Shell脚本基础知识

1. Shell脚本结构与执行

1.1脚本结构

第一行一定是：#! /bin/bash。该命令说明，该文件使用的是bash语法，如果不设置该行，则该脚本不会被执行。以#开头的行作为解释说明。Shell脚本通常以sh为后缀，用于区分这是一个shell脚本。

下面我们来编写一个shell脚本，如下所示：

[root@localhost ~]# mkdir shell

[root@localhost ~]# cd shell

[root@localhost shell]# vi 9\*9.sh //写入如下内容

#!/bin/bash

for i in `seq 9`

do

for j in `seq $i`

do

printf "$j\*$i=%2d " $((j\*i))

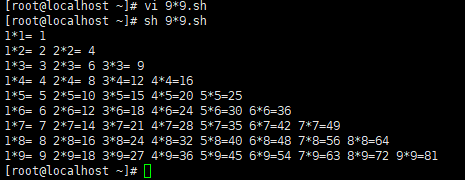
#echo -n "$j\*$i=$((j\*i)) "

#为了美观调换i和j位置

done

echo "" #换行

done



1.2 脚本执行

脚本的四种执行方法

1. sh执行，指的是用脚本对应的sh或bash来接着脚本执行

[root@shell ~]# bash 9\*9.sh

b 工作目录执行，指的是执行脚本时，先进入到脚本所在的目录（此时，称为工作目录），然后使用 ./脚本方式执行

[root@localhost shell]# ./9\*9.sh

-bash: ./1.sh: Permission denied

[root@localhost shell]# chmod +x 9\*9.sh //赋予权限 x代表执行权限

[root@localhost shell]# ./9\*9.sh //执行脚本

c. 绝对路径执行，指的是直接从根目录/到脚本目录的绝对路径

[root@localhost ~]# /root/9\*9.sh

d. .[shell环境](https://so.csdn.net/so/search?q=shell%E7%8E%AF%E5%A2%83&spm=1001.2101.3001.7020" \o "shell环境)执行，指的是在当前的shell环境中执行，可以使用 . 接脚本 或 source 接脚本

[root@localhost ~]# source 9\*9.sh

mv：是否覆盖"/tmp/2.txt"？ y

1. shell基础命令

2.1 基础命令

查看脚本执行过程

[root@localhost shell]# bash -x 9\*9.sh

++ seq 9

+ for i in '`seq 9`'

++ seq 1

+ for j in '`seq $i`'

+ printf '1\*1=%2d ' 1

1\*1= 1 + echo ''

查看脚本是否有语法错误

[root@localhost shell]# bash -n 9\*9.sh



2.2 Date命令

a. 显示年月日

三种方法

[root@localhost ~]# date +%Y-%m-%d

2024-10-08

[root@localhost ~]# date +%y-%m-%d

24-10-08

[root@localhost ~]# date +%F

2024-10-08

b. 显示时分秒

两种方法

[root@localhost ~]# date +%T

15:04:36

[root@localhost ~]# date +%H:%M:%S

15:04:52

c. 显示星期

[root@shell ~]# date +%w

5

[root@shell ~]# date +%W

41

d. 时间戳

使用命令date +%s 显示从 1970 年 1 月 1 日 00:00:00  到目前为止的秒数。

[root@shell ~]# date +%s

1728608466

使用date -d@1728608466 显示输入描述前的时间

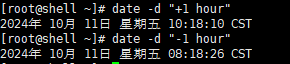
[root@shell ~]# date -d@1728608466

2024年 10月 11日 星期五 09:01:06 CST

e. 显示一个小时之前、后

date -d "+1 hour"    #一个小时后

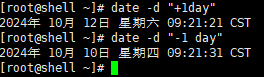
date -d "-1 hour"    #一个小时前



f. 显示一天之前、之后

date -d "+1day"     #一天后

date -d "-1 day"     #一天前



2.5 重定向及管道

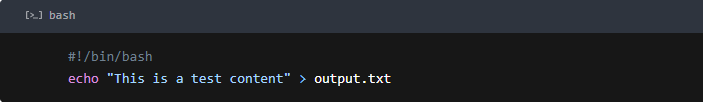
1.输出重定向(>和>>)

****1.1标准输出重定向（>）****

****作用****：将命令的标准输出重定向到文件。如果文件不存在，则创建文件；如果文件存在，则覆盖原有内容。

****示例1****：ls -l > file\_list.txt，此命令会将ls -l（列出文件详细信息）的输出结果保存到file\_list.txt文件中。如果file\_list.txt之前有内容，会被覆盖。

实例2：在script.sh脚本中，将echo命令的输出重定向到一个文件。



****1.2标准输出追加重定向（>>）****

****作用****：将命令的标准输出追加到文件末尾。如果文件不存在，则创建文件。

****示例****：echo "New line" >> log.txt，会将字符串 “New line” 添加到log.txt文件的末尾。

****1.3标准错误重定向（2>）****

****作用****：将命令执行过程中产生的标准错误输出重定向到文件。

****示例****：grep non\_existent\_pattern file.txt 2> error.log，当grep命令在file.txt中找不到指定模式时，产生的错误信息会被保存到error.log文件中，而不是显示在终端。

****1.4标准错误追加重定向（2>>）****

****作用****：将标准错误输出追加到文件末尾。

****示例****：在一个循环执行可能出错的命令脚本中，for i in $(seq 1 5); do command\_that\_might\_fail 2>> error\_accumulate.log; done，每次循环中命令产生的错误信息都会被追加到error\_accumulate.log文件中。

****1.5合并标准输出和标准错误重定向（&> 或 >&）****

****作用****：将命令的标准输出和标准错误输出同时重定向到一个文件。

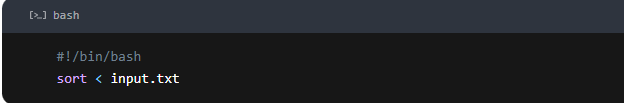
****示例****：find. -name "\*.bak" &> output\_and\_error.log，find命令的所有输出（包括正常输出和可能出现的错误输出）都会被保存到output\_and\_error.log文件中。

2.输入重定向(<)

****用法****：将文件的内容作为命令的输入。

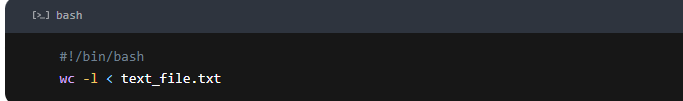
****示例 1****：

假设有一个input.txt文件，内容为一些数字 “1 2 3 4 5”，使用sort命令对这些数字进行排序，通过输入重定向来读取文件内容作为输入。



运行脚本后，会输出排序后的数字 “1 2 3 4 5”。如果没有输入重定向，sort命令会等待用户从键盘输入内容进行排序。

****示例 2****：使用wc -l命令（用于统计行数）来统计text\_file.txt文件的行数，通过输入重定向。



3.管道和重定向结合(|和>或>>)

****用法****：管道（|）用于将一个命令的输出作为另一个命令的输入，再结合重定向可以进一步处理输出结果。

****1.保存管道中间结果并继续处理****：

****示例****：cat data.txt | grep "important" > intermediate\_result.txt，这个命令先通过cat和grep组合从data.txt中筛选出包含 “important” 的行，然后将这些筛选后的结果保存到intermediate\_result.txt文件中。之后可以对这个文件进行其他操作，如sort intermediate\_result.txt。

****2.实时处理并记录结果****：

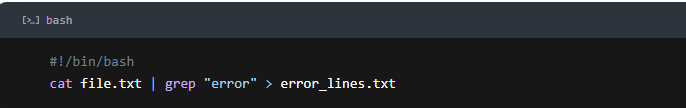
****示例****：tail -f access.log | grep "404" > error\_404.log，tail -f access.log会实时跟踪access.log文件的新增内容，通过管道将这些内容传递给grep命令来查找包含 “404” 的行，然后将这些行保存到error\_404.log文件中，用于实时监控网站的 404 错误。

****3.复杂数据处理流程****：

****示例****：find. -type f -name "\*.txt" -exec cat {} \; | awk '{print $1}' | sort | uniq -c > result.txt，这个命令首先使用find命令查找当前目录下所有的.txt文件，并将这些文件的内容通过cat命令输出，然后通过管道将这些内容传递给awk命令提取每行的第一个字段，接着进行排序和去重统计，最后将结果重定向到result.txt文件中，用于分析.txt文件内容的第一个字段的统计信息。

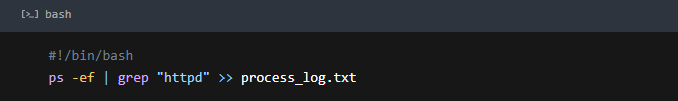
****示例 1****：

先使用cat命令读取file.txt文件内容，然后通过管道将输出传递给grep命令来查找包含特定单词（如 “error”）的行，最后将结果覆盖写入到error\_lines.txt文件。



****示例 2****：

先使用ps -ef命令获取所有进程信息，然后通过管道将输出传递给grep命令查找包含特定进程名（如 “httpd”）的行，最后将结果追加到process\_log.txt文件。



3. shell脚本中的变量

在shell脚本中使用变量可以节省时间并且使我们的脚本更加专业，所以当我们编写一个脚本时，就可以使用变量来代替某个使用频繁并且长度很长的字符串。变量的格式：“变量名=变量的值”。

3.1 引用命令

当我们引用某个命令的结果时，可以使用变量替代

示例两种形式（效果一样）：

[root@shell ~]# date +%w

5

a.

[root@shell ~]# a=`date +%w`

[root@shell ~]# echo $a

5

b.

[root@shell ~]# a=$(date +%w)

[root@shell ~]# echo $a

5

3.2 与用户交互

这里相当于把用户交互时写的值赋予给变量n

[root@shell ~]# read -p "请输入一个数字:" n

请输入一个数字:20

[root@shell ~]# echo $n

20

由于没有创造一个变量接收值，系统默认变量REPLY（英文 ：回答）

[root@shell ~]# read -p "请输入一个数字:"

请输入一个数字:30

[root@shell ~]# echo $REPLY

30

3.3 常见的内置变量

|  |  |
| --- | --- |
| 变量 | **含义** |
| $0 | 脚本名 |
| $1~$9 | 位置参数1~9 |
| ${10} | 位置参数10 |
| $# | 位置参数的个数 |
| “$\*” | 所有位置参数（整体作为单个字符串） |
| “$@” | 所有位置参数(每个作为单独字符串) |
| ${#\*} | 传递到脚本中的命令行参数的个数 |
| $? | 返回值，用于判断前一命令是否执行成功，0为成功，1则失败 |
| $$ | 脚本进程的PID |
| $- | 传递到脚本中的标识 |
| $\_ | 之前命令的最后一个参数 |
| $! | 运行在后台的最后一个作业的进程ID(PID) |

编辑脚本：

[root@localhost ~]# vi cbd.sh

#!/bin/bash

echo "$1"

echo "第二个参数是$2"

echo "第三个参数是$3"

echo "本脚本一共有$#个参数"

echo "所有位置参数(作为单个字符串)为:"$\*""

echo "所有位置参数每个作为单独字符串)为:"$@""

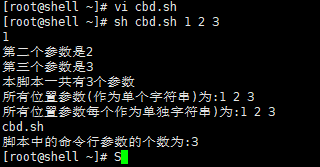
echo "$0"

echo "脚本中的命令行参数的个数为:${#\*}"

echo "脚本中的标识为:$-"

echo "命令的最后一个参数为:$\_"

执行脚本：



判断上一命令是否执行成功

[root@shell ~]# echo $?

0

3.4 常见的系统环境变量

#Shell常见的环境变量，主要是在程序运行时需要设置，环境变量详解如下：

|  |  |
| --- | --- |
| PATH | 命令所示路径，以冒号为分割； |
| HOME | 打印用户家目录； |
| SHELL | 显示当前Shell类型； |
| USER | 打印当前用户名； |
| ID | 打印当前用户id信息； |
| PWD | 显示当前所在路径； |
| TERM | 打印当前终端类型； |
| HOSTNAME | 显示当前主机名； |
| PS1 | 定义主机命令提示符的； |
| HISTSIZE | 历史命令大小，可通过 HISTTIMEFORMAT 变量设置命令执行时间; |
| RANDOM | 随机生成一个 0 至 32767 的整数; |
|  |  |

举一些案例

[root@shell ~]# echo $SHELL #显示当前Shell类型；

/bin/bash

[root@shell ~]# echo $PATH #命令所示路径，以冒号为分割；

/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin

[root@shell ~]# echo $PWD #显示当前所在路径；

/root

[root@shell ~]# echo $RANDOM #随机生成一个 0 至 32767 的整数;

7532

3.5 数学运算

创建一个脚本

[root@shell ~]# vi sum.sh

[root@shell ~]# cat sum.sh

#!/bin/bash

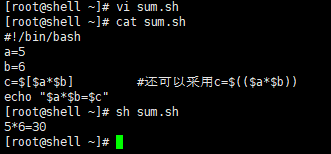
a=5

b=6

c=$[$a\*$b] #还可以采用c=$(($a\*$b))

echo "$a\*$b=$c"

执行脚本



4. Shell字符串和数组

4.1 Shell字符串

定义：

字符串可以由单引号' '包围，也可以由双引号" "包围，也可以不用引号。

三种形式的区别：

1. 不被引号包围的字符串：不被引号包围的字符串中出现变量时也会被解析；字符串中不能出现空格，否则空格后边的字符串会作为其他变量或者命令解析。

2. 由单引号包围的字符串：任何字符都会原样输出，在其中使用变量是无效的；字符串中不能出现单引号，对单引号进行转义也不行。

3. 由双引号" "包围的字符串： 如果其中包含了某个变量，那么该变量会被解析，得到该变量的值，而不是原样输出；字符串中可以出现双引号。

a. 获取字符串长度：

${#字符串名}

[root@shell ~]# rzc1="123456"

[root@shell ~]# echo ${#rzc1}

6

b.字符串拼接

将两个字符串并排放在一起就能实现拼接。$str1$str2这种写法不允许变量之间有空格，但是只要加上””就可以

[root@localhost ~]# rzc2="8910jqk"

[root@localhost ~]# rzc3=$rzc1$rzc2

[root@localhost ~]# echo $rzc3

12345678910jqk

c. 字符串截取

shell截取字符串通常有两种方式：从指定位置开始截取和从指定字符（子字符串）开始截取。

* 从左边开始计数时，起始数字是0（这符合程序员思维）；从右边开始计数时，起始数字是1（这符合常人思维）。计数方向不同，起始数字也不同。
* 不管从哪边开始计数，截取方向都是从左到右。且截取的原则为[ ),即左闭右开

使用指定位置从左边开始截取：

[root@shell ~]# echo ${rzc1:2:2} #2代表从左往右数第二个数 2代表输出两个数

34

[root@shell ~]# echo ${rzc1:1:2} #1 从左往右数第一个数 2 输出两个数字

23

[root@shell ~]# echo ${rzc1:1:4}

2345

使用指定位置从右边开始截取：

[root@shell ~]# echo ${rzc1:0-3:2}

45

[root@shell ~]# echo ${rzc1:0-4:2}

34

使用字符指定从左边开始截取：

使用#号可以截取指定字符（或者子字符串）右边的所有字符。

\*是通配符的一种，表示任意长度的字符串。\*字符3连起来使用的意思是：忽略左边的所有字符，直到遇见字符3（3字符不会被截取）。

[root@shell ~]# echo ${rzc1#\*3}

456

使用字符指定从右边开始截取：

使用%号可以截取指定字符（或者子字符串）左边的所有字符。

注意\*的位置，因为要截取字符3左边的字符，而忽略3右边的字符，所以\*应该位于3的右侧。

[root@shell ~]# echo ${rzc1%3\*}

12

4.2 字符串替换

格式：${字符串/要替换的字符/替换的字符}

[root@shell ~]# rzc4="welcome my word"

[root@shell ~]# echo $rzc4

welcome my word

[root@shell ~]# echo ${rzc4/my/you}

welcome you word

4.3 shell数组

概述：

bash支持一维数组（不支持多维数组），并且没有限定数组的大小。数组是相同类型的元素按一定顺序排列的集合。 类似与 C 语言，数组元素的索引由 0 开始编号。获取数组中的元素要利用索引，索引可以是整数或算术表达式，其值应大于或等于 0。

定义：

在 Shell 中，用括号来表示数组，数组元素用“空格”符号分割开

格式：数组名=（元素1 元素2 元素3 元素n）

1. 获取数组元素的值：

${数组名[索引]}

[root@localhost ~]# array1=(aa bb cc dd)

[root@localhost ~]# echo ${array1[0]}

aa

[root@localhost ~]# echo ${array1[1]}

bb

[root@localhost ~]#

2. 获取数组长度

利用@或\*，可以将数组扩展成列表，然后使用#来获取数组元素的个数。

[root@localhost ~]# echo ${#array1[@]}

4

[root@localhost ~]# echo ${#array1[\*]}

4

3. 数据拼接、合并

 Shell数组拼接（数组合并），将两个数组连接成一个数组。拼接数组的思路是：先利用@或\*，将数组扩展成列表，然后再合并到一起。

[root@shell ~]# array1=(aa bb cc dd)

[root@shell ~]# echo ${array1[@]}

aa bb cc dd

[root@shell ~]# array2=(ee ff gg hh)

[root@shell ~]# echo ${array2[@]}

ee ff gg hh

[root@shell ~]# array3=(${array2[@]} ${array1[@]})

[root@shell ~]# echo ${array3[@]}

ee ff gg hh aa bb cc dd

4.数组的赋值

利用索引对指定元素进行赋值

[root@localhost ~]# echo ${array1[@]}

aa bb cc dd

[root@localhost ~]# array1[1]=20

[root@localhost ~]# echo ${array1[@]}

aa 20 cc dd

5.数组的删除

利用命令unset 对数组索引指定的元素进行删除

[root@localhost ~]# echo ${array1[@]}

aa 20 cc dd

[root@localhost ~]# unset array1[1]

[root@localhost ~]# echo ${array1[@]}

aa cc dd

6.数组切片

格式：${数组名[@]:索引开始：索引结束}

[root@shell ~]# echo ${array3[@]}

ee ff gg hh aa bb cc dd

[root@shell ~]# echo ${array3[@]:1:4}

ff gg hh aa

5. shell流程控制

If语句

**脚本中常见的逻辑判断运算符：**

-f 判断文件是否存在 eg: if [ -f filename ]；

-d 判断目录是否存在 eg: if [ -d dir ]；

-eq 等于，应用于整型比较 equal；

-ne 不等于，应用于整型比较 not equal；

-lt 小于，应用于整型比较 letter；

-gt 大于，应用于整型比较 greater；

-le 小于或等于，应用于整型比较；

-ge 大于或等于，应用于整型比较；

-a 双方都成立（and） 逻辑表达式 –a 逻辑表达式；

-o 单方成立（or） 逻辑表达式 –o 逻辑表达式；

-z 空字符串；

-x 是否具有可执行权限

|| 单方成立；

&& 双方都成立表达式。

1. 不带有else

基础结构：

if  判断语句; then

    command

fi

编辑脚本

[root@shell ~]# vi if1.sh

[root@shell ~]# cat if1.sh

#!/bin/bash

a=10

if [ $a -gt 4 ] #这里的gt为英文单词greater than（大于）

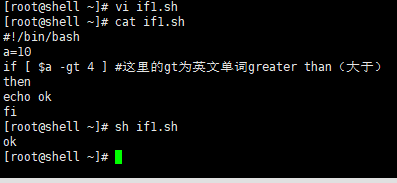
then

echo ok

fi

[root@shell ~]# sh if1.sh

ok



2. 带有else（否则的意思）

基础结构：

if  判断语句  ; then

    command

else

    command

fi

编写脚本if2.sh

[root@shell ~]# vi if2.sh

[root@shell ~]# cat if2.sh

#!/bin/bash

a=10

if [ $a -gt 4 ]

then

echo ok

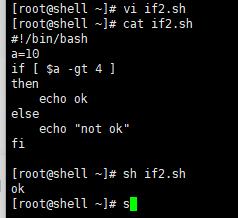
else

echo "not ok"

fi

[root@shell ~]# sh if2.sh

ok



3. 带有elif（相当于再则的意思）

编辑脚本if3.sh

[root@shell ~]# vi if3.sh

[root@shell ~]# cat if3.sh

#!/bin/bash

a=3

if [ $a -gt 4 ]

then

echo ok

elif [ $a -gt 8 ]

then

echo "very ok"

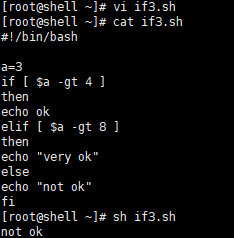
else

echo "not ok"

fi

[root@shell ~]# sh if3.sh

not ok



4. 嵌套（if句里在添加if语句）

编辑脚本if4.sh

[root@localhost ~]# vi if4.sh

#!/bin/bash

a=15

if [ $a -gt 4 ]

then

if [ $a -lt 20 ]

then

echo "ok"

else

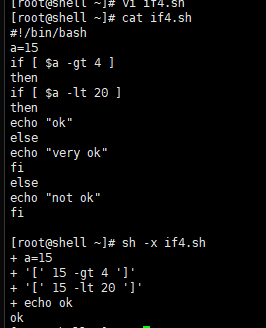
echo "very ok"

fi

else

echo "not ok"

fi



5. 多个条件

[root@shell ~]# cat if5.sh

#!/bin/bash

read -p "请输入任志财的一百米跑步成绩:" a

read -p "请输入王帅的一百米跑步成绩:" b

if [ $a -gt 9 ] && [ $a -lt 11 ]

then

echo "任志财对王帅说老弟你还得练”

if [ $b -gt 12 ]

then

echo "王帅对任志财说财哥我错了我再也不敢和你比赛了”

fi

[root@shell ~]# sh if5.sh

请输入任志财的一百米跑步成绩:10

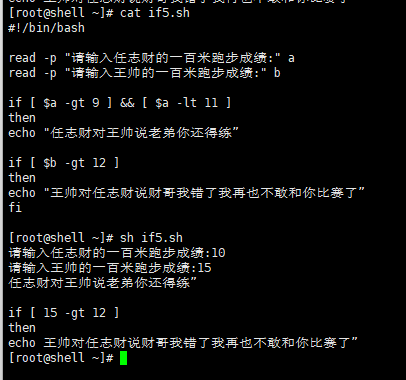
请输入王帅的一百米跑步成绩:15

任志财对王帅说老弟你还得练”

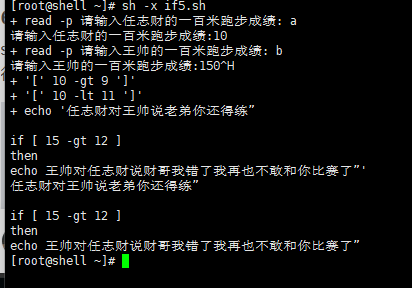
if [ 15 -gt 12 ]

then

echo 王帅对任志财说财哥我错了我再也不敢和你比赛了”



查看执行过程



6. if逻辑判断

shell脚本中if经常用于判断文档的属性，比如判断是普通文件还是目录，判断文件是否有读、写、执行权限等。If常用选项如下：

-e:判断文件或目录是否存在。

-d:判断是不是目录以及是否存在。

-f:判断是不是普通文件以及是否存在。

-r:判断是否有读权限。

-w:判断是否有写权限。

-x:判断是否可执行。

-h: file hard link（链接文件）

-L :file link（链接文件）

-b :file 块设备文件

-c :file 字符设备文件

-p :file 管道文件

-S :file socket套接字文件

-t :file 文件与终端相关联

-N :file文件最后一次读取后被修改过

-s :file 文件大小不为0，文件存在且非空

-z：判断是否为空字符串

-n：判断是否为非空字符串

注意：root用户对文件的读写比较特殊，即使一个文件没有给root用户读或者写的权限，root也可以读或者写。

[root@localhost ~]# vi test.sh

#!/bin/bash

if [ -d /etc ]; then

if [ -f if1.sh ]; then

if [ -r if1.sh ]; then

if [ ! -w if1.sh ]; then # ！-W 表示数学中的非，即本为ture，非本则为false

echo "文件不具有写权限"

else

echo "文件具有写权限"

fi

echo "文件具有读权限"

else

echo "文件不具有读权限"

fi

echo "文件存在"

else

echo "文件不存在"

fi

echo "目录存在"

else

echo "目录不存在"

fi

执行脚本



7. shell中的case判断

Case判断基础格式如下

case 变量 in

value1） #不限制value的个数

command

;; #一个分支的结束

value2）

command

;;

\*) #此处\*代表其他值

command

;;

esac

在[Shell脚本](https://so.csdn.net/so/search?q=Shell%E8%84%9A%E6%9C%AC&spm=1001.2101.3001.7020" \o "Shell脚本)中，case语句是一种多路选择结构，它允许一个变量等于多个值时分别执行不同的操作。case语句以case关键字开始，以esac（case的反写）关键字结束。

为了让我们能够更加清晰的理解case逻辑判断，接下来我们编写一个脚本来进行实验：

[root@localhost ~]# vi case.sh

#!/bin/bash

read -p "Please input a number:" n

if [ -z "$n" ]

then

echo "Please input a number."

exit 1

fi

if [ $n -lt 60 ] && [ $n -ge 1 ]

then

tag=1

elif [ $n -ge 60 ] && [ $n -lt 80 ]

then

tag=2

elif [ $n -ge 80 ] && [ $n -lt 90 ]

then

tag=3

then

tag=4

else

tag=0

fi

case $tag in #$tag匹配为1，输出not ok

1)

echo "not ok"

;;

2)

echo "ok"

;;

3)

echo "very ok"

;;

4)

echo "very good"

;;

\*) #$tag匹配除前面1，2，3，4以外的数。

elif [ $n -ge 90 ] && [ $n -le 100 ]

echo "The number range is 0-100."

;;

esac

6. shell中的循环

1. for循环

基础结构如下

for 变量名 in 循环条件；

do

command

done

案例：

基础案例

[root@localhost shell]# vi for1.sh

#！/bin/bash

sum=0

for i in `seq 1 10`

do

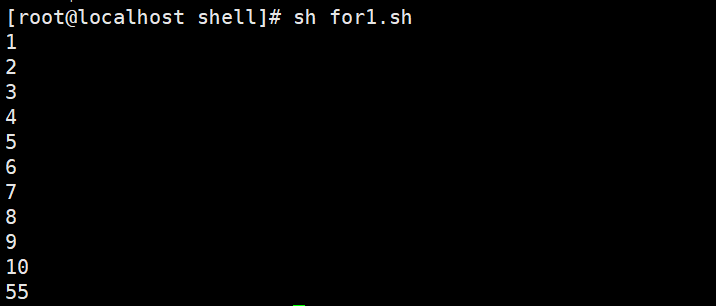
sum=$[$sum+$i]

echo $i

done

echo $sum

执行结果：



进阶案例（9\*9乘法表）

[root@shell ~]# vi for2.sh

[root@shell ~]# cat for2.sh

#!/bin/bash

for((i=1;i<10;i++)) *#定义i的初始值为1，且这个值不大于9，每次循环加1*

do

for((j=1;j<=i;j++))

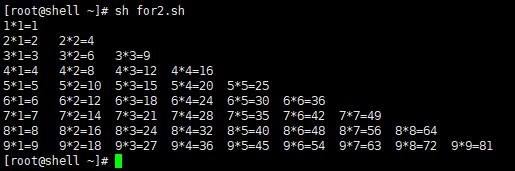
do

echo -en "$i\*$j=$[i\*j]\t" *#-n 表示不换行打印，\t表示制表符,属于转义字符，-e的作用就是使转义字符能够被解释*

done

echo

done



1. while循环

基础结构

while 条件; do

command

done

用while循环制作99乘法表

[root@shell ~]# vi while.sh

[root@shell ~]# cat while.sh

#!/bin/bash

i=1

while (( $i<=9 ))

do

j=1

while (( $j<=$i ))

do

echo -ne "$i\*$j=$((i \* j))\t"

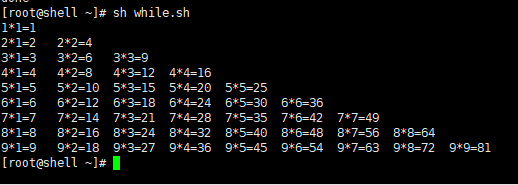
let j++ # let为赋值表达式，即j++ 《=》j=j+1

done

let i++

echo "" # 这步操作在于换行

done



7. shell中的中断与继续

7.1 跳出循环

break在脚本中表示跳出该层循环，示例如下：

[root@shell ~]# vi break1.sh

[root@shell ~]# cat break1.sh

#!/bin/bash

for i in `seq 1 5`

do

echo $i

if [ $i -eq 3 ]

then

break

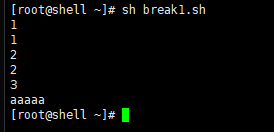
fi

echo $i

done

echo aaaaa

执行结果



7.2 结束本次循环

当在shell脚本中使用continue时，结束的不是整个循环，而是本次循环。忽略continue之下的代码，直接进行下一次循环。示例如下：

[root@shell ~]# vi continue1.sh

[root@shell ~]# cat continue1.sh

#!/bin/bash

for i in `seq 1 5 `

do

echo $i

if [ $i == 3 ]

then

continue #此处continue表示若 $i == 3 则结束本次循环

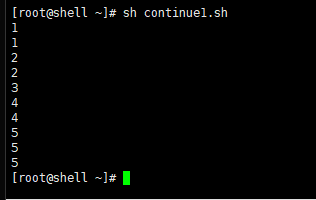
fi

echo $i

done

echo $i

执行结果：



7.3 退出整个脚本

当我们在shell脚本中遇到exit时，其表示直接退出整个shell脚本。示例如下：

[root@shell ~]# vi exit1.sh

[root@shell ~]# cat exit1.sh

#!/bin/bash

for i in `seq 1 5`

do

echo $i

if [ $i == 3 ]

then

exit

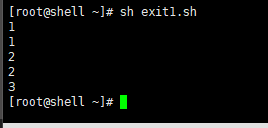
fi

echo $i

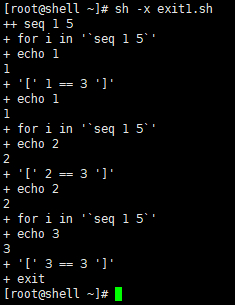
done

echo aaaa

执行结果：



执行过程



8. shell中的函数

shell脚本中的函数就是先把一段代码整理到了一个小单元中，并给这个小单元命名，当我们用到这段代码时直接调用这个小单元的名字就可以了，这样很方便，省时省力。但我们需要注意，在shell脚本中，**函数一定要写在前面**，因为函数要被调用的，如果还未出现就被调用就会出错。

基础格式：

function f\_name() # function 函数名()

{ #{

Command #  函数体（即命令序列）

} #}

8.1 打印出第一个、第二个参数、参数的个数及脚本名

示例如下：

[root@shell ~]# vi fun1.sh

[root@shell ~]# cat fun1.sh

#/bin/bash

input()

{

echo $1 $2 $# $0 # 函数的参数：$1 $2 $# ；$0则是脚本的名字

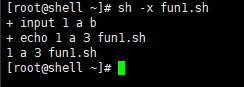
}

input 1 a b

[root@shell ~]# sh fun1.sh

1 a 3 fun1.sh

执行过程：



8.2加法的函数

[root@shell ~]# vi fun2.sh

[root@shell ~]# cat fun2.sh

#!/bin/bash

sum()

{

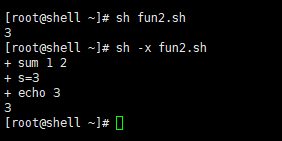
s=$[$1+$2]

echo $s

}

sum 1 2

执行过程



8.3 获得一个网卡的ip地址

[root@shell ~]# vi fun3.sh

[root@shell ~]# cat fun3.sh

#!/bin/bash

ip()

{

ifconfig |grep -A1 "$1: " |tail -1 |awk '{print $2}'

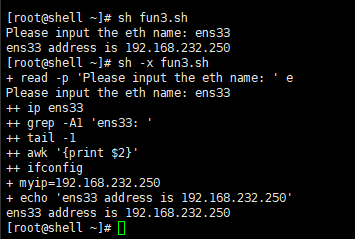
}

read -p "Please input the eth name: " e

myip=`ip $e`

echo "$e address is $myip"

执行过程与结果



9. shell中的正则表达式

概述

正则表达式是你所定义的模式模板， Linux工具可以用它来过滤文本。 Linux 工具（比如sed编辑器或gawk程序）

能够在处理数据时使用正则表达式对数据进行模式匹配。如果数据匹配模式，它就会被接受并进一步处理；

如果数据不匹配模式，它就会被滤掉。

它主要用于字符串的分割，匹配、査找及替换操作。即正则表达式是一种文本模式，该模式描述在搜索文本时要匹配的一个或多个字符串

简单来说就是通过一些特殊字符的排序，用以删除、查找、替换一行或者多行文字字符串的程序。

1. 基础正则

基础正则常见元字符（支持的工具：grep、egrep、sed、awk）

\ ：转义字符，用于取消特殊符号的含义，例：\! 、\n、\$等

^ ：匹配字符串开始的位置，例：^a、^the、^#、^[a-z]

$ ：匹配字符串结束的位置，例: word$、 ^$匹配空行

\*：匹配除\n之外的任意的一个字符，例: lo.\*k、lo.k、l..k

.\*：匹配任意长度的字符

+：匹配前面的字符出现过最少一次

\{n\} : 匹配前面一个字符n次，例: lo\{2\}k、 '[0-9]\{2\}'匹配两位数字

\{n,\} : 匹配前面一个字符不少于n次，例: lo\{2,\}k、 '[0-9]\{2,\}'匹配两位及两位以上数字

\{n,m\} : 匹配前面一个字符n到m次，例: lo\{2,3\}k、 '[0-9]\{2,3\}'匹配两位到三位数字

注: egrep、 awk使用{n}、{n,}、 {n, m}匹配时“{}"前不用加“\”

1. 常用命令

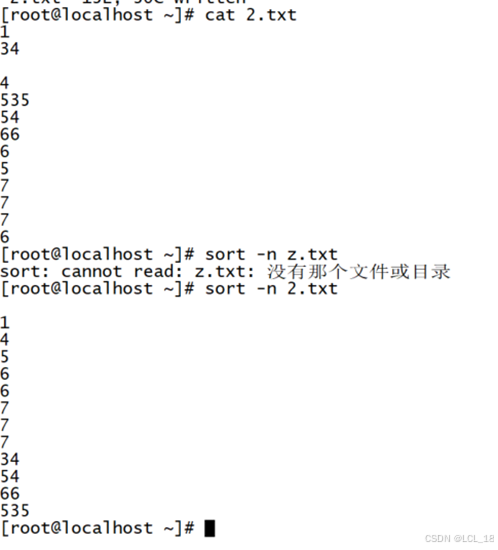
**Sort排序：**

**以行对文件进行排序**

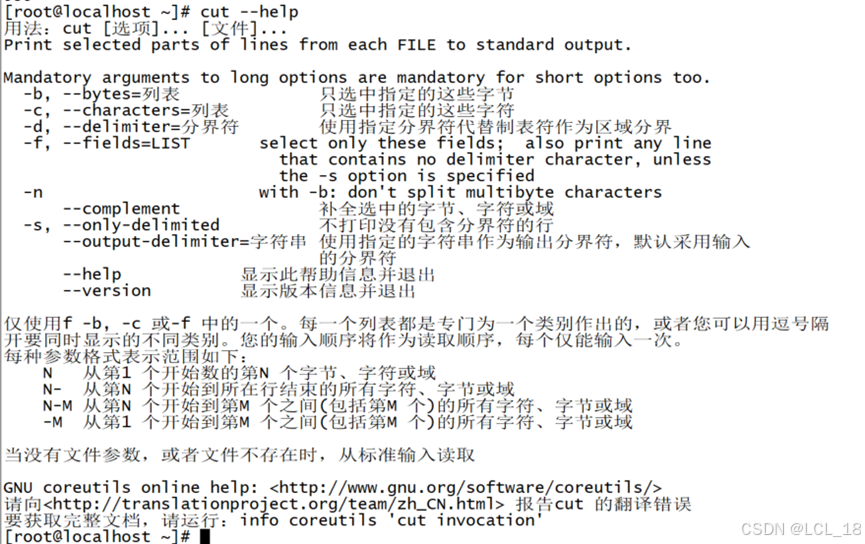
可以通过linux自带的man手册去查看更多的命令



这里举一个例子



查看一些常用命令可用 命令 –help



**tac：**倒序打印文件

**rev：**反向打印每一行

**cut：**字符切割，常用选项-d 分割，-f输出第几段

**tr：**替换或删除字符

**seq：**打印序列化数字

**uniq：**去重 -c 打印出现次数、-u ：打印不重复的行

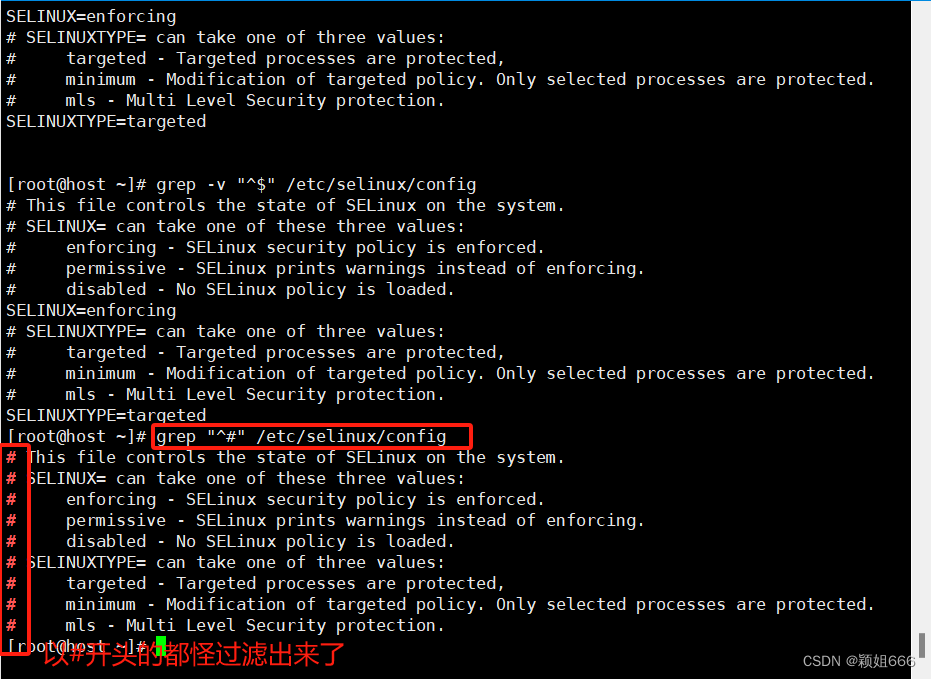
1. 三剑客

3.1 grep

1. 功能

筛选过滤出包含<匹配字符串>的<整行>

例如：筛选出以#开头的打印出来



2.语法

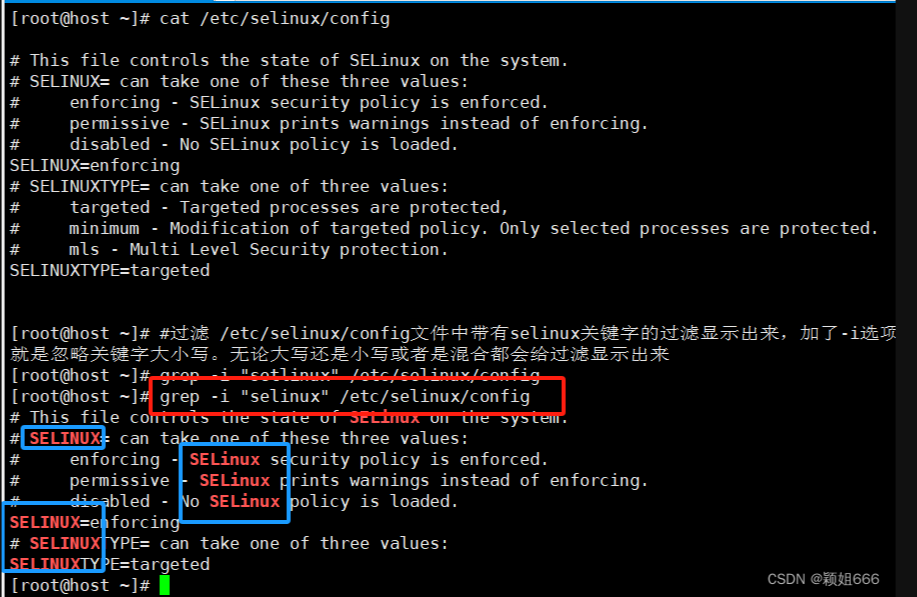
grep [选项]   PATTERN  [FILE....]

PATTERN：就是正则表达式，默认情况下，grep命令使用基本正则

3.选项

-i 表示忽略大小写

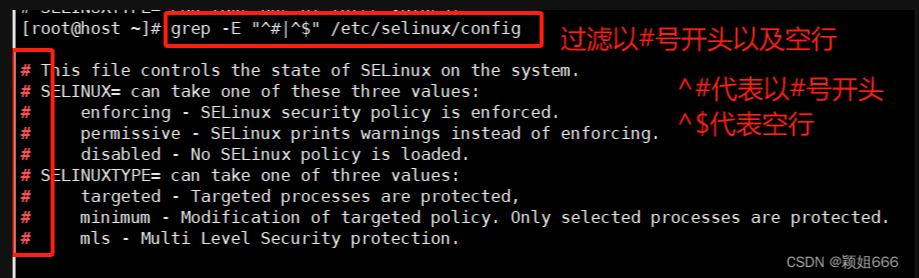
[root@shell ~]# grep -i "rzc" grep.sh # “”里面表示要找的内容 后面加上文件的路径



-E 表示启用<扩展正则>

，将模式 PATTERN 作为一个扩展的正则表达式来解释

[root@shell ~]# grep -E "^#|^$" grep.sh







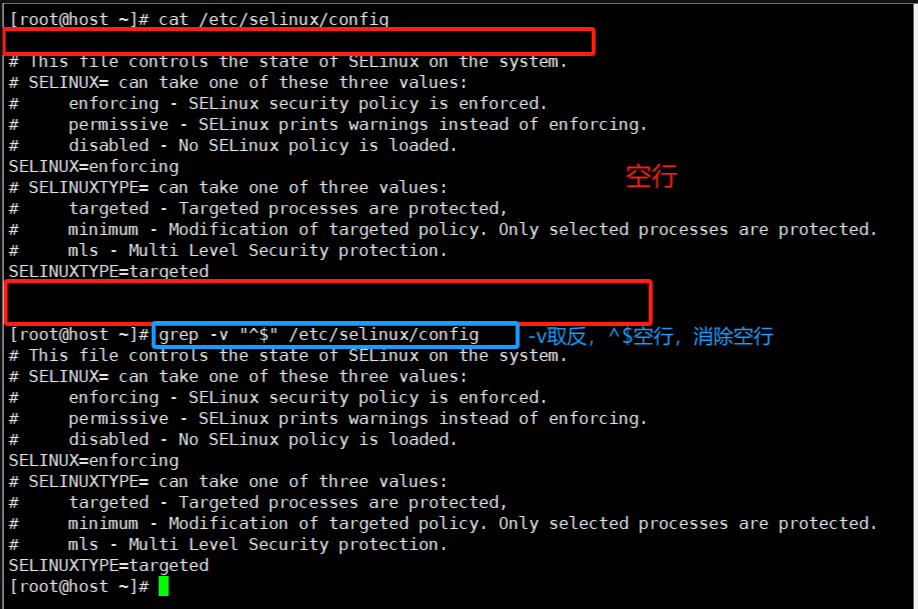
-o只显示匹配的行中与 PATTERN 相匹配的部分。

默认情况下，是输出包含<匹配内容>的<一行数据>。



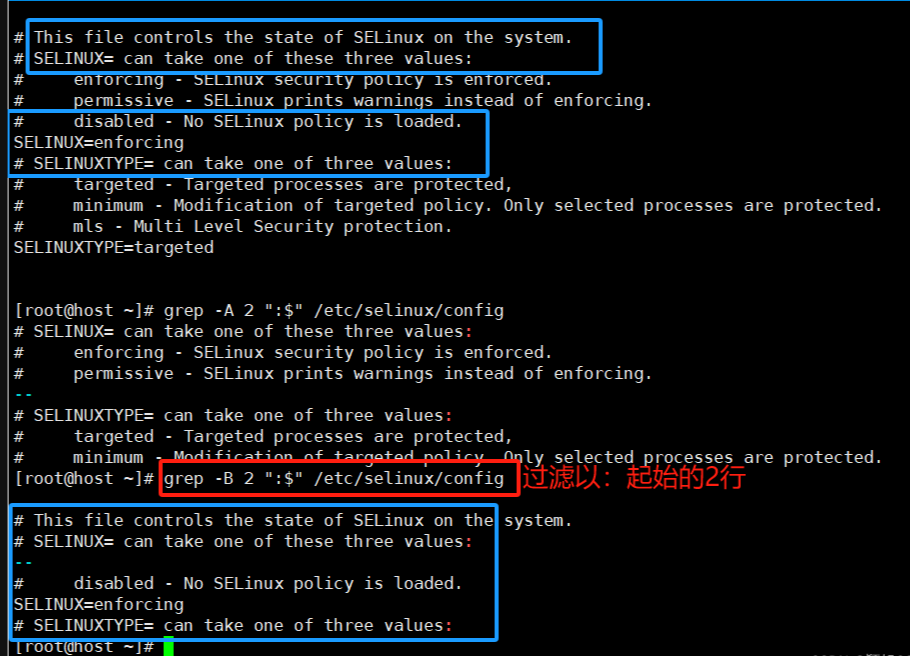
-v表示仅输出<不匹配的数据行> 注意是<不匹配的行>，也就是取反

^和$代表空行



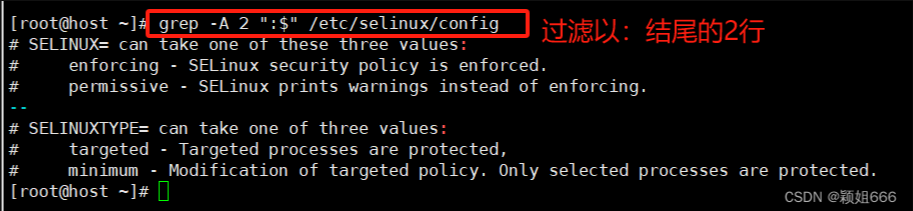
-B 打印出匹配的行之前的上文 NUM 行。

在相邻的匹配组之间将会打印内容是 -- 的一行。



-A打印出紧随匹配的行之后的下文NUM行。

在相邻的匹配组之间将会打印内容是 -- 的一行。



-C显示匹配的<数据行>的<行总数>



3.2 sed

1. 基本语法和功能

****语法格式****：sed [选项] '脚本命令' 输入文件。例如，sed's/old/new/g' file.txt，其中s表示替换操作，old是要被替换的字符串，new是替换后的字符串，g表示全局替换（即文件中所有匹配的字符串都被替换），file.txt是操作的目标文件。

****功能概述****：sed是一个流编辑器，主要用于对文本文件进行编辑操作。它可以在不打开文件的情况下，对文件内容进行查找、替换、删除、插入等操作。sed的操作是基于行的，它逐行读取文件内容，按照指定的脚本命令进行处理，然后输出结果。

2. 替换操作 （s命令）

****简单替换****：

示例：sed's/hello/hi/' file.txt，此命令会将file.txt文件中每行第一次出现的 “hello” 替换为 “hi”。如果要进行全局替换，需要在替换命令最后添加g选项，如sed's/hello/hi/g' file.txt。

****使用正则表达式进行替换****：

sed支持使用正则表达式来匹配要替换的内容。例如，sed's/[0 - 9]+/number/' file.txt，会将file.txt文件中每行出现的数字（一个或多个数字）替换为 “number”。

****替换部分匹配的字符串****：

可以通过在替换命令中使用&符号来引用匹配的字符串部分。例如，sed's/(\w+), (\w+)/\2, \1/' file.txt（在支持扩展正则表达式的情况下），假设文件中有 “John, Doe” 这样的内容，此命令会将其转换为 “Doe, John”，其中\1和\2分别代表第一个和第二个括号内匹配的内容。

3. 删除操作 （d命令）

****删除匹配行****：

示例：sed '/^#/d' file.txt，此命令会删除file.txt文件中所有以 “#” 开头的行。^符号表示行的开头，所以^#匹配以 “#” 开头的行。

****根据条件删除行****：

可以结合多个条件来删除行。例如，sed -n '/pattern1/ { /pattern2/!d }' file.txt（-n选项用于抑制默认输出，只输出经过脚本命令处理后的行），这个命令会删除file.txt中包含 “pattern1” 但不包含 “pattern2” 的行。

4插入和追加操作（i和a命令）

****插入操作（i）****：

示例：sed '1i This is a new line' file.txt，会在file.txt文件的第一行之前插入 “This is a new line”。

****追加操作（a）****：

示例：sed '$a This is an appended line' file.txt，会在file.txt文件的最后一行之后追加 “This is an appended line”。

5.读取和写入文件（r和w命令）

****读取文件（r）****：

示例：sed '3r other\_file.txt' file.txt，会在file.txt文件的第三行之后读取并插入other\_file.txt的内容。

****写入文件（w）****：

示例：sed -n '/pattern/w output\_file.txt' file.txt，会将file.txt中包含 “pattern” 的行写入到output\_file.txt文件中。

6.在shell脚本中的应用场景

****配置文件修改****：

在脚本中，可以使用sed修改配置文件。例如，修改nginx.conf文件中服务器监听端口，sed -i's/listen 80/listen 8080/g' nginx.conf，-i选项表示直接在原文件上进行修改。

****日志文件处理****：

对于日志文件，可以使用sed提取或清理特定的日志行。如sed -n '/ERROR/p' log\_file.txt会只打印log\_file.txt中包含 “ERROR” 的行，用于快速定位错误日志。

3.3 awk

1.打印操作

****打印整行****：

****命令格式****：awk '{print $0}' 文件名。例如：awk '{print $0}' log.txt会将log.txt文件中的每一行内容都打印出来。这在查看文件的原始内容时非常有用，比如查看配置文件或者简单的文本记录文件。

****打印指定列****：

****命令格式****：awk '{print $列数}' 文件名。例如：awk '{print $1}' data.csv，假设data.csv是一个以逗号分隔的文件（如包含姓名、年龄、地址等信息），这个命令会打印出每一行的第一列内容，也就是姓名列。如果想打印多列，可以用逗号分隔，如awk '{print $1, $3}' file.txt会打印第一列和第三列的内容。

2.条件判断操作

****基于文本内容筛选行****：

****命令格式****：awk '/正则表达式/ {动作}' 文件名。例如：awk '/error/ {print $0}' error.log会将error.log文件中包含 “error” 字样的行全部打印出来。这在日志文件分析中经常使用，用于快速定位包含特定错误信息的行。

****基于列内容筛选行****：

****命令格式****：awk '$列数 比较运算符 值 {动作}' 文件名。例如：awk '$2 > 10 {print $0}' numbers.txt，假设numbers.txt是一个包含数字的文件，这个命令会将第二列数字大于 10 的行全部打印出来。比较运算符可以是>（大于）、<（小于）、==（等于）、>=（大于等于）、<=（小于等于）和!=（不等于）。

3.数据统计操作

****计算列总和****：

****命令格式****：awk '{sum += $列数} END {print sum}' 文件名。例如：awk '{sum += $3} END {print sum}' sales.csv，假设sales.csv是一个销售数据文件，第三列是销售额，这个命令会计算出所有行销售额的总和并打印出来。

****计算列平均值****：

****命令格式****：awk '{sum += $列数; count++} END {print sum/count}' 文件名。例如：awk '{sum += $4; count++} END {print sum/count}' scores.csv，假设scores.csv是一个学生成绩文件，第四列是某一科目成绩，这个命令会计算出该科目成绩的平均值。

1. 文本格式化操作

****自定义输出格式****：

****命令格式****：awk '{printf "格式化字符串", $列数}' 文件名。例如：awk '{printf "%-10s %-10s\n", $1, $2}' contacts.txt，假设contacts.txt是一个包含联系人姓名和电话的文件，这个命令会将姓名和电话以左对齐、宽度为 10 个字符的格式输出，并且每行末尾添加换行符。格式化字符串中常用的格式控制符有%s（字符串）、%d（整数）、%f（浮点数）等，-表示左对齐，数字表示宽度。

****字符串拼接与替换****：

****命令格式****：awk '{new\_column = $列数1 "连接字符" $列数2; print new\_column}' 文件名。例如：awk '{new\_column = $1 " - " $2; print new\_column}' products.txt，假设products.txt是一个产品信息文件，第一列是产品名称，第二列是产品型号，这个命令会将名称和型号用 “-” 连接起来并打印新的列。也可以使用sub或gsub函数进行字符串替换，如awk '{sub("旧字符串", "新字符串", $列数); print $列数}' 文件名。