依赖库和方法简单介绍

Java的正则表达的相关类都在`import java.util.regex.*`里面，这里先介绍一些类和方法

`Pattern`和`String regex`

后面很多方法要求的是`String regex`参数，其会将传入的`regex`作为一个`pattern`生成一个正则匹配器

获得`Pattern`对象

和别的对象不同，`Pattern`没有公用的构造器，必须使用`static`的方法`.compile()`构造

比如如果想要构造一个匹配正整数的`Pattern`对象，就要写：`Pattern pattern =Pattern.compile("\+?[1-9][0-9]*");`，我们稍后关注正则表达式应该怎么写

匹配

借助`Pattern`对象

`Pattern`不能直接用于匹配，要调用`.matcher`然后`matches`

```
Pattern pattern =Pattern.compile("\\+?[1-9][0-9]*"); // 匹配正整数
String [] strs={"123","-034","0314","-231","+312","0+123"};
for(var str:strs){
    System.out.println(str+" : "+pattern.matcher(str).matches());
}
```

返回的是整个字符串是否满足`pattern`的规则，注意部分满足并不会被匹配

输入的也可以是`StringBuffer`等，只要实现了`CharSequence`就行（主要是`String`、`StringBuffer`和`StringBuilder`）

使用`Pattern.matches()`方法

还可以使用`Pattern.matches()`方法来进行匹配：

```
Pattern pattern =Pattern.compile("\\+?[1-9][0-9]*");
System.out.println(Pattern.matches(pattern.pattern(),"+102")); // pattern.pattern()返回pattern正则表达式的字符串
System.out.println(Pattern.matches("\\+?[1-9][0-9]*","+102"));
```

其中`pattern.pattern()`返回`pattern`存储的字符串

相信我，看到`matches`的实现的时你会和我一样懵逼

```
public static boolean matches(String regex, CharSequence input) { // 传入必须是字符串
    Pattern p = Pattern.compile(regex);
    Matcher m = p.matcher(input);
    return m.matches();
}
```

（搞得好麻烦啊

使用`String.matches()`

这样简单很多

```
Pattern pattern =Pattern.compile("\\+?[1-9][0-9]*");
System.out.println("+120".matches(pattern.pattern())); // String.matches()传入的也必须是字符串
```

可以用这个直接对某个字符串匹配

分割

使用`Pattern.split()`对字符串分割

也可以使用正则表达式分割文本：

```
Pattern patternright=Pattern.compile("[{},;\\s]+");
Pattern patternwrong=Pattern.compile("[{},;\\s]");
String str1="{192.168.1.1},{10.10.43.3}{}";
String str2="192.168.1.1 , {168.13.1.1}";
System.out.println(Arrays.toString(patternright.split(str1)));
System.out.println(Arrays.toString(patternwrong.split(str1)));
System.out.println(Arrays.toString(patternright.split(str2)));
System.out.println(Arrays.toString(patternwrong.split(str2)));
```

输出：

```
[, 192.168.1.1, 10.10.43.3]
[, 192.168.1.1, , , 10.10.43.3]
[192.168.1.1, 168.13.1.1]
[192.168.1.1, , , , 168.13.1.1]
```

这里需要注意两点：

1. 匹配字符串会作为分割符分割字符串，并且如果一开始就匹配了分割字符（比如`str1`的情况），那么就会在返回的数组一开始就有一个空字符串

   但是如果在最后匹配了分割字符，那么不会在返回数组的尾部多上空字符串

2. 如果想要匹配多个字符，需要在字符(集)后加上`+`

（可以看上面的例子了解更多）

直接使用`String.split()`分割

直接看代码

```
String patternStr="[^a-zA-Z0-9']+"; // 匹配除了a-zA-Z0-9'之外的所有字符
String buf="iplee is a cool man, but he doesn't do well in ACM.";
System.out.println(Arrays.toString(buf.split(patternStr)));
```

注意`String.split()`里面只接收`String`作为参数，不允许`Pattern`作为参数

配合流使用

`Scanner.skip()`

这个输入可以是`Pattern`也可以是`String`（输入后`String`还是会被`Pattern.compile()`成`Pattern`）

用来跳过一个满足输入的串，注意只能跳过一个！之后的内容会被正常处理

如果没能正常找到需要被跳过的串，那么就会抛出`java.util.NoSuchElementException`异常，捕捉异常之后就可以继续运行了

看下面的代码理解

```
Pattern pattern=Pattern.compile("[^a-zA-Z0-9']+"); // 跳过所有字母数字一撇外的字符
Scanner scNoException=new Scanner(" iplee is a cool man, but-he doesn't do well in
ACM.");
Scanner scException=new Scanner("iplee is a cool man, but-he doesn't do well in ACM.");
try{
    scNoException.skip(pattern);
}catch (java.util.NoSuchElementException e){
    System.out.println("Caught Exception 1");
    System.out.println(scNoException.nextLine());
}finally {
    System.out.println("str 1 : "+scNoException.nextLine());
}
try{
    scException.skip(pattern);
}catch (java.util.NoSuchElementException e){
    System.out.println("Caught Exception 2");
```

```
}finally {
    System.out.println("str 2 : "+scException.nextLine());
}
```

输出：

```
str 1 : iplee is a cool man, but-he doesn't do well in ACM.
Caught Exception 2
str 2 : iplee is a cool man, but-he doesn't do well in ACM.
```

注意首先第一个`Scanner`是正确跳过了空格的，其次第二个`Scanner`读取的时候发生了异常，最后捕捉异常之后第二个`Scanner`还是可以接着读的

查找和替换

`String`的替换相关方法

注意`String`有两个`replace`方法，但是和正则表达式无关，我们就不介绍了

和正则表达式有关的是`replaceAll`和`replaceFirst`方法，这两个方法的返回的都是处理过的串（原串不会变），并且返回的串里相对于原串，全部(`raplaceAll`)或者第一个(`raplaceFirst`)满足匹配的子串被替换

```
String str=" iplee dfkalfj iplee jkfdsa iplee ";
System.out.println(str.replaceAll("\\siplee\\s","snprin_fish"));
System.out.println(str.replaceFirst("\\siplee\\s","snprin_fish"));
```

输出：

```
snprin_fishdfkalfjsnprin_fishjkfdsasnprin_fish
snprin_fishdfkalfj iplee jkfdsa iplee
```

`String`的查找相关方法

并没有和正则表达式相关的！需要使用`Matcher`

`Matcher`

获取`Matcher`对象

和`Pattern`一样，`Matcher`并没有一个构造器，需要使用一个`Pattern`实例的`matcher`获得`Matcher`

```
Pattern pattern=Pattern.compile("[A-Z][a-z]+"); // 匹配英文字母
String str="Dom 10 21 31; Alice 21 4 0; Bob 231 43 453";
Matcher matcher=pattern.matcher(str);
```

三种关于匹配是否成功的方法

`Matcher`有三个关于匹配是否成功的方法，分别是

- `public boolean matches()`

  尝试对整个目标字符展开匹配检测，只有整个目标字符串完全匹配时才返回真值

- `public boolean find()`

  一个**连续匹配方法**，之所以叫连续匹配是因为它并不是一次性匹配完全部的串，而是一个个串地“查找”，并在查找完之后使用`Matcher`的`start`、`end`和`group`等方法来获取详细的信息

  常用的一种写法是：`while(matcher.find()){...}`

  `find`支持带`start`参数查找，此时会重置查找状态

- `public boolean lookingAt()`

  在开头匹配的方法，如果匹配成功就可以使用`start`和`end`之类的方法获取详细的信息

关于`group`，这是正则表达式的内容，我们现在只需要零组的内容，一般情况下零组（即`.group()`方法）就是完整的匹配内容

我们分别介绍

`matches`的使用

这个前面介绍过，如果匹配成功就返回`True`

可以通过`start`、`end`和`group`获得匹配的开始结束和内容（就是全串的信息）

`find`的使用

我们先看一个例子

```
Pattern pattern=Pattern.compile("[A-Z][a-z]+"); //
String str="Dom 10 21 31; Alice 21 4 0; Bob 231 43 453";
Matcher matcher=pattern.matcher(str);
while(matcher.find()){
    System.out.println("["+matcher.start()+" "+matcher.end()+") : "+matcher.group());
}
```

输出：

```
[0 3) : Dom
[14 19) : Alice
[28 31) : Bob
```

这就是`find`的作用，和迭代器一样，是向前一路“扫过去”并获得信息的方法

另外，`find`有一个带参数`start`的重载，使用了之后会**重新**从传入的`start`下标开始匹配

`lookingAt`的使用

和`find`不同，`lookingAt`不会自动“滑动”，反而会一直关注最前面的内容是不是匹配

```
Pattern pattern=Pattern.compile("[A-Z][a-z]+"); //
String str="Dom 10 21 31; Alice 21 4 0; Bob 231 43 453";
Matcher matcher=pattern.matcher(str);
System.out.println(matcher.lookingAt());
matcher.reset("chris 10 31 44"); // 换一个string
System.out.println(matcher.lookingAt());
```

输出：

```
true
false
```

其他方法

`reset`

重设状态，如果没有参数就将扫描位置放回当前串最前面，如果有`String`参数就把`Matcher`的字符串换成传入的`String`

比如`find`之后`reset`再`find`就会从最前面开始

`usePattern`

参数为`Pattern`，让`Matcher`使用新的`Pattern`工作

`public Matcher region(int start, int end)`

返回一个新的`Matcher`，使用原`Matcher`的$[start, end)$内容作为新`Matcher`的字符内容

`MatcherResult`

这是个接口，直接看源码：

```
public interface MatchResult {
    public int start();
    public int start(int group);
    public int end();
    public int end(int group);
    public String group();
    public String group(int group);
    public int groupCount();
}
```

`Matcher`实现了`MatcherResult`接口（听起来很滑稽）

正则表达式

参考网页：

[菜鸟教程-正则表达式]

注意在`Java`中正则表达式的转译必须写成`\\`，比如空白符是`\s`，在`Java`里需要写成`\\s`

为了方便讲述，后面`...`表示一些内容

语法

单个字符(集)

特殊字符集表示

内容	说明
`[ABC]`	对应`A`、`B`和`C`三个字符
`[^...]`	除了`...`外的所有内容
`[A-Z]`、`[a-z]`和`[0-9]`	所有大写字母、小写字母、数字
`.`	匹配`\n`和`\r`之外的所有内容
`\w`	对应字母、数字和下划线
`\s`	对应所有空白符，比如换行、空格
`\S`	对应所有非空白符
`^`和`$`	分别匹配输入字符串开始、结尾的位置
`\d`	匹配数字

需要被特别转译的字符

1. `()`表示自表达式，如果想要使用`(`、`)`本身，需要使用`\`转译

   `{}`、`[]`同理
2. `*`、`+`、`.`、`?`、`|`都有特殊用处，所以需要使用`\`转译
3. 反斜杠`\`本身表示转译，自然也需要转译

   需要注意的是，Java里面需要写`\\\\`转译一个`\`

限定符号

介绍

内容	说明
`*`	匹配前面的子表达式零次或多次
`+`	匹配前面的子表达式一次或多次
`?`	匹配前面的子表达式零次或一次
`{n}`	匹配前面的子表达式确定的`n`次
`{n,}`	至少匹配前面的子表达式`n`次
`{n,m}`	至少匹配前面的子表达式`n`次、最多`m`次

贪婪性

`+`和`*`默认都是贪婪的，也就是会尽可能多地匹配内容，我们可以在其后加上`?`使之变得不贪婪

比如我们想要匹配某个标签`<...>`，使用`<.*>`和`<.*?>`的效果如下：

```
Pattern patternGreed=Pattern.compile("<.*>");
Pattern patternNotGreed=Pattern.compile("<.*?>");
String str="<h1>head1</h1>  <h2>head2</h2>";
Matcher matcher=patternGreed.matcher(str);
System.out.println(patternGreed);
while(matcher.find()){
    System.out.println(matcher.group());
}
System.out.println();
System.out.println(patternNotGreed);
matcher=patternNotGreed.matcher(str);
while(matcher.find()){
    System.out.println(matcher.group());
}
```

输出：

```
<.*>
<h1>head1</h1>  <h2>head2</h2>

<.*?>
<h1>
</h1>
<h2>
</h2>
```

位置符

符号	说明
`^`	匹配输入字符串开始的位置 如果设置了RegExp对象的Multiline属性，`^`还会与`\n`或`\r`之后的位置匹配。
`$`	匹配输入字符串结束的位置 如果设置了RegExp对象的Multiline属性，`^`还会与`\n`或`\r`之前的位置匹配。
`\b`	匹配一个单词边界，即字母与符号间的位置
`\B`	非单词边界匹配，即前面不应该是空格

注意`^*`是非法的，实际上所有位置符后面都不应该跟限定符

在Java中，想要开启Multiline属性，需要在`Pattern.compile`传入一个`int`

下面给几个例子了解：

- 匹配章节行

```
Pattern pattern=Pattern.compile("^Chap\\.[1-9]\\d*$",Pattern.MULTILINE); // 这里的^和$不需要转译!
String str="Chap.1\n" +
        "chap1\n" +
        "Chap.2\n" +
        "chap2\n";
Matcher matcher=pattern.matcher(str);
while(matcher.find()){
    System.out.println(matcher.group());
}
/*输出:
Chap.1
Chap.2
*/
```

- 匹配`ed`后缀的内容

```
Pattern pattern=Pattern.compile("\\Bed\\b");
String str="I played LOL. I loved her. I think ledlight is great.";
Matcher matcher=pattern.matcher(str);
while(matcher.find()){
    System.out.println(matcher.end());
}
/*输出
8
21
*/
```

“或”

考虑校赛上黄哥出的题，就是要屏蔽脏话的

可能可以想到这样的程序：

```
Pattern pattern=Pattern.compile("(cao)|(fuck)|(ri)|(ckll)");
String str="fuckyou,caonima,rileguile,fucklly";
Matcher matcher=pattern.matcher(str);
while(matcher.find()){
    System.out.println("["+matcher.start()+" , "+matcher.end()+")"+" :
"+matcher.group());
}
```

但是我们看输出：

```
[0 , 4) : fuck
[8 , 11) : cao
[16 , 18) : ri
[26 , 30) : fuck
```

可以看出，`fucklly`并没有因为`ckll`的存在被变成`**`，也就是或匹配是直接向前的，不会将`fuckll`匹配成`**`

高级内容

我觉得暂时用不上的，包括分组内容和先行断言、后行断言

先行断言和后行断言

捕获和分组

一些常用的

需要匹配内容	正则表达式（非Java版）
QQ号	`[1-9]([0-9]{5,11})`
日期（简陋）	`\d{4}-\d{1,2}-\d{2}`
正整数	`\+?[1-9]\d*`
负整数	`-[1-9]\d*`
负浮点数	`-([1-9]\d*.\d*|0.\d*[1-9]\d*)`