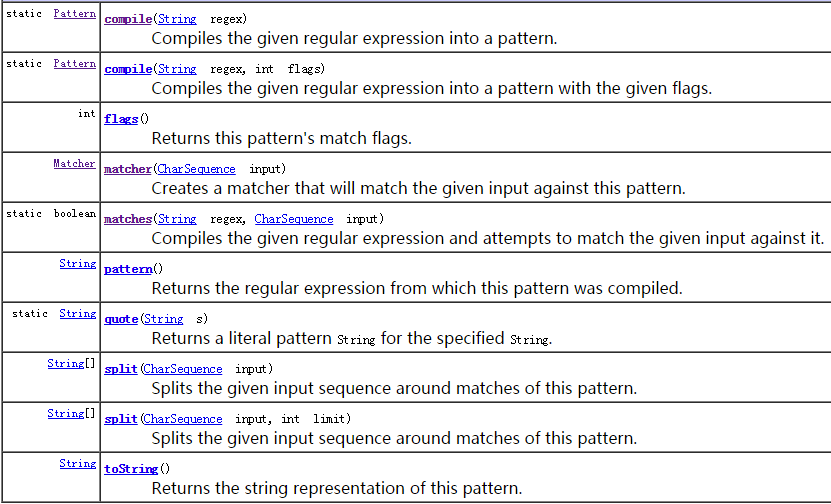
3、创建一个范围，开始字符值小于连字符，而结束字符值等于或大于连字符

[!--]、[!-~]

. 在中括号表达式时, 如[.]只会匹配.字符，等价于[\.]

二、详解Pattern类和Matcher类

java正则表达式通过java.util.regex包下的Pattern类与Matcher类实现。



Pattern类用于创建一个正则表达式匹配模式，构造方法为private, 不可直接创建, 可以通过Pattern.complie(String regex) 静态方法创建一个正则表达式模式：

Pattern p = Pattern.compile("\\w+");

p.pattern(); //返回 \w+

pattern() 返回正则表达式的字符串形式，即返回Pattern.complile(String regex)的regex参数

关于Java转义符\:



1、string[] split(CharSequence input)

Pattern有一个split(CharSequence input)方法, 用指定的模式分隔字符串, 并返回一个String[], 估计String.split(String regex)就是通过Pattern.split(CharSequence input)来实现的.

// 以数字分隔字符串

Pattern p = Pattern.compile("\\d+");

String[] str=p.split("我的QQ是:456456我的电话是:0532214我的邮箱

是:aaa@aaa.com");

结果:

str[0]="我的QQ是:"

str[1]="我的电话是:"

str[2]="我的邮箱是:aaa@aaa.com"

1. static boolean Pattern.matches(String regex, CharSequence input)

一个静态方法, 检测字符串是否和正则表达式匹配, 该方法需要匹配全部字符串.

Pattern.matches("\\d+","2223"); //返回true

Pattern.matches("\\d+","2223aa"); //返回false,需要匹配到所有字符串才能返回

true, 这里aa不能匹配

3、Matcher matcher(CharSequence input)

返回一个Matcher对象.

Matcher类的构造方法也是私有的, 只能通过Pattern对象的matcher

(CharSequence input)方法得到该类的实例。Pattern类只能做一些简单的匹配操作, 要想得到更强的正则匹配, 需要Pattern与Matcher一起合作. Matcher类提供了对正则表达式的分组支持, 以及对正则表达式的多次匹配支持.

Pattern p = Pattern.compile("\\d+"); // 将指定的正则表达式编译进Pattern对象

Matcher m = p.matcher("22bb23"); // 用指定的Pattern对象去匹配输入字符串，返回匹配器对象

m.pattern();//返回p，也就是返回创建该Matcher对象的Pattern对象

4、matches()/ lookingAt()/ find()

Matcher类提供三个匹配操作方法, 三个方法均返回boolean类型, 当匹配到时返回true.

matches()对整个字符串进行匹配, 只有整个字符串都匹配了才返回true

Pattern p = Pattern.compile("\\d+");

Matcher m = p.matcher("22bb23");

m.matches(); //返回false, 因为bb不能被\d+匹配,导致整个字符串匹配未成功.

Matcher m2 = p.matcher("2223");

m2.matches(); //返回true, 因为\d+匹配到了整个字符串

代码Pattern.matcher(String regex,CharSequence input)与下面这段代码等价

Pattern.compile(regex).matcher(input).matches()

lookingAt()对前面的字符串进行匹配, 只有匹配到的字符串在最前面才返回true

Pattern p = Pattern.compile("\\d+");

Matcher m = p.matcher("22bb23");

m.lookingAt(); //返回true,因为\d+匹配到了前面的22

Matcher m2 = p.matcher("aa2223");

m2.lookingAt(); //返回false,因为\d+不能匹配前面的aa

find()对字符串进行匹配, 匹配到的字符串可以在任何位置.

Pattern p=Pattern.compile("\\d+");

Matcher m=p.matcher("22bb23");

m.find(); //返回true

Matcher m2=p.matcher("aa2223");

m2.find(); //返回true

Matcher m3=p.matcher("aa2223bb");

m3.find(); //返回true

Matcher m4=p.matcher("aabb");

m4.find(); //返回false

5、start()/ end()/ group()

当使用matches(), lookingAt(), find( )执行匹配操作后, 就可以利用以上三个方法得到更详细的信息.

start()：返回匹配到的子字符串在字符串中的索引位置.

end()：返回匹配到的子字符串最后一个字符在字符串中的索引位置+ 1

group()：返回匹配到的子字符串

Pattern p = Pattern.compile("\\d+");

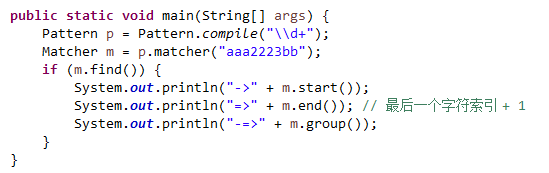
Matcher m = p.matcher("aaa2223bb");

m.find(); //匹配2223, 返回true

m.start(); //返回3

m.end(); //返回7,返回的是2223后的索引号

m.group(); //返回2223



结果：



Mathcer m2=m.matcher("2223bb");

m.lookingAt(); //匹配2223

m.start(); //返回0, 由于lookingAt()只能匹配前面的字符串,所以当使用lookingAt()

匹配时, start()方法总是返回0

m.end(); //返回4

m.group(); //返回2223

下面说说正则表达式的分组在java中是怎么使用的.

start(), end(), group()均有一个重载方法为：start(int i), end(int i), group(int i)专用于分组操作, Mathcer类还有一个groupCount()用于返回有多少组.

start(int i): 返回第i个分组匹配到的子字符串的索引位置

end(int i): 返回第i个分组匹配到的子字符串最后一个字符的索引位置 + 1

group(int i): 返回第i个分组匹配的子字符串

Pattern p=Pattern.compile("([a-z]+)(\\d+)");

Matcher m=p.matcher("aaa2223bb");

m.find(); //匹配aaa2223

m.groupCount(); //返回2,因为有2组

m.start(1); //返回0 返回第一组匹配到的子字符串在字符串中的索引号

m.start(2); //返回3

m.end(1); //返回3 返回第一组匹配到的子字符串的最后一个字符在字符串中的

索引位置.

m.end(2); //返回7

m.group(1); //返回aaa,返回第一组匹配到的子字符串

m.group(2); //返回2223,返回第二组匹配到的子字符串

现在使用一下稍微高级点的正则匹配操作, 例如有一段文本, 里面有很多数字, 而且这些数字是分开的, 现在要将文本中所有数字都取出来, 利用java的正则操作如下：

Pattern p=Pattern.compile("\\d+");

Matcher m=p.matcher("我的QQ是:456456 我的电话是:0532214 我的邮箱

是:aaa123@aaa.com");

while(m.find()) {

System.out.println(m.group());

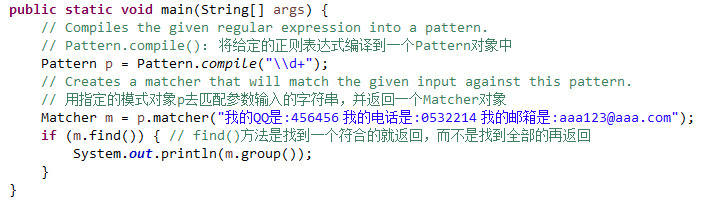
}

输出:

456456

0532214

123



结果：

IMG_256

如将以上while()循环替换成

while(m.find()) {

System.out.println(m.group());

System.out.print("start:"+m.start());

System.out.println(" end:"+m.end());

}

则输出:

456456

start:6 end:12

0532214

start:19 end:26

123

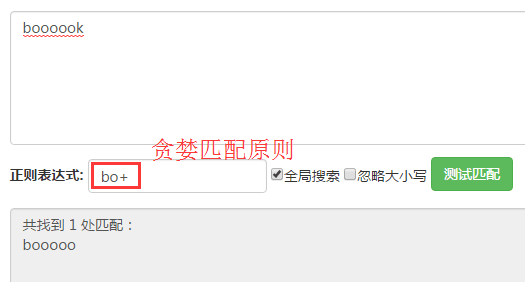
start:36 end:39

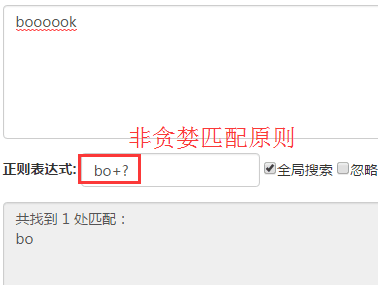
注意:只有当匹配操作成功, 才可以使用start(), end(), group()三个方法,否则会抛出java.lang.IllegalStateException, 也就是当matches(), lookingAt(), find()其中任意一个方法返回true时,才可以使用.

贪婪匹配与非贪婪匹配：

默认情况下，正则表达式使用最长匹配原则，即贪婪匹配原则；

当字符“？”紧跟在任何其他限定符之后时，匹配模式将变成最短匹配原则，即非贪婪匹配原则





子表达式反向引用：

