**Groovy 正则表达式**

**Groovy正则表达式语法**

背景

项目使用Gradle作为自动化构建的工具, 闲暇之余对这个工具的使用方式以及其配置文件依赖的Groovy语法进行了巩固, 在学习Groovy语法的时候发现其中一个比较有意思的东西, 就是Groovy的正则表达式,于是本篇文章总结了一下Groovy中的正则表达式的特点以及Groovy正则表达式与Java正则表达式的区别:

Groovy正则表达式语法

Groovy是Java语言的一种扩展, 可以无缝的使用Java的JDK, 并且其自身还有SDK对Java进行了扩展. Groovy中的正则表达式本质上还是会使用到JDK中的java.lang.regex 包中的class, 其实这一部分, 个人认为可以看成一种"语法糖", 只不过更方便大家在Groovy中使用正则表达式

来看一个最简单的正则表达式

def reg1 = ~'he\*llo'

def reg2 = /he\*llo/

println "reg1 type is ${reg1.class}"

println "reg2 type is ${reg2.class}"

println "hello".matches(reg1)

println "hello".matches(reg2)

运行结果:

reg1 type is class java.util.regex.Pattern

reg2 type is class java.lang.String

true

true

上式中使用了~ + 字符串(以及双斜线分隔符模式)的方式定义了一个正则表达式

Groovy中支持使用~ 来定义正则表达式, 打印出来的reg 类型都为Pattern 类型而不是一个字符串, 需要注意的点是上述例子中的~ 和= 之间有一个空格, 因为Groovy中存在=~ 操作符号, 这个操作符为查询操作符, 使用在字符串之后, 要求接一个正则表达式, 返回的是一个java.util.regex.Matcher 对象. 还有一个操作符==~ 也比较容易混淆,这个操作符为匹配操作符, 后面跟一个正则表达式, 返回的类型为Boolean 类型. 这个操作符要求前面给定的字符串与后面的正则表达式完全匹配才可返回true 比如以下的列子

def val1 = "hello" =~ "he\*llo"

println val1.class

print val1.matches()

运行结果

class java.util.regex.Matcher

true

使用Groovy中的匹配操作符可以简化上述的操作

def val1 = "hello" ==~ "he\*llo"

println val1.class

print val1

运行结果:

class java.lang.Boolean

true

原字符问题

我们知道正则表达式中存在一些特殊的字符(比如\w 表示的是[a-zA-Z0-9])用于文本的匹配, 这些字符一般是以\ 开头, 所以这个地方涉及到了转义字符问题. 举个例子:

def val1 = "test value"

println 'value is ${val1}'

println "value is ${val1}"

运行结果:

value is ${val1}

value is test value

如果在构建正则表达式字符串的时候, 使用双引号表示字符串,就需要使用\\ 来表示单斜线,比如:

def reg1 = "hello \\w\*"

def reg2 = /hello \w\*/

println "hello world" ==~ reg1

println "hello world" ==~ reg2

运行结果为true 当然使用双斜线字符串的话就不需要额外的斜线进行转义. 我们知道groovy的单引号中的字符串是以原字符的形式存在的,即是字符串本身就是它显示的意思,尝试使用单引号原字符来进行正则匹配:

def reg1 = 'hello \w\*' // 更改为 'hello \\w\*' 则运行正确

println "hello world" ==~ reg1

但是最终却是一个error 使用单引号依然需要进行转义, 仔细想想,Groovy的单引号场景是参数解析的场景, 而此处是含有斜线的正则表达式字符的匹配问题, 两个问题应该不一样,因此无论是使用单引号还是双引号,遇到正则表达式的含有斜线的特殊字符都要进行转义.不想进行转义可以使用斜线取代(单)双引号.

Pattern 和 Matcher

在Groovy中正则表达式中相关联的依然是这两个Java类. 依然回归到Java中的这两个类. JDK1.8 中java.util.regex 包中最核心的就是这两个类, Pattern表示的即是正则表达式的"模式", 这是一个抽象的概念, 在编程过程中我们使用的是字符串表示正则, 它只是一种抽象的模式,实际上需要将字符串表示的抽象模式"编译"成这个类才能够正常工作, 当在Groovy中可以理解为使用~ 操作符将字符串编译为一个Pattern对象. 回顾这个类中的一些重要的概念和方法:

Pattern.matches 和 Pattern.matcher

Matcher matcher(charsequence input)

这个函数返回一个Matcher匹配器对象, 这个匹配器匹配给定的输入与模式

def reg = ~/^hello \w\*world$/

def str = "hello world"

def matcher = reg.matcher(str)

println matcher.class

输出的类型就是java.util.regex.Matcher 然而上述的Matcher对象在groovy中可以用=~ 操作符号一步完成

def matcher = "hello world"=~/^hello \w\*world$/

println matcher.class

static boolean matches(string regex, charsequence input)

这个函数编译给定的正则表达式并且尝试匹配给定的输入, 这个在Java中是一个静态的函数, 可以理解为一种快速判断字符串与给定的正则表达式模式是否匹配的工具, 同样在Groovy中也有简单的实现方式

println "hello world"==~/^hello \w\*world$/

运行结果为true 可以看到Groovy中使用两个操作符号=~ 和 ==~ 完成了Matcher匹配对象的构建, 以及快速验证给定字符串是否和给定正则表达式模式匹配的功能.

Matcher 中的capturing group概念

首先Matcher 的概念是解释Pattern. 我理解的是有时候我们使用正则表达式不仅仅是完成简单的验证字符串是否和模式匹配, 而是需要更加灵活和高级的操作(比如获取部分匹配成功的子字符串功能), 此时就需要这个Matcher对象. Java中需要调用Pattern中的matcher方法返回这个对象, 而groovy中只需要使用=~ 操作符号即可创建这样的对象.抽象的讲, 这个对象即是存储一个正则表达式模式与一个给定输入字符串的所有匹配相关的信息. capturing group 这个概念是针对正则表达式中的() 引入的, 正则表达式中的括号表示group，捕获组是从左往右计算其开始括号进行编号的(因为具有括号嵌套的情况, 括号层次越高那么它的组编号自然越小), 其中0表示整个表达式, 如下例子:

(A (BC))

group 0: (A(BC))

group 1: (A(BC))

group 2: (BC)

计算表达式的group就从左括号开始算遇到一个左括号group number就加1. 使用group可以用于捕获输入字符串与模式匹配上的部分对应group位置的子字符串.

def str = "hello wrold hello"

def reg = /((el)(l))o/

def matcher = str=~reg

def num = 0

while(matcher.find()){

println "the ${num} match sub sequenc"

num++

groupnum = matcher.groupCount()

println "group count ${matcher.groupCount()}"

println "group string ${matcher.group()}"

println "group 0 string ${matcher.group(0)}"

for(id in 1..groupnum){

println "group ${id} string ${matcher.group(id)}"

println "start index is ${matcher.start(id)} and end index is ${matcher.end(id)}"

}

}

运行结果:

the 0 match sub sequenc

group count 3

group string ello

group 0 string ello

group 1 string ell

start index is 1 and end index is 4

group 2 string el

start index is 1 and end index is 3

group 3 string l

start index is 3 and end index is 4

the 1 match sub sequenc

group count 3

group string ello

group 0 string ello

group 1 string ell

start index is 13 and end index is 16

group 2 string el

start index is 13 and end index is 15

group 3 string l

start index is 15 and end index is 16

有一个点比较重要就是groupCount 组数量是不会将group 0计算在内的,组的数量是和括号数量保持一致, 其次是matcher.group() 方法和matcher.group(0) 方法返回内容都一样, 都是和模式匹配的完成的子序列, 当传递参数时返回的就是相应编号的捕获组获取的子序列.当然可以使用start end等方法获取到匹配的字符串(或者捕获组匹配到的字符串)的偏移量(end的偏移量位置始终是最后一个字符的位置加1).

注意在groovy中由于GDK实现了getAt 方法那么其实可以通过索引的方式访问捕获组中的内容,如下例子:

def reg = ~/h(el)(lo)/

def str = 'hello world hello nihao'

def matcher = str=~reg

println "first matched substring"

println matcher[0]

println matcher[0][0]

println matcher[0][1]

println matcher[0][2]

println "second matched substring"

println matcher[0]

println matcher[0][0]

println matcher[0][1]

println matcher[0][2]

运行结果

first matched substring

[hello, el, lo]

hello

el

lo

second matched substring

[hello, el, lo]

hello

el

lo

可以看到通过索引访问捕获字符串的规律, 返回的matcher对象第一维度索引表示子字符串的索引, 返回值为一个ArrayList 包含的内容是全部的子字符串以及与捕获组编号对应的子字符串. 这些等价的操作即是matcher.group(index)

Matcher 重置

匹配器的重置涉及到两个方法find() 和reset 其中find 方法可以指定从哪一个位置重新开始寻找模式匹配的字符串.比如:

def reg = /el/

def str = "hello world hello"

def matcher = str=~reg

while(matcher.find()){

println matcher.group()

}

matcher.find(0) // 重置matcher 从头开始寻找匹配字符串

// 但此时第一个匹配的子字符串已经获取到了，下一次调用find则是查询下一个匹配字符串

println "reset the matcher"

while(matcher.find()){

println matcher.group()

}

结果:

el

el

reset the matcher

el

当然最好使用reset来完成这个过程:

def reg = /el/

def str = "hello world hello"

def matcher = str=~reg

while(matcher.find()){

println matcher.group()

}

matcher.reset()

println "reset the matcher"

while(matcher.find()){

println matcher.group()

}

输出结果:

el

el

reset the matcher

el

el

小结

以上就是Groovy中的正则表达式一些比较常见的知识点, 主要是Groovy中特有的操作符号来构建正则表达式使用过程中依赖的各种对象. 最本质的还是回归到java中正则表达式相关的两个核心类(Pattern和Matcher). Groovy有时候表现得更像是一种"语法糖", 以脚本的方式来完成Java的编程.后续遇到更多跟Groovy正则相关的内容会持续更新本文.