From: Linux 命令行与 shell 脚本编程大全（第 3 版）

Linux 与 shell 命令和脚本

Summary Edition By Myself

Kulapati Admirnoon (Kuladmir)

2025-2-20

Version: 1.02

目录/Content

Linux 与 shell 命令和脚本

1. 基本命令

1.1 更改背景颜色

方法一：

setterm -inversescreen on/off

可以将背景从黑变白，字体从白变黑（on，off 反之）。

```
:~$ setterm -inversescreen on
:~$
```

```
$ setterm -inversescreen on
$ setterm -inversescreen off
$
```

方法二：

setterm –background white

setterm –foreground black

颜色选择：black red green yellow blue magenta cyan white，可以自定义背景颜色。

不过这个方法在 WSL 上使用会变成这样：

```
(base) kuladmir@kuladmir:~$ setterm -foreground cyan -background white
(base) kuladmir@kuladmir:~$ ls
```

1.2 帮助命令

man 工具名

可以查询工具的相关命令和帮助，man 命令后必须加工具名。

输入：man setterm，输出：

```
SETTERM(1)                                User Commands                                SETTERM(1)

NAME
       setterm - set terminal attributes

SYNOPSIS
       setterm [options]

DESCRIPTION
       setterm writes to standard output a character string that will invoke the specified terminal capabilities.
       Where possible terminfo is consulted to find the string to use. Some options however (marked "virtual consoles
       only" below) do not correspond to a terminfo(5) capability. In this case, if the terminal type is "con" or
       "linux" the string that invokes the specified capabilities on the PC Minix virtual console driver is output.
       Options that are not implemented by the terminal are ignored.

OPTIONS
       For boolean options (on or off), the default is on.

       Below, an 8-color can be black, red, green, yellow, blue, magenta, cyan, or white.

       A 16-color can be an 8-color, or grey, or bright followed by red, green, yellow, blue, magenta, cyan, or
       white.

       The various color options may be set independently, at least on virtual consoles, though the results of
       setting multiple modes (for example, --underline and --half-bright) are hardware-dependent.

       The optional arguments are recommended with '=' (equals sign) and not space between the option and the
       argument. For example --option=argument. setterm can interpret the next non-option argument as an optional
       argument too.
```

1.3 查询监测命令

who

展示出所有在线的用户（查看自己终端的命令）

```
kuladmir@kuladmir:~$ who
kuladmir pts/1        2024-12-18 15:41
```

whoami

展示出自己的账号。

```
kuladmir@kuladmir:~$ whoami
kuladmir
```

who | wc

展示出已登录人数。

```
kuladmir@kuladmir:~$ who | wc
      1       4      39
```

uname

展示出操作系统名，可以额外添加以下参数，也可以不添加。

-a 或--all　显示全部的信息。

-m 或--machine　显示电脑类型。

-n 或-nodename　显示在网络上的主机名称。

-r 或--release　显示操作系统的发行编号。

-s 或--sysname　显示操作系统名称。

-v　显示操作系统的版本。

```
kuladmir@kuladmir:~$ uname
Linux
kuladmir@kuladmir:~$ uname -a
Linux kuladmir 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64 x86_64 x86_64 GNU/Linux
kuladmir@kuladmir:~$ uname -n
kuladmir
kuladmir@kuladmir:~$ uname -r
5.15.167.4-microsoft-standard-WSL2
kuladmir@kuladmir:~$
```

top

动态查看进程（CPU 占用率等），按 Q 退出。

```
kuladmir@kuladmir:~$ top
top - 16:00:05 up 18 min,  1 user,  load average: 0.00, 0.00, 0.00
Tasks:  23 total,   1 running,  22 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   7786.4 total,   7170.4 free,    650.4 used,    190.2 buff/cache
MiB Swap:   2048.0 total,   2048.0 free,      0.0 used.   7136.0 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
      1 root      20   0   21500  12784   9532 S   0.0   0.2   0:00.24 systemd
      2 root      20   0    2616   1508   1384 S   0.0   0.0   0:00.00 init-systemd(Ub
      7 root      20   0    2616    132    132 S   0.0   0.0   0:00.00 init
     55 root      19  -1   66820  17240  16120 S   0.0   0.2   0:00.08 systemd-journal
     97 root      20   0   23992   6196   4924 S   0.0   0.1   0:00.10 systemd-udevd
    146 systemd+  20   0   21452  11836   9644 S   0.0   0.1   0:00.04 systemd-resolve
    147 systemd+  20   0   91020   6488   5644 S   0.0   0.1   0:00.05 systemd-timesyn
```

date

展示出当前时间，date +‘参数’可以输出时间，参数可以多个存在：%Y 表示年，%m 表示月，%d 表示天，%H 表示小时，%M 表示分钟，%S 表示秒，%s 表示时间戳的秒数。

df (-h)

可以展示出磁盘使用情况，-h 能够更易读。

```
(base) kuladmir@kuladmir:~$ df
Filesystem      1K-blocks      Used Available Use% Mounted on
none              3986612         0   3986612   0% /usr/lib/modules/5.15.167.4-
none              3986612         4   3986608   1% /mnt/wsl
drivers         475971580 260596544 215375036  55% /usr/lib/wsl/drivers
```

```
(base) kuladmir@kuladmir:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
none            3.9G     0  3.9G   0% /usr/lib/modules/5.15.167.4-microsoft-standard-WSL2
none            3.9G  4.0K  3.9G   1% /mnt/wsl
drivers         454G  249G  206G  55% /usr/lib/wsl/drivers
```

du

可以展示出文件和目录的占用空间大小，通常会输出很多内容：

```
(base) kuladmir@kuladmir:~$ du
4       ./.vscode-server/data/User/globalStorage/vscode.json-
8       ./.vscode-server/data/User/globalStorage/vscode.json-
12      ./.vscode-server/data/User/globalStorage
8       ./.vscode-server/data/User/History/20c86b32
```

可以使用参数-h 让结果更易读，-c 显示出文件的总大小，-s 显示每个参数的总计。

```
(base) kuladmir@kuladmir:~$ du -h
4.0K    ./.vscode-server/data/User/globalStorage/vscode.json-language-featu
8.0K    ./.vscode-server/data/User/globalStorage/vscode.json-language-featu
12K     ./.vscode-server/data/User/globalStorage
```

```
(base) kuladmir@kuladmir:~$ du -s
858848  .
(base) kuladmir@kuladmir:~$ du -c
4       ./.vscode-server/data/User/globalStorage/vscode.json-language-features/json-sch
8       ./.vscode-server/data/User/globalStorage/vscode.json-language-features
12      ./.vscode-server/data/User/globalStorage
```

```
8       ./miniconda3/ssl
667264  ./miniconda3
858848  .
858848  total
```

free

可以展示出内存状态。

```
(base) kuladmir@kuladmir:~$ free
               total        used        free      shared  buff/cache   available
Mem:         7973228      662336     7334740        3100      217832     7310892
Swap:        2097152           0     2097152
```

sort *file_name*

可以进行排序操作，根据字符顺序进行排序，-n 参数可以把数字识别为数字而不是字符。-M 参数可以识别月份的三字母，用于对月份进行合理排序。

wc *file_name*

可以计算文件字数，结果显示分别为：行数（-l），字数（-w），字符数（-m），字节数（-c，默认不显示），可以在 wc 命令后加入参数，显示对应数值。

```
(base) kuladmir@kuladmir:~/test$ cat 1.c

#include<stdio.h>
int main()
{
        printf("Hello World\n");
        printf("Good\n");
        return 0;
}
(base) kuladmir@kuladmir:~/test$ wc 1.c
 8 10 90 1.c
```

grep *Search* *file_name*

用于在对应文件内查询内容，可以添加通配符*进行查询，其实不加通配符也不影响结果，比如下例中，in 和 in*结果一致。

```
(base) kuladmir@kuladmir:~/test$ grep include 1.c
#include<stdio.h>
(base) kuladmir@kuladmir:~/test$ grep in* 1.c
#include<stdio.h>
int main()
        printf("Hello World\n");
        printf("Good\n");
```

可用的参数：

-e 可以指定多个内容。

-v 反向搜索，输出不含 *Search* 的内容。

-n 输出匹配内容的行号。

-c 输出有多少行匹配。

```
(base) kuladmir@kuladmir:~/test$ grep -c in 1.c
4
(base) kuladmir@kuladmir:~/test$ grep -v in 1.c

{
        return 0;
}
(base) kuladmir@kuladmir:~/test$ grep -n in 1.c
2:#include<stdio.h>
3:int main()
5:      printf("Hello World\n");
6:      printf("Good\n");
(base) kuladmir@kuladmir:~/test$ grep -e i -e n 1.c
#include<stdio.h>
int main()
        printf("Hello World\n");
        printf("Good\n");
        return 0;
```

netstat

可以展示 Linux 使用情况，可以加入如下参数：

-a 或--all 显示所有连线中的 Socket。

-A<网络类型>或--<网络类型> 列出该网络类型连线中的相关地址。

-c 或--continuous 持续列出网络状态。

-C 或--cache 显示路由器配置的快取信息。

-e 或--extend 显示网络其他相关信息。

-F 或--fib 显示 FIB。

-g 或--groups 显示多重广播功能群组组员名单。

-i 或--interfaces 显示网络界面信息表单。

-l 或--listening 显示监控中的服务器的 Socket。

-M 或--masquerade 显示伪装的网络连线。

-n 或--numeric 直接使用 IP 地址，而不通过域名服务器。

-N 或--netlink 或--symbolic 显示网络硬件外围设备的符号连接名称。

-o 或--timers 显示计时器。

-p 或--programs 显示正在使用 Socket 的程序识别码和程序名称。

-r 或--route 显示 Routing Table。

-s 或--statistice 显示网络工作信息统计表。

-t 或--tcp 显示 TCP 传输协议的连线状况。

-u 或--udp 显示 UDP 传输协议的连线状况。

-v 或--verbose 显示指令执行过程。

-V 或--version 显示版本信息。

-w 或--raw 显示 RAW 传输协议的连线状况。

-x 或--unix 此参数的效果和指定"-A unix"参数相同。

--ip 或--inet 此参数的效果和指定"-A inet"参数相同。

ps

显示目前进程。

```
(base) kuladmir@kuladmir:~$ ps
    PID TTY          TIME CMD
    400 pts/0    00:00:00 bash
    566 pts/0    00:00:00 ps
```

可以额外添加参数（部分），参数可以组合使用：

-A/-e　显示所有进程。

-a　显示除控制进程（session leader）和无终端进程外的所有进程。

-N　显示与指定参数不符的所有进程。

-d　显示除控制进程外的所有进程。

-c　显示包含在 cmdlist 列表中的进程。

-G　显示组 ID 在 grplist 列表中的进程。

-U　显示属主的用户 ID 在 userlist 列表中的进程。

-g　显示会话或组 ID 在 grplist 列表中的进程。

-p　显示 PID 在 pidlist 列表中的进程。

-f　显示完整格式的输出。

```
(base) kuladmir@kuladmir:~$ ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 08:34 ?        00:00:00 /sbin/init
root           2       1  0 08:34 ?        00:00:00 /init
root           7       2  0 08:35 ?        00:00:00 plan9 --control-socket 6 --log-level 4 --
root          55       1  0 08:35 ?        00:00:00 /usr/lib/systemd/systemd-journald
root         100       1  0 08:35 ?        00:00:00 /usr/lib/systemd/systemd-udevd
systemd+     175       1  0 08:35 ?        00:00:00 /usr/lib/systemd/systemd-resolved
systemd+     176       1  0 08:35 ?        00:00:00 /usr/lib/systemd/systemd-timesyncd
root         189       1  0 08:35 ?        00:00:00 /usr/sbin/cron -f -P
```

kill

用于结束进程。kill PID 可以结束某 PID 进程，使用 kill -s PID 可以强制结束，killall *uid_name* 可以结束所有以这个进程名（*uid_name*）为开头的进程。

mount

用于查看系统挂载的列表。

umount

用于卸载设备。umount [device | directory]可以卸载设备，如果该设备正被使用，那么系统不会卸载。

1.4 文件操作

ls

展示出当前目录中的内容，可以跟随以下参数：

-l　展示出目录下的属性（属于谁、文件权限），可以在后面添加文件名以执行精确搜索，也可以使用？（一个字符）或*（多个字符）进行模糊匹配【通配符】。也可以使用[a-z]这种方式，进行单字符定域模糊匹配。或者使用[!a]方式，进行排除 a 的单字符定域模糊匹配。

-a　展示出所有文件（包含隐藏文件）。

-i　展示出 iNode 号。

-F　区分文件和文件夹。

```
kuladmir@kuladmir:~$ ls
run_vina.sh  vina_1.2.5_linux_x86_64
kuladmir@kuladmir:~$ ls -l
total 3996
-rw-r--r-- 1 kuladmir kuladmir       0 Dec 17 15:04 run_vina.sh
-rwxrwxrwx 1 kuladmir kuladmir 4089576 Dec 17 14:59 vina_1.2.5_linux_x86_64
kuladmir@kuladmir:~$ ls -i
51494 run_vina.sh  49264 vina_1.2.5_linux_x86_64
kuladmir@kuladmir:~$
```

pwd

展示出当前工作目录。

```
kuladmir@kuladmir:~$ pwd
/home/kuladmir
```

cd *directort_name*

可更改当前目录（需要为当前目录中有的目录），目录名处可以选择以下参数，不添加时，可以直接从当前目录切到主目录（/home/user）：

/　更改到根目录。

/bin　更改到二进制可执行文件目录。

/etc　更换到配置文件目录。

/home　更换到用户 home 目录。

/dev　更换到配置文件。

..　回到上级目录。

~　可以回到主目录。

可以使用绝对路径进行路径切换（cd /home/user/else），若想访问当前目录下的一个文件夹，可以使用相对路径进行路径切换，但需要保证当前目录下有该文件夹（cd else）。也可以使用.（当前目录）和..（当前目录的父目录），进行切换。

```
kuladmir@kuladmir:~$ ls
run_vina.sh  vina_1.2.5_linux_x86_64
kuladmir@kuladmir:~$ cd /
kuladmir@kuladmir:/$ ls
bin                dev   init              lib64       mnt   root  sbin.usr-is-merged  sys  var         wslIGclBe
bin.usr-is-merged  etc   lib               lost+found  opt   run   snap                tmp  wslGIkbCe   wslgCmMAe
boot               home  lib.usr-is-merged  media       proc  sbin  srv                 usr  wslHJLaCe   wsllpLOCe
kuladmir@kuladmir:/$ cd /home/kuladmir
kuladmir@kuladmir:~$ ls
run_vina.sh  vina_1.2.5_linux_x86_64
kuladmir@kuladmir:~$ cd ..
kuladmir@kuladmir:/home$ ls
kuladmir
kuladmir@kuladmir:/home$
```

在 WSL 中，/mnt/c 或/mnt/d 可以访问到 Windows 的 C 盘和 D 盘。

echo $HOME

可以看到自己的家目录。

```
kuladmir@kuladmir:~$ echo $HOME
/home/kuladmir
```

clear

清屏。

nano *file_name*

可以修改某文件，里面的^为 Ctrl，保存可以使用 Ctrl+O/S，之后 Enter 保存，Ctrl+X 退出。

vim *file_name*

可以打开一个文件，之后点击 i 即可修改（进入插入模式），修改后点击 esc 退出修改（进入普通模式）。

普通模式下：h、j、k、l 可以分别作为向左、向下、向上、向右移动，Ctrl+F 可以下翻一页，Ctrl+B 可以上翻一页）。

:wq　即可退出并保存。

:q　可以在未修改时退出。

:q!　取消修改后退出。

:w filename　可以把文件内容保存到另一个文件里。

除此之外，/ 意为查询：在 / 后输入要查询的内容即可。每次查询到后，会把光标放到首次出现的位置，如果查找相同的内容，下次查询光标则会放到下次出现的位置（第二次查询 in 时，光标会放在第二次出现 in 的地方），或者可以按/加回车或者 n 完成快速查询。

:s 为替换。:s/*old word*/*new word*/可以把光标出的 old word 替换为 new word（光标必须在 old word 处，否则不能替换）。

:s/*old*/*new*/g　可以把一行内的 old word 都替换。

:n,ms/*old*/*new*/g　可以把 n 行到 m 行的 old 都替换。

:%s/*old*/*new*/g　可以把全文的 old 替换。

:%s/*old*/*new*/gc　可以全文替换，但在出现时提示。

普通模式下，存在一些指令可以对文本进行修改（命令前可以加入数字，完成多个字符删除）：

x　删除当前光标所在位置的字符。

dd　删除当前光标所在行。

dw　删除当前光标所在位置的单词。

d$　删除当前光标所在位置至行尾的内容。

J　删除当前光标所在行行尾的换行符（拼接行）。

u　撤销前一编辑命令。

a　在当前光标后追加数据，激活插入模式，和 i 效果相同。

A　在当前光标所在行行尾追加数据，激活插入模式，同时光标跳到行尾。

r char　用 char 替换当前光标所在位置的单个字符。

R test　用 text 覆盖当前光标所在位置的数据，直到按下 ESC 键。

在普通模式删除后（其实类似于 Windows 剪切），可以使用 p 将删除内容在复制回来（类似于 Windows 粘贴）。y 可以实现复制，不过为了可视化复制的内容，将光标放在复制的开头，按 v 可以可视化处理。

cat *file_name*

可以把文件内容展示出来，但是不像 nano 和 vim 一样能够编辑。tac 命令与之相反，可以反向输出内容。

```
(base) kuladmir@kuladmir:~/test$ cat 1.c

#include<stdio.h>
int main()
{
        printf("Hello World\n");
        printf("Good\n");
        return 0;
}
```

cat 命令中的一些参数：

-n 显示行数，-b 只显示有文本内容的行数。

```
(base) kuladmir@kuladmir:~/test$ cat -n 1.c
     1
     2  #include<stdio.h>
     3  int main()
     4  {
     5          printf("Hello World\n");
     6          printf("Good\n");
     7          return 0;
     8  }
```

```
(base) kuladmir@kuladmir:~/test$ cat -b 1.c
     1  #include<stdio.h>
     2  int main()
     3  {
     4          printf("Hello World\n");
     5          printf("Good\n");
     6          return 0;
     7  }
```

less/more　*file_name*

对于文本较长的文件，cat 会全部输出，less 和 more 则会先输出一页，但是 more 只能使用空格或 Enter 完成前进阅读，less 则可以上下翻页。

tail/head　*file_name*

tail 可以从后输出默认 10 行内容，可以加入参数-n 和数字，进行控制输出。可以加入-f 参数，实时显示文件内容。head 与此相反，则是从头输出，也可以使用-n 和数字调控，但不存在-f 参数。

```
(base) kuladmir@kuladmir:~/test$ tail -n 5 1.c
{
        printf("Hello World\n");
        printf("Good\n");
        return 0;
}
```

mkdir *directory_name*

用来创建新目录，但是如果想一次性产生多个目录，可以使用-p 参数。

```
(base) kuladmir@kuladmir:~$ mkdir test/test1
mkdir: cannot create directory ‘test/test1’: No such file or directory
(base) kuladmir@kuladmir:~$ mkdir -p test/test
(base) kuladmir@kuladmir:~$ ls
miniconda3  test  vina
(base) kuladmir@kuladmir:~$ cd ./test/test
(base) kuladmir@kuladmir:~/test/test$ ls
(base) kuladmir@kuladmir:~/test/test$ pwd
/home/kuladmir/test/test
(base) kuladmir@kuladmir:~/test/test$
```

rm -r *directory_name*

用来删除目录，使用时建议加上参数：-i 使得系统询问是否要删除。终极大法：rm -rf *directory_name* 可以删除该目录下所有内容（!）。

rmdir *directory_name*

用来删除一个空目录，非空目录无法删除。

```
(base) kuladmir@kuladmir:~$ ls ./test
test
(base) kuladmir@kuladmir:~$ ls
miniconda3  test  vina
(base) kuladmir@kuladmir:~$ rmdir test
rmdir: failed to remove 'test': Directory not empty
```

mv *file_name newname*

用来修改文件名，如果新文件名处为一个位置，则会将此文件移到对应位置。建议添加参数-i 使得系统提示是否覆盖已存在文件。mv 只会改变文件的名，不会影响时间戳和 inode 编号。注意，mv 也可以对文件夹进行操作。

```
(base) kuladmir@kuladmir:~$ mkdir test
(base) kuladmir@kuladmir:~$ ls
miniconda3  test  vina
(base) kuladmir@kuladmir:~$ cd test
(base) kuladmir@kuladmir:~/test$ ls
(base) kuladmir@kuladmir:~/test$ mkdir test1
(base) kuladmir@kuladmir:~/test$ ls
test1
(base) kuladmir@kuladmir:~/test$ mkdir tet3
(base) kuladmir@kuladmir:~/test$ ls
test1  tet3
(base) kuladmir@kuladmir:~/test$ rm -r tet3
(base) kuladmir@kuladmir:~/test$ ls
test1
(base) kuladmir@kuladmir:~/test$ mv test1 ..
(base) kuladmir@kuladmir:~/test$ ls
(base) kuladmir@kuladmir:~/test$ cd
(base) kuladmir@kuladmir:~$ ls
miniconda3  test  test1  vina
(base) kuladmir@kuladmir:~$ mv test1 ./test2
(base) kuladmir@kuladmir:~$ ls
miniconda3  test  test2  vina
(base) kuladmir@kuladmir:~$
```

cp *file_name-ed*　*new_filename*

可以用来复制文件，使用时建议加上参数：-i 以保证有重复文件时，系统会询问时候覆盖。*file_name-ed* 是被复制的文件名，其他参数：

-R　将一个文件夹的内容复制，并保存在新的一个文件夹内（做副本）

cp *../file_name-ed*　*new_filename*

用以将上一目录的文件复制到该目录。

rm *file_name*

用来删除文件。使用时建议加上参数：-i 使得系统询问是否要删除。此时文件名可以引入通配符（?或*）进行文件的删除操作。-f 可以使得系统强制删除。

touch *file_name*

可以用来创建文件，只是创建出来，不会打开写入。

```
(base) kuladmir@kuladmir:~$ cd ./test
(base) kuladmir@kuladmir:~/test$ ls
(base) kuladmir@kuladmir:~/test$ touch 1
(base) kuladmir@kuladmir:~/test$ ls
1
(base) kuladmir@kuladmir:~/test$ cp 1 2
(base) kuladmir@kuladmir:~/test$ ls
1  2
(base) kuladmir@kuladmir:~/test$ rm 1
(base) kuladmir@kuladmir:~/test$ ls
2
(base) kuladmir@kuladmir:~/test$
```

echo “*content*” > *file_name*

可以将内容写入文件。

```
(base) kuladmir@kuladmir:~/test$ echo "Hello World" > 1.txt
(base) kuladmir@kuladmir:~/test$ ls
1.txt  2
(base) kuladmir@kuladmir:~/test$ cat 1.txt
Hello World
```

tar

可以用来对文件进行归档，往往与参数共同使用。可用参数：

-c 创建一个新的归档文件。

-d 检查归档文件和文件系统的不同之处。

-r 追加文件到已有 tar 归档文件末尾。

-t 列出已有 tar 归档文件的内容。

-u 将比 tar 归档文件中已有的同名文件新的文件追加到该 tar 归档文件中。

-x 从已有 tar 归档文件中提取文件。

-f 输出结果到文件或设备。

-v 在处理文件时显示文件。

-z 将输出重定向给 gzip 命令来压缩内容 。

-p 保留所有文件权限。

常用组合：

tar -cvf *name.tar [directory]* 将某文件创建归档成一个新文件 name.tar，似乎压缩文件不一定要为 tar，可以 tar 改成 rar 和 zip。

tar -tf *name.tar* 可以列出 name.tar 里的文件。

tar -xvf *name.tar* 用于对文件进行解压

```
(base) kuladmir@kuladmir:~/test$ tar -cvf test.tar 1.c
1.c
(base) kuladmir@kuladmir:~/test$ ls
1.c  a.out  test  test.tar
(base) kuladmir@kuladmir:~/test$ tar -tf test.tar
1.c
(base) kuladmir@kuladmir:~/test$ ls
1.c  a.out  test  test.tar
(base) kuladmir@kuladmir:~/test$ rm 1.c
(base) kuladmir@kuladmir:~/test$ ls
a.out  test  test.tar
(base) kuladmir@kuladmir:~/test$ tar -xvf test.tar
1.c
(base) kuladmir@kuladmir:~/test$ ;s
-bash: syntax error near unexpected token `;'
(base) kuladmir@kuladmir:~/test$ ls
1.c  a.out  test  test.tar
```

zip *f_name.zip file_name*

可以用来压缩对应文件，并生成目标文件。

unzip -d *directory_name* *file_name.zip*

可以将 zip 文件压缩到给定目录中（如果当前目录没有则会创建）。

gzip *file_name*

用于把文件压缩生成压缩文件，gzip 会把源文件直接压缩生成新文件。使用通配符，可以批量完成压缩。gunzip 可以解压缩。

```
(base) kuladmir@kuladmir:~/test$ ls
1.c  a.out  s1.c  test
(base) kuladmir@kuladmir:~/test$ gzip 1.c
(base) kuladmir@kuladmir:~/test$ ;s
-bash: syntax error near unexpected token `;'
(base) kuladmir@kuladmir:~/test$ ls
1.c.gz  a.out  s1.c  test
```

write *user*

用来和他人聊天。

wall *info*

广播信息。

ln -s *file_name* *new_name*（符号链接）

用于给已有的文件建立一个指针（自身就是个新文件），指针指向这个文件。

```
(base) kuladmir@kuladmir:~/test$ ln -s 1.c s1.c
(base) kuladmir@kuladmir:~/test$ ls
1.c  s1.c
(base) kuladmir@kuladmir:~/test$ ls
1.c  s1.c
(base) kuladmir@kuladmir:~/test$ ls -l
total 0
-rw-r--r-- 1 kuladmir kuladmir 0 Dec 19 13:21 1.c
lrwxrwxrwx 1 kuladmir kuladmir 3 Dec 19 13:21 s1.c -> 1.c
```

ln *file_name* *new_name*（硬链接）

用于创建包含了源文件信息和位置的文件，但本质上，两者是同一个文件。

```
(base) kuladmir@kuladmir:~/test$ ln 1.c s2.c
(base) kuladmir@kuladmir:~/test$ ls
1.c  s1.c  s2.c
(base) kuladmir@kuladmir:~/test$ ls -i
37399 1.c  37401 s1.c  37399 s2.c
```

file *file_name*

可以查看文件的类型（例如是文件夹、ASCII、链接等）。

```
(base) kuladmir@kuladmir:~/test$ ls
1.c  s1.c  test
(base) kuladmir@kuladmir:~/test$ file 1.c
1.c: C source, ASCII text
(base) kuladmir@kuladmir:~/test$ file test
test: directory
(base) kuladmir@kuladmir:~/test$ file s1.c
s1.c: symbolic link to 1.c
(base) kuladmir@kuladmir:~/test$
```

1.5 用户指令

useradd -d *main_directory* -m *directory*

可以创建一个用户，并产生一个目录。可选参数：

-c 给新用户添加备注。

-d 为主目录指定一个名字。

-e 用 YYYY-MM-DD 方式，设置一个账户过期的日期。

-f 指定密码过期后多久禁用，0 表示过期就禁用，-1 表示禁用此功能。

-g 指定用户登录组的 GID 或组名。

-m 创建用户目录。

-k 必须和-m 同用，将/etc/skel 目录的内容复制到用户的 HOME 目录。

-n 创建一个与用户登录名同名的新组。

-D 修改默认值，后面可以跟-e（账户过期日期） -f（密码过期到禁用的天数）-g（更改组名或 GID） -s（登录 shell）。

userdel -r *user*

可以删除用户，并删除主目录。不加-r 则只会删除/etc/passwd 下的内容。

usermod *user*

可以修改已有用户的信息。可选参数：

-l 修改用户账户登录名。

-L 锁定，使无法登录。

-p 修改密码。

-U 解除锁定。

-G 添加进组（usermod -G *group_name user*）

passwd *user*

可以对用户密码进行操作，只有 passwd 时会执行修改密码，可以添加如下参数：

-l 锁定口令，即禁用账号。

-u 口令解锁。

-d 使账号无口令。

-e 强迫用户下次登录时修改口令

chpasswd

能从标准输入自动读取登录名和密码对（由冒号分割）列表，给密码加密，然后为用户账户设置。

chsh -s shell *absolute_path* **user**

用于快速修改用户的登录 shell。

chfh *user*

可以在/etc/passwd 中添加一些备注信息。

chage

用于管理用户的有效期。可选参数：

-d 设置上次修改密码到现在的天数。

-E 设置密码过期的日期。

-I 设置密码过期到锁定账户的天数。

-m 设置修改密码之间最少要多少天。

-W 设置密码过期前多久开始出现提醒信息。

finger

可以查看用户的 Linux 有关信息，一般会禁用。

groupadd *group_name*

可以增加一个新的用户组，可以加入如下参数：

-g GID 号 可以指定用户的组标识号

groupdel *group_name*

可以删除一个用户组。

groupmod -g GID *group_name*

可以修改用户组名的 GID 号。

groupmod -n 新用户组名 旧用户组名

可以修改用户组名。

hostname

用来显示机器主机名。

文件权限：

```
(base) kuladmir@kuladmir:~$ ls -l
total 12
-rw-r--r--  1 kuladmir kuladmir    0 Dec 23 21:19 1.txt
drwxr-xr-x 19 kuladmir kuladmir 4096 Dec 19 10:39 miniconda3
```

在第一列中，第一位 d，表示这是个文件夹，-表示这是个文件。2-4 位的三个字母表示用户的权限，5-7 位表示同组成员的权限，8-10 位表示其他人权限。r 为可读，w 为可写，x 为可执行。

chmod o(+/-)r *file_name*

删除文件阅读权限，+表示给权限，-表示取消权限。o 表示其他人，u 表示用户，g 表示同组，a 表示所有。

chmod *3*/o_number file_name*

增加文件阅读和读写权限，也可以使用三位数字表示权限，r=4，w=2，x=1。用数字之和表示权限，4 为只读，6 为读写，7 为全部，0 为无权限。文件权限顺序：用户、同组、其他人。当使用 chmod +x 将其设为程序时，可以使用./文件名执行，当编译.c 文件时，需要执行命令：gcc file.c 进行编译。

```
(base) kuladmir@kuladmir:~/test$ ls -l
total 0
-rw-r--r-- 1 kuladmir kuladmir 0 Dec 24 19:03 1.txt
(base) kuladmir@kuladmir:~/test$ chmod a+w 1.txt
(base) kuladmir@kuladmir:~/test$ ls -l
total 0
-rw-rw-rw- 1 kuladmir kuladmir 0 Dec 24 19:03 1.txt
(base) kuladmir@kuladmir:~/test$ chmod 711 1.txt
(base) kuladmir@kuladmir:~/test$ ls -l
total 0
-rwx--x--x 1 kuladmir kuladmir 0 Dec 24 19:03 1.txt
```

chown username/UID *filename*

用以更改文件的归属权，并且也可以更改属组，只要 username.groupname 即可。只有 root 才能更改归属权。

chgrp groupname *filename*

用以更改文件的属组，这个文件需要为用户所有的。

umask *3*/o_number filename*

用于减少文件的权限，最后的权限为：源文件权限减去 umask 的参数权限值，即：一个文件为 777，umask 022 后，会成为 755 权限。

1.6 网络查询命令

ping 网址

用以检查和对应网址连接情况 Ctrl+C 退出

tracepath 网址

用以检查和网址连接

traceroute 网址

用以检查和网址连接

ifconfig

用以查看本地 IP{Windows 为 ipconfig}

eth0 表示第一块网卡，HWaddr 表示网卡物理地址。第一行：连接类型：Ethernet（以太网）HWaddr（硬件 mac 地址）第二行：网卡的 IP 地址、子网、掩码，第三行：UP（代表网卡开启状态）RUNNING（代表网卡的网线被接上）MULTICAST（支持组播）MTU:1500（最大传输单元）：1500 字节，第四、五行：接收、发送数据包情况统计。

route

用以查看网关，配置文件：/etc/resolv.conf

netstat | more

用以查看分页显示程序，netstat 后加-n，可以让本地地址显示为地址。

1.7 包管理系统（PMS，package management system）

sudo dpkg -i deb *file_name*

可以安装一个 deb 包。

sudo dpkg -r *file_name*

可以删除安装的 deb 包。

sudo dpkg -p *file_name*

可以删除安装的 deb 包，并删除配置文件。

dpkg -L *package_name*

可以显示某个包相关的所有文件列表。

```
(base) kuladmir@kuladmir:~/test$ dpkg -L vim
/.
/usr
/usr/bin
/usr/bin/vim.basic
/usr/share
/usr/share/bug
/usr/share/bug/vim
/usr/share/bug/vim/presubj
/usr/share/bug/vim/script
/usr/share/doc
```

sudo apt-get install *file_name*

可以安装一个文件。

sudo apt-get remove *file_name*

可以卸载一个文件。

sudo apt-get update

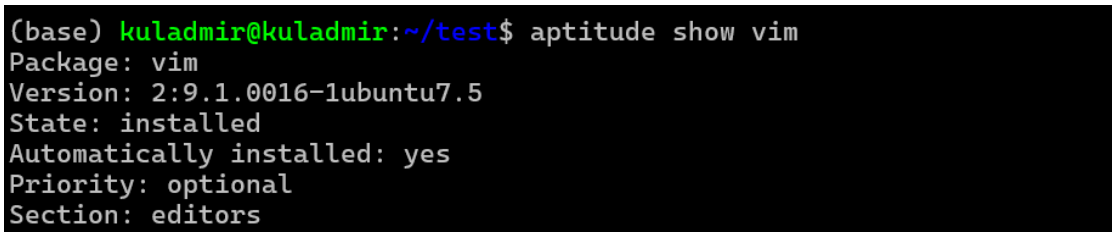
可以更新源。

aptitude

只输入 aptitude 可以显示出目前安装包的情况，按 q 可以退出。

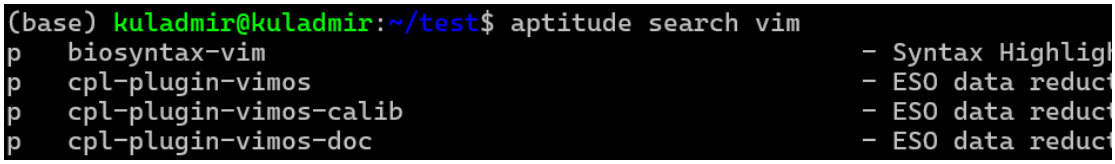
aptitude show *package_name*

可以显示包的详情。



aptitude search *pakage_name*

可以查找任何与 package_name 相关的包，这就说明它隐含一个通配符。如果包前面有 i 表示这个包已经安装，否则说明没安装。



sudo aptitude install *package_name*

可以安装一个包。

sudo aptitude safe-upgrade

可以将所有已安装的包全部更新，并且比较稳定。

sudo aptitude purge/remove *package_name*

可以删除一个包。remove 时可以保留数据和配置文件，purge 则不会。并且，删除后在使用 aptitude search *package_name* 进行搜索时，如果显示 c，则表示配置文件未清除，否则显示未 p。

以上都是 Ubuntu（Debian 下属的）系统常用的安装方法，以下是 Red Hat 系统常用的命令。

yum list installed > *filename*

可以用来把已安装的包信息显示，并重定向到一个文件内，方便查看。

yum list (installed) *package_name*

可以用来显示某个包的具体信息，添加 installed 后，会显示这个包是否安装。

sudo yum install *package_name*

可以用来安装一个软件包。

sudo yum localinstall *package_name.rpm*

使用 rpm 下载软件包后，用 yum 安装（本地安装）。

yum list updates

可以检测是否有需要更新的包，没有则返回空。

yum update (*package_name*)

可以更新一个包或者所有包（不加特定 package_name）。

yum remove/erase *package_name*

可以卸载一个软件包，remove 会保留数据和配置文件，erase 则不会。

yum clean all

当发生一个包的依赖关系被破坏时，可以使用处理。如果不行，可以使用

yum deplist *package_name*

显示包的依赖关系，以此保证用户可以安装。如果还是不可以，那么可以使用

yum update --skip-broken

跳过存在问题的包的安装，完成接下来的安装。

从源码安装，往往需要在官网下载一个版本的软件包（.tar.gz），之后使用命令 tar -zxvf name.tar.gz 进行解压，完成解压后会产生一个文件夹。根据需求，可以使用./configure 完成系统配置，保证系统能够编码源码。make 能够构建二进制文件，产生可执行文件。

2. 使用 shell

Shell 本质上是 Linux 上一个介于用户和计算机之间的程序，用于将用户的命令转化为电信号，使得计算机明白，并执行命令。bash shell 是 Ubuntu 发行的 Linux 下的 shell 程序，在 CLI（Command Line Interface，命令行界面）中，已经存在一个 shell 程序，可以使用 bash 再开一个 shell 程序，这个新开的 shell 是子 shell 程序，已存在的为父 shell 程序。

有趣的是，可以通过 ps -f 查询到这一情况，并且随时可以使用 exit 退出一层 shell（类似于嵌套）。

```
(base) kuladmir@kuladmir:~$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
kuladmir     263     259  0 14:45 pts/0    00:00:00 -bash
kuladmir     422     263  0 15:01 pts/0    00:00:00 ps -f
(base) kuladmir@kuladmir:~$ bash
(base) kuladmir@kuladmir:~$ bash
(base) kuladmir@kuladmir:~$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
kuladmir     263     259  0 14:45 pts/0    00:00:00 -bash
kuladmir     423     263  0 15:01 pts/0    00:00:00 bash
kuladmir     433     423  0 15:01 pts/0    00:00:00 bash
kuladmir     443     433  0 15:01 pts/0    00:00:00 ps -f
(base) kuladmir@kuladmir:~$ exit
exit
(base) kuladmir@kuladmir:~$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
kuladmir     263     259  0 14:45 pts/0    00:00:00 -bash
kuladmir     423     263  0 15:01 pts/0    00:00:00 bash
kuladmir     444     423  0 15:01 pts/0    00:00:00 ps -f
```

而所有可用命令中，一部分命令为外部命令，一部分为内部命令。可以使用 which command 和 type command 的方式进行确认。如果返回一个路径，例如/bin/ps 之类的，则这是个外部命令，否则返回一个 shell builtin。然而，which 命令似乎不适用于内部命令。

```
(base) kuladmir@kuladmir:~$ which ps
/usr/bin/ps
(base) kuladmir@kuladmir:~$ type -a ps
ps is /usr/bin/ps
ps is /bin/ps
(base) kuladmir@kuladmir:~$ which pwd
/usr/bin/pwd
(base) kuladmir@kuladmir:~$ type cd
cd is a shell builtin
(base) kuladmir@kuladmir:~$ which cd
```

可以在一行内输入多个指令，只要每个指令用 ; 隔开，则每个指令都能按顺序执行。

```
(base) kuladmir@kuladmir:~$ pwd; ls
/home/kuladmir
miniconda3  test  vina
(base) kuladmir@kuladmir:~$ pwd; ls; ls -l; ps -f
/home/kuladmir
miniconda3  test  vina
total 12
drwxr-xr-x 19 kuladmir kuladmir 4096 Dec 19 10:39 miniconda3
drwxr-xr-x  3 kuladmir kuladmir 4096 Dec 23 09:58 test
drwxr-xr-x  2 kuladmir kuladmir 4096 Dec 19 09:02 vina
UID          PID    PPID  C STIME TTY          TIME CMD
kuladmir     263     259  0 14:45 pts/0    00:00:00 -bash
kuladmir     463     263  0 15:04 pts/0    00:00:00 ps -f
```

需要注意的是，上述方式为命令分组，如果在命令分组外加入一个()，则可以使得这个命令分组变成一个进程列表，特点是，这些命令会在新的 shell 中执行，之后执行完自动退出。

```
(base) kuladmir@kuladmir:~$ (ps; ls; ls -l)
    PID TTY          TIME CMD
    263 pts/0    00:00:00 bash
    465 pts/0    00:00:00 bash
    466 pts/0    00:00:00 ps
miniconda3  test  vina
total 12
drwxr-xr-x 19 kuladmir kuladmir 4096 Dec 19 10:39 miniconda3
drwxr-xr-x  3 kuladmir kuladmir 4096 Dec 23 09:58 test
drwxr-xr-x  2 kuladmir kuladmir 4096 Dec 19 09:02 vina
(base) kuladmir@kuladmir:~$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
kuladmir     263     259  0 14:45 pts/0    00:00:00 -bash
kuladmir     468     263  0 15:07 pts/0    00:00:00 ps -f
```

也可以使用一个命令进行检查：echo $BASH_SUBSHELL。如果返回非 0，说明这是在新 shell 运行的，否则不是。

```
(base) kuladmir@kuladmir:~$ pwd; ls -l; echo $BASH_SUBSHELL
/home/kuladmir
total 12
drwxr-xr-x 19 kuladmir kuladmir 4096 Dec 19 10:39 miniconda3
drwxr-xr-x  3 kuladmir kuladmir 4096 Dec 23 09:58 test
drwxr-xr-x  2 kuladmir kuladmir 4096 Dec 19 09:02 vina
0
(base) kuladmir@kuladmir:~$ (pwd; ls -l;echo $BASH_SUBSHELL)
/home/kuladmir
total 12
drwxr-xr-x 19 kuladmir kuladmir 4096 Dec 19 10:39 miniconda3
drwxr-xr-x  3 kuladmir kuladmir 4096 Dec 23 09:58 test
drwxr-xr-x  2 kuladmir kuladmir 4096 Dec 19 09:02 vina
1
```

注意，在进程列表中，也可以存在一个进程列表，根据层数，echo $BASH_SUBSHELL 会返回不同的值。

sleep *time*(s)

可以让程序主动睡眠一定时间，在后面加上&可以让其转为后台睡眠。ps -f 可以看到执行。睡眠结束后，会显示一个 Done。

jobs

可以显示出目前正在执行的进程。可用的参数：

-l　列出进程的 PID 以及作业号。

-p　只列出作业的 PID。

-n　只列出上次 shell 发出的通知后改变了状态的作业。

-r　只列出运行中的作业。

-s　只列出已停止的作业。

使用 jobs 命令后，能看到[n]后面有+/-，+表示这是默认作业，并不是所有[n]后面都有+/-。

```
(base) kuladmir@kuladmir:~$ sleep 10&
[1] 501
(base) kuladmir@kuladmir:~$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
kuladmir     263     259  0 14:45 pts/0    00:00:00 -bash
kuladmir     501     263  0 15:19 pts/0    00:00:00 sleep 10
kuladmir     502     263  0 15:19 pts/0    00:00:00 ps -f
(base) kuladmir@kuladmir:~$ jobs
[1]+  Running                 sleep 10 &
(base) kuladmir@kuladmir:~$ ls
miniconda3  test  vina
```

coproc *DIY_name* (command;) &

该命令允许生成一个子 shell，并完成对应任务。注意，必须要保持格式，每个部分需要包有一个空格的间距。

```
(base) kuladmir@kuladmir:~$ coproc My (sleep 10; ls -l; ) &
[1] 506
(base) kuladmir@kuladmir:~$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
kuladmir     263     259  0 14:45 pts/0    00:00:00 -bash
kuladmir     506     263  0 15:25 pts/0    00:00:00 -bash
kuladmir     507     506  0 15:25 pts/0    00:00:00 sleep 10
kuladmir     508     263  0 15:25 pts/0    00:00:00 ps -f
```

histroy

可以显示最近用的 1000 条命令，在环境变量中可以修改其展示的条数。使 -a 参数可以强制将一些历史记录写入保存的文件中，-n 可以更新历史记录。需要说明的是，一些历史会先存在内存，当 shell 退出后才会写入历史文件中。

用 !! 也可以再次完成上一次执行的命令。或者 ! n 也可以执行第 n 条指令。

```
(base) kuladmir@kuladmir:~$ history
    1  ls
    2  touch test
    3  echo 1 > test
    4  code test
    5  code run_vina/sh
    6  code run_vina.sh
    7  ls
    8  mv /mnt/c/Users/11750/Desktop/vina_1.2.5_linux_x86_64  .
    9  ls
   10  ls -a
   11  ssh-keygen -t rsa
```

```
(base) kuladmir@kuladmir:~$ ls -l
total 12
drwxr-xr-x 19 kuladmir kuladmir 4096 Dec 19 10:39 miniconda3
drwxr-xr-x  3 kuladmir kuladmir 4096 Dec 23 09:58 test
drwxr-xr-x  2 kuladmir kuladmir 4096 Dec 19 09:02 vina
(base) kuladmir@kuladmir:~$ !!
ls -l
total 12
drwxr-xr-x 19 kuladmir kuladmir 4096 Dec 19 10:39 miniconda3
drwxr-xr-x  3 kuladmir kuladmir 4096 Dec 23 09:58 test
drwxr-xr-x  2 kuladmir kuladmir 4096 Dec 19 09:02 vina
```

alias

可以允许对一些命令进行别名化，有一些命令已被自动别名，alias -p 可以显示。或者也可以自己定义：alias *DIY_command*='std_command'，注意格式：等号之间没有空格。

```
(base) kuladmir@kuladmir:~$ alias lii='ls -l'
(base) kuladmir@kuladmir:~$ ls
miniconda3  test  vina
(base) kuladmir@kuladmir:~$ lii
total 12
drwxr-xr-x 19 kuladmir kuladmir 4096 Dec 19 10:39 miniconda3
drwxr-xr-x  3 kuladmir kuladmir 4096 Dec 23 09:58 test
drwxr-xr-x  2 kuladmir kuladmir 4096 Dec 19 09:02 vina
```

注意，自己别名处理的一些命令，只在当前 shell 下有效，在别的 shell 下，不会生效。

3. 环境变量

printenv 或 env

可以显示所有的全局变量，如果要显示个别的，应使用 printenv，echo 也可以显示变量的值，不过要加个$。

```
(base) kuladmir@kuladmir:~$ printenv HOME
/home/kuladmir
(base) kuladmir@kuladmir:~$ echo $HOME
/home/kuladmir
```

set

可以显示出所有的全局变量和环境变量。

可以自定义**局部环境变量**。第一种方法是使用 echo $name，之后给 name 赋值；或者直接定义（注意下面的格式，=之间没有空格）。在自定义局部变量时，最好把变量定义为小写的，避免未知麻烦。

```
(base) kuladmir@kuladmir:~$ echo $test

(base) kuladmir@kuladmir:~$ test=kula
(base) kuladmir@kuladmir:~$ echo $test
kula
```

```
(base) kuladmir@kuladmir:~$ testa="Kula"
(base) kuladmir@kuladmir:~$ echo $testa
Kula
```

可以自定义全局变量，只要使用 export 变量，将其转化为全局变量就可以了。不过，在子 shell 中修改全局变量后，父 shell 中不会显示修改后的值。子 shell 的权限相对低，并且子 shell 中定义的全局变量，也不会生效；同时，在子 shell 删除全局变量后，也不会对父 shell 造成影响。

```
(base) kuladmir@kuladmir:~$ eho="Test"
(base) kuladmir@kuladmir:~$ echo $eho
Test
(base) kuladmir@kuladmir:~$ bash
(base) kuladmir@kuladmir:~$ eho="TEST"
(base) kuladmir@kuladmir:~$ echo $ eho
$ eho
(base) kuladmir@kuladmir:~$ echo $eho
TEST
(base) kuladmir@kuladmir:~$ exit
exit
(base) kuladmir@kuladmir:~$ echo $eho
Test
```

unset envir_var

可以删除一个环境变量。

在环境变量中引入一个新的路径：PATH=$PATH:new_path。之后可以使用 echo $PATH 进行查看已有的环境变量，不过这种方法只能保留一时。最好是在 /etc/profile.d 目录中创建一个以.sh 结尾的文件。把所有新的或修改过的全局环境变量设置放在这个文件中。存储个人用户永久性 bash shell 变量的地方是 $HOME/.bashrc 文件。

数组变量的定义：arr_name=(a b c d e)。

数组变量的遍历：echo $arr_name 只会显示第一个值，即 arr_name[0]，如果要访问其他值，需要加入索引：echo ${arr_name[n]}，当 n 为*时，输出数组所有值。

数组变量的修改：arr_name[n]=new_value。

数组变量删除：unset arr_name 可以删除整个数组，unset arr_name[n]可以删除对应的值。

```
(base) kuladmir@kuladmir:~$ arr=(one two three four)
(base) kuladmir@kuladmir:~$ echo $arr
one
(base) kuladmir@kuladmir:~$ echo ${arr[*]}
one two three four
(base) kuladmir@kuladmir:~$ echo ${arr[1]}
two
(base) kuladmir@kuladmir:~$ arr[1]=five
(base) kuladmir@kuladmir:~$ echo ${arr[1]}
five
(base) kuladmir@kuladmir:~$ unset arr[2]
(base) kuladmir@kuladmir:~$ echo ${arr[*]}
one five four
(base) kuladmir@kuladmir:~$ unset arr
```

4. 普通 shell 脚本

4.1 构建脚本

第一步：创建.sh 文件。vim *file_name*.sh，进入后在一行输入#! /bin/bash，# 可以作为注释标志，其后的内容不会被编译，但是对于第一行例外。建议还是加一些别的注释，保证可读性。

第二步：编辑.sh 文件。在.sh 文件内写入命令，之后保存。

```
#!/bin/bash
#test
        date
        ls -l
~
```

第三步：修改.sh 文件的权限，可以使用 chmod u+x *name*.sh。

```
(base) kuladmir@kuladmir:~/test$ ls -l
total 20
-rw-r--r-- 1 kuladmir kuladmir    32 Dec 25 09:57 1.sh
-rwxr-xr-x 1 kuladmir kuladmir 15960 Dec 24 20:24 a.out
(base) kuladmir@kuladmir:~/test$ chmod u+x 1.sh
(base) kuladmir@kuladmir:~/test$ ls -l
total 20
-rwxr--r-- 1 kuladmir kuladmir    32 Dec 25 09:57 1.sh
-rwxr-xr-x 1 kuladmir kuladmir 15960 Dec 24 20:24 a.out
```

第四步：.sh 文件执行。为了保证系统能够找到，可以将这个目录加入到环境变量中 PATH=$PATH:*new_path*，或者在执行时使用./name.sh 即可。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
Wed Dec 25 09:58:05 CST 2024
total 20
-rwxr--r-- 1 kuladmir kuladmir    32 Dec 25 09:57 1.sh
-rwxr-xr-x 1 kuladmir kuladmir 15960 Dec 24 20:24 a.out
```

4.2 显示

echo *content*

可以显示出 content 的内容，如果 content 里面有单引号或双引号，想让它显示出来，那么整个语句应该被双引号和单引号包括。而且单引号和双引号必须成对出现。如果 content 中没有出现引号的必要，则整个语句都不需要括号。

```
(base) kuladmir@kuladmir:~/test$ echo "we're the world"
we're the world
(base) kuladmir@kuladmir:~/test$ echo We're the world'
Were the world
```

echo 也可以加到 shell 脚本中。但是在不同命令输出后会有换行，可以使用 echo -n *content*，这样可以阻止换行。echo 也可以调用环境变量里的值，只要在环境变量前加一个$就可以。不过$最后不会被显示出来，如果想在文本显示$，那么需要使用转义字符\$。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
Today is:Wed Dec 25 10:11:33 CST 2024
UID: 1000
test: The pen is
test1: The pen is $5
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash
#test
        echo -n Today is:
        date
        echo UID: $UID
        echo test: "The pen is $5"
        echo test1: "The pen is \$5"
```

echo 也可以调用局部变量的值。首先要给变量赋值，其次才能调用；调用时，变量前要加$。如果在赋值时，用到另一个变量，则对于数值时，也要加$，否则会识别为一个字符串并赋值。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
The a is 5
The b is b
The a is b
The a is 6
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash
        a=5
        b=6
        echo The a is $a
        echo The b is b
        a=b
        echo The a is $a
        a=$b
        echo The a is $a
```

shell 提供了命令替换功能。variant=`command`或者 variant=$(command)，可以把 variant 替换为一个命令，在使用$调用 variant 时，就可以调用已替换的命令。但$variant 不能在行内单独使用。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
The date is:> Wed Dec 25 10:47:43 CST 2024
The file is:> 1.sh a.out
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash
        var=`date`
        testa=$(ls)
        echo "The date is:>" $var
        echo "The file is:>" $testa
```

做日志（生成一个唯一的文件）：

today=$(date +%y%m%d)　　ls /usr/bin -al > log.$today

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
(base) kuladmir@kuladmir:~/test$ ls
1.sh  a.out  log.241225
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash
        today=$(date +%y%m%d)
        ls /home/kuladmir/test -al > log.$today
```

cat /dev/null > *file_name*

可以清除文件。由于 null 文件不含任何内容，直接将内容重定向到 file_name 里，可以直接清除 file_name 里的文件，但是 file_name 里的内容会被备份。

```
(base) kuladmir@kuladmir:~/test$ cat 1.txt
We are the world
We are the children
(base) kuladmir@kuladmir:~/test$ cat /dev/null > 1.txt
(base) kuladmir@kuladmir:~/test$ cat 1.txt
```

mktemp *file_name*.XXX(XXX...)

可以在当前目录创建一个临时文件，XXX(XXX...)会随机生成多个后缀（X 最少为 3 个）。-t 参数可以强制在/tmp 目录下创建一个命令。-d 参数可以创建一个临时文件夹。

```
(base) kuladmir@kuladmir:~/test$ ls
1.sh  1.txt  2.sh  2.txt
(base) kuladmir@kuladmir:~/test$ mktemp test.XXXXXX
test.IbfqKV
(base) kuladmir@kuladmir:~/test$ mktemp -d te.XXXXXX
te.c8d2TG
(base) kuladmir@kuladmir:~/test$ ls
1.sh  1.txt  2.sh  2.txt  te.c8d2TG  test.IbfqKV
```

tee *file_name*

可以实现 T 型信息输出。往往配合管道使用，可以将一个命令的输出输出在屏幕上，也可以备份一份到文件中。

```
(base) kuladmir@kuladmir:~/test$ date | tee 1.txt
Sat Dec 28 11:17:51 CST 2024
(base) kuladmir@kuladmir:~/test$ cat 1.txt
Sat Dec 28 11:17:51 CST 2024
```

4.3 重定向和管道

command > *file_name*

可以让命令的输出，定向到一个文件中，并保存。文件不存在时会自动创建一个新文件。使用 > 会把文件原内容覆盖，>> 则会追加。

```
(base) kuladmir@kuladmir:~/test$ ls
1.sh  1.txt
(base) kuladmir@kuladmir:~/test$ cat 1.txt
test

(base) kuladmir@kuladmir:~/test$ ls -l > 1.txt
(base) kuladmir@kuladmir:~/test$ cat 1.txt
total 4
-rwxr--r-- 1 kuladmir kuladmir 77 Dec 25 10:57 1.sh
-rw-r--r-- 1 kuladmir kuladmir  0 Dec 26 13:29 1.txt
(base) kuladmir@kuladmir:~/test$ ls -l > 2.txt
(base) kuladmir@kuladmir:~/test$ cat 2.txt
total 8
-rwxr--r-- 1 kuladmir kuladmir  77 Dec 25 10:57 1.sh
-rw-r--r-- 1 kuladmir kuladmir 113 Dec 26 13:29 1.txt
-rw-r--r-- 1 kuladmir kuladmir   0 Dec 26 13:29 2.txt
(base) kuladmir@kuladmir:~/test$ date >> 1.txt
(base) kuladmir@kuladmir:~/test$ cat 1.txt
total 4
-rwxr--r-- 1 kuladmir kuladmir 77 Dec 25 10:57 1.sh
-rw-r--r-- 1 kuladmir kuladmir  0 Dec 26 13:29 1.txt
Thu Dec 26 13:30:52 CST 2024
```

command < file_name

可以让文件内容重定向到命令，<< 可以作为内联输入重定向符号（command << marker），重定向中要有一个标志，可以为 EOF(end of file)。

command_1 | command_2

可以让命令一和命令二链接起来：命令一执行产生的结果将传递到命令二。可以并列多个命令，以从左到右的顺序进行执行。管道也可以和重定向连接。

echo $?

可以显示错误代码。没错误返回 0，有错误则返回非 0。如下是一些举例：

1　一般未知错误。

2　不适合 shell 命令。

126　命令不可执行。

127　没找到命令。

128　无效参数。

文件描述符：STDIN (0)　STDOUT (1)　STDERR (2)。使用文件描述符，可以重定向错误到某个文件，而不会在 CLI 中直接输出。

可以同时重定向错误和数据，只要在后面加上对应的文件描述符数字。

```
(base) kuladmir@kuladmir:~/test$ ls
1.sh  1.txt  2.sh  2.txt
(base) kuladmir@kuladmir:~/test$ ls -l test 1.sh ls.sh 2>1.txt 1>2.txt
(base) kuladmir@kuladmir:~/test$ cat 1.txt
ls: cannot access 'test': No such file or directory
ls: cannot access 'ls.sh': No such file or directory
(base) kuladmir@kuladmir:~/test$ cat 2.txt
-rwxr--r-- 1 kuladmir kuladmir 77 Dec 27 21:55 1.sh
```

上例解释：使用 ls -l 查看 test 1.sh ls.sh 文件，2>1.txt 表示将错误信息传到 1.txt，之后将 ls -l 的输出结果传输到了 2.txt（1>2.txt）。

脚本里也可以实现重定向，包括临时重定向和永久重定向。临时重定向中：需要在重定向的语句后加入> &n，n 是文件描述符数字，之后在执行时，也需要重定向到另一个文件，就可以把指定内容重定向到其他文件。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh 2>1.txt
The file is correct
(base) kuladmir@kuladmir:~/test$ cat 1.txt
The file is error
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash

echo "The file is error" >&2
echo "The file is correct"
```

exec n>*file_name*

可以在脚本实现大量文本的重定向。exec 内的语句也可以使用上述方法，再次实现重定向。【exec 规定了下面一些结果的输出位置，文件描述符为 1 时，会让输出重定向到新位置，不会再输出到屏幕；而文件描述符为 2 时，相当于是定义了一个“如果有错误信息，应该传到哪”，并没真正重定向，只有语句后存在 &2，说明这句是个错误信息，然后被重定向。】

```
(base) kuladmir@kuladmir:~/test$ vim 1.sh
(base) kuladmir@kuladmir:~/test$ ./1.sh
This is a line
This is a line
(base) kuladmir@kuladmir:~/test$ cat 1.txt
This is a test
(base) kuladmir@kuladmir:~/test$ cat 2.txt
This is a test
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash

exec 2>1.txt
echo "This is a line"
echo "This is a line"
exec 1>2.txt
echo "This is a test"
echo "This is a test">&2
```

脚本中，参数也可以由文件中读取获得，而不是键盘输入。只需要修改文本描述符 0 的数据来源，使用文件重定向，然后结合 read line 即可实现，从文件中逐行读取内容。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
This is a test., #1
This is not an apple., #2
This is a banana., #3
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash

exec 0< 1.txt
count=1
while read line
do
        echo "$line, #$count"
        count=$[$count+1]
done

(base) kuladmir@kuladmir:~/test$ cat 1.txt
This is a test.
This is not an apple.
This is a banana.
```

重定向可以自己定义，前文提及了 0、1、2 三个标准文本描述符。3~8 可以自己定义。

创建输出文件描述符：exec 3>*file_name*，然后在需要的地方加上>&3。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
This is a test.
(base) kuladmir@kuladmir:~/test$ cat 1.txt
This is a test2.
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash

exec 3>1.txt
echo "This is a test."
echo "This is a test2.">&3
```

恢复重定向文件：如果有 exec 3>&1　exec 1>*file_name*，这意思是：文本操作符 3 重定向到了 1，即屏幕；之后发送到屏幕的内容又被重定向到了 file_name 里。可以使用 exec 1>&3 实现恢复，这样就可以重新在屏幕上显示发送到屏幕的内容。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
This is over
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash


exec 3>&1
exec 1>1.txt
echo "We are the world"
echo "We are the children"
# 进入txt文件
exec 1>&3
echo "This is over"

(base) kuladmir@kuladmir:~/test$ cat 1.txt
We are the world
We are the children
```

创建输入文件描述符：exec 6<&0　exec 0<*file_name*　exec 0<&6。首先将 STDIN 的输入保存到 6 里，之后把输入源定向到文件，之后恢复 STDIN。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
We are the world
We are the children
This is alright
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash


exec 6<&0
exec 0<1.txt
while read line
do
        echo "$line"
done
exec 0<&6
echo "This is alright"
(base) kuladmir@kuladmir:~/test$ cat 1.txt
We are the world
We are the children
```

关闭文件描述符：exec 3>&-。关闭后，就不能再次使用。

lsof

可以列出 Linux 的所有打开的文件描述符。-p 参数可以允许指定进程 PID，-d 参数可以允许指定要显示的文本描述符。

```
(base) kuladmir@kuladmir:~/test$ lsof -p 1
COMMAND PID USER   FD      TYPE DEVICE SIZE/OFF NODE NAME
systemd   1 root  cwd   unknown                  /proc/1/cwd (readlink: Permission denied)
systemd   1 root  rtd   unknown                  /proc/1/root (readlink: Permission denied)
systemd   1 root  txt   unknown                  /proc/1/exe (readlink: Permission denied)
systemd   1 root NOFD                            /proc/1/fd (opendir: Permission denied)
```

4.4 数学计算

expr *number_1* {?} *number_2*

使用 expr 可以实现计算或者字符串操作。数组和符号之间要有空格，字符串有空格时，需要用引号包括。?代表一个操作符：

逻辑操作符：| 或。如果 number_1 不为 0 或者 null，则返回 number_1，否则返回 number_2。& 且。两个数都不为 0 或者 null，则返回 number_1，否则返回 0。

比较操作符：> >= < <= = !=(不等于)，以上操作如果成立，则返回 1，否则返回 0。

算数操作符：+ - *(转义为乘法) / %(取余数)，以上操作返回结果，但是只能做整数运算。

字符串操作符：length str 返回字符串长度，如果字符串里有空格，应该使用引号括起来。index str chars 在 str 里寻找 chars 的位置，若无返回 0。substr str pos length 在 str 里，从第 pos 个位置，返回一个长度为 length 的字符串。

最后得到的结果也可以重定向到文件内。

$[*number_1* {?} *number_2*]

使用如上方式也可以进行计算，特别是在 shell 脚本中。但是 shell 脚本中，调用变量的值，或者赋值时，需要$variant 的形式。

```
(base) kuladmir@kuladmir:~/test$ expr 5+5
5+5
(base) kuladmir@kuladmir:~/test$ expr 5 + 5
10
(base) kuladmir@kuladmir:~/test$ expr 5 \* 6
30
(base) kuladmir@kuladmir:~/test$ expr length kuladmir
8
(base) kuladmir@kuladmir:~/test$ expr length Kula pati
expr: syntax error: unexpected argument ‘pati’
(base) kuladmir@kuladmir:~/test$ expr length 'Kula pati'
9
(base) kuladmir@kuladmir:~/test$ expr index kulapatiadmirmoon pati
4
```

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
result is 11
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash
        number1=5
        number2=$[$number1 + 6]
        echo "result is $number2"
```

bc -q

用于呼出计算器，-q 阻止了一些欢迎文字的输出。bc 里可以执行计算，bc 也可以执行浮点数计算，并且计算参数没有限制，且不需要数字和符号之间有空格。同时 scale = n 可以控制小数位数为 n，默认是 0，quit 可以退出。

```
(base) kuladmir@kuladmir:~/test$ bc -q
5+6+7+8+9+1.1
36.1
5/7
0
scale=2
5/7
.71
6.315+5.323
11.638
print(5)
5
5+6
11
quit
```

variant=$(echo “scale=*n*; *expression* | bc”)

可以允许在 shell 脚本中，使用 bc 完成计算。expression 为计算表达式。后者使用重定向也可以完成计算。

variant=$(bc << EOF \n scale=*n* \n *expression* \n EOF)

\n 表示两句之间有换行（此处为了表示方便而用\n，shell 中换行即可），expression 中为表达式，可以有多句，包含但不限于赋值、计算等，每句后有换行。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
The var1 is 26
The var is 1.66
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash
        var1=$(echo "scale=2; 5+6+7+8" | bc)
        echo The var1 is $var1
        var=$(echo "scale=2; 30/6/3" | bc)
        echo The var is $var
```

在 shell 脚本中，如果要用一个变量对另一个进行赋值，需要按如下格式进行：

var_1=$[$var_2+*number*]

4.5 普通 if 结构命令

if command \n then \n command \n fi

if command; then \n command fi

if 的判断语句，如果 if 后的命令顺利执行，错误码为 0，则执行 then 后的命令（可以是多个）。如果发生错误，则不执行 then 后的命令，但也会输出错误原因。\n 如上，只是为了表明换行。

if command \n then \n command \n else command \n fi

if 的判断语句，执行同上，如果错误码不为 0，则执行 else 后的命令。

if command \n then command \n elif command \n then command \n fi

if 的判断语句，if 后的命令错误码为 0，则执行 then；否则判断 elif 后的命令错误码是否 0，是则执行 then，否则结束。elif 后也可以再添加一个 else 语句。构成：

if command \n then command \n elif command \n then command \n else command \n fi

这就结束了？显然不是。还可以构成一个大的 if 语句。

if command \n then command elif command then command elif command then command …… fi

【要点】if 后要有一个 then 作为判定成功后的结果，else 可以作为判定失败的结果，或者 elif 作为判定失败的结果，并且进行下一次判定。elif 本质是个 else if 的混合，具备了 else 和 if 的特征。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
1.sh  1.txt
The command is worked
./1.sh: line 6: lts: command not found
The command is failed
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash
if ls
then
        echo "The command is worked"
fi
if lts
then
        echo "The command is worked"
else
        echo "The command is failed"
fi
```

test (*content*)

用于对命令进行测试，可以用在 if 里，作为 if 后的命令。在 if 里，test 后面如果不加 content，则会产生非 0 的错误码，如果加内容，不管是命令与否，都不会产生非 0 错误码。

```
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash
if test ls
then
        echo "The command is true"
fi
if test k
then
        echo "The command is true"
else
        echo "The command is failed"
fi
(base) kuladmir@kuladmir:~/test$ ./1.sh
The command is true
The command is true
```

注意，如果 test 后是一个变量（有$），则会根据这个变量是否为空，进行判断，如果为空，则产生非 0 错误码，否则不会，空和值为 0 不一样，也就是说，即使这个变量的值为 0，test 也认为它不是空，不会产生非 0 的错误码。

以下判断中，如果满足判定条件，则执行 then 语句，否则执行 else 或其他。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
The command is true
The command is failed
The command is true
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash
if test ls
then
        echo "The command is true"
fi

if test $k
then
        echo "The command is true"
else
        echo "The command is failed"
fi
```

再者，if 后可以是一个数值判断，例如：

[*number_1* -eq *number_2*] 判断 number_1 是否等于 number_2。

[*number_1* -ge *number_2*] 判断 number_1 是否大于等于 number_2。

[*number_1* -gt *number_2*] 判断 number_1 是否大于 number_2。

[*number_1* -le *number_2*] 判断 number_1 是否小于等于 number_2。

[*number_1* -lt *number_2*] 判断 number_1 是否小于 number_2。

[*number_1* -ne *number_2*] 判断 number_1 是否不等于 number_2。

number_1 和 number_2 可以为常数，也可以为变量，但都要是整数。方括号和数字必须有空格，数字和参数之间也必须有空格，下同。

```
(base) kuladmir@kuladmir:~/test$ cat 2.sh
#!/bin/bash
if [ 10 -gt 5 ]
then
        echo "10>5"
fi
if [ 10 -eq 6 ]
then
        echo "10=6"
else
        echo "10!=6"
fi

(base) kuladmir@kuladmir:~/test$ ./2.sh
10>5
10!=6
```

再者，if 后也可以是一个字符串比较，例如：

[*str_1* = *str_2*]　判断 str_1 和 str_2 是否相同。

[*str_1* != *str_2*]　判断 str_1 和 str_2 是否不同。

[*str_1* \> *str_2*]　判断 str_1 是否大于 str_2。

[*str_1* \< *str_2*]　判断 str_1 是否小于 str_2。

[-n *str_1*]　判断 str_1 长度是否非 0。

[-z *str_2*]　判断 str_1 长度是否为 0。

\< 和 \> 是转义字符，如果只用 < 和 > 会出错，字符串比较以 ASCII 码为准。

```
(base) kuladmir@kuladmir:~/test$ ./2.sh
var!=var1
var!>var1
(base) kuladmir@kuladmir:~/test$ cat 2.sh
#!/bin/bash
var="Kula"
var1="dmir"
if [ var = var1 ]
then
        echo "var=var1"
else
        echo "var!=var1"
fi
if [ var \> var2 ]
then
        echo "var>var1"
else
        echo "var!>var1"
fi
```

最后，if 后可以是一个文件比较，例如：

[-d *file*]　检查 file 是否存在并是一个目录 。

[-e *file*]　检查 file 是否存在。

[-f *file*]　检查 file 是否存在并是一个文件。

[-r *file*]　检查 file 是否存在并可读。

[-n *file*]　检查 file 是否存在并非空。

[-w *file*]　检查 file 是否存在并可写。

[-x *file*]　检查 file 是否存在并可执行。

[-O *file*]　检查 file 是否存在并属当前用户所有。

[-G *file*]　检查 file 是否存在并且默认组与当前用户相同。

[*file_1* -nt *file_2*]　检查 file_1 是否比 file_2 新。

[*file_1* -ot *file_2*]　检查 file_1 是否比 file_2 旧。

```
(base) kuladmir@kuladmir:~/test$ ./2.sh
The file is exist
The file is not exist
(base) kuladmir@kuladmir:~/test$ cat 2.sh
#!/bin/bash
if [ -d /home/kuladmir/test ]
then
        echo "The file is exist"
else
        echo "The file is not exist"
fi
if [ -f /home/kuladmir/test/2.c ]
then
        echo "The file is exist"
else
        echo "The file is not exist"
fi

(base) kuladmir@kuladmir:~/test$ pwd
/home/kuladmir/test
(base) kuladmir@kuladmir:~/test$ ls
1.sh  1.txt  2.sh
```

上述的数值判断、字符判断、文件判断，都可以额外添加 && 和 || 实现多条件判断。&&为且，全真才真，一假则假；||为或，一真即真，全假即假。并列的条件数没有限制。

```
(base) kuladmir@kuladmir:~/test$ ./2.sh
The file is not exist
dmir = Kula or  Kula < dmir
(base) kuladmir@kuladmir:~/test$ cat 2.sh
#!/bin/bash
if [ -d /home/kuladmir/test ] && [ -f /home/kuladmir/test/2.c ]
then
        echo "The file is exist"
else
        echo "The file is not exist"
fi
var="Kula"
var1="dmir"
if [ var = var1 ] || [ var \< var1 ]
then
        echo "$var1 = $var or  $var < $var1"
else
        echo "test"
fi
```

4.6 高级 if 结构命令

if ((expression))

(())内可以写入更高级的表达式（其实就是 C 语句），[[]]内可以写入正则表达式。例如：

var++/--　先用后+1 或-1。

++/--var　先+1 或-1 后用。

var**　幂运算。

!　求反。

~　位求反。

<</>>　左移位或右移位。

同时，在使用这些高级运算符的基础上，也可以完成大小比较、赋值等。

case $variant in \n pattern_1 | pattern_2) command_1;; \n pattern_3) command_2;; \n *) default command_3;; \n esac

使用 case，能够避免 if...elif...elif...else 等的嵌套。根据 variant 的值，对应不同 pattern_n 的结果，如果符合，则执行对应语句，如果全都不匹配，则会执行*)，产生默认结果。

不一定必须只有一个 pattern_1 | pattern_2)，可以有多个或没有。如果变量的值为数字也可以使用 case。如果 pattern_n 部分有重合，则只会输出最早对应的结果。

```
(base) kuladmir@kuladmir:~/test$ ./2.sh
Welcome, Option2
(base) kuladmir@kuladmir:~/test$ cat 2.sh
#!/bin/bash
var="test"
case $var in
tps | bar )
        echo "Welcome"
        echo "Option1";;
test)
        echo "Welcome, Option2";;
*)
        echo "Welcome, default option";;
esac
```

4.7 循环结构命令

任何循环体内都要有迭代条件，要逐渐趋向于结束循环，不能无限循环。

for *variant* in *list* \n do \n command \n done

for variant in list; do

for ((变量初始化; 循环条件; 迭代条件))

可以实现循环，list 中为一系列值，即值域。variant 会获得 list 的值，每循环一次，值都会被更换，直到循环结束。同时，list 里会按照空格进行分割为单个的数据值，如果一个数据值为一个短语，则需要用引号”包括。对于内部的引号，也要使用转义字符\’。

循环中的 list，也可以是一个已经定义的字符串变量，只需要$list 解引用即可。字符串的延长：list=$list“char”，即可添加。

循环中的 list，也可以是一个目录，目的是便利该目录下的所有文件。为了保证运行，需要加入一个通配符*，保证对该目录下所有文件进行遍历，例如：

for *variant* in /home/usr/test/*

```
(base) kuladmir@kuladmir:~/test$ vim 1.sh
(base) kuladmir@kuladmir:~/test$ ./1.sh
This is a word: kula
This is a word: dmir
This is a test: Kula dmir
This is a test: id  ill
This is a test: i'd
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash
for variant in kula dmir
do
        echo "This is a word: $variant"
done
for var in "Kula dmir" i'd  i'll i\'d
do
        echo "This is a test: $var"
done
```

循环中的 list 的值来源，也可以是一个已有的文件，只需要完成如下几步：

第一步：定义变量。variant=“*file_name*”

第二步：调用文件。for *var* in $(cat $*variant*)

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
The word is this
The word is is
The word is this
The word is is
(base) kuladmir@kuladmir:~/test$ cat 1.txt 2.txt 1.sh
this
is
this is
#!/bin/bash

var="1.txt"
for car in $( cat $var)
do
        echo "The word is $car"
done
var1="2.txt"
for car in $( cat $var1)
do
        echo "The word is $car"
done
```

在获取 list 的数据和遍历时，程序会根据多个符号进行分割，包括但不限于空格和换行。如果想控制不输出含有空格的字符串，就要使用 IFS 进行限制，使 for 进行获取和便利 list 数据时，只会根据设定分割。

IFS=$‘*symbol*’

symbol 处，可以是各种分割标志，例如 :\n,;等，按照需求设定。当面对随时更改的需求，有两种方法：

IFS.OLD=$IFS　　IFS=$‘*new_symbol*’　　IFS=$IFS.OLD

备份旧的换行符，需要时进行替换。

IFS=$“*n of new symbols*”

设置多个换行标志。

while *content_1*　\n　do　\n　*content_2*　\n　done

while 风格的循环结构。content 内可以是一个命令，或者一个表达式。当 content_1 的条件满足或错误码为 0 时循环，content_1 处可以为多个语句。content_1 处的多条语句中，最后一条满足才会决定循环与否，下同。

until *content_1*　\n　do　\n　*content_2*　\n　done

until 风格的循环结构。content 内可以是一个命令，或者一个表达式。直到 content_1 的条件满足时结束循环，content_1 处可以为多个语句。

循环可以嵌套使用，也可以被控制。

break (n)　　continue (n)

break 可以结束本次循环，不在执行本循环。break n 可以结束 n 层循环。continue 可以不再运行之后的命令，从新一轮开始执行。continue n 可以跳过 n 层循环，之后开启新的第 m-n+1 层循环。

循环可以用于把循环结果重定向到文件中。done > *file_name*

循环可以快速遍历文件，并输出可以执行等文件；也可以快速创建多个账户。

4.8 处理输入

shell 文件内的参数，可以在执行这个文件的时候输入，这样就能够灵活的使用不同参数执行这个脚本，这只需要几步：

第一步：shell 脚本里，对于表达式里需要的参数，以此可以写为$1 $2 …… ${10} ……

第二步：CLI 执行文件时，手动输入对应位置的值，可以是数字，也可以是字符串，字符串建议用引号包括。

```
(base) kuladmir@kuladmir:~/test$ ./2.sh "Kula pati" "Kuladmir" 5
Hello, This is Kula pati
Nice to meet you, Kuladmir
The price of the cake is 5
(base) kuladmir@kuladmir:~/test$ cat 2.sh
#!/bin/bash
echo "Hello, This is $1"
echo "Nice to meet you, $2"
echo "The price of the cake is $3"
```

$0 是专门用来接收文件名的参数。如果直接向上文一样编写，实际返回结果是这样的：

```
(base) kuladmir@kuladmir:~/test$ ./2.sh
The file name is ./2.sh
(base) kuladmir@kuladmir:~/test$ cat 2.sh
#!/bin/bash

echo "The file name is $0"
```

name=$(basename $0)

这个命令可以很好解决如上的问题。

```
(base) kuladmir@kuladmir:~/test$ ./2.sh
the file name is 2.sh
(base) kuladmir@kuladmir:~/test$ cat 2.sh
#!/bin/bash

name=$(basename $0)
echo "the file name is $name"
```

一些时候，可以使用 if [-n “$1”]判断内容是否为非空。$#能够反馈 CLI 中输入的参数数量，之后可以结合 if [$# -ne *number*]继续参数数量的判断。在{}里，不能使用$#，但可以使用!#作为代替。并且在没有传参数时，$#和!#的值都是 0。

```
(base) kuladmir@kuladmir:~/test$ ./2.sh 5 6 7
There is(are) 3 parameter(s).
(base) kuladmir@kuladmir:~/test$ cat 2.sh
#!/bin/bash
echo "There is(are) $# parameter(s)."
```

$*和$@都可以一次性将 CLI 的参数获取，并暂时保存。但是$*把参数获取后，处理方式类似于形成一个长字符串，而$@则是形成一个字符数组。

```
(base) kuladmir@kuladmir:~/test$ ./2.sh kula dmir
The content is kula dmir
The content is kula
The content is dmir
(base) kuladmir@kuladmir:~/test$ cat 2.sh
#!/bin/bash
for para in "$*"
do
        echo "The content is $para"
done
for para in "$@"
do
        echo "The content is $para"
done
```

shift n，就是这个命令，不是键盘键。shift n 能够让参数左移 n 个单位，例在 CLI 里传入了三个参数，分别对应$1、$2、$3，shift 1 可以让$2 变为$1，$3 变为$2，且之前的值会丢失。不加 n 时，默认 n=1。

```
(base) kuladmir@kuladmir:~/test$ ./2.sh 5 6 7 8 9
There are some numbers: 5 6 7 8 9
The first is 5
The first is 7
(base) kuladmir@kuladmir:~/test$ cat 2.sh
#!/bin/bash
echo "There are some numbers: $@"
echo "The first is $1"
shift 2
echo "The first is $1"
```

将 shift 和 CLI 参数结合，就可以实现“做核酸”一样的运作。

```
(base) kuladmir@kuladmir:~/test$ ./2.sh -a -b -c -d
4
-a command is exist
-b command is exist
-c command is exist
-d command is not exist
(base) kuladmir@kuladmir:~/test$ cat 2.sh
#!/bin/bash
num=$#
echo $num
for((i=0;i<num;i++))
do
        case "$1" in
                -a)
                        echo "-a command is exist";;
                -b)
                        echo "-b command is exist";;
                -c)
                        echo "-c command is exist";;
                *)
                        echo "$1 command is not exist";;
        esac
        shift
done
```

shift 也可以实现选项和参数的分离，使用 -- 即可，但 case 语句里要添加这个选项。

```
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash

while [ -n "$1" ]
do
   case "$1" in
      -a) echo "Found the -a option" ;;
      -b) echo "Found the -b option";;
      -c) echo "Found the -c option" ;;
      --) shift
          break ;;
       *) echo "$1 is not an option";;
   esac
   shift
done

(base) kuladmir@kuladmir:~/test$ ./1.sh -a -b -c -- test1
Found the -a option
Found the -b option
Found the -c option
```

也可以实现选项后带参数的情况。

```
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash
while [ -n "$1" ]
do
   case "$1" in
      -a) echo "Found the -a option";;
      -b) param="$2"
          echo "Found the -b option, with parameter value $param"
          shift ;;
      -c) echo "Found the -c option";;
      --) shift
          break ;;
       *) echo "$1 is not an option";;
   esac
   shift
done
```

但是，如上只能实现单个选项执行，无法实现多选项执行。但一些内置命令可以同时带 2-3 个及以上的选项。

getopt *optstring* *parameters*

可以允许定义命令选项。optstring 里要写入选项字母和参数需求情况，例如 -a -b -c -d 要使用，且-b 后需要一个参数，则应写 getopt ab:cd parameters。parameters 里就可以写传入的参数了。如果传入的参数里，有未定义的 optstring 选项，则会发生错误，在 getopt 后加入-q 选项，可以屏蔽错误提示。

```
(base) kuladmir@kuladmir:~/test$ getopt abc:d -a -b -c test -d test1 test2
 -a -b -c test -d -- test1 test2
(base) kuladmir@kuladmir:~/test$ getopt a:cd -a -b -cd
 -a -b -c -d --
(base) kuladmir@kuladmir:~/test$ getopt ab:d -a -b ts -d -e
getopt: invalid option -- 'e'
 -a -b ts -d --
(base) kuladmir@kuladmir:~/test$ getopt -q ab:d -a -b ts -d -e
 -a -b 'ts' -d --
```

不仅可以在 CLI 中使用 getopt，shell 脚本也可以使用。

set -- $(getopt -q *optstring* “$@”)

可以将输入的内容全部按照 getopt 的要求进行格式化，实现选项和参数的分类，但是面对含有空格的参数时，会出现问题。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh -ac

Found the -a option
Found the -c option
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash

set -- $(getopt -q ab:cd "$@")
echo
while [ -n "$1" ]
do
   case "$1" in
   -a) echo "Found the -a option" ;;
   -b) param="$2"
       echo "Found the -b option, with parameter value $param"
       shift ;;
   -c) echo "Found the -c option" ;;
   --) shift
       break ;;
    *) echo "$1 is not an option";;
   esac
   shift
done
count=1
for param in "$@"
do
   echo "Parameter #$count: $param"
   count=$[ $count + 1 ]
done
```

getopts *optstring* *parameters*

可以允许定义命令选项。optstring 里要写入选项字母和参数需求情况，例如 -a -b -c -d 要使用，且-b 后需要一个参数，则应写 getopt ab:cd *parameters*。parameters 里就可以写传入的参数了。如果传入的参数里，有未定义的 optstring 选项，则会发生错误，在 optstring 部分最前端加一个 : ，可以屏蔽错误提示。由于 getopts 解析时，会消除-，因此 case 里无需加 - ；并且，选项和参数即使之间没有空格，也是可以实现分割。如果输入了未定义的选项，则会直接输出 ? 。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh -abcde
Found the -a option
Found the -b option, with value cde
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash
while getopts :ab:c opt
do
   case "$opt" in
      a) echo "Found the -a option" ;;
      b) echo "Found the -b option, with value $OPTARG";;
      c) echo "Found the -c option" ;;
      *) echo "Unknown option: $opt";;
   esac
done
(base) kuladmir@kuladmir:~/test$ ./1.sh -ab -e
Found the -a option
Found the -b option, with value -e
(base) kuladmir@kuladmir:~/test$ ./1.sh -ac -e
Found the -a option
Found the -c option
Unknown option: ?
```

read (-s) (-t *time*) (-nN) (-p “*print_info*”) (*variant*)

read 可以实现类似 C 里的 scanf 输入的功能。加入-p 选项，可以使用提示符；variant 可指定也可不指定，如果不指定，则输入的数据会存在 REPLY 中。variant 部分可以不仅只有一个参数，会按照默认方式进行数值分配，因此对于字符串应该加引号。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
Please input your name: kuladmir
Please input your weight: 65
Wow, your weight is 65
Please input your all name:Kulapati Admirmoon
Admirmoon Kulapati, welcome! Your age is kuladmir
Please input a chars "We are the world"
The chars is "We are the world"
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash

read -p "Please input your name: " age
read -p "Please input your weight: "
echo "Wow, your weight is $REPLY"
read -p "Please input your all name:" first second
echo "$second $first, welcome! Your age is $age"
read -p "Please input a chars" chars
echo "The chars is $chars"
```

想限制思考时间？-t 参数可以实现，read 命令中加入-t 参数，可以限制输入的时间为 time 秒。一般放在 if 后做判断条件，如果超时，则会返回非 0 错误码。

想限制输入量？-n 参数可以实现，read 命令中加入-n 参数，可以限制输入的文本量为 N 个，当输入的文本量达到 N 时，会自动完成输入，不需回车。

输入密码的时候怕被看到？-s 参数可以实现加密，或者说不是加密，只是把 read 后输入的内容字体颜色改成了底板颜色，看不到而已。

除此之外，read 不仅可以用键盘输入，也可以读取文件内容。

cat *file_name* | while read line

通过这个命令，read 能够逐行的读取文本，可选择用 echo 输出，或作他用。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
Line content: wow
Line content: i like it, do you want to sell it?
Line content: yes, of course
Line content: now, it is time
Line content: to pay for it.
Line content:
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash
cat 1.txt | while read line
do
        echo "Line content: $line"
done
```

4.9 控制脚本

终止信号（SIGINT），使用 Ctrl+C，可以终止一个进程；

暂停信号（SIGTSTP），使用 Ctrl+Z，可以暂停一个进程，显示为 Stopped。

无条件终止（SIGKILL），使用 kill PID 可以直接终止一个对应 PID 的进程。

挂起进程（SIGHUP），尽可能终止进程（SIGTREM），无条件暂停（SIGSTOP）。

trap *command* *signals*

trap 命令可以捕获一个信号，command 部分可以写入一个展示命令，signals 部分可以写入不同的信号。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
The time is 3
The time is 4
The time is 5
^Cwe find the SIGINT;
The time is 6
^Cwe find the SIGINT;
The time is 7
The time is 8
The time is 9
The time is 10
The time is 11
Finish
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash

trap "echo 'we find the SIGINT;'" SIGINT
count=2
while [ $count -le 10 ]
do
        sleep 2
        count=$[$count+1]
        echo "The time is $count"
done
echo "Finish"
```

如果 trap 的 signals 信号为 exit，则只要捕获到信号就会直接退出。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
The time is 3
The time is 4
^Cwe find the SIGINT;

(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash

trap "echo 'we find the SIGINT;'" exit
count=2
while [ $count -le 10 ]
do
        sleep 2
        count=$[$count+1]
        echo "The time is $count"
done
echo "Finish"
```

trap 的判定和输出可以根据需求在 shell 脚本的部分进行更改，trap 命令会对全文环境都生效，如果想使 trap 失效，应使用取消命令。

trap -- *signals*

取消命令里的 signals 内容应该与设定的 signals 内容一致。

可以后台实行 shell 脚本，只要在执行命令后加一个&。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh &
[1] 432
(base) kuladmir@kuladmir:~/test$ The time is 3
```

后台执行 shell 脚本时，如果有输出，可能会出现在新的一行，如果此时输入新的命令，新的输出可能会和后台运行脚本的输出混杂。

nohup *./file_name* &

可以允许程序一直运行，直到结束。即使中途关闭 CLI 也可以。如果不加&，则在当前目录产生一个 nohup.out 的可执行文件。该文件保存了 file_name 的运行结果。下次运行结果会继续写入，不会覆盖。

bg/fg (*job_number*)

可以恢复一个暂停的任务，使用 bg 重启后，会在后台进行，使用 fg 重启后，会在前台进行。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
^Z
[1]+  Stopped                 ./1.sh
(base) kuladmir@kuladmir:~/test$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
kuladmir     286     285  0 19:38 pts/0    00:00:00 -bash
kuladmir     415     286  0 19:40 pts/0    00:00:00 /bin/bash ./1.sh
kuladmir     416     415  0 19:40 pts/0    00:00:00 sleep 50
kuladmir     417     286  0 19:40 pts/0    00:00:00 ps -f
(base) kuladmir@kuladmir:~/test$ bg 1
[1]+ ./1.sh &
```

nice (-n) number 或 -number ./file.sh > file.out

可以调整任务优先级，nice 只能把优先级降低（-20～19，优先级依次降低）。-n 参数不是必须的，可以使用-n number 或-number 把任务优先级降到 number。之后可以使用 ps -p PID -o pid,ppid,ni,cmd 进行查看。

```
(base) kuladmir@kuladmir:~/test$ nice -10 ./1.sh > 1.out
^Z
[1]+  Stopped                 nice -10 ./1.sh > 1.out
(base) kuladmir@kuladmir:~/test$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
kuladmir     286     285  0 19:38 pts/0    00:00:00 -bash
kuladmir     463     286  0 19:53 pts/0    00:00:00 /bin/bash ./1.sh
kuladmir     464     463  0 19:53 pts/0    00:00:00 sleep 50
kuladmir     465     286  0 19:53 pts/0    00:00:00 ps -f
(base) kuladmir@kuladmir:~/test$ ps -p 464 -o pid,ppid,ni,cmd
    PID    PPID  NI CMD
    464     463  10 sleep 50
```

renice -n *number* -p PID

可以调整以运行命令的优先级，并且只能降低优先级。

at -f *file_name*　*time*

可以设置文件的执行时间，time 部分可以使用多种格式。

标准时间，24 小时制。例如：　20:05

AM/PM 时间。例如：8:05 PM

特定时间。例如：now　noon　midnight　teatime

标准日期。例如：MMDDYY　MM/DD/YY　DD.MM.YY

atq

可以显示排队进行的任务。

atrm *number*

可以删除排队进行的任务，number 是前面序号。

```
(base) kuladmir@kuladmir:~/test$ at -f 1.sh 8:10 PM
warning: commands will be executed using /bin/sh
job 5 at Sat Dec 28 20:10:00 2024
(base) kuladmir@kuladmir:~/test$ at -f 1.sh 8:11 PM
warning: commands will be executed using /bin/sh
job 6 at Sat Dec 28 20:11:00 2024
(base) kuladmir@kuladmir:~/test$ atq
5       Sat Dec 28 20:10:00 2024 a kuladmir
6       Sat Dec 28 20:11:00 2024 a kuladmir
(base) kuladmir@kuladmir:~/test$ atrm 5
(base) kuladmir@kuladmir:~/test$ atq
6       Sat Dec 28 20:11:00 2024 a kuladmir
```

min hour day month weekday command

使用 corn 时间表可以设置命令的执行情况。例如：周一 10:20 执行，则命令写成：20 10 * * 1 command 或者 20 10 * * mon command。weekday 处写星期几，0-6 分别为星期日到星期六；或者用 mon tue wed thu fri sat sun 表示。

cron -l

可以查找目前的 cron 时间表，一般都没有。cron -e 可以创建 cron 时间表。

ls /etc/cron.*

可以浏览 cron 目录，这个目录是预配置的。

5. 高级 shell 脚本

5.1 函数

function *name* { *command* }

name() { *command* }

可以创建一个函数，name 是不同函数唯一的区别。要在 shell 里使用函数，直接在新行调用这个函数（name）即可。但是函数需要定义后，才能调用使用。在 CLI 里也可以定义一个函数，但退出后就会消失。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
This is a function
Tht function is out
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash

name()
{
        echo "This is a function"
}
name
echo "Tht function is out"
```

标准变量$?可以显示函数调用执行后的反馈码。但这个反馈码是根据最后一句是否正确执行而确定的，如果没有成功执行，则反馈 1，成功执行则反馈 0。但是过程中有错，最后一句正确，也会反馈 0。

return number

可以返回一个状态码，使用$?可以显示出来。但用$?提取 return 值之前，有别的命令，则会丢失。且状态码范围必须是 0~255。

```
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash

name()
{
        echo "This is a function"
        return 5
}
name
echo "The final number is $?"

name1()
{
        echo "This is an another function"
        return 10
}
name1
echo "-------"
echo "The final number is $?"



(base) kuladmir@kuladmir:~/test$ ./1.sh
This is a function
The final number is 5
This is an another function
-------
The final number is 0
```

result=`*function_name*`

可以将函数的返回值，赋给 result。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
Enter a number:>  50
The number is 250
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash

name()
{
        read -p "Enter a number:>  "
        echo  $[$REPLY * 5]
}
result=`name`
echo "The number is $result"
```

同时，函数会被做为一种脚本进行执行，因此 shell 里可以对函数的变量进行赋值。就和之前介绍的相同，函数内需要赋值的变量可以写为$1、$2 等，$#可以显示总共传入的参数数量。脚本其他行写函数调用和赋值。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
val=15  val1=14
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash

name()
{
        if [ $# -eq 1 ]
        then
                echo $[$1+$1]
        else
                echo $[$1+$2]
        fi
}
val=$(name 5 10)
val1=$(name 7)
echo "val=$val  val1=$val1"
```

函数体被定义后，必须被调用才会执行。如果在 if 或循环结构中调用函数，如果函数有参数需求，应该在调用后加上参数。以上例函数为例：if 里必须要写成 value=$(*name* $1 $2)，这样才能接收 CLI 里传入的参数，写成 value=$(name) 会报错。

函数体内的变量，为 shell 脚本的局部变量；函数体外，shell 脚本内的变量，为全局变量。一般建议函数内外变量名不同，如果必要，函数内变量可以写为 local variant，保证这个变量为局部变量，其变化只局限于函数。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
var=50  tet=5  result=9  gar=28
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash

name()
{
        local var=$[$tet +5]
        result=$[$tet +4]
        gar=$[$gar *2]
}
var=50
tet=5
gar=14
name
echo "var=$var  tet=$tet  result=$result  gar=$gar"
```

如果传入函数的参数为数组，则不能使用上述方法，应该使用$@，这可以让函数接收每一个值，而不像$*变成一个字符串。

```
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash

name()
{
        new_arr=($@)
        echo "Function arr is ${new_arr[*]}"
}
arr=(1 2 3 4 5)
name ${arr[*]}
echo "The arr is ${arr[*]}"
```

函数的返回值（数组）也可以被新的变量接收。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
The arr is 1 2 3 4 5
The ARR is 3 4 5 6 7
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash

name()
{
        old_arr=($(echo "$@"))
        els=$[ $# - 1 ]
        for((i=0;i<=$els;i++))
        {

                new_arr[$i]=$[ ${old_arr[$i]} + 2 ]
        }
        echo ${new_arr[*]}
}
arr=(1 2 3 4 5)
echo "The arr is ${arr[*]}"
Ar=$(echo ${arr[*]})
ARR=($(name $Ar))
echo "The ARR is ${ARR[*]}"
```

函数递归，在函数体里调用函数，但要保证每次递归后，都会接近结束条件。

```
(base) kuladmir@kuladmir:~/test$ ./1.sh
Input a number:> 5
The result is 120
(base) kuladmir@kuladmir:~/test$ cat 1.sh
#!/bin/bash

name()
{
        if [ $1 -eq 1 ]
        then
                echo 1
        else
                local temp=$[ $1 -1 ]
                local res=$(name $temp)
                echo $[ $res * $1 ]
        fi
}
read -p "Input a number:> " value
result=$(name $value)
echo "The result is $result"
```

创建一个库，能够随时在其他 shell 脚本里调用其中的函数而不需要再重新定义。

source(.) *PATH*

可以允许 shell 访问对应位置（库），直接使用其中的函数。source 可以简写为一个.。

```
(base) kuladmir@kuladmir:~/test$ cat func
add()
{
        echo $[ $1 + $2]
}
minus()
{
        echo $[ $1 - $2]
}

(base) kuladmir@kuladmir:~/test$ cat 2.sh
#!/bin/bash

. ./func
val=4
val1=6
res=$(add $val $val1)
res1=$(minus $val1 $val)
echo "The res=$res   res1=$res1"

#
(base) kuladmir@kuladmir:~/test$ ./2.sh
The res=10   res1=2
```

5.2 正则表达式

正则表达式：用户定义的模式模板。即“筛子”，能够从一堆数据中筛出符合模式模板的数据，过滤不符合的数据。正则表达式是通过正则表达式引擎（Regular Expression Engine）实现，例如基础正则表达式（Basic Regular Expression）和扩展正则表达式（Extended Regular Expression）。

后两节会使用 sed 和 gawk 命令实现正则表达式的匹配。总之，正则表达式匹配的原则是：只要正则表达式和一些数据的一部分能匹配，就可以将这些得到反馈；如果没有匹配到，则无反馈。这意味着大小写也要被区分。

正则表达式能识别一些特殊字符：. * [] $ {} \ + ? | ()，这意味着在查找时，需要对一些特殊字符进行转义（\），才能正确实现。

```
(base) kuladmir@kuladmir:~/test$ sed -n '/This/p' 1.txt
This is a test.
This is not a test.
(base) kuladmir@kuladmir:~/test$ cat 1.txt
This is a test.
This is not a test.
this is a test.
Are you ok?

(base) kuladmir@kuladmir:~/test$ sed -n '/this/p' 1.txt
this is a test.
```

以下是基础正则表达式符号，常用 sed 命令。

如果未加限制时，只要在文本的任意处找到正则表达式，就会输出匹配的结果。^（脱字符）能够限制只匹配行首，并且^要放在正则表达式前。$能够限制只在行尾匹配，并且$要放在正则表达式后。限制字符可以组合使用。.能够代替任意一个字符。

```
(base) kuladmir@kuladmir:~/test$ cat 1.txt
This is a test.
This is not a test.
this is a test.
Are you ok?

(base) kuladmir@kuladmir:~/test$ sed -n '/.ou/p' 1.txt
Are you ok?
(base) kuladmir@kuladmir:~/test$ sed -n '/you/p' 1.txt
Are you ok?
(base) kuladmir@kuladmir:~/test$ sed -n '/^you/p' 1.txt
(base) kuladmir@kuladmir:~/test$ sed -n '/ok$/p' 1.txt
```

与.的效果相似，使用[]设定一个字符组，并在里面填入可供匹配的一些字符，就能够实现**一个字符位置**的多字符匹配。[^]可以设定一个字符组，并在里面填入排除匹配的一些字符，也能够实现一个位置的多字符匹配。由于[]里面可能会写入多个字符，为了简洁，可以使用区间的方式：[n-m]更为简洁，n、m 可以分别为数字或字母，并且是包括两端的。每个字符之间无需空格分开。*让匹配时，其之前的这个字符，出现的字数不固定（可以 0 次或多次）。

```
(base) kuladmir@kuladmir:~/test$ sed -n '/^[C-GI-Z]/p' 1.txt
This is a test.
We are the world.
Remember the day, i full in love with you.
It is not time to say goodbay.
(base) kuladmir@kuladmir:~/test$ sed -n '/^[G-M]/p' 1.txt
It is not time to say goodbay.
(base) kuladmir@kuladmir:~/test$ cat 1.txt
This is a test.
We are the world.
Remember the day, i full in love with you.
It is not time to say goodbay.
```

```
(base) kuladmir@kuladmir:~/test$ echo 'ieee' | sed -n '/ie*/p'
ieee
(base) kuladmir@kuladmir:~/test$ echo 'Ieee' | sed -n '/Ie*/p'
Ieee
```

Linux 里也有一些默认的字符组：

[[:alpha:]] 匹配任意字母字符，不管是大写还是小写

[[:alnum:]] 匹配任意字母数字字符 0~9

[[:blank:]] A~Z 或 a~z 匹配空格或制表符

[[:digit:]] 匹配 0~9 之间的数字

[[:lower:]] 匹配小写字母字符 a~z

[[:print:]] 匹配任意可打印字符

[[:punct:]] 匹配标点符号

[[:space:]] 匹配任意空白字符：空格、制表符、NL、FF、VT 和 CR

[[:upper:]] 匹配任意大写字母字符 A~Z

以下是扩展正则表达式符号，一般使用 gawk 命令。?和*很像，但是?限制前面的一个字符出现的次数要么为 0，要么为 1。

```
(base) kuladmir@kuladmir:~/test$ echo 'btt' | gawk '/b[aoe]?/{print $0}'
btt
(base) kuladmir@kuladmir:~/test$ echo 'baoeaoeaoet' | gawk '/b[aoe]?/{print $0}'
baoeaoeaoet
(base) kuladmir@kuladmir:~/test$ echo 'baoeaoeaoet' | gawk '/b[aoe]?t/{print $0}'
(base) kuladmir@kuladmir:~/test$ echo 'baoeaoeaoet' | sed -n '/b[aoe]*t/p'
baoeaoeaoet
```

+和?类似，作为互补，+限制前面的一个字符出现的次数要大于等于 1。{n}可以限制前面字符出现的次数为 n 次，如果为{n,m}，则限制出现次数至少为 n，至多为 m。

```
(base) kuladmir@kuladmir:~/test$ echo 'beaet' | gawk '/b[ae]+t/{print $0}'
beaet
(base) kuladmir@kuladmir:~/test$ echo 'bt' | gawk '/b[ae]+t/{print $0}'
(base) kuladmir@kuladmir:~/test$ echo 'bat' | gawk '/b[ae]+t/{print $0}'
bat
(base) kuladmir@kuladmir:~/test$ echo "beet" | gawk '/be{1}t/{print $0}'
(base) kuladmir@kuladmir:~/test$ echo "beet" | gawk '/be{2}t/{print $0}'
beet
```

|可以允许并列两个供匹配内容。两个供匹配内容和|之间不能有空格，否则空格也会被当成匹配的一部分。()可以将任意一些内容视为一组，一个整体字符。

```
(base) kuladmir@kuladmir:~/test$ echo "The dog" | gawk '/dog|cat/{print $0}'
The dog
(base) kuladmir@kuladmir:~/test$ echo "the cat" | gawk '/dog|cat/{print $0}'
the cat
(base) kuladmir@kuladmir:~/test$ echo "The flower" | gawk '/(a|f)l(t|o|M)/{print $0}'
The flower
```

5.3 初级和进阶 sed

sed *options script file*

流编辑器，可以根据提供的内容，对数据进行处理。options 部分提供了一些参数：

-e script　在处理输入时，将 script 中指定的命令添加到已有的命令中，或者进行多数据处理。

-f file_1　在处理输入时，将 file 中指定的命令添加到已有的命令中。

-n　不产生命令输出，使用 print 命令来完成输出。

script 部分，为数据处理的模板，有一些说明。

's/*old_word*/*new_word*/*flag*' 可以将文本里的词进行替换，这个替换不会修改源内容，只是一个读取后修改，然后把结果发送到 OUTPUT。-e 选项可以添加多个修改部分。-f 可以读取 file_1 里的修改部分，然后对 file 进行修改。如果 old_word 和 new_word 部分是一个路径，为了更加直观，可以使用!作为分割符号。

file 部分，如果要对文件进行操作，则后面跟文件名；如果使用管道对 echo 内容进行操作，则不需要文件名。

```
(base) kuladmir@kuladmir:~/test$ echo "we are not happy" | sed "s/not/none/"
we are none happy
(base) kuladmir@kuladmir:~/test$ echo "we are not happy" | sed -e "s/not/none/ ; s/we/You/"
You are none happy
(base) kuladmir@kuladmir:~/test$ sed -f 1.sed 2.txt
Mu is a man who likes to play.
kuku was happy because she won a prize.
(base) kuladmir@kuladmir:~/test$ cat 1.sed
s/Ku/Mu/
s/mumu/kuku/
(base) kuladmir@kuladmir:~/test$ cat 2.txt
Ku is a man who likes to play.
mumu was happy because she won a prize.
```

's/*old_word*/*new_word*/*flag*'其中 flag 为替换标记，有以下参数：

number　替换这个词第 number 次出现的位置。

g　替换所有。

p　原先行内容先打印出来，往往与-n 配合使用，输出修改的行。

w file　替换结果写入文件。

```
(base) kuladmir@kuladmir:~/test$ cat 1.txt
can you can a bar?
can you can a can?
can you play a canner
(base) kuladmir@kuladmir:~/test$ sed 's/can/CAN/2' 1.txt
can you CAN a bar?
can you CAN a can?
can you play a CANner
(base) kuladmir@kuladmir:~/test$ sed 's/can/CAN/' 1.txt
CAN you can a bar?
CAN you can a can?
CAN you play a canner
(base) kuladmir@kuladmir:~/test$ sed 's/can/CAN/g' 1.txt
CAN you CAN a bar?
CAN you CAN a CAN?
CAN you play a CANner
(base) kuladmir@kuladmir:~/test$ sed -n 's/bar/CAN/p' 1.txt
can you can a CAN?
```

从前文看到，默认情况下会对每行进行替换。使用行地址则可以控制哪些行修改，哪些行不改。'n,ms/*old_word*/*new_word*/*flag*'可以设定第 n 行和第 m 行修改，其他行不修改。'n,$s/*old_word*/*new_word*/*flag*'可以设定从第 n 行开始修改。

sed 语句里，如果要实现多个替换，则每个替换语句应该用{}包括。

```
(base) kuladmir@kuladmir:~/test$ cat 1.txt
can you can a bar?
can you can a can?
can you play a canner
(base) kuladmir@kuladmir:~/test$ sed '2s/can/CAN/2' 1.txt
can you can a bar?
can you CAN a can?
can you play a canner
(base) kuladmir@kuladmir:~/test$ sed '2,3s/can/CAN/' 1.txt
can you can a bar?
CAN you can a can?
CAN you play a canner
(base) kuladmir@kuladmir:~/test$  sed '2,$s/can/CAN/' 1.txt
can you can a bar?
CAN you can a can?
CAN you play a canner
```

sed 'nd' *file_name*

可以删除第 n 行的内容，没有 n 时则全删。这个删除不会影响文本实际内容。

sed ‘na*word*’

可以在第 n 行之后添加一行内容，如果没有 n，则默认为所有行。如果是$，则表示最后一行。

sed ‘ni*word*’

可以在第 n 行之前插入一行内容，如果没有 n，则默认为所有行。如果是$，则表示最后一行。

sed ‘nc*word*’

可以修改第 n 行为新内容，如果没有 n，则默认为第一行。如果是$，则表示最后一行。如果是 n,mc，实际表现为修改第 n 行，删除第 m 行；本质是，用了新内容替换掉了两行。

```
(base) kuladmir@kuladmir:~/test$ sed '2i\This is a test.' 1.txt
can you can a bar?
This is a test.
can you can a can?
can you play a canner
(base) kuladmir@kuladmir:~/test$ sed 'a\This is a line' 1.txt
can you can a bar?
This is a line
can you can a can?
This is a line
can you play a canner
This is a line
(base) kuladmir@kuladmir:~/test$ sed '2c\This is a change.' 1.txt
can you can a bar?
This is a change.
can you play a canner
(base) kuladmir@kuladmir:~/test$ sed '2d' 1.txt
can you can a bar?
can you play a canner
```

sed ‘y/*chars*/*new_chars*’

可以转化 chars 为 new_chars。本质上是 new_chars 和 chars 进行一对一替换。

/content/可以用在操作前，进行特征搜索；‘=’可以输出行号。‘l’可以输出文本中存在的不可打印符号。

```
(base) kuladmir@kuladmir:~/test$  sed '=' 1.txt
1
can you can a bar?
2
can you can a can?
3
can you play a canner
(base) kuladmir@kuladmir:~/test$ sed -n '/canner/{
> =
> p
> }' 1.txt
3
can you play a canner
(base) kuladmir@kuladmir:~/test$ sed 'l' 1.txt
can you can a bar?$
can you can a bar?
can you can a can\357\274\237$
can you can a can?
can you play a canner$
can you play a canner
```

sed ‘n,mw *file_name*’ *file*

可以向文件内写入行。将 file 里的第 n,m 行写入 file_name 里。写入会在 file_name 里写入指定文本，会改变文件。

sed ‘n,mr *file_name*’ *file*

可以从文件读入行，之后写入。将 file_name 里的第 n,m 行读出，写入 file 里，如果要在文本后写入而不覆盖，则使用$r，这个写入不会影响 file 文件内容。

```
(base) kuladmir@kuladmir:~/test$ sed '2,3w 2.txt' 1.txt
can you can a bar?
can you can a can?
can you play a canner
(base) kuladmir@kuladmir:~/test$ cat 2.txt
can you can a can?
can you play a canner
(base) kuladmir@kuladmir:~/test$ vim 2.txt
(base) kuladmir@kuladmir:~/test$ cat 2.txt
can you can a can?
can you play a canner
line1:This is a line.
line2:This is a pie.
(base) kuladmir@kuladmir:~/test$ sed '3,4r 2.txt' 1.txt
can you can a bar?
can you can a can?
can you play a canner
can you can a can?
can you play a canner
line1:This is a line.
line2:This is a pie.
```

----------进阶+----------

sed 提供了新的参数：

N　将数据流中的下一行加进来创建一个多行组来处理。

D　删除多行组中的一行。

P　打印多行组中的一行。

n　使 sed 编辑器移动到数据流中的下一文本行。

使用 n 命令时，可以和其他命令一起使用，例如/content/，如果有其他字母参数，则应该使用{ , }方式包括。在使用‘s/*old_word*/*new_word*/’进行替换时，如果 old_word 的两个词不在一行，可以在两个词之间加一个.。

```
(base) kuladmir@kuladmir:~/test$ cat 2.txt
We are the test.
Yes, we are the command test.
But is it true?
Ich wil mit dir sein
(base) kuladmir@kuladmir:~/test$ sed '/command/{ N ; s/\n/ / }' 2.txt
We are the test.
Yes, we are the command test. But is it true?
Ich wil mit dir sein
(base) kuladmir@kuladmir:~/test$ sed '/not/{ n ; d }' 1.txt
We are not the only.
We are the only.

We will never give up.
(base) kuladmir@kuladmir:~/test$ cat 1.txt
We are not the only.

We are the only.

We will never give up.
```

```
(base) kuladmir@kuladmir:~/test$  sed 'N ; s/give.up/let go/' 1.txt
We are not the only.

We are the only.

We will never let go.
(base) kuladmir@kuladmir:~/test$ cat 1.txt
We are not the only.

We are the only.

We will never give
up.
```

sed 编辑器有两块区域：模式空间和保持空间。操作保持空间的命令共有五种：

h/H　将模式空间　复制/附加　到保持空间

g/G　将保持空间　复制/附加　到模式空间

x　交换模式空间和保持空间的内容

排除命令（!）可以让原命令失效，而发挥与之互补的作用。

sed '{n,mb [label]; *command*}' *file_name*

分支命令（b）可以跳过一些行，在特定行不执行一些命令。如果不加 label，则默认跳到结尾，如果加了 label 部分，则可以跳到 label 命令行的位置。

```
(base) kuladmir@kuladmir:~/test$ vim 1.txt
(base) kuladmir@kuladmir:~/test$ sed -n '{1!G ; h ; $p }' 1.txt
This is the last line.
This is the second data line.
This is the first data line.
This is the header line.
(base) kuladmir@kuladmir:~/test$ cat 1.txt
This is the header line.
This is the first data line.
This is the second data line.
This is the last line.
(base) kuladmir@kuladmir:~/test$  sed '{2,3b ; s/This is/Is this/ ; s/line./test?/}' 1.txt
Is this the header test?
This is the first data line.
This is the second data line.
Is this the last test?
```

设置分支：冒号加一个名字（:jump），可以设定一个分支节点。在 b 后面添加这个 label 后，执行完 b 命令则会跳到这个分支命令行处，执行后面的命令。

```
(base) kuladmir@kuladmir:~/test$ echo "We, have, no, time:" | sed -n '{
> :start
> s/,//1p
> /,/b start
> }'
We have, no, time:
We have no, time:
We have no time:
```

sed '{t [label]; *command*}' *file_name*

测试命令（t）可以进行测试，成功时跳到脚本结尾。理解：如果 t 上面的一个命令使得一些行内容被修改，则测试成功，t 后面的命令将不会再修改已经改动的内容。

```
(base) kuladmir@kuladmir:~/test$ sed '{
> s/first/matched/
> t
> s/This is/NULL/
> }' 1.txt
NULL the header line.
This is the matched data line.
NULL the second data line.
NULL the last line.
```

万能替代符号（&），在替换时，new_word 部分可以使用&，代表命令里的匹配模式。替代单个单词（\(\)，必须要转义），将这个单词用()包括，后续使用\1、\2 等进行分配即可。

```
(base) kuladmir@kuladmir:~/test$ echo "This is a test." | sed 's/test/"&"/'
This is a "test".
(base) kuladmir@kuladmir:~/test$ echo "This is a test." | sed 's/test/""/'
This is a "".
(base) kuladmir@kuladmir:~/test$ echo "The apple is sweet" | sed 's/apple \(is\)/peer \1/'
The peer is sweet
```

一些应用：

1）增加行间距。

```
(base) kuladmir@kuladmir:~/test$ sed '$!G' 1.txt
This is the header line.

This is the first data line.

This is the second data line.

This is the last line.
```

2）有空行的段落增加行间距。

```
(base) kuladmir@kuladmir:~/test$ sed '/^$/d ; $!G' 2.txt
We are the test.

Yes, we are the command test.

But is it true?

Ich wil mit dir sein
```

3）列出行数，虽然使用=可以实现，不过会存在换行情况。

```
(base) kuladmir@kuladmir:~/test$ sed '=' 1.txt | sed 'N ; s/\n/ /'
1 This is the header line.
2 This is the first data line.
3 This is the second data line.
4 This is the last line.
```

4）删除连续空白行。sed '/./,/^$/!d' *file_name*

5）删除开头空白行。sed '/./,$!d' *file_name*

6）删除结尾空白行。sed '{ :start /^\n*$/{$d; N; b start } }'

7）删除 HTML 标签。sed 's/]*>//g' *file_name*

5.4 初级和进阶 gawk

gawk *options program file*

流编译器，可以在内部填入语句和结构化语句。options 处可以填入选项：

-F fs　指定行中划分数据字段的字段分隔符。

-f file　从指定的文件中读取程序。

-v var=value　定义 gawk 程序中的一个变量及其默认值。

-mf N　指定要处理的数据文件中的最大字段数。

-mr N　指定数据文件中的最大数据行数。

-W keyboard　指定 gawk 的兼容模式或警告等级。

program 部分，需要为一条语句，并用{}包括，不加 file 时，由于需要输入，则需要使用键盘进行 STDIN 输入，根据设置则会输出{}里的内容；Ctrl-D 可以传入一个 EOF，结束 gawk 命令。添加 file 时，可以使用$n 在文件内进行数据匹配。$0 表示整行，$1 表示行中第一个数据，$2 表示行中第二个数据，$n 表示行中第三个数据。由于在分割时，会根据空格和制表符进行分割，如果要指定其他分隔符文件，应使用-F program file。

语句中可以使用$n 对文本内容进行读取和修改，之后可以重新输出。或者预先写一个 gawk 文件，使用-F : -f file.gawk 进行处理。

```
(base) kuladmir@kuladmir:~/test$ gawk '{print $2}' 1.txt
is
this
this
(base) kuladmir@kuladmir:~/test$ gawk '{print $0}' 1.txt
This is only a test.
Yes, this is a real test.
No, this is a test.
(base) kuladmir@kuladmir:~/test$  gawk '{print "Hello World"}'
ls
Hello World
time
Hello World
```

在 program 部分，如果添加了 BEGIN 关键词，则会强制 gawk 读取前执行 BEGIN 后的命令，如果后面的语句没有任何读取，则需要 STDIN 输入一些内容，才会显示后面的部分。如果后面的语句有读取，仍然可以使用$n。END 和 BEGIN 相似，在 gawk 读取完数据后执行 END 后的内容。

```
(base) kuladmir@kuladmir:~/test$ gawk 'BEGIN{print "Hello World"} {print "So it is " $1}' 1.txt
Hello World
So it is This
So it is Yes,
So it is No,
(base) kuladmir@kuladmir:~/test$ cat 1.txt
This is only a test.
Yes, this is a real test.
No, this is a test.
(base) kuladmir@kuladmir:~/test$ gawk 'BEGIN{print "Hello World"} {print "So it is over?"}'
Hello World
ls
So it is over?
clear
So it is over?
```

----------进阶+----------

gawk 的内置变量：

FS　输入字段分隔符。

RS　输入记录分隔符。

OFS　输出字段分隔符。

ORS　输出记录分隔符。

FIELDWIDTHS　由空格分隔的一列数字，定义了每个数据字段确切宽度。

通过设置 OFS，可以使得输出的内容之间存在 OFS 定义的分隔符。FIELDWIDTHS 和 FS 不会同时生效，使用 FIELDWIDTHS 时，根据变量设置长度，可以限制变量输出的长度。

要使用这些参数，需要在 BEGIN{}中添加，不一定必须要叫 BEGIN，随意都可以。

```
(base) kuladmir@kuladmir:~/test$ gawk 'BEGIN{OFS="-"}  {print $4,$5}' 1.txt
header-line.
first-data
second-data
last-line.
(base) kuladmir@kuladmir:~/test$ gawk 'BEGIN{OFS="--~--"} {print $4,$5}' 1.txt
header--~--line.
first--~--data
second--~--data
last--~--line.
```

gawk 内部自定义变量和赋值。定义和赋值时，需要在一个{}里完成。除此之外，CLI 里也可以实现变量赋值。

gawk 里数组管理。数组定义：var[index]=value。数组遍历：for (var in array)。删除数组变量：delete array[index]。

gawk 里结构化变量。

判断结构：

if(condition) { progress } else { progress }

循环结构：

while(condition) { progress }

do { progress }while(condition)

for(initialization ; condition ; refinement)

输出语法：

printf “statement %_”, variant

%_是一个格式化指定符，_处为控制字母。%c 对应一个字符，%d 或%i 对应一个 int 值，%e 对应科学计数法的值，%f 对应一个 float 值，%lf 对应一个 double 值，%o 对应八进制，%x 对应十六进制。

gawk 内建函数。

数学函数：int()　取整；sqrt()　求平方根；log()　求自然对数；exp()　求指数函数；rand()　产生随机数（0-1）。

按位函数：and (v1,v2)　按位与；or (v1,v2)　按位或；comp1 (val)　补运算；lshift (val,count)　将值左移；rshift (val,count)　将值右移；xor(v1,v2)　按位异或。

字符函数：asort(s[,d]) 将数组 s 排序，如果有 d，则排序后的数组会保存到 s。index(s,t) 在 s 里查找 t 出现的位置。length([s]) 返回字符串长度。tolower()和toupper() 将字符转换为小写/大写。

时间函数：mktime() 即日期[YYYY MM DD HH MM SS]转化为时间戳。systime()返回当前时间戳。

自定义函数：function name(){}

6. ssh

ssh 是一个网络协议，用于在网络上的计算机之间进行加密通信，以确保数据传输的安全性。它主要用于远程登录到服务器、执行远程命令、文件传输等。

OpenSSH 是一个软件，用于构建 ssh 协议，并实现 ssh 的功能。其包含了多个组件：

sshd：SSH 服务器守护进程，运行在服务器上，负责处理客户端的连接请求和认证；

ssh：SSH 客户端程序，用于从本地计算机连接到远程服务器；

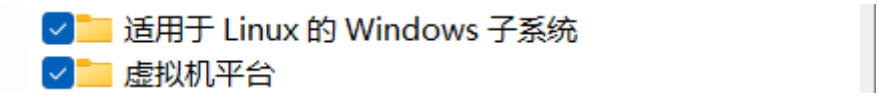
scp：用于在本地和远程计算机之间安全地复制文件；

sftp：SSH 文件传输协议客户端，提供类似于 FTP 的文件传输功能，但更加安全。

安装和使用 Linux 和 ssh：

1. 开启 WSL(windows subsystem for linux)前的配置

要求：CPU 虚拟化开启（一般都会开，任务管理器里可以查看）；在 windows 功能里，开启两个功能选项，之后完成重启；



2. 安装 WSL

第一步：下载 WSL。

cmd 中输入：wsl --install --web-download 默认下载的是 Ubuntu。

【可选】wsl --list --online 可以列出所有可供安装的 Linux 发行版。

【可选】wsl --list -v 可以查看 windows 上已经安装了的 Linux 系统。

【可选】wsl --set-default *new_linuxname* 可以修改默认的 Linux 系统。

【可选】wsl --unregister *linuxname* 可以卸载一个 Linux 系统。

【可选】wsl --export *linuxname filename* 可以备份一个 Linux 系统，生成一个 filename 的备份文件。

第二步：下载 Ubuntu。

第三步：部署 ssh。

在 shell 里输入：sudo apt update；

安装 ssh 服务器：sudo apt install openssh-server；

启动 ssh 服务：sudo service ssh start；

第三步：生成 ssh 密钥。

在 shell 里输入：ssh-keygen，会生成两个密钥：公钥和私钥。

【可选】在 shell 中，输入 cd ~/.ssh 可以访问该目录，查看公钥和私钥。

7. Slurm 作业调度

Slurm(Simple Linux Utility for Resource Management)，是一种可扩展的工作负载管理器。

用户登入节点或平台后，可以向服务器提交任务，Slurm 系统会根据任务需求，分配给不同的计算节点，完成计算。

sbatch　test.sh/test.slurm

该命令用以向服务器提交任务。在任务脚本里，可以添加一些参数：

#SBATCH --job-name=*Test_name*	设定任务名
#SBATCH --output= *File_name*	设定标准输出文件
#SBATCH --error= *File_name*	设定标准错误文件
#SBATCH --qos=*Type*	设定 QOS 状态
#SBATCH --n=*number*	设定总进程数
#SBATCH --time=*dd:hh:mm:ss*	设定作业最大运行时间
#SBATCH --exclusive	独占节点
#SBATCH --p	作业队列
#SBATCH --nodes=*number*	设定节点需求数
#SBATCH –gres=gpu:*n*	设定 GPU 需求数

```
#!/bin/bash
#SBATCH --job-name=Gro_test
#SBATCH --nodes=1
```

```
[jqzang02@k216 gromacs]$ sbatch run_md_gpu.sh
Submitted batch job 20792821
```

squeue [-j job_id]/[-l]

该命令可以查询作业信息。作业状态：R(正在运行)，PD(正在排队)，CG(即将完成)，CD(已完成)。只用 squeue 命令时，会输出所有队列中的任务，添加-j job_id 时，可以确定查询某个 id 的作业。添加-l 可以显示该任务的全部细节。

```
[jqzang02@k216 gromacs]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
          20792821    ampere Gro_test jqzang02 PD       0:00      1 (Priority)
```

```
[jqzang02@k216 gromacs]$ squeue -l
Thu Feb 20 15:06:12 2025
             JOBID PARTITION     NAME     USER    STATE       TIME TIME_LIMI  NODES NODELIST(REASON)
          20792821    ampere Gro_test jqzang02  PENDING       0:00 UNLIMITED      1 (Priority)
```

scancel *job_id*

该命令可以结束任务。

scontrol show job *job_id*

该命令可以查看对应任务的具体参数。

```
[jqzang02@k216 gromacs]$ scontrol show job 20792779
JobId=20792779 JobName=OYE_2
   UserId=jqzang02(7099) GroupId=student(5000) MCS_label=N/A
   Priority=500000000 Nice=0 Account=root QOS=normal
   JobState=PENDING Reason=Job's_QOS_not_permitted_to_use_this_partition_(ampere_allows_gpu,debug_not_normal) Dependency=(null)
   Requeue=1 Restarts=0 BatchFlag=1 Reboot=0 ExitCode=0:0
   RunTime=00:00:00 TimeLimit=01:12:00 TimeMin=N/A
   SubmitTime=2025-02-20T14:05:02 EligibleTime=2025-02-20T14:05:02
   AccrueTime=2025-02-20T14:05:02
   StartTime=Unknown EndTime=Unknown Deadline=N/A
   SuspendTime=None SecsPreSuspend=0 LastSchedEval=2025-02-20T14:20:31 Scheduler=Main
   Partition=ampere AllocNode:Sid=x254:289101
   ReqNodeList=(null) ExcNodeList=(null)
```

scontrol hold job_id

该命令可以暂停任务。

scontrol release job_id

该命令可以恢复任务。

sacct

该命令可以查看作业记录（默认是过去 24 小时）。