上章回顾

- for循环的格式是什么?

```
#!/bin/bash
for 变量名 in 循环变量值
    do
        命令序列
        命令序列
    done
```

- while循环的格式是什么?

```
#!/bin/bash
while [条件表达式]
do
    命令序列
done
```

- case语句的格式是什么?

```
#!/bin/bash
case "$1" in
    位置变量值)
        命令
        ;;
    位置变量值)
        命令
        ;;
esac
```

本章目标

- 熟悉 awk 的命令行模式基本语法结构。
- 熟悉 awk 的相关内部变量。
- 熟悉 awk 常用的打印函数 print。
- 能够在 awk 中匹配正则表达式并打印相关的行。

- 掌握正则表达式以及 grep 和 egrep 的进阶应用。

一、AWK概述

1、AWK 是一种编程语言，主要用于 Linux/Unix 系统下处理文本和数据，是系统的工具之一。数据来源可以是标准输入、一个或多个文件，或者其他命令的输出。

2、处理文本和数据时，它会逐行扫描文件，默认从第一行扫描到最后一行，寻找匹配的特定行，并对这些数据进行输出或执行相关操作。

3、AWK 名称源于其三位作者 Alfred Aho、Brian Kernighan 和 Peter Weinberger 名字的首字母。

4、gawk 是 awk 的 GNU 版本，提供了贝尔实验室和 GNU 的一些扩展功能。

二、AWK的工作原理

1、awk 是 Linux 中可针对列进行筛选的工具，默认列分隔符为空格，可在命令行使用-F选项指定分隔符。

1）基本语法为：awk选项 '{[文件开始前]}; {print第几列}'。例如：

- awk '{print $0}'文件路径：打印文件每一行的所有内容。
- awk -F : '{print $0}'文件路径：以冒号为分隔符，打印文件每一行的所有内容。

2、awk的执行步骤

1）文件处理前（BEGIN）：在开始检查文件前执行，是一个代码块，每次执行 awk 命令时 BEGIN 仅执行一次。通常用于定义开头显示的字样，也可利用其灵活性完成一些数值运算。例如：awk 'BEGIN {print 1+3}'。

```
awk 'BEGIN {print 1+3}'
```

2）文件读取中：通过内置变量或正则条件筛选数据，awk 逐行检查数据，找到满足条件的行后，提取对应列的数据。例如

```
#以冒号为分隔符，打印文件每一行的第一列和第三列，并在开头打印“开始”。
awk -F : 'BEGIN{print "开始"}；{print $1,$3}；' 文件位置

#将字段分隔符设为冒号，输出字段分隔符设为制表符，打印/etc/passwd文件每一行的第一列和第三列。
awk 'BEGIN { FS = ":";  OFS = "\t"} { print $1, $3}' /etc/passwd

#打印表头“用户名 用户 ID”，并打印/etc/passwd文件每一行的第一列和第三列。
awk 'BEGIN {    print "用户名\t用户ID"} {    print $1, $3}' /etc/passwd
```

3）文件读取后（END）：在最后一行数据筛选完成，即将输出结果时执行。例如：awk 'BEGIN{print "开始"}; {print $1,$3}; END {print "结束"}'文件位置。

```
awk 'BEGIN{print "开始"}; {print $1,$3}; END {print "结束"}'  文件位置
```

3、awk 使用 print 函数打印字段，默认字段间用空格分隔。

1）例如：{print $1,$2}中，逗号映射的是 OFS 内部变量，其默认值为空格。

2）可在使用 awk 前设置 OFS 函数改变列之间的分隔内容。例如：awk -F: 'BEGIN{OFS=" --- "};{print $1,$3}'文件位置。

4、awk 默认分隔符是空格，因为其内置变量 FS 默认值为空格，也可通过指定 FS 的值来更改分隔符。

1）例如：awk 'BEGIN{FS=":"};{print $1,$NF}'文件位置。OFS 和 FS 都是 awk 的内部变量，编写 awk 脚本时，若需动态更改间隔符和输出字段分隔符，可使用 FS 和 OFS。

2）-F属于命令行选项，不能在 awk 程序内修改，更适用于处理多个文件且使用同一种分隔符的情况。

```
awk 'BEGIN{FS=":"};{print $1,$NF}' 文件位置
```

三、awk的内部变量

变量	变量说明	备注
$0	当前处理行的所有记录	
$1,$2,$3...$n	文件中每行以间隔符号分割的不同字段	awk -F: '{print $1,$3}'
NF	当前记录的字段数（列数）	awk -F: '{print NF}'
$NF	最后一列	$(NF-1)表示倒数第二列
NR	行号	
FS	定义间隔符，默认空格	'BEGIN{FS=":"};{print $1,$3}'
OFS	定义输出字段分隔符，默认空格	'BEGIN{OFS="\t"};print $1,$3}'

四、print和printf

1、print：标准化输出，正常输出需要显示的内容

2、printf：通过选项添加输出的格式，不会自动换行

- %15s：表示第一列占用 15 个字符，以字符串形式输出。
- \n：用于自动换行，由于 printf 不会自动换行，需要换行时在末尾添加。
- %-15：其中 “-” 表示当前列左对齐，默认列是右对齐。
- 例如：

```
awk 'BEGIN {FS=":"};{printf "%-15s %15s\n",$1,$3}'文件位置。
```

五、awk命令行中赋予变量

```
awk -F : -v a=1 '{print $a}' 文件位置
awk -F : 'BEGIN {a=1;b=2};{print $a,$b}'  文件位置
awk -v a=10 'BEGIN {print a}'
```

六、awk脚本使用

1、脚本编写

```
#!/usr/bin/awk -f
# 设置字段分隔符为冒号
BEGIN {
    FS = ":"
}
# 处理第 1 行到第 3 行
NR == 1, NR == 3 {
    print $1 "\t" $NF
}
```

- #!/usr/bin/awk -f：这是脚本的魔法字符，表明该脚本要使用 awk 来执行。
- BEGIN { FS = ":" }：在处理文件之前，设置字段分隔符为冒号 :。
- NR == 1, NR == 3 { print $1 "\t" $NF }：NR 代表当前处理的行号，此命令会处理第 1 行到第 3 行的数据，打印出每行的第一个字段和最后一个字段，并用制表符 \t 分隔。

2、使用方法

1）把上述代码保存为一个文件，例如 script.awk。

2）给脚本添加可执行权限

3）执行脚本，添加参数

```
[root@localhost ~] chmod +x test.awk
[root@localhost ~] ./test.awk /etc/passwd
root    /bin/bash
```

```
bin     /sbin/nologin
daemon  /sbin/nologin
```

七、awk语法实例

1、基于语法

1）打印第一列和倒数第二列

```
awk -F: '{print $1,$(NF-1)}' /etc/passwd | head -3
```

- -F:：指定字段分隔符为冒号 :。
- {print $1,$(NF-1)}：对文件的每一行，打印第一个字段和倒数第二个字段。
- | head -3：将 awk 的输出结果截取前 3 行。

2）打印第一列、最后一列和倒数第二列，并且显示列数

```
awk -F: '{print $1,$(NF-1),$NF,NF}' /etc/passwd | head -3
```

- -F:：指定字段分隔符为冒号 :。
- {print $1,$(NF-1),$NF,NF}：对文件的每一行，打印第一个字段、倒数第二个字段、最后一个字段以及当前行的字段数量。
- | head -3：将 awk 的输出结果截取前 3 行。

3）打印包含 root 的行，显示所有列

```
awk '/root/{print $0}' /etc/passwd
```

- /root/：这是一个正则表达式，用于匹配包含 root 的行。
- {print $0}：打印匹配到的行的所有内容。

4）打印包含 root 的行

```
awk '/root/' /etc/passwd
```

- /root/：正则表达式，匹配包含 root 的行。在 awk 中，如果没有指定动作，默认动作就是打印匹配到的行。

5）打印包括 root 的行的第一列和最后一列

```
awk -F: '/root/{print $1,$NF}' /etc/passwd
```

- -F:：指定字段分隔符为冒号 :。
- /root/：正则表达式，匹配包含 root 的行。
- {print $1,$NF}：对匹配到的行，打印第一个字段和最后一个字段。

6）打印包括 root 的行，以 : 分隔的第一、第三列

```
awk -F: '/root/{print $1,$3}' /etc/passwd
```

- -F:：指定字段分隔符为冒号 :。
- /root/：正则表达式，匹配包含 root 的行。
- {print $1,$3}：对匹配到的行，打印第一个字段和第三个字段。

7）打印第一行到第五行

```
awk 'NR==1,NR==5' /etc/passwd
```

- NR==1,NR==5：这是一个范围模式，NR 表示当前行号，此模式会匹配第 1 行到第 5 行。默认动作是打印匹配到的行。

8）打印第一行到第五行以及 root 开头的行

```
awk 'NR==1,NR==5;/^root/{print $0}' /etc/passwd
```

- NR==1,NR==5：匹配第 1 行到第 5 行。
- /^root/：正则表达式，匹配以 root 开头的行。
- {print $0}：对匹配到的行，打印所有内容。

2、OFS和FS语法

```
awk 'BEGIN{FS=":"};/^root/,/^lp/{print $1,$NF}' /etc/passwd
```

- BEGIN{FS=":"}：在处理文件之前，`BEGIN` 块会被执行，这里将字段分隔符 `FS`（Field Separator）设置为冒号 :。这意味着 `awk` 在处理每行文本时，会以冒号为分隔符来划分字段。
- /^root/,/^lp/：这是一个范围模式，它会匹配从以 `root` 开头的行开始，到以 `lp` 开头的行结束的所有行。

- {print $1,$NF}：对于匹配到的行，打印该行的第一个字段 `$1` 和最后一个字段 `$NF`，字段之间默认以空格分隔。

```
awk -F: 'BEGIN{OFS="\t\t"};/^root/,/^lp/{print $1,$NF}' /etc/passwd
```

- -F:：通过 `-F` 选项指定字段分隔符为冒号 `:`。
- BEGIN{OFS="\t\t"}：在 `BEGIN` 块中，将输出字段分隔符 `OFS`（Output Field Separator）设置为两个制表符 `\t\t`。这意味着当使用 `print` 语句输出多个字段时，字段之间会以两个制表符分隔。
- /^root/,/^lp/：同样是范围模式，匹配从以 `root` 开头的行到以 `lp` 开头的行。
- {print $1,$NF}：对匹配到的行，打印第一个字段和最后一个字段，字段之间用两个制表符分隔。

```
awk -F: 'BEGIN{OFS="@@@"};/^root/,/^lp/{print $1,$NF}' /etc/passwd
```

- -F:：指定字段分隔符为冒号 `:`。
- BEGIN{OFS="@@@"}：在 `BEGIN` 块中，将输出字段分隔符 `OFS` 设置为 `@@@`。这样，当使用 `print` 语句输出多个字段时，字段之间会以 `@@@` 分隔。
- /^root/,/^lp/：范围模式，匹配从以 `root` 开头的行到以 `lp` 开头的行。
- {print $1,$NF}：对匹配到的行，打印第一个字段和最后一个字段，字段之间用 `@@@` 分隔。

3、printf格式化输出

```
awk 'BEGIN{FS=":"};{printf "%-15s %-15s %-15s\n",$1,$6,$NF}' /etc/passwd
```

- BEGIN{FS=":"}：`BEGIN` 块在处理文件之前执行，这里把字段分隔符 `FS` 设为冒号 `:`，意味着 `awk` 会以冒号来划分每行文本的字段。
- {printf "%-15s %-15s %-15s\n",$1,$6,$NF}：
  - printf 是格式化输出函数。
  - %-15s 表示以字符串形式输出，并且左对齐，宽度为 15 个字符。若字符串长度不足 15 个字符，会在右侧用空格补齐。
  - $1、`$6` 和 `$NF` 分别代表每行的第一个字段、第六个字段和最后一个字段。
  - \n 是换行符，使每行输出后换行。

```
awk -F: '{printf "|%15s| %10s| %15s|\n", $1,$6,$NF}' /etc/passwd
```

- -F:：通过 `-F` 选项指定字段分隔符为冒号 `:`。
- {printf "|%15s| %10s| %15s|\n", $1,$6,$NF}：

- printf 用于格式化输出。
- | 是普通字符，会直接输出，用于构建表格的边界。
- %15s 和 `%10s` 表示以字符串形式输出，并且右对齐，宽度分别为 15 个字符和 10 个字符。若字符串长度不足相应宽度，会在左侧用空格补齐。
- $1、`$6` 和 `$NF` 分别为每行的第一个字段、第六个字段和最后一个字段。
- \n 是换行符，让每行输出后换行。

```
awk -F: '{printf "|%-15s| %-10s| %-15s|\n", $1,$6,$NF}' /etc/passwd
```

- -F:：指定字段分隔符为冒号 :。
- {printf "|%-15s| %-10s| %-15s|\n", $1,$6,$NF}：
  - printf 用于格式化输出。
  - | 是普通字符，用于构建表格的边界。
  - %-15s 和 `%-10s` 表示以字符串形式输出，并且左对齐，宽度分别为 15 个字符和 10 个字符。若字符串长度不足相应宽度，会在右侧用空格补齐。
  - $1、`$6` 和 `$NF` 分别代表每行的第一个字段、第六个字段和最后一个字段。
  - \n 是换行符，使每行输出后换行。

4、BEGIN和END用法

```
head -3 /etc/passwd | awk -F: 'BEGIN{print
"U_name\t\tLog_Shell\t\tU_home\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"};{print
$1"\t\t"$NF"\t\t"$(NF-1)};END{print" ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"}'
```

- BEGIN块
  - 在处理输入数据之前执行，打印表头 U_name、Log_Shell、U_home，并用制表符分隔，然后换行打印一排波浪线作为分隔线。
- 主体部分
  - 对输入的每一行，打印第一个字段 $1、最后一个字段 $NF 和倒数第二个字段 $(NF - 1)，字段之间用两个制表符分隔。
- END块
  - 在处理完所有输入数据后执行，打印一排波浪线作为结束分隔线。

```
head -3 /etc/passwd | awk -F : 'BEGIN{print
"u_name\t\th_dir\t\tshell\n**********************"}{printf "%-20s %-20s %-20s\n",$1,$(NF-
1),$NF};END{print "****************"}'
```

- BEGIN块

- 在处理输入数据前执行，打印表头 u_name、h_dir、shell，用制表符分隔，然后换行打印一排星号作为分隔线。
- 主体部分
  - 使用 printf 函数进行格式化输出，%-20s 表示以左对齐的方式输出字符串，宽度为 20 个字符。依次输出第一个字段 $1、倒数第二个字段 $(NF - 1) 和最后一个字段 $NF，字段之间用空格分隔，最后换行。
- END块
  - 在处理完所有输入数据后执行，打印一排星号作为结束分隔线。

八、正则表达式

1、正则表达式（Regular Expression，常简称为 regex 或 regexp，缩写为 RE），又被称作正规表示法、常规表示法 。它本质上是一种字符模式，主要用于在文本查找过程中匹配特定的字符序列。

2、在众多程序设计语言里，都支持借助正则表达式来开展字符串操作。正则表达式这一概念，最初是由 Unix 系统中的工具软件得以广泛传播，像 sed、grep、awk 等工具便是典型代表。
软件（例如se,grep,awk）普及开的。

3、元字符是正则表达式里一类具备特殊含义的专用字符，例如：点（.）、星号（*）、问号（?）等都属于元字符范畴。

- **点（.）通常代表匹配任意一个字符；**
- **星号（*）代表匹配前面的字符零次或多次；**
- **问号（?）代表匹配前面的字符零次或一次。**
- 这些元字符在正则表达式中用于构建复杂的匹配模式。

4、前导字符指的是处于元字符前面的字符。比如在 “abc*” 中，“c” 是 “*” 的前导字符；在 “aooo.” 里，“o” 是 “.” 的前导字符。

5、常见符号：

- “==”：表示等于关系。
- “!=”：代表不等于关系。
- “>”：表示大于关系。
- “<”：代表小于关系。
- “>=”：表示大于等于关系。
- “<=”：代表小于等于关系。
- “~”：用于匹配操作，即判断左侧字符串是否与右侧正则表达式模式相匹配。
- “!~”：表示不匹配操作，与 “~” 相反，判断左侧字符串是否不匹配右侧正则表达式模式。
- “!”：为逻辑非运算符，用于对逻辑值取反。
- “&&”：是逻辑与运算符，只有当左右两侧逻辑表达式都为真时，整个表达式结果才为真。
- “||”：为逻辑或运算符，只要左右两侧逻辑表达式有一个为真，整个表达式结果即为真。

6、正则表达式和awk结合实例

```
从第一行开始匹配到以lp开头行
awk -F: 'NR==1,/^lp/{print $0 }' passwd
从第一行到第5行
```

```
awk -F: 'NR==1,NR==5{print $0 }' passwd
从以lp开头的行匹配到第10行
awk -F: '/^lp/,NR==10{print $0 }' passwd
从以root开头的行匹配到以lp开头的行
awk -F: '/^root/,/^lp/{print $0}' passwd
打印以root开头或者以lp开头的行
awk -F: '/^root/ || /^lp/{print $0}' passwd
awk -F: '/^root/;/^lp/{print $0}' passwd

显示5-10行
awk -F':' 'NR>=5 && NR<=10 {print $0}' /etc/passwd
awk -F: 'NR<10 && NR>5 {print $0}' passwd
列出uid在0-20之间的用户信息
awk -F: '$3>=0 && $3 <= 20{print $0}' /etc/passwd
列出uid是0或者14的用户信息
awk -F: '$3==0 || $3 == 14{print $0}' /etc/passwd
列出第7列不以nologin结尾的正行内容
awk -F: '$7!~/nologin$/{print $0}' /etc/passwd
列出第3列的值不等于38的正行内容
awk -F: '$3!=38{print $0}' /etc/passwd
打印奇数行
awk -F: 'NR%2==1{print $0}' /etc/passwd
打印30-39行以bash结尾的内容：
awk 'NR>=30 && NR<=39 && /bash$/{print $0}' passwd
awk 'NR>=3 && NR<=8 && /bash$/' 文件名称
```

九、正则表达式：grep工具使用

1、grep 是一个强大的文本处理工具，主要用于依据指定的关键字对文本内容进行行过滤操作。其基本语法为：grep [options] 'keys' [filename]。

2、以下是 grep 常用选项及其功能说明：

- **-i：该选项使 grep 在搜索时不区分大小写。**
  - **例如，当搜索关键字为 “hello” 时，无论文本中出现的是 “Hello”、“HELLO” 还是 “hELLo” 等，都能被匹配到。**
- **-v：执行反向选择操作，即查找不包含指定关键字内容的行。**
  - **例如，若要查找文件中不包含 “error” 字样的行，就可以使用 grep -v 'error' filename 命令。**
- **-w：按单词进行搜索，实现精准匹配。当使用该选项时，只有当关键字作为一个完整的单词出现时才会被匹配。**
  - **例如，搜索 “the”，文本中 “there” 就不会被匹配到，只有单独的 “the” 才会被识别。**
- **-c：用于统计匹配到指定关键字的次数。执行命令后，grep 会输出符合条件的行数，而不是具体的行内容**
  - **例如 grep -c 'apple' fruit.txt 会显示在 fruit.txt 文件中 “apple” 出现的行数。**

- **-n：在输出匹配到的行时，同时显示该行在文件中的行号。这对于定位特定内容在文件中的位置非常有用，**
  - **例如 grep -n 'world' hello.txt 会输出包含 “world” 的行及其行号。**

3、grep 还支持一些基于正则表达式的特殊匹配模式：

- ^key：表示匹配以指定关键字开头的行。
  - 例如，grep '^start' log.txt 会找出 log.txt 文件中所有以 “start” 开头的行。
- key$：表示匹配以指定关键字结尾的行。
  - 例如 grep 'end$' story.txt 会找到 story.txt 文件中所有以 “end” 结尾的行。
- ^$：用于匹配空行。当执行 grep '^$' file.txt 命令时，会筛选出 file.txt 文件中所有不包含任何字符的空行。

4、示例

```
grep -i root passwd 忽略大小写匹配包含root的行
grep -w ftp passwd 精确匹配ftp单词
grep -c root passwd 匹配次数
grep -c 'bash$' passwd 统计以bash结尾的行数
grep -n root passwd 打印匹配到root关键字的行号
grep -ni root passwd 忽略大小写匹配统计包含关键字root的行
grep -nic root passwd 忽略大小写匹配统计包含关键字root的行数
grep -i ^root passwd 忽略大小写匹配以root开头的行
grep bash$ passwd 匹配以bash结尾的行
grep -n ^$ passwd 匹配空行并打印行号
grep ^# /etc/vsftpd/vsftpd.conf 匹配以#号开头的行
grep -v ^# /etc/vsftpd/vsftpd.conf 匹配不以#号开头的行
```

5、正则的使用

1）建议新建文件测试

2）正则表达式标准概念解析

- .：匹配任意单个字符，但不包括换行符。例如，若有文本 “apple”“banana”“cat”，使用 `.at` 进行匹配时，只有 “cat” 会被匹配到，因为 . 替代了 “c” 这个字符。
- *：表示前导字符可以匹配 0 次或连续多次。以 “ab*” 为例，它能够匹配 “a”（此时 `b` 出现 0 次）、“ab”（`b` 出现 1 次）以及 “abb”（`b` 出现 2 次）等，但像 “cb” 就无法匹配，因为其前导字符不是 `a`。
- .*：可匹配任意长度的字符序列。例如对于 “ab.*”，在 “ab123”“abbb”“abab” 等字符串中都能成功匹配，因为它能涵盖 `ab` 之后任意数量和种类的字符。
- ^：匹配行的开头位置。比如，当文本中有 “apple”“banana”“cherry”，使用 `^a` 进行匹配，只有 “apple” 和 “banana” 所在行能被匹配，因为它们是以 `a` 开头的行。
- $：匹配行的结尾位置。假设文本中有 “pear”“grape”“melon”，使用 `r$` 进行匹配，只有 “pear” 所在行能被找到，因其以 `r` 结尾。
- ^$：专门用于匹配空行，即不包含任何可见字符的行。
- []：能够匹配指定字符组内的任一单个字符。例如 `[abc]`，若文本为 “apple”“banana”“cherry”，它可以匹配到 “apple” 中的 `a`、“banana” 中的 `b` 以及 “cherry” 中的 `c`。

- [^]：匹配不在指定字符组内的任一字符。比如 `[^abc]`，在 “apple”“banana”“cherry” 中，像 “cherry” 中的 `h` 就符合匹配条件，因为 `h` 不在 `[abc]` 这个字符组内。
- ^[]：匹配以指定字符组内的任一字符开头的行。例如 `^[abc]`，对于文本 “apple”“banana”“cherry” “date”，“apple”“banana”“cherry” 所在行都会被匹配，因为它们都以 `a`、`b` 或 `c` 开头。
- ^[^]：匹配不以指定字符组内的任一字符开头的行。比如 `^[^abc]`，在上述文本中，“date” 所在行就会被匹配，因为它不以 `a`、`b` 或 `c` 开头。
- \<：用于取单词的开头部分。例如在文本 “apple pie” 中，使用 `\<app` 可以匹配到 “apple”，因为它定位到了单词 “apple” 的开头。
- \>：用于取单词的结尾部分。比如在 “apple pie” 中，`pie\>` 可以匹配到 “pie”，确定了 “pie” 这个单词的结尾位置。
- \<\>：实现精确匹配整个单词，等同于使用 `grep -w 'xxx'` 命令。例如，对于文本 “apple”“apples” “banana”，使用 `\<apple\>` 只会匹配到 “apple”，而不会匹配 “apples”。
- \{n\}：匹配前导字符连续出现 `n` 次。例如 `go\{2\}`，在 “google”“gooogle” 等文本中，只有 “google” 能被匹配，因为其中 `o` 连续出现了 2 次。
- \{n,\}：匹配前导字符至少出现 `n` 次。例如 `go\{2,\}`，“google”“gooo” 等都能被匹配，因为它们中 `o` 的出现次数都大于等于 2 次。
- \{n,m\}：匹配前导字符出现的次数介于 `n` 次与 `m` 次之间（包含 `n` 和 `m`）。比如 `go\{2,3\}`，“google” 和 “gooo” 可以被匹配，因为 `o` 的出现次数在 2 到 3 次这个区间内。
- (strings)：用于保存被匹配的字符，在一些文本编辑工具（如 vim）中常与替换命令结合使用。例如在 vim 中，`:%s/10.1\(.1.1\)/192.2\1/g` 这条命令，会将匹配到的 `10.1` 后面跟着的 `.1.1` 部分保存（即 `\1` 代表 `.1.1`），然后将 `10.1` 替换为 `192.2`，并保留后面匹配的部分，最终实现类似将 `10.1.1.1` 替换为 `192.2.1.1` 的效果。
- 预定义字符组：`[0 - 9]` 匹配任意一个数字；`[a - z]` 匹配任意一个小写字母；`[A - Z]` 匹配任意一个大写字母；`[a - zA - Z]` 匹配任意一个字母；`[a - Z]` 这种写法在大多数正则表达式实现中等同于 `[a - zA - Z]`，匹配任意一个字母。

◦ Perl 内置正则表达式特殊字符

- \d：等同于 `[0 - 9]`，用于匹配数字。例如在文本 “123abc” 中，`\d` 可以匹配到 `1`、`2`、`3`。
- \w：匹配字母、数字以及下划线，即 `[a - zA - Z0 - 9_]`。比如在 “abc_123” 中，`\w` 能匹配到其中的每个字符。
- \s：匹配空格、制表符、换页符等空白字符，对应 `[\t\r\n]`。在文本 “apple\tbanana\ncherry” 中，`\s` 可以匹配到 `\t` 和 `\n`。

```
[root@server ~]# cat 1.txt
ggle
gogle
google
gooogle
goooooogle
gooooooogle
taobao.com
```

```
taotaobaobao.com

jingdong.com
dingdingdongdong.com
10.1.1.1

Adfjd8789JHfdsdf/
a87fdjfkdLKJK


7kdjfd989KJK;
bSKJjkksdjf878.
cidufKJHJ6576,
hello world
helloworld yourself

```

实例

```
[root@localhost ~]# grep 'go*' 1.txt
[root@localhost ~]# grep 'go.' 1.txt
[root@localhost ~]# grep 'goo.*' 1.txt
[root@localhost ~]# grep '[abc]' 1.txt
[root@localhost ~]# grep '[^abc]' 1.txt
[root@localhost ~]# grep '^[abc]' 1.txt
[root@localhost ~]# grep '^[^abc]' 1.txt
[root@localhost ~]# grep '\<hello' 1.txt
[root@localhost ~]# grep 'word>\' 1.txt
[root@localhost ~]# grep '\<world\>' 1.txt
[root@localhost ~]# grep -w 'world' 1.txt
[root@localhost ~]# grep 'go\{2\}' 1.txt
[root@localhost ~]# grep 'go\{2,\}' 1.txt
[root@localhost ~]# grep 'go\{2,3\}' 1.txt
[root@localhost ~]# grep '[0-9]' 1.txt
[root@localhost ~]# grep -P '\d' 1.txt
[root@localhost ~]# grep '[a-zA-Z0-9_]' 1.txt
[root@localhost ~]# grep -P '\w' 1.txt
```

3）扩展正则表达式

- 在使用正则表达式进行文本匹配时，可以借助 grep 命令搭配 -E 选项，或者直接使用 egrep 工具来支持扩展正则表达式。扩展正则表达式提供了比基础正则表达式更丰富的匹配功能。
- 扩展正则表达式元字符及示例
  - +：用于匹配一个或多个前导字符。
    - 例如，正则表达式bo+ 可以匹配 “bo” 以及 “boo” 等字符串。因为 + 要求前导字符 o 至少出现一次，所以 “b” 无法被匹配，而 “bo” 中 o 出现了 1 次，“boo” o 出现了 2 次，都满足匹配条件。
  - ?：该元字符用于匹配零个或一个前导字符。以正则表达式bo? 为例
    - 它能够匹配 “b” 和 “bo”。对于 “b”，o 出现了 0 次；对于 “bo”，o 出现了 1 次，均符合 ? 的匹配规则。
  - a|b：此为或匹配模式，表示匹配 a 或者 b。
    - 例如，正则表达式 apple|banana 可以匹配字符串 “apple” 或者 “banana”。
  - ()：用于将字符组合成一个整体，也就是组字符。
    - 例如在hello myself yourself (my|your)self 这个表达式中，(my|your) 构成了一个组，它可以匹配 “my” 或者 “your”，进而整个表达式可以匹配 “myself” 或者 “yourself”。
  - {n}：表示前导字符重复 n 次。
    - 例如，正则表达式 go{2} 可以匹配 “google” 中的 “goo” 部分，因为这里前导字符 o 重复了 2 次。
  - {n,}：意味着前导字符重复至少 n 次。
    - 例如，正则表达式 go{2,} 可以匹配 “google”（o 重复 2 次）、“gooo”（o 重复 3 次）等字符串，只要 o 的重复次数大于等于 2 次即可。
  - {n,m}：表示前导字符重复的次数在 n 到 m 次之间（包含 n 和 m）。
    - 例如，正则表达式 go{2,3} 可以匹配 “google”（o 重复 2 次）和 “gooo”（o 重复 3 次），但无法匹配 “go”（o 重复 1 次）或者 “goooo”（o 重复 4 次）。

```
#举例
[root@localhost ~]# egrep 'go+' 1.txt
[root@localhost ~]# grep -E 'go+' 1.txt
[root@localhost ~]# egrep 'go?' 1.txt
[root@localhost ~]# egrep '^go|world' 1.txt
[root@localhost ~]# egrep '(go)+' 1.txt
[root@localhost ~]# egrep '(taobao|jingdong).com' 1.txt
[root@localhost ~]# egrep 'go{2}' 1.txt
[root@localhost ~]# egrep 'go{2,3}' 1.txt
```

作业（AWK）：
获取当前IP地址、子网掩码、广播地址
使用awk工具统计使用bash作为解释器的用户数量

统计还有多少空闲内存（仅统计物理内存的剩余容量）

统计磁盘使用率（仅统计根目录的可用容量）

#课后作业（正则）

1、查找不以大写字母开头的行。
2、查找有数字的行
3、查找一个数字和一个字母连起来的
4、查找不以r开头的行
5、查找以数字开头的
6、查找以大写字母开头的
7、查找以小写字母开头的
8、查找以点结束的
9、去掉空行
10、查找完全匹配abc的行
11、查找A后有三个数字的行

```
#新建文本文件，完成上述要求
# cat a.txt
Aieur459Root0000
9h847RkjfkIIIhello

rootHllow88000dfjj


8ikuifhupliooking
hello world
192.168.0.254
welcome to uplooking.

abcderfkdjfkdtest
rlllA899kdfkdfj
iiiA848890ldkfjdkfj
abc
1234567890837498483
123456@qq.com
123456@163.com
abcdefg@itcast.com23ed
```