正则表达式学习笔记

LEE BEN
PKU

2019 年 3 月 21 日

序言

正则表达式在处理字符串程序中可以发挥很重要的作用。在文本搜索、合法性检验、网页爬取等方面应用广泛。突然想起学习正则表达式，其实是想利用起这一段宝贵的时光，学习一些新的知识。后续我会继续学习 Python、TensorFlow 和机器学习相关的知识，并且尽量使用 LaTeX 进行排版。总之这一段时光应该有不错的收获。

目录

1 正则表达式的定义 3

2 正则表达式用在何处 3

3 字符匹配表达式 3

3.1 正向字符匹配 3

3.2 反向字符匹配 4

4 数量匹配 5

4.1 固定长度或数量 5

4.2 不定长度或数量 5

4.2.1 贪婪数量匹配与懒惰数量匹配 6

5 位置匹配 6

5.1 元字符匹配位置 6

5.2 零宽断言 7

5.2.1 正向零宽断言 7

5.2.2 反向零宽断言 7

6 分组 7

6.1 分组后的引用 8

6.2 平衡组和递归引用 8

7 转义 9

8 程序中的正则表达式应用 10

8.1 C++ 中的 regex 应用 10

8.1.1 match_result 类 10

8.1.2 匹配 11

8.1.3 查找 11

8.1.4 逐一查找 11

8.1.5 分割 12

8.1.6 替换 12

8.2 Python 中的 regex 应用 12

8.2.1 re 的构造 13

8.2.2 匹配 13

8.2.3 查找 13

8.2.4 逐一查找 13

8.2.5 分割 13

8.2.6 修饰符的选择 14

1 正则表达式的定义

正则表达式描述了一类规则，用于匹配大量字符中的指定字符串。我们处理最多的数据类型就是字符和字符串，这在计算机内部也是最容易被识别和处理的。每个字符占用 1 字节 (8 位) 的内存，采用指定的编码方式进行呈现。无论是字母、数字、符号，还是由他们组成的长串，有时会按特定的规则进行排列，有着特别的意义。而想要从大量混杂的文本中找到我们所需的字符串们，就需要借助正则表达式来完成。

2 正则表达式用在何处

在微软的 microoffice 中，我们知道有通配符这种东西。但是通配符不能用于很多编程语言。正则表达式则不同，他可以成为各种编程语言的库文件，而匹配规则不变。在 C++、C#、Python 中都有各自的库文件和方法调用。具体方法的不同暂且不表，掌握正则表达式的基本规则后，才是最根本最重要的。

3 字符匹配表达式

3.1 正向字符匹配

所谓正向匹配，就是得到需要匹配的字符。

如果指定到某一个字符，比如 a，那么在正则表达式中直接写 a 即可。但是对于完全指定的字符或者字符串，其实没有匹配的必要：因为我们都

\w 表示字母、下划线、数字或汉字
\s 表示空白符
\d 表示数字
. 表示除换行符之外的任意字符

已经知道了该字符串。更多情况下，我们想要匹配的是一类字符。下面就是用于匹配一类字符的元字符 (metacharacters)。

使用以上的元字符，就可以匹配文本中对应含义的一个字符。

如果要匹配字符集合中的一个，那么就可以使用方括号将这些集合包在一起。比如 [abc] 匹配 abc 三个字母中的其中一个，或 a 或 b 或 c。[\d\s] 匹配数字或者空白符。

3.2 反向字符匹配

所谓反向匹配，就是要避开某个或者某类字符的匹配。

比如需要取得一个不是小写字母 a 的字符，就应该表示为 [^ a] 。这里 [ ] 的作用是将和^a 看做绑定在一起的东西。因为单独的一个是^ 由其自身意义的 (匹配字符串开端的位置)。而对于一类反向匹配字符，同正向匹配字符一样，是存在特殊命令的。

\W 表示非字母、下划线、数字或汉字，可以是标点符号
\S 表示非空字符
\D 表示非数字

4 数量匹配

4.1 固定长度或数量

当我们想要让一个或一类字符重复出现指定次数时 (比如手机号码为 11 位以 1 开头的字符串)，就需要让\d 重复 10 遍 (除了第一位我们用 1)。于是固定长度我们可以用重复次数来表示。于是 1\d10 就可以匹配手机号码了。

exp{n} 表示对表达式 exp 重复 n 次的字符串

4.2 不定长度或数量

当我们想要获取 exp 重复的数量在一定范围或者我们不知道 exp 会重复多少次的字符串 (比如一个非常长的英文单词)，这时需要用到花括号的另一种用法或者另外的元字符。

exp*	表示 exp 可以重复 0 次及以上
exp+	表示 exp 可以重复 1 次及以上
exp?	表示 exp 可以重复 0 次或 1 次（可有可无的意思）
exp{n,}	表示 exp 可以重复 n 次及以上
exp{n,m}	表示 exp 可以重复 n 到 m 次

所以如果我们想要匹配-、空格或者 作为电话号码之间的连词符，而又要面临着有些电话号码并没有连词符的情况，便可以使用 [-]? 来表示。

4.2.1 贪婪数量匹配与懒惰数量匹配

如果我们在 aabbcb 中查找以 a 开头以 b 结尾的字符串，直接使用 a\w*b 得到的结果会是 aabbcb，而不是 aab。这是因为 * 默认为贪婪匹配，即符合条件的最长字符串。

如果我们想要得到 aab，则要在数量匹配符 * 后面加一个?，来表示懒惰匹配，即重复表达式 exp 次数最少的那个匹配。a[a-z]*?b 就可得出 aab 的匹配。同理，当? 加在不定长度数量符后，就都表示懒惰匹配。

5 位置匹配

前面所说的匹配，都是 exp 要在文本中找到 0 个以上字符。也就是说，匹配得到的结果是包含着 exp 中的字符的。而在某些时候，我们知道 exp 前后的格式，而想要从中抽出 exp 时，就要找到这些特殊的位置。比如单词之间可能会被数字、标点、空白所隔，也可能处于字符串的开头，要找到这些单词，就要分别找到单词的开头和结尾位置。\b 的作用就是找到这个开头或者结尾。因此找到一个单词的正则表达式为\b[a-zA-Z]+\b。这只是一个简单的例子，实际上匹配位置有元字符和用户自定义两种。

5.1 元字符匹配位置

\b 匹配英文单词的开端和结束位置
^ 匹配整个字符串的开端
$ 匹配整个字符串的结尾

5.2 用户自定义匹配位置

用户自己知道想要找到的字符串前后有着特殊字符出现或者特殊格式，比如前缀为 re 的单词，我们想要找到 re 之后的词；又或者后缀为 ing 的单词，我们想找出其原型。这是我们就要匹配 re 后面的位置或者 ing 前面的位置。

5.2.1 正向零宽断言

正向零宽断言就是要找到符合某格式的位置，分为前后两种。要匹配的字符串前面有固定格式时 (比如 re 后面的单词，其前面总是 re 两个字母)，此时用前缀匹配：(?<=exp)，表示 exp 后面紧接的位置；当我们要找带有固定后缀的字符串时，使用后缀匹配 (?=exp)，表示 exp 前面的位置。

5.2.2 反向零宽断言

反向零宽断言匹配的非 exp 前缀或者后缀的位置，使用！替换正向零宽断言中的 = 即可。这个用法还不常见，所以在这里暂且不表。

6 分组

单个字符的重复是直接在其后面加数量匹配的字符或者 n，如果是字符串的重复就要用到分组的概念。用 () 将需要重复的字符串包括起来，从而成为一个整体，用来重复。比如 IPv4 的表达：((2[0-4]\d|25[0-5]|[01]?[0-9]?[0-9])\.)3(2[0-4]\d|25[0-5]|[01]?[0-9]?[0-9])\。可以看出这其中有两层分组，

小组是用 | 分隔开的不同情况中的一个 (或门)，大组是 0-255.，这个大组将会重复 3 次，在第四个重复时没有.，所以要单独再写一次。

一般来说，圆括号会对 exp 进行自动分组编号，但是如果想要人为地给小组进行编号，那么使用的语法为：(?<GroupName>exp) 或者 (?'GroupName'exp)，这样的话不仅会匹配 exp，而且还会将其组名变为 GroupName，这在之后的平衡小组中会起到作用。

6.1 分组后的引用

使用分组后，正则表达式会对其进行编号，并且可以直接引用。引用的方法为\编号，表示该编号对应的组的匹配。比如：(1[0-9])\1 这个正则表达式，如果 (1[0-9]) 匹配到了 12，那么\1 也表示 12，整个的匹配结果就是 1212。

6.2 平衡组和递归引用

当需要匹配出一段字符串中最长的相互匹配的括号之间的字符时，单纯使用简单匹配是无法做到的，因为这不能保证匹配结果中左右括号的数量一致。我们在数据结构中学过栈的知识，并且知道利用栈可以检查字符串中的括号匹配问题。在这里，由于我们可以对分组进行编号，所以同样可以利用这个原理来找到最长的匹配字符串。

思路为遇到左括号压栈，遇到右括号出栈，最终判断栈是否为空。之前分组之所以可以被后向引用，实际上是被压入了栈，即被捕获。

正则表达式为：首先是最外层的一对括号，接着向后匹配，如果是左括号，则分组编号 +1，如果是右括号，则分组编号-1，如果不是括号，那么继续。当匹配到字符尾或者分组编号等于 0 的时候，进入 (?('name')(?!)) 判断。如果为零，则匹配成功，否则匹配失败。

7 转义

上述的很多元字符，如果成为需要匹配的一部分，则需要经过转义才能够表示。这和其他编程语言是相同的。需要转移的元字符包括：

- .
- ?
- -
- \
- *
- +
- (
-)
- {
- }
-
- ^$

需要注意的是，如果放在 [] 中，则所有的元字符都不表示转义，可以直接用于匹配。

有关? 的用法，到现在为止一共有五种：可以转义为?；可以表示 0 次或 1 次重复；加在重复字符之后表示懒惰匹配；与 = 或者! 结合表示零宽断言；自定义编号中用于强制取消分组或指定分组名称。

8 程序中的正则表达式应用

正则表达式的基本语法是通用的，但是到了不同的编程语言中，会有不同的用法。因为常用 C++ 和 Python，所以这里主要介绍在这两种语言中的 regular expression 的应用方法。(regex)

8.1 C++ 中的 regex 应用

C++11 中的 STL 库中有头文件 <regex>,包含着 regex 相关的类:regex 类、match_result 模板类 (包括 smatch 类、cmatch 类)、sregex_iterator 类等，并且包含相关方法：regex_match()、regex_search()、regex_repalce() 等。下面我们从需求方面来运用这些类和方法。

因为 regex 是 STL 中的部分，所以其源文本都使用 std::string 类型的数据。但是经过试验，可能 C++ 中并不支持零宽后行断言。但 Python 支持零宽后行断言。

8.1.1 match_result 类

正则表达式最为重要的部分就是对匹配结果的分析。C++11 中使用 match_result 类来表示符合 regex 的**一个**匹配。在这个匹配中有很多的 sub_match。如果 regex 是分 n 组的，那么一个 match_result 就有 n+1 个 sub_match。第一个 sub_match 是整个 regex 的匹配，剩下的 n 个分别对应 n 个组的匹配。每一个 sub_match 都有方法 str()，用来返回匹配的字符串，同时有 first 和 second 属性，反映该匹配在源文本中的位置。

在匹配、查找的时候，match_result 类很重要。如果要直接查看匹配得到的文本，那么可以使用数组的方括号索引取出第 0 个 submatch。

8.1.2 匹配

匹配的含义是对源字符串**整体**进行检查，查看其是否符合 regex 的模式。使用的方法为 regex_match()，用到的类有 regex 类、match_result 类。方法返回 bool 值告知是否匹配成功。

regex_match() 有输入参数和输出参数。输入参数为源文本和正则表达式，输出参数主要是匹配结果，存放在 match_result 类型的变量中。smatch 类型内部实质上是 string，cmatch 类型内部实质上是一个 C_style 字符串，两者差距不大。

8.1.3 查找

相比匹配是检查**整个**源文本是否符合 regex 的情况，search 是在源文本**之中**找寻是否存在符合 regex 的文本。regex_match() 方法同样返回布尔值来表示是否找到结果。

输入参数和输出参数与 regex_match() 的相同，需要注意的是对 match_result 的填充只是源文本中第一个匹配到 regex 的字符串，并不是源文本中符合 regex 的全部字符串。

8.1.4 逐一查找

使用到 sregex_iterator 类，此外还有 cregex_iterator，类似 smatch 和 cmatch 类的区分。其默认构造函数返回迭代器超尾，可用于遍历结果。

首先构造匹配的迭代器，输入参数为源文本的开始和超尾、正则表达式 regex。将解地址符 (*) 作用于每个迭代器，得到的恰好是对应的 match_result，

如果是 s 就对应 smatch，c 就对应 cmatch。之后便可以使用对 match_result 的处理方法来查看各个匹配结果了。

```
sregex_iterator it(s.begin(),s.end(),re),itend;

for(auto i=it;i!=itend;i++) cout«(*i).str()«endl;
```

8.1.5 分割

当我们需要使用 regex 的匹配文本去对源文本进行分割时，类似 regex_iterator 一样，有一个类为 regex_token_iterator，其构造函数还接受一个 flag 值，如果为 0，效果同 regex_iterator，如果为-1，则正好是以所有匹配字符串为分割的字符串集合；如果是正整数 n，则是以每个 matchresult 的第 n 个 sub_match 为结果。

8.1.6 替换

替换是我们常常需要做的工作，这让我联想到很多编辑器中的 Ctrl+F。在 C++ 中使用的方法为 regex_repalce()。该方法要求源文本、regex 和用于替换的格式化字符串，或者直接用该字符串替代匹配结果。只需要调用一次，replace 会将源文本中所有匹配的地方替换成我们指定的字符串。

8.2 Python 中的 regex 应用

Python 同样是面向对象的语言，其对正则表达式的应用同 C++ 中类似。比较不同的地方就是 Python 并不需要手动创建 regex 对象，而是直接 import re 这个模块，使用 re 的方法。

8.2.1 re 的构造

给入一个字符串，调用 re.compile() 方法，自动返回一个 regex 对象 (pattern)，之后便可以使用这个 pattern 去匹配字符了。

8.2.2 匹配

re.match()。含义和用法同 C++ 的 regex_match() 一样，都是对整个字符串的格式进行检查。如果匹配不成功，返回 none，如果匹配成功，返回 true。

8.2.3 查找

re.search()

8.2.4 逐一查找

这里有两个方法：re.findall() 和 re.finditer()。前者返回一个 list，里面包含所有匹配成功的字符串，后者返回一个迭代器，类型与 match_result 的 sub_match 相似，有成员 start、end，但不同之处是 start、end 直接指示第几位而非一个抽象的迭代器，还有方法 group() 类似于 sub_match 的 str() 方法。

8.2.5 分割

Python 的正则表达式分割很直接，即 re.split()，不像 C++ 中的使用 tokeniterator 才能实现。

8.2.6 修饰符的选择

re 模块有很多修饰符，即 flag，这在调用其匹配方法的时候可以选择，具体可参考 http://www.runoob.com/python/python-reg-expressions.html。