java正则表达式学习笔记

1、字符缩略表示法
\a [\b] \e \f \n \r \t \0octal \x## \u#### \cchar
1)只有在字符组内部[],\b才代表退格字符，在其他场合，\b都代表单词分界符。
2)\cchar是区分大小写的，直接对后面字符的十进制编码进行异或操作。比如,\cA和\ca是不同的，\cA等于传统意义上的\x01,\ca则等价于\x21,匹配'!'。

2、对单词分界符元字符\b和\B来说，“单词字符”的规定不同于\w和\W。单词分界符能够识别Unicode字符，而\w和\W只能识别ASCII字符。

3、顺序环视结构中可以使用任意正则表达式，但是逆序环视中的子表达式只能匹配长度有限的文本。也就是说，？可以出现在逆序环视中，但是*和+则不行。

4、java.util.regex中编译选项
编译选项 (?mode) 描述
Pattern.UNIX_LINES d 更改点号和^的匹配
Pattern.DOTALL s 点号能匹配任何字符
Pattern.MULTILINE m 扩展^和$的匹配规定
Pattern.COMMENTS x 宽松排列和注释模式(在字符组内部也有效)
Pattern.CASE_INSENSITIVE i 对ASCII字符进行不区分大小写的匹配
Pattern.UNICODE_CASE u 对Unicode字符进行不区分大小写的匹配
Pattern.CANNON_EQ Unicode按等价模式匹配，也就是不同编码中的同样字符视为相等。
Pattern.LITERAL 将regex参数作为文字文本，也就是普通文本，而非正则表达式。
这些编译选项主要用于Pattern.compile(String regex, int flags)方法的flags参数。
而(?mode)则用于regex参数中，例如：Matcher mImg = Pattern.compile("(?id)<IMG\\s+(.*?)/?>").matcher(html);其中(?id)，i表示对ASCII字符进行不区分大小写的匹配和d表示更改点号和^的匹配。

5、java.util.regex使用正则表达式非常简单，功能由两个类，一个接口和一个unchecked exception组成。
java.util.regex.Pattern
java.util.regex.Matcher
java.util.regex.MatchResult
java.util.regex.PatternSyntaxException
Pattern对象就是编译好的正则表达式，可以应用于任意多个字符串，Matcher对象则对应单独的实例，表示将正则表达式应用到某个具体的目标字符串上。
MatchResult封装了成功匹配的数据，匹配数据可以在下一次匹配尝试之前从Matcher本身获得，也可以提取出来作为MatchResult保存。

如果匹配尝试所使用的正则表达式格式不正确，就会抛出PatternSyntaxException异常。
这是一个运行时异常，继承自java.lang.IllegalAgumentException
通常Pattern.compile()编译是最耗时间的，所以一般要把编译独立出来，编译一次重复使用。

6、java.util.regex.Matcher的用法
1)程序员能够对Matcher的进行设置和修改的是：
a)Matcher封装的Pattern对象,可以在创建Matcher后，通过Matcher.usePattern(newPattern)进行修改，Pattern对象可以通过pattern()方法获得。
usePattern(newPattern)这个方法会用给定的pattern对象替换与matcher关联的pattern对象。这个方法不会重置Matcher，所以能够在文本的当前位置开始使用不同的pattern。
b)要匹配的目标字符串可以通过Matcher.reset(String text)方法进行修改。需要注意的是调用了此方法之后，匹配范围等信息都将重置。
c)目标字符串的匹配范围可以通过Matcher.region(start, end)进行修改，默认匹配范围为整个字符串，设置了匹配的起始和结束偏移值之后，
程序可以通过Matcher.regionStart()和Matcher.regionEnd()获取前面的匹配范围设置信息。
d)anchoring bounds标志位。如果匹配范围不等于整个目标字符串，可以设定是否将匹配范围的边界设置为"文本起始位置"和“文本结束位置”，
这会影响文本行边界元字符(\A ^ $ \z \Z)。这个标志位默认为true，可以通过Matcher.useAnchoringBouds()进行修改，
通过Matcher.hasAnchoringBounds()方法查询标志位状态，注意Matcher.reset()方法不修改此标志位。
e)transparent bounds标志位。如果匹配范围不是整个目标字符串，而是一部分，那么如果此标志位设为true的话，则允许顺序环视、逆序环视
以及单词分界符超越匹配范围边界的设置，匹配目标字符串的其他部分，也就是可以稍微有越界行为。此标志位默认为false，可以通过useTransparentBounds()进行修改设置。
通过hasTransparentBounds()方法查询标志位状态。
下面的例子说明了transparent bounds设置为false的情况：

```
String regex="\\bcar\\b";
String text = "Madagascar is best seen by car or bike.";
Matcher m = Pattern.compile(regex).matcher(text);
m.region(7,text.length());
m.find();
System.out.println("Matches starting at character "+m.start());
```

结果是：也就是说尽管匹配范围的起止位置可能在某个单词内部，\b仍然能够匹配，也就是他看不到之前的之母。
matches starting at character 7
单词分界符的确匹配了匹配范围的起始位置，即Madagascar中的car，尽管此处根本不是单词的边界。
如果不设定transparent bounds标志位，单词分界符就“受骗”了。如果在find之前添加这

条语句：
m.useTransparentBounds(true);
结果就是：matches starting at character 27
因为边界现在是透明的，引擎可以感知到起始边界之前有个字母‘s’，所以\b在此处无法匹配。于是结果就成了上面的结果。

2)程序员只能够对Matcher的进行查询的是：
a)Matcher当前的pattern中包含的捕获型括号的数目可以通过groupCount()方法查询获取。
b)目标字符串中的match pointer或current location，用于支持"寻找下一个匹配"的操作。
c)目标字符串中的append pointer，在查找替换、复制未匹配的文本操作时使用。
d)表示到达字符串结尾的上一次匹配尝试是否成功的标志位，可以通过hitEnd()方法获得这个标志位的值。
e)match result,如果最近一次匹配尝试成功，java会将各种数据收集起来，合称为match result。包括匹配文本的范围(通过group()方法),
匹配文本在目标字符串中的起始和结束偏移值(通过start()和end()方法),以及每一组捕获型括号对应的信息(通过group(num)、start(num)和end(num)方法)。
3）Matcher appendReplacement(StringBuffer result,String replacement)
此方法首先会把目标字符中，匹配到的字符串之前的文本，拷贝到result中，然后是拷贝replacement字符串到result中。
4)public StringBuffer appendTail(StringBuffer sb),这个方法将目标字符串中剩下的文本附加到提供的StringBuffer中。
5)String quoteReplacement(String s),这个方法主要功能是把字符串s中的\和$进行了转义处理。

7、一个具体的例子：

1）Java的Pattern.matches()函数会自动在正则表达式两端添加^和$。

2）java的Matcher.group(1)是指匹配第一个“()”括号里的正则表达式，Matcher.group(2)匹配第二个括号的，以此类推。

3）(?:表示在算group()时忽略此括号。

4）如下代码的功能是匹配字符串，取出类似:

rule

<attibutes>

when

<conditions>

then

<actions>

end

或

query

<conditions>

end

的drools规则表达式，并对when，then等部分进行DSL翻译处理。

```
private static final String ruleOrQuery =
"^(?: " + // alternatives rule...end, query...end
"\\p{Blank}*(rule\\b.+?^\\s*when\\b)" + // 1: rule, name, attributes. when starts a line
"(.*?) " + // 2: condition
"^(\\s*then) " + // 3: then starts a line
"(.*?) " + // 4: consequence
"(^\\s*end.*?$) " + // 5: end starts a line
"|\\s*(query\\s+ " +
"(?:\"[^\"]+\"|'[^']+'|\\S+)" +
"(?:\\s+\\([^)]+)?) " + // 6: query, name, arguments
"(.*?) " + // 7: condition
"(^\\s*end.*?$) " + // 8: end starts a line
")";


private static final Pattern finder = Pattern.compile(ruleOrQuery,
Pattern.DOTALL | Pattern.MULTILINE | Pattern.COMMENTS);
private StringBuffer expandConstructions(final String drl) {

// parse and expand specific areas
final Matcher m = finder.matcher(drl);
final StringBuffer buf = new StringBuffer();
int drlPos = 0;
int linecount = 0;
while (m.find()) {
final StringBuilder expanded = new StringBuilder();

int newPos = m.start();
```

```
linecount += countNewlines(drl, drlPos, newPos);
drlPos = newPos;

String constr = m.group().trim();
if (constr.startsWith("rule")) {
//m.group(1)是指匹配第一个括号里的正则表达式。
String headerFragment = m.group(1);
expanded.append(headerFragment); // adding rule header and
// attributes
String lhsFragment = m.group(2);
//这里使用antlr when部分
expanded.append(this.expandLHS(lhsFragment, linecount
+ countNewlines(drl, drlPos, m.start(2)) + 1));
//这里使用antlr then部分
String thenFragment = m.group(3);
expanded.append(thenFragment); // adding "then" header
String rhsFragment = this.expandRHS(m.group(4), linecount
+ countNewlines(drl, drlPos, m.start(4)) + 1);
expanded.append(rhsFragment);
expanded.append(m.group(5)); // adding rule trailer

} else if (constr.startsWith("query")) {
String fragment = m.group(6);
expanded.append(fragment); // adding query header and attributes
String lhsFragment = this.expandLHS(m.group(7), linecount
+ countNewlines(drl, drlPos, m.start(7)) + 1);
expanded.append(lhsFragment);
expanded.append(m.group(8)); // adding query trailer

} else {
// strange behavior
this.addError(new ExpanderException(
"Unable to expand statement: " + constr, 0));
}
m.appendReplacement(buf,
Matcher.quoteReplacement(expanded.toString()));
}
m.appendTail(buf);
return buf;
}
```