骏马金龙

基础正则https://edu.51cto.com/sd/73e2f

作用：搜索文本

学习正则表达式本质就是学习各种各样的元字符并记住这些元字符的含义。编程语言支持高级正则表达式

grep/sed、awk、文本编辑器、IDE

搜索字符的3种方式

1. 精确搜索
2. 通配符搜索：sql `% _ [] [^]`
3. 正则表达式搜索
    - 元字符：预定义好的具有特殊含义的符号，这些符号能够进行通配
    - 可读性非常的差
    - 写正则表达式不难

正则分类

正则表达式的分类：

- 基础正则表达式(BRE:basic regular expression)
- 扩展正则表达式(ERE:extended regular expression)
- 编程语言支持的高级正则表达式

BRE和ERE语法基本一致，只有部分元字符（预定义好的带有特殊含义的一些符号）需要区别对待。

扩展正则中这些元字符可直接使用：`?`、`+`、`{`、`}`、`|`、`(`和`)`。

基础正则中这些元字符前需要加反斜线转义：`\?`、`\+`、`\{`、`\}`、`\|`、`\(`和`\)`。

> grep sed默认使用基础正则表达式
>
> grep -E、sed -r、egrep、awk扩展正则表达式

grep基本使用

例如：

```
# grep -E 'pattern' filename
# cmd | grep -E 'pattern'

# 基础正则
echo "hello world" | grep '[a-z]\+'
echo "hello world" | grep '\([a-z]\+\)d'

# 扩展正则
echo "hello world" | grep -E '[a-z]+'
echo "hello world" | grep -E '([a-z]+)d'
```

纯普通字符匹配

普通字符的匹配模式相当于是精确匹配。

```
echo "hello world" | grep 'world'
```

基本的正则元字符

匹配字符

- `abc` 匹配字符串"abc"，普通字符的匹配
- `[abcde...]`：匹配中括号内的任意单个字符
  - `a[xyz]b` :axb、ayb、azb，不能匹配aab amb
- `\n`：匹配换行符
- `\t`：匹配制表符
- `\w`：匹配单词字符 `[a-zA-Z0-9_]`
- `\W`：匹配非单词字符 `[^a-zA-Z0-9_]`
- `\s`：匹配空白字符
- `\S`：匹配非空白字符
- `\d`：匹配数字
- `\D`：匹配非数字
- `.` 表示匹配任意单个字符
  - 默认情况下 `.` 无法匹配换行符，可在多行模式下设置匹配模式修饰符使之真正匹配任意字符，包括换行符

注意有些程序并不完全支持上面的反斜线转义元字符。例如gnu grep 2.6版本不支持 `\s` 和 `\d`，而gnu grep 2.20支持 `\s` 但不支持 `\d`，sed不支持 `\d`。

上面所说的**单词(word)**，在正则表达式中的含义：`[a-zA-Z0-9_]` 组成的字符或字符串都是单词，例如 `nihao, hello world_` 第一个单词是 `nihao`，第二个单词是 `hello`，第三个单词是 `world_`。

中括号表达式

字符组

普通中括号包围的字符组：表示某单个字符匹配中括号内的任一字符即匹配成功

- `x[abc]z`：可以匹配包含"xaz"、"xbz"、"xcz"的字符串
- 取反表示法：中括号内开头使用`^`，表示只要不是中括号中的字符就匹配
  - `x[^abc]z`：可匹配包含"xdz"、"xez"等的字符串，但不能匹配包含"xaz"、"xbz"、"xcz"的字符串
- 范围表示法：
  - `[a-z]`：代表任一单个小写字母
    - `[^a-z]`：只要单个非小写字母的其它任一单个字符
  - `[A-Z]`：代表任一单个大写字母
  - `[0-9]`：代表任一单个数字
    - 注：`[0-59]`表示匹配`0、1、2、3、4、5、9`而不是0到59中间的数值
  - `[a-z0-9A-Z]`：代表任一字母或数字
  - `[a-z0-9A-Z_]`：代表任一字母、数字或下划线，即匹配**单词字符**（word）
  - `[A-z]或[a-Z]`：建议不要使用这种横跨大小写字母的范围表达式，不同地方表达的含义不同
  - 甚至有些按照字典顺序排序时，`[a-d]`不是等价于abcd，而是等价于aBbCcDd。如果想要等价于abcd，应将locale环境设置为C：`LC_ALL=C`
- 特殊元字符在中括号中的匹配：
  - 想要在中括号中匹配`^`，需将其放在中括号的非开头位置，如`[a^]`
  - 想要在中括号中匹配`-`，需将其放在开头位置或结尾位置，如`[abc-]`、`[-abc]`
  - 想要在中括号中匹配`]`，需将其放在开头位置，如`[]abc]`
  - 想要匹配上面2个或三个元字符，`[]^]`、`[-^]`、`[]-]`、`[]^-]`

字符类

将字符分成不同的类别，称为字符类(character class)。

下面是POSIX标准的字符类。

字符类	含义
`[:lower:]`	等价于 `a-z`
`[:upper:]`	等价于 `A-Z`
`[:alpha:]`	等价于 `A-Za-z`，也等价于 `[:lower:]` + `[:upper:]`
`[:digit:]`	等价于 `0-9`
`[:alnum:]`	等价于 `0-9A-Za-z`，也等价于 `[:lower:]` + `[:upper:]` + `[:digit:]`
`[:xdigit:]`	匹配十六进制数字 `0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f`
`[:blank:]`	匹配空格或制表符
`[:space:]`	匹配空格、制表符、换行符、换页符、垂直制表符、回车符等等所有空白符号
`[:punct:]`	(Punctuation)匹配所有标点符号，`! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ` {`
`[:print:]`	可打印字符，等价于 `[:alnum:]` + `[:punct:]` +空格
`[:graph:]`	图形字符，即能展现字符颜色的符号，等价于 `[:alnum:]` + `[:punct:]`
`[:cntrl:]`	所有的控制符号，八进制的 000 到 037，以及 177 (DEL)

有些语言还额外提供一些非POSIX的字符类，比如 `[:ascii:]` 表示任一ASCII表中的字符。

使用字符类时，需使用中括号包围：

- `[[:alpha:]]`：任一字母
- `[^[:alpha:]]`：任一非字母
- `[^[:alpha:]0-3]`：任一非字母且非0 1 2 3

等价类和排序类(了解)

`[=x=]` 包围时表示**等价类**（equivalence class），等价类表示将普通字母和带有重音的字母分为一类。例如 `aāáǎà` 是一类，它等价于 `[=a=]`、`oōóǒò` 又是一类，等价于 `[=o=]`。注：没有 `[=ao=]` 这样连在一起的类。

例如：

```
1   echo "aāáǎà" | grep '[[=a=]]'
```

`[.xyz.]` 包围时表示**排序类**(collating symbol)，排序类表示将 `[..]` 包围中的多个符号当作一个字符，但要求在字符集中预先定义映射关系，例如已经预定义了xyz对应于R，那么 `a[[.xyz.]]a` 可以可以匹配 `aRa`。

位置匹配(锚定)

字符匹配会消耗字符。位置匹配，只是匹配位置，不会消耗字符。

只匹配位置，不匹配字符，所以不会消耗字符数量，也称为即**零宽断言**。

- `^` 匹配行首
- `$` 匹配行尾
- `\<`：匹配单词开头处的位置

- `\>`：匹配单词结尾处的位置
- `\b`：匹配单词边界处的位置（开头和结尾），所以 `\bword\b` 等价于 `\<word\>`
- `\B`：匹配非单词边界处的位置

正则表达式的匹配过程

`reg:abc`

`str:"aabxabcxyz"`

每一轮正则的匹配，都需要从正则的第一个元素从头开始匹配。

第一轮匹配：

- 扫描第一个字符，和正则表达式的第一个元素进行匹配
- 如果匹配失败，则意味着这一轮的正则匹配失败

第二轮匹配：

- 扫描下一个字符，从头开始和正则表达式进行匹配
- 如果匹配成功，则继续扫描下一个字符和正则表达式的下一个元素进行匹配
    - 如果这个字符匹配失败，则导致这一轮正则匹配失败
    - 交还除第二轮匹配开始的首字符外的所有字符
    - 交还之后，从交还的第一个字符开始进入下一轮匹配

第三轮匹配：

- 如果这一轮正则匹配成功，则不交还匹配成功的字符
- 然后从匹配成功的下一个字符进入下一轮匹配

字符消耗问题：

- **某轮正则匹配成功，则消耗所有匹配成功的字符**
- **某轮匹配失败，则消耗本轮匹配的首字符，剩余的字符被交还**

量词(重复匹配次数)

1.在基础正则表达式中，对于量词的元字符需要加上反斜线进行转义。

关于量词需要注意结论：**量词它是正则表达式中的一个隐含的修饰符，它修饰它前面一个字符或前面一个子表达式，它自身不是正则表达式中的独立元素，量词和它所修饰的字符或子表达式组合起来才是正则表达式中的独立元素。**

量词可能会出现大量的回溯，而回溯所正则表达式中的性能杀手

- `{m}` 表示匹配前一个字符或前一个子表达式m次
    - "a{3}"：匹配a3次。aaa aab aaaa
    - "[abc]{3}"： aaa bbb ccc aba abc cba
- `{m,n}` (m<n)表示匹配前一个字符或前一个子表达式最少m次，最多n次
    - "a{3,5}" aaa aaaa aaaaa
- `{m,}` 表示匹配前一个字符或前一个子表达式至少m次
    - "3{2,}" 33 333 33333333 333333333333333333333
- `{,n}` 表示匹配前一个字符或前一个子表达式至多n次（注：不一定会支持该语法）

    - "a{,3}" a aa aaa aaaa
    - 注意：**匹配0次也算匹配成功，只不过这时候匹配的是空字符，grep没法显示出来**
- `?` 表示匹配前一个字符或前一个子表达式0或1次，等价于 `{0,1}` 等价于 `{,1}`
    - "ab?c" "abc" "ac"
- `*` 表示前一个字符或前一个子表达式匹配0或多次，即任意次数 等价于 `{0,}`
    - `a3*b` ：可以匹配ab、a3b、a33b、a33333333b
    - `.*` ：匹配任意长度的任意字符
        - 问：为什么 `.*` 匹配的是任意字符任意长度，而不是某个字符任意长度？
        - `.*` 3 333 33333 abcdef
- `+` 表示匹配前一个字符或前一个子表达式1或多次，即至少一次 等价于 `{1,}`
    - "a+" aa aaaa a aaaaaaaa
    - "[abc]+" abcabcabc

这些量词均为贪婪匹配模式，即尽可能多的去匹配符合条件的字符。例如正则表达式 `ab.*c` 去匹配字符串 `abbcdecfc` ，其中 `.*` 部分匹配的将是 `bcdecf` 。

基础正则和扩展正则都只支持贪婪匹配，不支持非贪婪匹配。如果想要在基础正则或扩展正则上实现非贪婪匹配，比较复杂，但也能实现。例如：

```
# 字符串："abc:def:ghi:jkl"

# 匹配目标：取第一列，即"abc"
# 正则表达式：
    # 错误写法："[^:]*:"
    # 正确写法："[^:]*"

# 匹配目标：取前两列，即"abc:def"
# 正则表达式："[^:]*:[^:]*"
```

怎么实现：以中括号取反的方式排除分隔符，然后用量词去修饰。

二选一表达式

基础正则表达式中，需要对 `|` 进行转义，使用 `\|` 来表示二选一表达式。

竖线 `|` 分隔左右两个正则子表达式，表示匹配任一一个即可。例如 `a|b` 表示a或者b，在结果上等价于 `[ab]` ；`[0-5]|\sa` 表示0、1、2、3、4、5、" a"。

使用二选一子表达式注意几点：

1. 二选一元字符优先级很低，所以 `abc|def` 表示的是abc或def，等价于 `(abc)|(def)` ，而不是 `ab(c|d)ef`
2. 二选一结构和中括号表达式的性能比较：
    - 对于DFA引擎(grep egrep awk sed)来说，二选一结构和中括号的性能是完全一样的，比如 `a|b|c|d|e` 和 `[abcde]` 完全等价
        - 对于NFA引擎(编程语言支持的正则表达式全所NFA引擎)来说，二选一结构性能远低于中括号性能， `a|b|c|d|e` 意味着5倍的回溯数量，而 `[abcde]` 它只有单次回溯数量
        - 能用中括号表达式就不要用二选一结构
        - 把尽可能匹配到的选择写在二选一结构的前面
3. 二选一结构在分组捕获时，只有成功匹配时才会能反向引用

- `x(abc)|(def)y` 中，要么 `\1` 可用，要么 `\2` 可用，不会同时有用
- `([ab])x|cdy\1` 无法匹配cdya或cdyb

分组捕获

使用小括号包围一部分正则表达式 `(pattern)` ，这部分正则表达式即成为一个分组整体，也即成为一个子表达式。

小括号有两个隐含的功能

- 1.分组
- 2.自动捕获：保存分组的匹配结果，以后可以去引用这个捕获的结果

根据左括号的位置决定第几个分组。

例如： `(abc)def` 、 `([a-d] ){3}` 、 `(([0-9])abc(def){2})(hgi)` 。

分组后可以使用 `\N` 来反向引用对应的**分组的匹配结果**，N是1-9的正整数， `\1` 表示第一个分组表达式的匹配结果， `\2` 表示第二个分组表达式的匹配结果。

例如：

```
echo "you see see you" | grep -E '(.*) \1' # "a a" "ab ab" "13 13"
echo "you see see you" | grep -E '((.*) (.*)) \3 \2'
# "a b b a"
# "ab abc abc ab"
# "abcd xyz xyz abcd"
```

注意：**反向引用所引用的是分组匹配后的结果，不是分组表达式。**

正则表达式 `(abc|def) and \1xyz` 可以匹配字符串"abc and abcxyz"或"def and defxyz"，但是不能匹配"abc and defxyz"或"def and abcxyz"。

如果想要引用分组表达式而不是分组捕获的结果，需使用递归正则。Perl

```
$ echo "abc and defxyz" | grep -P '(abc|def) and \1xyz'
$ echo "abc and defxyz" | grep -P '(abc|def) and (?1)xyz'
# grep -P '(abc|def) and (abc|def)xyz'
abc and defxyz
```

匹配模式修饰符

i修饰符：忽略大小写的匹配 "abcABC" "/ab/i"

g修饰符：全局匹配