※ABeen※ 正则表达式

学习摘要

目录

一、 正则表达式有什么用

常有人会问，正则表达式有什么用？那么复杂，还要学，普通的字符串处理，复杂一点也够用了。实际上正则表达式能帮你节省时间、大量的时间，好多程序员在处理文本这个繁琐问题上花费“巨额”的时间，如果把这些时间省了，就会有更多时间做爱做的事。

正则表达式(Regular Expression)是强大，便捷、高效的文本处理工具。配合特定工具的支持，正则表达式能够添加、删除、分离、叠加、插入和修整各种类型的文本和数据。一旦掌握了它，就是知道它简直是工具中的无价之宝，个人感觉正则表达式就像一把瑞士军刀，经常让你方便的很，特别是在网络数据采集 HTML 处理方面。

目前许多工具都支持正则表达式（文本编辑器、文字处理软件、系统工具、数据库引擎等等），很多编程放言也对正则表达式做了很好的支持。例如：Java 、Jscript 、VB、JavaScript、C 、C++、C#、PHP 等。正则表达式能得到众多语言的支持是有原因的：

较低层次上来说：正则表达式描述的是一串文本的特征。我们也可用来验证用户的输入数据，或者检索大量的文本。

较高层次上来说：正则表达式容许我们掌控自己的数据－－控制这些数据。

常用元字符

元字符	名称	匹配对象
.	点号	单个任意字符
[…]	字符组	列出的任意字符
[^…]	排除型字符组	未列出的任意字符
^	脱字符	行的起始位置

$	美元符	行的结束位置
\<	反斜线－小于	单词的起如位置(需要环境支持)
\>	反斜线－大于	单词的结束位置(需要环境支持)
|	竖线	匹配任意分隔的表达式
(…)	括号	限制多选结构的范围，标注量词作用的元素，为反向引用“捕获”文本
\1,\2…	反向引用	匹配之前的第一、第二组括号内的子表达式匹配的文本

在字符组的内部，元字符的定义规则是不一样的。例号.点号在外部表示任意一下字符，但在字符组内只是表示一个.点号。

计数功能元字符		
？	0~1	可以出现，也可以只出现一次（单次可选）
*	0~N	可以重复出现，也可以不出现（任意次数均可）
+	1~N	可以重复出现，但至少要出现一次（至少一次）
{min, max}	Min ~ max	至少出现 mix 次，至多出现 max 次

环视元字符		
(?=……)	肯定顺序环视	子表达式能够匹配**右侧**文本
(?!……)	否定顺序环视	子表达式不能匹配**右侧**文本
(?<=……)	肯定逆序环视	子表达式能够匹配**左侧**文本
(?<!……)	否定逆序环视	子表达式产能匹配**左侧**文本

二、 分组、捕获、分组不捕获

普通的无特殊意义的括号通常有两种功能：分组和捕获。

捕获型括号的编号是按照括号出现的次序，从左到右计算的。如果提供反向引用，可以在表达式的后面用\1,\2 来引用匹配的文本。如：(a)(b)可以用\1 引用(a),\2 引用(b).

分组、捕获、分组不捕获

(?:……)	仅用于分组的括号
(?<Name>……)	命名捕获
(?>……)	固化分组
*** + ? {min, max}**	匹配优先量词
***? +? ?? {min, max}?**	忽略优先量词
***+ ++ ?+ {min, max}+**	占有优先量词

仅用于分组的括号（?:…）

仅用于分组的括号（?:…） 不能用来提取文本，只能用来规定多选结构或者量词的作用对象。如 (1|one)(?:and|or)(2|two) 匹配之后，\1 包含“1”或者“one”,\2q 包含“2”或者“two”。只用于分组的括号也叫**非捕获型括号**.

命名捕获 (?<Name>)

.Net 引擎能够能捕获内容命名。如果在正则表达式内部引用捕获的文，.Net 中使用 \k<Name>. 如 (?<Name>a)[a-z]+\k<Name> 来匹配 abeena.

固化分组 (?>…)

如果了解了正则引擎的匹配原理，就很容易理解固化分组。固化分组一旦括号内的子表达式匹配之后，匹配的内容就固定下来，在接下来的匹配过程中不会变化，除非整个固化分组的括号都被弃用，在外部回溯中重新应用。固化分组能够提高匹配效率，而且自己能对什么能匹配，什么不能匹配进行准确地控制。

匹配优先量词 * + ? {min, max}

匹配优先量词在优先匹配尽量多的字符。

忽略优先量词 *? +? ?? {min, max}?

忽略优先量词与匹配优先量词正好相反，匹配尽可能少的内容。

占有优先量词 *+ ++ ?+ {min, max}+

占有优先量词类似普通的匹配优先量词，不过他们一旦匹配某些内容，就不会“交还”。类似固化分组。

三、 正则表达式引擎的构造

正则引擎一般分为:DFA 引擎和传统 NFA 引擎. 一般支持忽略优量词就是传统 NFA 引擎,否则就是 DFA 或其他引擎.

顾名思义,既然是引擎肯定会有多种零件构成,要真正了解引擎的工作原理, 还就必须先了解其零件.

正则引擎的零件分为文字字符、量词、字符组、括号等等。

文字文本 （如: a、*、!、好）

如果一个正则表达式只包含纯文本字符，如 abeen 正则引擎会将其视为：一个 a,接着一个 b,接着一个 e,接着一个 e，接着一个 n. 这应该不难理解。呵呵

字符组[…]、点号.、Unicode 属性及其他

需要说明的一点就是无论字符组的长度是多少，它都只能匹配一个字符。

捕获型括号()

用于捕获文本的括号（而不是用于分组的括号）不会影响匹配的过程。

锚点

锚点可以分为：简单锚点和复杂锚点。

简单锚点只是检查目标字符中的特定位置(如^,$)，或者是比较两个相邻字符(如\<,\b)。

复杂锚点(环视) 能包含任意复杂的子表达式。

非“电动”的括号、反向引用和忽略优先量词

捕获括号、以用相应的反向引用和表示法（如$1 或\1）、忽略优先量词只对 NFA 引擎起作用，DFA 不支持。

四、 正则表达式的匹配原理原则

原则一：优先选择最左端(最靠开头)的匹配结果

根据这条规则，起始位置最靠左端的匹配结果总是优先于其他可能的匹配结果。这里没有规定优先匹配结果的长度，而只是规定，在所有可能的匹配结果中，优先选择开始位置靠左端的。

匹配过程：正则匹配先从需要查找的字符串的起始位置开始，尝试匹配整个表达式能匹配的所有样式文本，如果在当前位置测试了所有可能之后不能找到匹配结果，正则引擎就是启动传动装置向右移动一个位置，也就是从字符串的第二个字符之前的位置开始重新尝试。在找到匹配结果以前必须在所有位置重复此过程。只能在尝试了所有位置(从第一个字符直到最后一个字符)都不能匹配结果的情况下，才会报告“匹配失败”。

所以如果用 ee 来匹配 abeen 的话，从字符串开始第一轮尝试失败(因为 ee 不能匹配 ab)，第二轮尝试也失败(因为 ee 也不能匹配 be),直到第三软尝试能够成功，引擎会停止下来报告匹配 ee 成功.

如果不了解这个规则，有时候就不能理解匹配的结果。如用 abeen 来匹配：“this is 123abeen321 test by abeen”. 结果会是 123abeen321 中的 abeen 被匹配，而不是最后的 abeen。

原则二：标准的匹配量词*、+、？、{min, max}是匹配优先的

标准量词是匹配的，在匹配成功之前，进行尝试的次数是存在上限和下限的，匹配优先量词之所以得名，是因为他们总是(或者至少是尝试)匹配多于匹配成功下限的字符。也就是说，标准匹配量词的结果可能并非所有可能结果中最长的，但它们总是尝试匹配可能多的字符，直到匹配上限为止。

例如： [0-9]+为什么能匹配 ABeen 19820527 中的所有数字。1 匹配成功之后，实际上已经满足了成功的下限，但此表达式是匹配优先的，所以它不会停止在此处，而会继续下去。继续匹配 19820527,直到字符串结束。

过度的匹配优先

^.*(?<Test>[0-9][0-9])能够匹配一行字符串的最后两位数字，如果有的话将它们存于 Test 组中。来看一下匹配过程， .* 首先匹配整行，而[0-9][0-9]是必须匹配的，在尝试匹配行末的时候会失败，它样它会通知.*自己没法匹配了，为了大局着想，大哥你还是交出点来吧。于是.*很有大哥风范，以大局为重交出一个字符，如果不够还会继续交出字符。当然前提是大哥得先留下自己那份，也就是匹配成功的下限。

这样我们来看 ^.*(?<Test>[0-9][0-9]) 匹配 this is a test abeen 1982abeen 的过程。.*首先匹配整个字符串以后，第一个[0-9]的匹配要求.*释放一个字符 n,但是[0-9]还不能匹配，所以.*必须继续交还字符 e,如此循环只到交还 2 为止。但不幸的是第一个[0-9]匹配成功后，第二个[0-9]不能匹配成功，于是.*大哥再次以大局为重，又释放一个字符 8,这样整个表达式就匹配成功，结果是 this is a test abeen 1982，Test 内容为 82.

先来先服务

如果想用^.*(?<Test>[0-9]+)来匹配一行的最后整个数字，比如匹配 this is a test abeen 1982,结果 Test 只捕获了 2,这是为什么呢？

也许你的本意是想捕获 1982,但结果为什么是 2 呢！首先.*会捕获整个字符串，然后[0-9]+必须要捕获(注意是必须)一个数字，所以.*交还了一个 2 出来，接下来整个表达式中没有必须要匹配的元素了，所以.*不会再交还字符。结果 Test 只能捕获 2 了。

这就是“先来先服务”原则，匹配优先的结构只会在被迫的情况下交还字符。

五、 回溯

NFA 引擎最重要的性质就是，它会依次处理各个表达式或组合元素，遇到需要在两个可能成功的途经中进行选择的时候，它会选择其一，然后记住另一个选项，以备稍后可能的需要。一般需要做出选择的情形包括量词和多选结构。

真实世界中的面包屑

回溯就像在道路的每个分岔口留下一小堆面包屑。如果走了死路，就可以原路返回，直

到遇到面包屑标示的尚未尝试过的道路。如果还是死路，继续返回，找到下一堆面包屑，如些重复，直到找到出路，或者走完所有没有尝试过的道路。

不论选择那一种途经，如果它能成功匹配，而且余下的正则也匹配成功了，匹配即告完成。如果余下的正则匹配失败，引擎会回溯到之前做选择时记录的备用途经。这样引擎最终会尝试表达式的所有可能途经(如果没有匹配成功的话)。

回溯的两个要点

如果需要在“进行尝试”和“跳过尝试”之间选择，对于匹配优先量词，引擎会优先选择“进行尝试”，而对于忽略优先量词，会选择“跳过尝试”。

在需要回溯时，距离当前最近储存的选项就是当本地失败强制回溯时返回的。使用的原则是 LIFO(last in first out,后进先出)。