第3天-Shell 数组并发正则

一、Shell 编程之数组

1、什么是数组

Shell 的数组就是把有限个元素（变量或字符内容）用一个名字命名，然后用编号对它们进行区分的元素集合。这个名字就称为数组名，用于区分不同内容的编号就称为数组下标。组成数组的各个元素（变量）称为数组的元素，有时也称为下标变量。有了Shell数组后，就可以用相同名字引用一系列变量及变量值，并通过数字（索引）来识别使用它们。在许多场合，使用数组可以缩短和简化程序开发。数组的本质还是变量，是特殊的变量形式

2、数组的操作

1、 数组的定义

- 方法一：用小括号将变量值括起来赋值给数组变量，每个变量值之间要用空格分隔

```
[root@qfedu.com ~]# array=(value1 value2 value3 ... )
示例：
[root@qfedu.com ~]# array=(1 2 3)     # 用小括号将数组内容赋值给数组变量，数组元素用“空格”分隔开。
[root@qfedu.com ~]# echo ${array[*]} # 输出上面定义的数组的所有元素值，注意语法。
```

- 方法二：用小括号将变量值括起来，同时采用键值对的形式赋值

```
[root@qfedu.com ~]# array=([0]=one [1]=two [3]=three [4]=four)
```

- 方法三：通过分别定义数组变量的方法来定义

```
[root@qfedu.com ~]# arry[0]=one
[root@qfedu.com ~]# arry[1]=two
[root@qfedu.com ~]# arry[2]=three
[root@qfedu.com ~]# arry[3]=four
[root@qfedu.com ~]# echo ${arry[@]} #查看所有数组的值
one two three four
```

- 方法四：动态地定义数组变量，并使用命令的输出结果作为数组的内容

```
array=($(命令))
array=(`命令`)
示例：
[root@qfedu.com ~]# arry=(`ls ./`)
[root@qfedu.com ~]# echo ${arry[@]}
10.sh 172.16.1.7 1.sh 2.sh 3.sh 4.sh 5.sh 6.sh 7.sh 8.sh access_2010-12-8.log
rsync.sh
```

2、数组中常用变量

```
${ARRAY_NAME[INDEX]}        # 引用数组中的元素 注意：引用时，只给数组名，表示引用下标为0的元素
${#ARRAY_NAME[*]}           # 数组中元素的个数
${#ARRAY_NAME[@]}           # 数组中元素的个数

${ARRAY_NAME[*]}            # 引用数组中的所有元素
${ARRAY_NAME[@]}            # 引用数组中的所有元素
${#ARRAY_NAME}              # 数组中下标为 0 的字符个数
```

3、数组的打印

```
[root@qfedu.com ~]# array=(one two three)
[root@qfedu.com ~]# echo ${array[0]}    # 打印单个数组元素用${数组名[下标]}，当未指定数组下标时，数组的下标是从0开始。
one
[root@qfedu.com ~]# echo ${array[1]}
two
[root@qfedu.com ~]# echo ${array[2]}
three
[root@qfedu.com ~]# echo ${array[*]}    # 使用*或者@可以得到整个数组内容。
one two three
[root@qfedu.com ~]# echo ${array[@]}    # 使用*或者@可以得到整个数组内容。
one two three
```

4、数组元素的个数的打印

```
[root@qfedu.com ~]# echo ${array[*]}    # 使用*或者@可以得到整个数组内容。
one two three
[root@qfedu.com ~]# echo ${#array[*]}   # 用${#数组名[@或*]}可以得到数组长度，这和前文讲解的变量子串知识是一样的，因为数组也是变量，只不过是特殊的变量，因此也适合变量的子串替换等知识。
3
[root@qfedu.com ~]# echo ${array[@]}    # 使用*或者@可以得到整个数组内容。
one two three
[root@qfedu.com ~]# echo ${#array[@]}   # 用${#数组名[@或*]}可以得到数组长度
3
```

5、数组赋值

直接通过"数组名[下标] "对数组进行引用赋值，如果下标不存在，自动添加新一个数组元素，如果下标存在就覆盖原来的值

数组中元素的赋值方式主要有几个方式：

```
# 一次只能赋值一个元素
[root@qfedu.com ~]# ARRAY_NAME[INDEX]=value
#  一次赋值全部元素
[root@qfedu.com ~]# ARRAY_NAME=("VAL1"  "VAL2"  "VAL3"  ...)
# 只赋值特定元素
[root@qfedu.com ~]# ARRAY_NAME=([0]="VAL1"  [3]="VAL4" ...)
# 交互式赋值
[root@qfedu.com ~]# read  -a  ARRAY_NAME

[root@qfedu.com ~]# array=(one two three)
[root@qfedu.com ~]# echo ${array[*]}
```

```
one two three
[root@qfedu.com ~]# array[3]=four     # 增加下标为3的数组元素
[root@qfedu.com ~]# echo ${array[*]}
one two three four
[root@qfedu.com ~]# array[0]=qfedu    # 修改数组元素
[root@qfedu.com ~]# echo ${array[*]}
 qfedu two three four
```

6、数组的删除

因为数组本质上还是变量，因此可通过“unset 数组[下标]”清除相应的数组元素，如果不带下标，表示清除整个数组的所有数据

```
[root@qfedu.com ~]# echo ${array[*]}
 qfedu two three four
[root@qfedu.com ~]# unset array[1]        # 取消下标为1的数组元素
[root@qfedu.com ~]# echo ${array[*]}      # 打印输出后发现数组元素“two”，不见了
 qfedu three four
[root@qfedu.com ~]# unset array           # 删除整个数组
[root@qfedu.com ~]# echo ${array[*]}
                                          # 没有任何内容了。
```

7、数组的截取

```
[root@qfedu.com ~]# arrary=(1 2 3 4 5)
[root@qfedu.com ~]# echo ${arrary[@]:1:3 }  # 从下标为1的元素开始截取，共取3个数组元素
2 3 4
[root@qfedu.com ~]# arrary=({a..z})
[root@qfedu.com ~]# echo ${arrary[@]}
a b c d e f g h i j k l m n o p q r s t u v w x y z
[root@qfedu.com ~]# echo ${arrary[@]:3:3}   # 从下标为1的元素开始截取，共取3个数组元素。
d e f
[root@qfedu.com ~]# echo ${arrary[@]:0:3}   # 从下标为0的元素开始截取，共取3个数组元素
a b c
```

8、数组的替换

```
[root@qfedu.com ~]# arrary=(1 2 3 4)
[root@qfedu.com ~]# echo ${arrary[@]}
1 2 3 4
[root@qfedu.com ~]# echo ${arrary[@]/2/oldoby} # 把数组中的1替换成b，原数组未被修改，和sed很像。
1 oldoby 3 4
```

9、数组元素删除

```
[root@qfedu.com ~]# array=(one two three four five)
[root@qfedu.com ~]# echo ${array[@]}
one two three four five
[root@qfedu.com ~]# echo ${array[@]#o*}    # 从左边开始匹配最短的，并删除。
ne two three four five
[root@qfedu.com ~]# echo ${array1[@]##o*}  # 从左边开始匹配最长的，并删除。
two three four five
[root@qfedu.com ~]# echo ${array[@]%f*}    # 从右边开始匹配最短的，并删除。
one two three
[root@qfedu.com ~]# echo ${array[@]%%f*}   # 从右边开始匹配最长的，并删除。
one two three
```

2、数组切片

1、命令格式

```
${ARRAY_NAME[@]:offset:number}
    # offset：要路过的元素个数
    # number：要取出的元素个数；省略number时，表示取偏移量之后的所有元素
```

2、实例

```
[root@qfedu.com ~] echo ${array[@]:0:1}
Zero
[root@qfedu.com ~] echo ${array[@]:0:2}
Zero One
[root@qfedu.com ~] echo ${array[@]:0:3}
Zero One Two
[root@qfedu.com ~] echo ${array[@]:1}
One Two
[root@qfedu.com ~] echo ${array[@]:2}
Two
[root@qfedu.com ~]# echo ${#array[*]}
3
[root@qfedu.com ~]# echo ${#array[@]}
3
[root@qfedu.com ~]# echo ${array[*]}
Zero One Two
[root@qfedu.com ~]# echo ${#array}
8
[root@qfedu.com ~]# echo ${array[0]}
Zero
```

3、遍历数组

- 创建一个数组 array=(A B C D 1 2 3 4)

1、标准的 for 循环

```
for(( i=0;i<${#array[@]};i++))
do
# ${#array[@]}  获取数组长度用于循环
echo ${array[i]}
done
```

2、for ... in 循环（不带数组下标）

```
for var in ${array[@]}
# 也可以写成for var in ${array[*]}
do
    echo $var
done
```

3、While 循环

```
i=0
while [ $i -lt ${#array[@]} ]
#当变量（下标）小于数组长度时进入循环体
do
    echo ${ array[$i] }
    # 按下标打印数组元素
    let i++
done

#while 遍历数组
i=0
while  [[ i -lt ${#arrayIndex[@]} ]];do
  echo ${arrayIndex[i]}
  let i++
done
```

4、关联数组

- Bash 支持关联数组，它可以使用字符串作为数组索引，关联数组一定要事先声明才行，不然会按照索引数组进行执行

1、定义关联数组

- 使用声明语句将一个变量声明为关联数组。

```
[root@qfedu.com ~]# declare -A assArray
```

- 声明之后，可以有两种方法将元素添加到关联数组中。

1、利用内嵌索引-值列表的方法

```
[root@qfedu.com ~]# assArray=([lucy]=beijing [yoona]=shanghai)l
[root@qfedu.com ~]# echo ${assArray[lucy]}
beijing
```

2、使用独立的索引-值进行赋值

```
[root@qfedu.com ~]# assArray[lily]=shandong
[root@qfedu.com ~]# assArray[sunny]=xian
[root@qfedu.com ~]# echo ${assArray[sunny]}
xian
[root@qfedu.com ~]# echo ${assArray[lily]}
shandong
```

2、列出数组索引

- 每一个数组都有一个索引用于查找。使用${!数组名[@或者*]}获取数组的索引列表

```
[root@qfedu.com ~]# echo ${!assArray[*]}
lily yoona sunny lucy
[root@qfedu.com ~]# echo ${!assArray[@]}
lily yoona sunny lucy
```

3、获取所有键值对

```
[root@qfedu.com ~]# cat array.sh
#! /bin/bash
declare -A cityArray
cityArray=([yoona]=beijing [lucy]=shanghai [lily]=shandong)
for key in ${!cityArray[*]}
do
  echo "${key} come from ${cityArray[$key]}"
done

[root@qfedu.com ~]# bash array.sh
lily come from shandong
yoona come from beijing
lucy come from shanghai
```

5、Shell 数组脚本开发实践

1、使用循环批量输出数组的元素

```
[root@qfedu.com ~]# cat array1.sh
#!/bin/bash
#################################################################
# File Name: array1.sh
# Version: V1.0
# Author: qfedu
# Organization:
# Created Time :
# Description:
#################################################################
#普通方式
arrary=(1 2 3 4 5)
for i in ${arrary[@]}
do
    echo $i
done
#c语言方式
arrary1=(1 2 3 4 5)
for ((i=0;i<${#arrary1[@]};i++))    # 从数组的第一个小标0开始，循环数组的所有下标
do
    echo ${arrary1[i]}              # 打印数组元素
done
# while循环打印
arrary2=(1 2 3 4 5)
i=0
while ((i<${#arrary2[@]}))
```

```
do
    echo ${arrary2[i]}
    ((i++))
done
```

2、通过竖向列举法定义数组元素并批量打印

```
[root@qfedu.com ~]# cat array2.sh
#!/bin/bash
#################################################################
# File Name: array2.sh
# Version: V1.0
# Author: qfedu
# Organization:
# Created Time :
# Description:
#################################################################
arrary=(
     qfedu
     qfedu.com
     yunjisuan
     anquan
)
for ((i=0;i<${#arrary[*]};i++))
do
    echo "This is num $i,then content is ${arrary[i]}"
done
```

3、把命令结果作为数组元素定义并打印

```
[root@qfedu.com ~]# cat array3.sh
#!/bin/bash
#################################################################
# File Name: array3.sh
# Version: V1.0
# Author: qfedu
# Organization:
# Created Time :
# Description:
#################################################################
arrary=(`ls / qfedu`)
for ((i=0;i<${#arrary[*]};i++))
do
    echo "This is num $i,then content is ${arrary[i]}"
done
echo  --------------------
arrary1=($(ls / qfedu))
num=0
for i in ${arrary1[*]}
do
    echo "This is num $num,then content is $i"
    ((num++))
done
```

二、Bash 并发及并发控制

默认的情况下，Shell脚本中的命令是串行执行的，必须等到前一条命令执行完后才执行接下来的命令，但是如果我有一大批的的命令需要执行，而且互相又没有影响的情况下（有影响的话就比较复杂了），那么就要使用命令的并发执行了。

1、初始脚本

```
#!/bin/bash
for(( i = 0; i < ${count}; i++ ))
do
        commands1
done
commands2
```

- 上面的脚本，因为每个commands1都挺耗时的，所以打算使用并发编程，这样就可以节省大量时间了。

2、修改后脚本

```
#!/bin/bash

for(( i = 0; i < ${count}; i++ ))
do
{
        commands1
}&
done

commands2
```

- 这样的话 commands1 就可以并行执行了。 实质是将 commands1 作为后台进程在执行，这样主进程就不用等待前面的命令执行完毕之后才开始执行接下来的命令。
- 但是本来目的是让 commands1 的这个循环都执行结束后，再用 command2 去处理前面的结果。如果像上面这样写的话，在 commands1 都还没结束时就已经开始执行commands2 了，得到了错误的结果。

3、再次修改脚本

```
#!/bin/bash
for(( i = 0; i < ${count}; i++ ))
do
{
        commands1
}&
done
wait
commands2
```

- 这样就可以达到预期的目的了，先是所有的commands1在后台并行执行，等到循环里面的命令都结束之后才执行接下来的 commands2。
- 上面的脚本，如果 count 值特别大的时候，应该控制并发进程的个数，不然会影响系统其他进程的运行，甚至死机。

4、控制进程的数量

- 如何控制进程的数量，主要都是利用管道的思想，实现脚本如下

```
#!bin/bash

PRONUM=10                  # 进程个数

tmpfile="$$.fifo"          # 临时生成管道文件
mkfifo $tmpfile
exec 6<>$tmpfile
rm $tmpfile

for(( i=0; i<$PRONUM; i++ ))
do
        echo "init."
done >&6

for(( i = 0; i < ${count}; i++ ))
do
        read line
        #echo $line
        {
                commands
                echo "line${i} finished."
        } >&6 &
done <&6

wait
```

初始时给管道内写入 PRONUM 个字符串，然后每从管道内读出一个字符串就生成一个子进程，当管道内没有字符串可读时就阻塞在那里，不能创建新的子进程，一直等到有新的字符串进来时才继续运行。当每个并发进程执行完毕时又向管道内写入一个字符串，表示当前子进程已执行完毕，可以创建新的子进程了。

5、文件描述符

- fd 进程打开的文件

1、查看当前进程打开的文件

```
[root@qfedu.com ~]# ll /proc/$$/fd
```

2、自定义当前进程用描述符号操作文件

- 手动定义文件描述符,只要没有被占用可以使用
- 自定义用当前进程打开一个文件

  ```
  [root@qfedu.com ~]# exec 6<> /file
  ```

- 自定义用当前进程关闭一个文件

  ```
  [root@qfedu.com ~]# exec 6<&-
  ```

3、给文件写入内容

- 文件表述符号（fd）的方式

```
[root@qfedu.com ~]# echo "1111" >> /proc/$$/fd/6
```

- 文件名的方式

```
[root@qfedu.com ~]# echo "1111">> /file
```

4、文件描述符恢复文件

- 删除源文件

```
[root@qfedu.com ~]# rm -rf /file
```

- 新文件产生新的文件 inode

```
[root@qfedu.com ~]# cp /proc/$$/fd/6 /file
```

- 查看状态为（deleted）

```
[root@qfedu.com ~]# ll /proc/$$/fd
```

- 释放文件描述符，重新读取就显示正常了

6、管道

1、匿名管道不能跨终端（|管道）

2、命名管道

- 创建命名管道文件（不是普通文件，管道内的文件只能读取一次，不会永久保存）

```
[root@qfedu.com ~]# mkfifo /tmp/fifo1
```

- 查看命名管道文件

```
[root@qfedu.com ~]# cat /tmp/fifo1
```

7、通过文件描述符号和命名管道实现多进程并发控制

```
#!/bin/env bash
# pint02 multi thread
# v1.0 by qfedu

ip=192.168.100.
use=rr
thread=5
tmp_fifofile=/tmp/$$.fifo     # 以当前进程 pid 创建 fifo 文件，防止冲突
mkfifo $tmp_fifofile
exec 8<> $tmp_fifofile        # 定义当前进程打开管道文件（手动指定打开的文件描述符为8）
rm $tmp_fifofile              # 删除管道文件（当前进程没有释放文件描述符8，不影响）
for i in $(seq $thread)       # 给管道文件（描述符号8）里循环写入内容
do
```

```
        echo >&8                # 给管道文件（描述符8）输入$thread个回车（写入任何记录都可以，内容不重要，添加$thread条记录.重定向，追加都可以,管道文件内容不会被覆盖）
done
for i in {1..254}
do
        read -u 8               # read -u 读取一次文件描述符的内容（read 读取不到内容就停止，管道文件只能读取一次，限制读取$thread次）
        {

        useradd $use$i
        echo "111" | passwd --stdin $use$i &>/dev/null
                if [ $? -eq 0 ]; then
                        echo "$use$i is created"

        ping -c1 -W1 $ip$i &>/dev/null
        if [ $? -eq 0 ]; then
                echo "$ip$i is up"
        else
                echo "$ip$i is down"
        fi
        echo >&8                # 给管道文件（描述符号8）写入一个回车（还会一条记录，内容随意）
        } &
done
wait
exec 8<&-
echo "all finish"
```

三、Expect 基本操作

1、Expect 介绍

- 通过 Shell 可以实现简单的控制流功能，如：循环、判断等。但是对于需要交互的场合则必须通过人工来干预，有时候可能会需要实现和交互程序如telnet服务器等进行交互的功能。而Expect就使用来实现这种功能的工具。
- Expect 是一个免费的编程工具语言，用来实现自动和交互式任务进行通信，而无需人的干预。Expect的作者Don Libes 在1990年开始编写Expect时对Expect做有如下定义。
- Expect 是一个用来实现自动交互功能的软件套件 (Expect [is a] software suite for automating interactive tools)。使用它系统管理员 的可以创建脚本用来实现对命令或程序提供输入，而这些命令和程序是期望从终端（terminal）得到输入，一般来说这些输入都需要手工输入进行的。Expect则可以根据程序的提示模拟标准输入提供给程序需要的输入来实现交互程序执行。甚至可以实现实现简单的BBS聊天机器人。
- Expect 是不断发展的，随着时间的流逝，其功能越来越强大，已经成为系统管理员的的一个强大助手。Expect 需要 Tcl 编程语言的支持，要在系统上运行 Expect 必须首先安装Tcl。

2、Expect工作原理

- Expect 的工作方式象一个通用化脚本工具。用来实现计算机之间需要建立连接时进行特定的登录会话的自动化。
- 脚本由一系列 expect-send 对组成：expect 等待输出中输出特定的字符，通常是一个提示符，然后发送特定的响应。解决人机交互的问题

3、Expect 安装

```
[root@qfedu.com ~]# yum install -y expect tcl tclx tcl-devel
```

4、Expect 命令

Expect 的核心是 spawn、expect、send、set。

- spawn 调用要执行的命令
- expect 监听交互输出
- send 进行交互输入
- set 设置变量值
- interact 交互完后，将控制权交给控制台。
- expect eof，与 spawn 对应，表示捕捉终端输出信息终止，类似 if...endif
- settimeout -1，设置 expect 永不超时
- settimeout 300，如果 300 后没有捕捉到 expect 的监听的内容，那么就退出

5、Expect 的语法

- Expect 使用的是 tcl 语法，语法结构如下

1、命令之后是参数，互相用空格间隔

```
cmd arg arg arg
```

2、使用变量

```
set var 1000  # 定义变量<br>set var [lindex $argv 0] #将 var = argv[0]<br>$foo
```

3、嵌套命令

- 将一个命令的输出，作为另一个命令的输入参数

```
cmd1 [cmd2 arg]
```

4、双引号

- 将词组标记为一个参数，双引号内$符号有效

```
cmd "hello world $foo"
```

5、大括号

- 将词组标记为一个参数，但大括号内无法扩展变量

```
cmd {hello world}
```

6、反斜线，转义

6、Expect 实例解决 ssh 登录验证免交互

1、命令选项说明

- spawn expect 内部命令，启动一个shell程序。
- expect 期望哪些内容
- yes/no 就send发送 yes ，\r 表示回车
- password 就 send 发送 centos
- exp_continue，跳过循环，就继续下一条语句。
- interact 允许用户交互

2、Expect 实现免交互公钥推送

1、安装和生成公钥

```
#!/bin/bash
#创建一个IP地址文件。
>ip.txt
#检测expect是否安装，检测公钥是否创建。
rpm -q expect &> /dev/null
if [ $? -ne 0 ] ;thenv
        yum install -y expect tcl tclx tcl-devel
fi
if [ ! -f ~/.ssh/id_rsa ];then
        ssh-keygen -P "" -f ~/.ssh/id_rsa
fi
#使用for循环ping测试主机是否在线。之前插入安装和准备秘钥。
```

注意！！！：缩进绝对不能用空格，必须回车

2、通过 shell 循环判断在线主机

```
#!/bin/bash
#创建一个IP地址文件。
>ip.txt
# 使用for循环ping测试主机是否在线。
for i in {2..254}
do
{
# 请注意练习环境的IP地址，可能与示例中不同。
        ip=192.168.0.$i
        ping -c1 -W1 $ip &> /dev/null
        if [ $? -eq 0 ];then
        echo "$ip" >> ip.txt
        fi
        }&
done
```

3、通过 expect 进行交互

- 使用 expect 解释器

```
#!/usr/bin/expect
set timeout 10
spawn ssh-copy-id 192.168.0.2
expect {
        "yes/no" { send "yes\r"; exp_continue }
        "password:" { send "centos\r" }
}
```

- 使用 shell 脚本

```
#!/bin/bash
# 创建一个IP地址文件。
>ip.txt
# 使用 for 循环 ping 测试主机是否在线。
for i in {2..254}
```

```
do
 {
# 请注意练习环境的 IP 地址，可能与示例中不同。
        ip=192.168.122.$i
        ping -c1 -W1 $ip &> /dev/null
        if [ $? -eq 0 ];then
        echo "$ip" >> ip.txt

/usr/bin/expect <<-EOF
set timeout 10
spawn ssh-copy-id $ip
expect {
        "yes/no" { send "yes\r"; exp_continue }
        "password:" { send "centos\r" }
}
expect eof
EOF
        fi
        }&
done
wait
echo "finish..."
```

四、Grep 命令

1、grep 介绍

- grep 系列是 Linux 中使用频率最高的文本查找命令。主要功能在一个或者多个文件中查找特定模式的字符串。如果该行有匹配的字符串，则输出整个行的内容。如果没有匹配的内容，则不输出任何内容。grep命令不改动源文件。
- Linux的grep家族包括grep、egrep、fgrep、rgrep。grep 可以通过 -G、-E、-F 命令行选项来使用 egrep 和 fgrep 的功能。
- grep：在文件中全局查找指定的正则表达式，并打印所有包含该表达式的行
- egrep：　扩展的egrep，支持更多的正则表达式元字符
- fgrep：固定grep(fixed grep)，有时也被称作快速(fast grep)，它按字面解释所有的字符

2、grep 命令格式

```
grep [选项] PATTERN filename filename ...
```

- 在每个 FILE 或是标准输入中查找 PATTERN。默认的 PATTERN 是一个基本正则表达式(缩写为 BRE)。

```
[root@qfedu.com ~]# grep -i 'hello world' menu.h main.c
```

3、grep 命令选项

FILE 文件控制		
-B	--before-context=NUM	打印以文本起始的NUM 行
-A	--after-context=NUM	打印以文本结尾的NUM 行
-C	--context=*NUM*	打印输出文本NUM 行
-NUM		等同 --context=NUM
	--color[=WHEN] --colour[=WHEN]	高亮颜色突出显示搜索的字符串。值'always', 'never', or 'auto'。
-U	--binary	将文件作为二进制文件处理。仅有MS-DOS和MS-Windows支持该选项
-u	--unix-byte-offsets	报告UNIX风格的字节偏移。这个选项仅在同时使用-b选项的情况下才有效；仅有MS-DOS和MS-Windows支持该选项
与 PATTERN 正则表达式相关的选项		
-E	--extended-regexp	PATTERN 是一个可扩展的正则表达式(缩写为 ERE)
-F	--fixed-strings	PATTERN 是一组由断行符分隔的定长字符串
-G	--basic-regexp	PATTERN 是一个基本正则表达式(缩写为 BRE)
-P	--perl-regexp	PATTERN 是一个 Perl 正则表达式
-e	--regexp= PATTERN	用 PATTERN 来进行匹配操作
-f	--file=FILE	从 FILE 中取得 PATTERN
-i	--ignore-case	忽略大小写
-w	--line-regexp	强制 PATTERN 仅完全匹配字词
-x	--extended-regexp	强制 PATTERN 仅完全匹配一行
-z	--null-data	一个 0 字节的数据行，但不是空行
输出控制选项		
-m	--max-count=NUM	在找到指定数量的匹配行后停止读文件
-b	--byte-offset	在显示符合样式的那一行之前，标示出该行第一个字符的编号

FILE 文件控制		
-n	--line-number	在显示符合样式的那一行之前，标示出该行的列数编号
	--line-buffered	刷新输出的每一行
-H	--with-filename	在显示符合样式的那一行之前，表示该行所属的文件名称
-h	--no-filename	在显示符合样式的那一行之前，不标示该行所属的文件名称
	--label=LABEL	打印标签作为文件名的标准输入(主要用于管道处理) 例如：cat test |grep --label=test -H 123
-o	--only-matching	仅输出匹配行的匹配部分
-q	--quiet --silent	抑制所有正常输出
	--binary-files=TYPE	假定二进制文件为TYPE类型文件TYPE可以为binary、text或without-match
-a	--text	等价于-binary-files=text
-I	--binary-files=without-match	等价于--binary-files=without-match
-d	--directories=ACTION	当grep的对象为目录时用，处理目录可以读取、递归或跳过
-D	--devices=ACTION	当grep的对象为处理设备、栈或套接字时必须用，处理对象可以读取或跳过
-r -R	--recursive --directories=recurse	相当于--directories=recurse 遍历目录
	--include=FILE_PATTERN	仅grep匹配的文件模式的文件
	--exclude=FILE_PATTERN	跳过匹配的文件模式的文件和目录进行grep匹配
	--exclude-from=FILE	跳过任一匹配文件模式的文件
	---exclude-dir=PATTERN	跳过匹配的目录文件目录
-L	--files-without-match	仅仅打印未匹配的文件的文件名
-l	--files-with-matches	仅仅打印匹配的文件的文件名
-c	--count	仅仅打印每个文件的匹配次数
-T	--initial-tab	将标签排队（标签即文件名）

FILE 文件控制		
-Z	--null	打印文件名，文件名与匹配行中间没有空字节 -z与-Z的区别之一：当一个文件有多个匹配行时-z只打印一次文件名，而-Z每匹配一次打印一次文件名
杂项		
-s	--no-messages	不显示错误信息
-v	--invert-match	打印不匹配的行

- ‘egrep’即‘grep -E’。‘fgrep’即‘grep -F’。直接使用‘egrep’或是‘fgrep’均已不可行了。
- 不带 FILE 参数，或是 FILE 为 -，将读取标准输入。如果少于两个 FILE 参数。
- 就要默认使用 -h 参数。如果选中任意一行，那退出状态为 0，否则为 1；如果有错误产生，且未指定 -q 参数，那退出状态为 2。

4、grep 实例

1、查找指定进程

```
[root@qfedu.com ~]# ps -ef | grep svn
root 4943   1      0  Dec05 ?   00:00:00 svnserve -d -r /opt/svndata/grape/
root 16867 16838  0 19:53 pts/0    00:00:00 grep svn
```

- 第一条记录是查找出的进程；第二条结果是grep进程本身，并非真正要找的进程。

2、查找指定进程个数

```
[root@qfedu.com ~]# ps -ef|grep svn -c
2
[root@qfedu.com ~]# ps -ef|grep -c svn
2
```

3、从文件中读取关键词进行搜索

```
[root@qfedu.com test]# cat test.txt
hnlinux
www.qfedu.com
ubuntu
ubuntu linux
redhat
Redhat
linuxmint

[root@qfedu.com test]# cat test2.txt
linux
Redhat

[root@qfedu.com test]# cat test.txt | grep -f test2.txt
hnlinux
ubuntu linux
Redhat
linuxmint
```

- 输出 test.txt 文件中含有从 test2.txt 文件中读取出的关键词的内容行

4、从文件中读取关键词进行搜索且显示行号

```
[root@qfedu.com test]# cat test.txt
hnlinux
www.qfedu.com
ubuntu
ubuntu linux
redhat
Redhat
linuxmint

[root@qfedu.com test]# cat test2.txt
linux
Redhat

[root@qfedu.com test]# cat test.txt | grep -nf test2.txt
1:hnlinux
4:ubuntu linux
6:Redhat
7:linuxmint
```

- 输出 test.txt 文件中含有从 test2.txt 文件中读取出的关键词的内容行，并显示每一行的行号

5、从文件中查找关键词

```
[root@qfedu.com test]# grep 'linux' test.txt
hnlinux
ubuntu linux
linuxmint

[root@qfedu.com test]# grep -n 'linux' test.txt
1:hnlinux
4:ubuntu linux
7:linuxmint
```

6、从多个文件中查找关键词

```
[root@qfedu.com test]# grep -n 'linux' test.txt test2.txt
test.txt:1:hnlinux
test.txt:4:ubuntu linux
test.txt:7:linuxmint
test2.txt:1:linux

[root@qfedu.com test]# grep 'linux' test.txt test2.txt
test.txt:hnlinux
test.txt:ubuntu linux
test.txt:linuxmint
test2.txt:linux
```

- 多文件时，输出查询到的信息内容行时，会把文件的命名在行最前面输出并且加上":"作为标示符

7、grep 不显示本身进程

```
[root@qfedu.com test]# ps aux| grep ssh
root   2720  0.0  0.0  62656  1212 ?      Ss   Nov02   0:00 /usr/sbin/sshd
root  16834  0.0  0.0  88088  3288 ?      Ss   19:53   0:00 sshd: root@pts/0
root  16901  0.0  0.0  61180   764 pts/0  S+   20:31   0:00 grep ssh
[root@qfedu.com test]# ps aux|grep \[s]sh]
[root@qfedu.com test]# ps aux|grep \[s]sh
root   2720  0.0  0.0  62656  1212 ?      Ss   Nov02   0:00 /usr/sbin/sshd
root  16834  0.0  0.0  88088  3288 ?      Ss   19:53   0:00 sshd: root@pts/0
[root@qfedu.com test]# ps aux | grep ssh | grep -v "grep"
root   2720  0.0  0.0  62656  1212 ?      Ss   Nov02   0:00 /usr/sbin/sshd
root  16834  0.0  0.0  88088  3288 ?      Ss   19:53   0:00 sshd: root@pts/0
```

8、找出已u开头的行内容

```
[root@qfedu.com test]# cat test.txt |grep ^u
ubuntu
ubuntu linux
```

9、输出非u开头的行内容

```
[root@qfedu.com test]# cat test.txt |grep ^[^u]
hnlinux
www.qfedu.com
redhat
Redhat
linuxmint
```

10、输出以 hat 结尾的行内容

```
[root@qfedu.com test]# cat test.txt |grep hat$
redhat
Redhat

[root@qfedu.com test]# ifconfig eth0|grep "[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\
{1,3\}\.[0-9]\{1,3\}"
          inet addr:192.168.120.204  Bcast:192.168.120.255  Mask:255.255.255.0
[root@qfedu.com test]# ifconfig eth0|grep -E "([0-9]{1,3}\.){3}[0-9]"
          inet addr:192.168.120.204  Bcast:192.168.120.255  Mask:255.255.255.0
```

11、显示包含 ed 或者 at 字符的内容行

```
[root@qfedu.com test]# cat test.txt |grep -E "peida|com"
www.qfedu.com

[root@qfedu.com test]# cat test.txt |grep -E "ed|at"
redhat
Redhat
```

12、显示当前目录下面以 .txt 结尾的文件中的所有包含每个字符串至少有7个连续小写字符的字符串的行

```
[root@qfedu.com test]# grep '[a-z]\{7\}' *.txt
test.txt:hnlinux
test.txt:www.qfedu.com
test.txt:linuxmint
```

13、grep -E 或 egrep 实例

```
[root@qfedu.com ~]# egrep'ifcfg' /etc/*                          # 文件
[root@qfedu.com ~]# egrep'root' /etc/passwd /etc/shadow /etc/hosts
/etc/passwd:root:x:0:0:root:/root:/bin/bash
/etc/passwd:operator:x:11:0:operator:/root:/sbin/nologin
/etc/shadow:root:$6$gcO6Vp4t$OX9LmVgpjtur67UQdUYfw7vJW.78.uRXCLIxw4mBk82Z99:7:::

[root@qfedu.com ~]# egrep -l 'root' /etc/passwd /etc/shadow /etc/hosts
/etc/passwd
/etc/shadow

[root@qfedu.com ~]# egrep -n 'root' /etc/passwd /etc/shadow /etc/hosts
/etc/passwd:1:root:x:0:0:root:/root:/bin/bash
/etc/passwd:11:operator:x:11:0:operator:/root:/sbin/nologin
/etc/shadow:1:root:$6$gcO6Vp4t$OX9LmVgpjtur67UQdUy8.M78.uRXCLIxw4mBk82ZrNlxyf54

[root@qfedu.com ~]# egrep'54:04:A6:CE:C2:1F' /etc/sysconfig/*

[root@qfedu.com ~]# egrep-R '54:04:A6:CE:C2:1F' /etc/sysconfig/

[root@qfedu.com ~]# egrep'^IPADDR' /etc/sysconfig/network-scripts/ifcfg-eth0
|egrep -o '[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}'
192.168.2.254
[root@qfedu.com ~]# egrep'^IPADDR' /etc/sysconfig/network-scripts/ifcfg-eth0
|egrep -o '([0-9]{1,3}\.){3}[0-9]{1,3}'
192.168.2.254

[root@qfedu.com ~]# egrep'NW' datafile
[root@qfedu.com ~]# egrep'NW' d*
[root@qfedu.com ~]# egrep'^n' datafile
[root@qfedu.com ~]# egrep'4$'  datafile
[root@qfedu.com ~]# egrepTB Savage datafile
[root@qfedu.com ~]# egrep'TB Savage' datafile
[root@qfedu.com ~]# egrep'5\..' datafile
[root@qfedu.com ~]# egrep'\.5' datafile
[root@qfedu.com ~]# egrep'^[we]' datafile
[root@qfedu.com ~]# egrep'[^0-9]' datafile
[root@qfedu.com ~]# egrep'[A-Z][A-Z] [A-Z]' datafile
[root@qfedu.com ~]# egrep'ss* ' datafile
[root@qfedu.com ~]# egrep'[a-z]{9}' datafile
[root@qfedu.com ~]# egrep'\<north' datafile
[root@qfedu.com ~]# egrep'\<north\>' datafile
[root@qfedu.com ~]# egrep'\<[a-r].*n\>' datafile
[root@qfedu.com ~]# egrep'^n\w*\W' datafile
[root@qfedu.com ~]# egrep'\bnorth\b' datafile
[root@qfedu.com ~]# egrep'NW|EA' datafile
[root@qfedu.com ~]# egrep'3+' datafile
[root@qfedu.com ~]# egrep'2\.?[0-9]' datafile
[root@qfedu.com ~]# egrep'(no)+' datafile
[root@qfedu.com ~]# egrep'S(h|u)' datafile
[root@qfedu.com ~]# egrep'Sh|u' datafile
```

14、grep 常用方式

```
# 将/etc/passwd，有出现 root 的行取出来
[root@qfedu.com ~]# grep root /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin

[root@qfedu.com ~]# cat /etc/passwd | grep root
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin

# 将/etc/passwd，有出现 root 的行取出来,同时显示这些行在/etc/passwd的行号
[root@qfedu.com ~]# grep -n root /etc/passwd
1:root:x:0:0:root:/root:/bin/bash
30:operator:x:11:0:operator:/root:/sbin/nologin

# 将/etc/passwd，将没有出现 root 的行取出来
[root@qfedu.com ~]# grep -v root /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin

# 将/etc/passwd，将没有出现 root 和nologin的行取出来
[root@qfedu.com ~]# grep -v root /etc/passwd | grep -v nologin
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin

# 用 dmesg 列出核心信息，再以 grep 找出内含 eth 那行,要将捉到的关键字显色，且加上行号来表示
[root@qfedu.com~]# dmesg | grep -n --color=auto 'eth'
247:eth0: RealTek RTL8139 at 0xee846000, 00:90:cc:a6:34:84, IRQ 10
248:eth0: Identified 8139 chip type 'RTL-8139C'
294:eth0: link up, 100Mbps, full-duplex, lpa 0xC5E1
305:eth0: no IPv6 routers present

[root@qfedu.com ~]# grep -R 'ifcfg' /etc                        # 目录
[root@qfedu.com ~]# grep --help |grep '\-R'
  -R, -r, --recursive       equivalent to --directories=recurse

[root@qfedu.com ~]# grep --help |egrep -A5 '\-R'
  -R, -r, --recursive       equivalent to --directories=recurse
      --include=FILE_PATTERN  search only files that match FILE_PATTERN
      --exclude=FILE_PATTERN  skip files and directories matching FILE_PATTERN
      --exclude-from=FILE   skip files matching any file pattern from FILE
      --exclude-dir=PATTERN  directories that match PATTERN will be skipped.
  -L, --files-without-match  print only names of FILEs containing no match
```

五、正则表达式 RE

1、正则表达式应用环境

- 重要的文本处理工具：vim、sed、awk、grep
- 各种语言和应用程序： mysql、oracle、php、python、Apache、Nginx ...

2、什么是正则表达式

- 正则表达式（Regular Expression，通常简称为 regex 或 RE）是一种字符表达方式，可以用它来查找匹配特定准则的文本。在许多编程语言中都有用到正则表达式，常用它来实现一些复杂的匹配。这里简单介绍一下 shell 中常用到的一些正则表达式。
- 正则表达式是对字符串进行操作的一种逻辑公式，即用事先定义好的的一些特定字符以及这些特定字符的组合，组成一个有一定规则的字符串（Regular Expression），使用这个有一定规则的字符串来表达对字符串的一种过滤逻辑。正则表达式被广泛应用于Linux和许多其他编程语言中，而且不论在哪里，其基本原理都是一样的。
- 正则表达式是由两个基本组成部分所建立：一般字符与特殊字符。一般字符是指没有任何特殊意义的字符；特殊字符，常称为元字符（metacharacter），或 meta 字符，正则表达式将匹配被查找行中任何位置出现的相同模式。在正则表达式中，元字符是最重要的概念。在某些情况下，特殊字符也可被视为一般字符（使用转义符 \ 进行转义）。
- POSIX 有两种风格的正则表达式，基本正则表达式（BRE）和扩展正则表达式（ERE）。这两种风格的正则表达式在一些字符含义上有细微的差距。以常用的 grep 指令来说，grep 指令默认支持的是 BRE，若要使用 ERE 进行匹配，可以使用 -E 选项，接下来的例子中均使用 grep 指令来演示正则表达式的使用。
- 正则表达式 Shell 使用场景

```
^[0-9]+$                                                    123 456 5y7
# 匹配数字
[a-z0-9_]+@[a-z0-9]+\.[a-z]+                                qfedu@1000phone.com
# 匹配Mail
[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}
# 匹配IP

# 或
[[:digit:]]{1,3}\.[[:digit:]]{1,3}\.[[:digit:]]{1,3}\.[[:digit:]]{1,3}

[root@qfedu.com ~]# egrep'[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}'
/etc/sysconfig/network-scripts/ifcfg-eth0
IPADDR=172.16.100.1
NETMASK=255.255.255.0
GATEWAY=172.16.100.254
[root@qfedu.com ~]# egrep'[[:digit:]]{1,3}\.[[:digit:]]{1,3}\.[[:digit:]]{1,3}\.
[[:digit:]]{1,3}' /etc/sysconfig/network-scripts/ifcfg-eth0
IPADDR=172.16.100.1
NETMASK=255.255.255.0
GATEWAY=172.16.100.254
```


3、正则表达式元字符和 Shell 通配符

- shell 元字符(也称为通配符)　由shell来解析，如rm -rf .pdf，元字符 Shell将其解析为任意多个字符
- 正则表达式元字符　　由各种执行模式匹配操作的程序来解析，比如vi、grep、sed、awk、python

```
# shell 元字符
[root@qfedu.com ~]# rm -rf *.pdf
# 正则表达式元字符
[root@qfedu.com ~]# grep 'abc*' /etc/passwd
abrt:x:173:173::/etc/abrt:/sbin/nologin

# vim示例
:1,$ s/tom/David/g                # 如tom、anatomy、tomatoes及tomorrow中的“tom”被替换了，而Tom确没被替换
:1,$ s/\<[Tt]om\>/David/g
```

4、正则表达式元字符

字符	BRE/ERE	含义
.	BRE&ERE	匹配任意单个字符（除字符串结束符 NUL）
^	BRE&ERE	匹配行首，如 ^abc，匹配以 abc 开头的字符串
$	BRE&ERE	匹配行尾，如 abc$，匹配以 abc 结尾的字符串
*	BRE&ERE	匹配 0 个或任意多的单个字符，前置字符可以是正则表达式
+	ERE	匹配前面正则表达式的 1 个或多个实例
？	ERE	匹配前面正则表达式的 0 个或 1 个实例
[...]	BRE&ERE	方括号表达式，匹配方括号内的任一字符，常配合 - 符使用，表示匹配一个连续的范围。 ^ 字符作为方括号内的第一个字符表示匹配不在方括号内的任意字符
-	BRE&ERE	连字符，在方括号表达式中使用，表示连续字符的范围（范围会因 locale 而有所不同，因此不具可移植性）
{n,m}	ERE	区间表达式，表示匹配在它前面的字符 n 到 m次。其中，n 与 m 的值必须介于 0-RE_DUM _MAX（含）之间，后者最小值为255
{n}	ERE	表示匹配在这之前的字符 n 次
{n,m}	BRE	功能同 {n,m}
{n}	BRE	功能同{n}
\	BRE&ERE	转义符
()	ERE	匹配位于方括号括起来的正则表达式群
( )	BRE	将(与)\ 之间的模式保存在特殊的“保留空间”中，最多可以存储9个，可以通过后续的转义序列 \n 来匹配保留空间中的模式
\n	BRE	与 ( ) 结合起来使用，\1匹配第一个子模式、 \2匹配第二个，最多到 \9
|	ERE	匹配位于 | 符号前或后的正则表达式

5、正则表达式应用举例

.* 所有字符
^[^] 非字符组内的字符开头的行
[a-z] 小写字母
[A-Z] 大写字母
[a-Z] 小写和大写字母
[0-9] 数字
< 单词头 单词一般以空格或特殊字符做分隔,连续的字符串被当做单词
> 单词尾

1、grep 使用的元字符

注：grep也可以使用扩展集中的元字符，仅需要对这些元字符前置一个反斜线

- grep: 使用基本元字符集 ^, $, ., *, [], [^], < >,(),{}, +, |
- egrep(或grep -E): 使用扩展元字符集 ?, +, { }, |, ( )
- \w 所有字母与数字，称为字符[a-zA-Z0-9] 'l[a-zA-Z0-9]ve' 'l\w*ve'
- \W 所有字母与数字之外的字符，称为非字符 'love[a-zA-Z0-9]+' 'love\W+'
- \b 词边界 '<love>' '\blove\b'
- 使用 grep 命令对 /etc/passwd 文件进行查找匹配操作：

1、匹配一般字符

```
[root@qfedu.com ~]# grep root /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
```

2、使用点字符 " . " 匹配任意字符

```
[root@qfedu.com ~]# grep r..t /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
```

3、使用星号字符 " * " 或问号字符 " ? "匹配0个或多个字符

```
[root@qfedu.com ~]# grep roo* /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
tss:x:59:59:Account used by the trousers package to sandbox the tcsd
daemon:/dev/null:/sbin/nologin
chrony:x:998:996::/var/lib/chrony:/sbin/nologin
```

- grep roo* /etc/passwd 命令将在 /etc/passwd 中匹配 ro ，后面可以接 0 个或多个 o。在 ERE 风格下，使用的是 ? 符号来达到和 * 号一样的效果：

```
[root@qfedu.com ~]# grep -E roo? /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
tss:x:59:59:Account used by the trousers package to sandbox the tcsd
daemon:/dev/null:/sbin/nologin
chrony:x:998:996::/var/lib/chrony:/sbin/nologin
```

4、使用加字符 " + " 匹配1个或多个字符

```
[root@qfedu.com ~]# grep -E roo+ /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin

[root@qfedu.com ~]# grep -E ro+ /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
tss:x:59:59:Account used by the trousers package to sandbox the tcsd
daemon:/dev/null:/sbin/nologin
chrony:x:998:996::/var/lib/chrony:/sbin/nologin
```

- 使用 + 字符可以匹配在其前面的 1 个或多个字符，与 * 字符有些许的差别，另外， + 字符实在 ERE 风格下使用的，故需要使用 grep 命令的 -E 选项

5、使用 ^ 匹配行首，$匹配行尾

```
[root@qfedu.com ~]# grep ^t /etc/passwd
tss:x:59:59:Account used by the trousers package to sandbox the tcsd
daemon:/dev/null:/sbin/nologin
tongye:x:1000:1000:tongye:/home/tongye:/bin/bash

[root@qfedu.com ~]# grep ^t.*h$ /etc/passwd
tongye:x:1000:1000:tongye:/home/tongye:/bin/bash
```

- .* 结合在一起表示匹配零个或多个任意字符，与 ^ 和 $ 结合起来使用的话就可以匹配一个指定开头和结尾的字符串了

6、使用方括号表达式匹配括号内的任一字符

```
[root@qfedu.com ~]# grep [Nn]et /etc/passwd
systemd-network:x:192:192:systemd Network Management:/:/sbin/nologin
```

- 结合 - 字符使用，可以表示匹配一个范围内的任一字符，如 [0-9] 表示匹配 0-9 中的任意一个数字、[a-z] 表示匹配一个小写字母、[A-Z] 表示匹配一个大写字母：

```
[root@qfedu.com ~]# grep [a-z]c /etc/passwd
sync:x:5:0:sync:/sbin:/bin/sync
tss:x:59:59:Account used by the trousers package to sandbox the tcsd
daemon:/dev/null:/sbin/nologin
abrt:x:173:173::/etc/abrt:/sbin/nologin
```

- 结合 ^ 字符使用，表示取反

```
[root@qfedu.com ~]# grep [^a-z]c /etc/passwd
tss:x:59:59:Account used by the trousers package to sandbox the tcsd
daemon:/dev/null:/sbin/nologin
chrony:x:998:996::/var/lib/chrony:/sbin/nologin
```

- 这里表示匹配一个非小写字母的字符后接一个字符 c 的字符串。注意，^ 放在方括号里面表示反向含义，放在方括号外面则表示的是匹配行首。

7、使用 {n.m} 区间表达式来匹配指定的次数

- 这个表达式可以用来匹配指定的次数，其中 {n,m} 表示匹配在其前面的字符 n 到 m次，{n,} 表示至少匹配 n 次，{,m} 表示最多匹配 m 次，而 {n} 则是精准匹配 n 次。在 BRE 中，使用的是 {n,m} 的形式来实现相同的功能。n 与 m 的值必须介于 0 至 RE_DUP_MAX（包含这个值）之间，后者的最小值为255

```
[root@qfedu.com ~]# grep 0'\{3\}' /etc/passwd
tongye:x:1000:1000:tongye:/home/tongye:/bin/bash

[root@qfedu.com ~]# grep -E 0{3} /etc/passwd
tongye:x:1000:1000:tongye:/home/tongye:/bin/bash
```

8、使用 ( ) 保存已匹配的字符，并通过 \n 来引用已保存的匹配字符串

- 使用 ( ) 会先匹配括号中的字符串，然后将匹配到的字符串保存在由正则表达式解析器预定义好的叫做寄存器的变量中，其编号从1到9，也就是说最多可以保存9组字符串，使用 \n 可以取出所保存的字符串，其中 n 为1到9，分别对应9个寄存器的值。

```
[root@qfedu.com ~]# grep '\(operator\).*\1' /etc/passwd
operator:x:11:0:operator:/root:/sbin/nologin
```

- 这个表达式还有一个有意思的用法：

```
[root@qfedu.com ~]# grep '^\(.\).*\1$' /etc/passwd
nobody:x:99:99:Nobody:/:/sbin/nologin
```

- 正则表达式 ^(.).*\1$ 将匹配一个行首字符和行尾字符相同的字符串。

2、POSIX 方括号表达式

- 为了配合非英语的环境，POSIX 标准强化其字符集范围的能力（如 [a-z]），以匹配非英文字母字符。POSIX 使用方括号表达式 [...] 来表示一个范围值，在方括号表达式里，除了字面上的字符外（a、b、c等），另有额外的组成部分，包括：

1. 字符集：以 [: ... :] 将关键字组合括起来的 POSIX 字符集，关键字描述各种不同的字符集；
2. 排序符号：排序符号将多个字符序列视为一个单位（如，locale 中将 ch 这两个字符视为一个单位），它使用 [. 与 .] 将字符组合括起来，在系统所使用的特定 locale 上各有其定义；
3. 等价字符集：等价字符集列出的是应视为等值的一组字符，它由取自于 locale 的名字元素组成，以 [= 与 =] 括住。
4. POSIX 字符集列表：

类别	匹配字符
[:alnum:]	数字字符
[:alpha:]	字母字符
[:blank:]	空格与定位符
[:cntrl:]	控制字符
[:digit:]	数字字符
[:graph:]	非空格字符
[:lower:]	小写字母字符
[:upper:]	大写字母字符
[:space:]	空白符
[:print:]	可显示的字符
[:punct:]	标点符号字符
[:xdigit:]	十六进制数字

- 注意，字符集要放到方括号表达式中，因此一般会出现类似 [[:alpha:]] 的表达式。
- 正则表达式匹配所有的大写字母

```
[root@qfedu.com ~]# grep [[:upper:]] /etc/passwd
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
nobody:x:99:99:Nobody:/:/sbin/nologin
systemd-network:x:192:192:systemd Network Management:/:/sbin/nologin
dbus:x:81:81:System message bus:/:/sbin/nologin
polkitd:x:999:998:User for polkitd:/:/sbin/nologin
tss:x:59:59:Account used by the trousers package to sandbox the tcsd
daemon:/dev/null:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
```