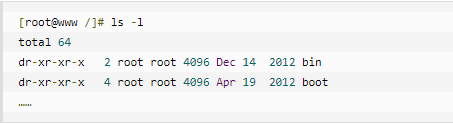
linux学习文档: https://linuxtools-rst.readthedocs.io/zh\_CN/latest/base/index.html

### linux 文件基本属性

使用ll命令来显示一个文件的属性以及文件所属的用户和组



第一个属性：

* 当为[ d ]则是目录
* 当为[ - ]则是文件；
* 若是[ l ]则表示为链接文档(link file)；
* 若是[ b ]则表示为装置文件里面的可供储存的接口设备(可随机存取装置)；
* 若是[ c ]则表示为装置文件里面的串行端口设备，例如键盘、鼠标(一次性读取装置)。



r=4，w=2，x=1

如：

rwx = 4 + 2 + 1 = 7

rw = 4 + 2 = 6

rx = 4 +1 = 5

rwx的顺序不会变，若没有对应的权限，就用-表示；

更改文件属性：

* chgrp：更改文件属组

|  |
| --- |
| chgrp [-R] 数组名 文件名 |

-R：递归更改文件属组，就是在更改某个目录文件的属组时，如果加上-R的参数，那么该目录下的所有文件的属组都会更改。

* chown：更改文件属主，也可以同时更改文件属组

|  |
| --- |
| chown [-R] 属主名 文件名  chown [-R] 属主名：属组名 文件名  chown [-R] :属组名 文件名  chown -R work test // -R为递归 work为登录人 test为文件名 |

* chmod：更改文件9个属性

第一种：

r:4 w:2 x:1

使用数字代表权限：

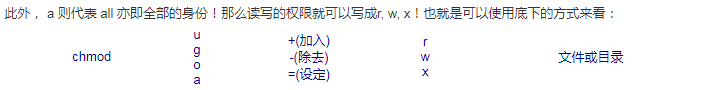
9个权限3个一组，分别是owner（user）/group/others,用每组权限的数字之和表示

赋予全部权限：

|  |
| --- |
| chmod [-R] 777 文件或目录 |

第二种：符号类型

三种身份user/group/others，分别用 u g o来表示



a代表三种身份同时操作

|  |
| --- |
| chmod a-x install.log |

|  |
| --- |
| chmod u+x,g-w,o-w install.log |

|  |
| --- |
| chmod u=r,g=rwx,o=--- install.log |

### 文件和目录管理

#### 绝对路径和相对路径

绝对路径：从根目录 / 写起 例如：/usr/share/doc

相对路径： ../..

#### ls

参数：

-a ：全部的文件，连同隐藏档( 开头为 . 的文件) 一起列出来(常用)

-d ：仅列出目录本身，而不是列出目录内的文件数据(常用)

-l ：长数据串列出，包含文件的属性与权限等等数据；(常用)

|  |
| --- |
| ls –al 目录名称 |

目录名称可以不写，默认显示当前目录下的

ll -t ：按修改时间排序降序（最新的在最前面）

ll -rt：按修改时间升序

#### cd

后可跟相对路径和绝对路径

/ 代表根目录

~代表 /root目录

#### pwd

显示当前目录（print working directory）

#### mkdir

|  |
| --- |
| mkdir [-mp] 目录名称 |

-m ：配置文件的权限喔！直接配置，不需要看默认权限 (umask) 的脸色～

-p ：帮助你直接将所需要的目录(包含上一级目录)递归创建起来！

|  |
| --- |
| mkdir -m 777 –p 1/2/3 |

#### rmdir

删除空目录

|  |
| --- |
| rmdir [-p] 目录名称 |

-p ：连同上一级『空的』目录也一起删除

|  |
| --- |
| rmdir 1/2/3 只删除3  rmdir -p 1/2/3 1、2、3都会被删除 |

#### rm

移除文件和目录

|  |
| --- |
| rm [-fir] 文件或目录 |

-f ：就是 force 的意思，忽略不存在的文件，不会出现警告信息；

-i ：互动模式，在删除前会询问使用者是否动作

-r ：递归删除啊！最常用在目录的删除了！这是非常危险的选项！！！

注意:只有 -r 的时候rm才能删除目录，其他情况不能。

#### cp

复制文件和目录，可以更改文件或目录名

|  |
| --- |
| cp [-adfilprsu] 来源档(source) 目标档(destination) |

-a：相当於 -pdr 的意思，至於 pdr 请参考下列说明；(常用)

-d：若来源档为连结档的属性(link file)，则复制连结档属性而非文件本身；

-f：为强制(force)的意思，若目标文件已经存在且无法开启，则移除后再尝试一次；

-i：若目标档(destination)已经存在时，在覆盖时会先询问动作的进行(常用)

-l：进行硬式连结(hard link)的连结档创建，而非复制文件本身；

-p：连同文件的属性一起复制过去，而非使用默认属性(备份常用)；

-r：递归持续复制，用於目录的复制行为；(常用)

-s：复制成为符号连结档 (symbolic link)，亦即『捷径』文件；

-u：若 destination 比 source 旧才升级 destination ！

#### mv

移动文件或目录，或修改名称

|  |
| --- |
| mv [-fiu] source destination |

-f ：force 强制的意思，如果目标文件已经存在，不会询问而直接覆盖；

-i ：若目标文件 (destination) 已经存在时，就会询问是否覆盖！

-u ：若目标文件已经存在，且 source 比较新，才会升级 (update)

**移动所有文件（含隐藏文件）：**

mv \* .[^.]\* ..

最后个参数指移动到的目标目录

第二个参数 \* : 除了隐藏文件的所有文件

第三个参数 .[^.]\* : 所有隐藏文件，第一个.表示匹配文件名第一位是.即以.开头，第二个[^.]表示文件名第二位不是. 第三个\* 表示从三位后没有限制；

注意：.\* 会匹配 目录 非隐藏文件 . .. 并不会匹配到以.开头的隐藏文件

扩展：移动除某个文件夹外的所有文件（含隐藏文件）

mv !(排除文件) .[^.]\* ..

#### touch

修改文件或者目录的时间属性，包括存取时间和更改时间。若文件不存在，**新建**一个。

touch 文件/目录

#### vi

vi 文件 : 新建文件

#### link

ln -s [源文件] [目标文件]

软链接：以路径的形式存在，类似于快捷方式，不占磁盘空间

ln [源文件] [目标文件]

硬链接：已文件副本形式存在，两文件各项属性相同

软链接和硬链接，无论改动哪一个文件，链接文件都会跟着改动！！

#### find

find 搜索路径 -iname 文件名 //参数i : 忽略大小写

find . -iname 文件名 //. 跟 ./ 效果一样，表示搜索范围在当前文件下

find 搜索路径 -iname 文件夹名 -type d // 搜索文件夹名称

find 搜索路径 -iname 文件名 2>/dev/null // 将错误信息转移，界面干净

find 搜索路径 -name 文件名 **-print0**

//默认情况下，find命令每输出一个文件名，后面带一个‘\n’换行符，因此输出都是一行一行的；

-print0 则让 find命令在打印出一个文件名之后接着输出一个 NULL 字符 ('') 而不是换行符；

搭配xargs -0，即将**空字符串''**当做多条记录的分隔符。这样就解决了文件名中有空格的问题。

\0表示空字符串**''，界面显示空白；**

Link: <https://www.cnblogs.com/xiaofeng666/p/10746243.html>

#### xargs

见印象笔记

#### tar

tar -czvf xxx.tar xxx // 压缩xxx为xxx.tar

tar -xzvf xxx.tar // 解压xxx.tar

tar -xzvf xxx.tar -C 目标路径 // 解压到指定位置

将多个文件压缩成一个压缩文件：

tar -czvf xxx.tar file1 file2 file3

将文件夹所有内容压缩：

tar -czvf xxx.tar folder**/\*** // **注意要加通配符 \***

-c(--create): 建立新的备份文件

-z(--gzip): 通过gzip指令处理备份文件

-v(--verbose): 显示指令执行过程

-f(--file): 指定备份文件，-f后要紧接备份文件

-x(--extract): 从备份文件中还原文件

--exclude=xxx 排除文件或目录

#### scp

文件上传：

scp -P 端口号 sourceFile root@10.2.3.56:targetFile

注意：-P, P要大写

文件下载：

scp -P 端口号root@10.2.3.56:sourceFile downloadFolder

#### zip

zip -r xxx.zip xxx // 将文件夹xxx递归压缩成xxx.zip

zip -r xxx.zip xxx -x 排除的文件 // 排除某文件压缩

#### unzip

unzip xxx.zip -d yyy // 解压到yyy文件夹

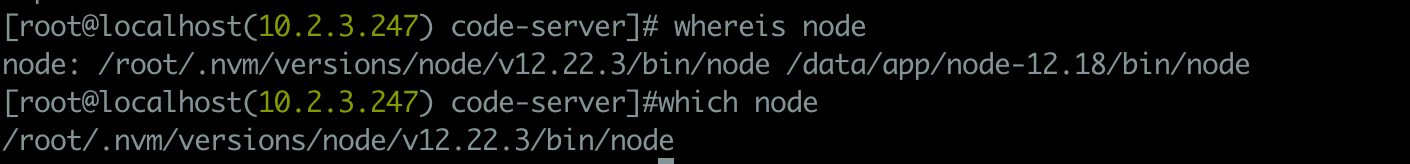
#### which

查找可执行文件的位置，原理是在当前登录用户的PATH环境变量记录的路径中进行查找

#### whereis

快速查找任何文件，是个文件搜索命令

会把所有包含关键字的路径都列出来



### 文件内容查看

#### cat

从第一行开始显示文件内容

|  |
| --- |
| cat [-AbEnTv] + 文件 |

-A ：相当於 -vET 的整合选项，可列出一些特殊字符而不是空白而已；

-b ：列出行号，仅针对非空白行做行号显示，空白行不标行号！

-E ：将结尾的断行字节 $ 显示出来；

-n ：列印出行号，连同空白行也会有行号，与 -b 的选项不同；

-T ：将 [tab] 按键以 ^I 显示出来；

-v ：列出一些看不出来的特殊字符

#### tac

从最后一行开始显示，即倒着显示，跟cat相反

#### nl

显示行号,输出文件内容

|  |
| --- |
| nl 文件 |

#### more

一页一页翻动

在 more 这个程序的运行过程中，你有几个按键可以按的：

空白键 (space)：代表向下翻一页；

Enter ：代表向下翻『一行』；

/字串 ：代表在这个显示的内容当中，向下搜寻『字串』这个关键字；

:f ：立刻显示出档名以及目前显示的行数；

q ：代表立刻离开 more ，不再显示该文件内容。

b 或 [ctrl]-b ：代表往回翻页，不过这动作只对文件有用，对管线无用。

#### less (比more好用)

一页一页翻动（可以往后翻）

less运行时可以输入的命令：

空白键 ：向下翻动一页；

b：往回翻一页

[pagedown]：向下翻动一页；

[pageup] ：向上翻动一页；

/字串 ：向下搜寻『字串』的功能；

?字串 ：向上搜寻『字串』的功能；

n ：重复前一个搜寻 (与 / 或 ? 有关！)

N ：反向的重复前一个搜寻 (与 / 或 ? 有关！)

q ：离开 less 这个程序；

less **-N** 文件名 // 显示行号

也可以在命令行模式，:-N 显示行号，:-n 不显示行号

#### head

取出前面几行

head -n 20

或直接head -20

#### tail

取出后面几行

tail -f 10 file 一般用于动态查看日志，显示最后10行

-f: 动态跟踪文件file的增长情况,tail会每隔1秒检查下文件是否增加新内容

-n:输出几行

#### wc

wc [选项] [文件名]

-c选项统计字节数

-w选项统计单词数

-l选项统计行数

#### cut

以每一行为一个处理对象的，剪切数据

-b: 按字节（bytes）定位

-c: 按字符（characters）定位

-f: 按域（fields）定位

cut **-b** 3-5,8 file

3-5指第3至5个**字节**；

逗号连接多个字符；

若写成8,3-5也是一样顺序输出，因为cut会将定位数先升序排列再剪切

cut -b -3 //从头到第3个字节（含第三个字节）

cut -b 3- //从第3个字节到末尾（含第三个字节）

cut -c 3-5,8 file

-c 以**字符**为单位来提取内容

对于多字节字符好用（一个汉字3个字节，一个英文字母1个字节）

cut **-d** : **-f** 1,3-5

按照冒号为间隔,提取第1个、第3-5个域

cut -d ‘ ’ f 1

以**一个**空格作为间隔符，注意cut不能以多个空格作为间隔符

### 文本处理

#### sed 文本替换

见印象笔记。

#### grep

见印象笔记

#### 批量kill掉多个进程

**ps -ef|grep php|grep -v grep|cut -c 9-15|xargs kill -9**

"ps - ef"是linux 里查看所有进程的命令。

"grep mcfcm\_st"的输出结果是，所有含有关键字"mcfcm\_st"的进程

"grep -v grep"是在列出的进程中去除含有关键字"grep"的进程。

"**cut** -c 9-15"是截取输入行的第9个字符到第15个字符，而这正好是进程号PID。

"xargs kill -9"中的xargs命令是用来把前面命令的输出结果（PID）作为"kill -9"命令的参数，并执行该令。kill不支持管道，用xargs来传递参数

"kill -9"会强行杀掉指定进程。

### 系统命令

top 实时显示 process 的动态

top -c：显示命令的全路径以及命令参数

top 状态下按 k pid, 杀死进程

按c 显示命令名称和完整命令行

按b键，高亮显示当前运行进程

按x键，排序列的加亮显示

shift + >”或”shift + <” 向右或左改变排序列

按M　根据内存使用大小排序

按P 根据CPU使用率进行排序 （默认排序）

按1　　　　展开多核cpu显示

按E top顶部内存信息，每次切换转换率为1000，切换的单位为 k,m,g,t,p

按e top下的进程信息，每次切换转换率为1000，切换的单位也是 k,m,g,t,p

**VIRT：virtual memory usage 虚拟内存**

1、进程“需要的”虚拟内存大小，包括进程使用的库、代码、数据等

2、假如进程申请100m的内存，但实际只使用了10m，那么它会增长100m，而不是实际的使用量

**RES：resident memory usage 常驻内存**

1、进程当前使用的内存大小，但不包括swap out

2、包含其他进程的共享

3、如果申请100m的内存，实际使用10m，它只增长10m，与VIRT相反

**SHR：shared memory 共享内存**

1、除了自身进程的共享内存，也包括其他进程的共享内存

2、虽然进程只使用了几个共享库的函数，但它包含了整个共享库的大小

3、**计算某个进程所占的物理内存大小公式：RES – SHR**

#### 查看磁盘空间(disk free)

df -hl

h:把显示的单位改成容易辨认的单位

l: 只显示本地磁盘分区，不包含的分区比如其他服务器共享的磁盘。

df命令使用statfs系统调用，是基于分区元数据的文件获取，是针对整个分区的。

df [dir/file name]来查看文件目录数据存放在哪个filesystem

#### 查看文件和目录大小(disk usage)

du -sh 目录：查看目录总大小，不加目录表示查看当前目录总大小

s: 查看总大小

h: 换成容易辨认的单位

du命令会对待统计文件逐个调用fstat，获取文件的大小。基于文件获取，可以获取多个分区的，可以跨分区操作，但文件过多时，逐个命令会比较慢，可以采取**du -h --max-depth=1 /文件夹** 去逐层查找文件。

du -sh \* 查看所有文件的大小

du -sh .[!.]\* 查看所有隐藏文件大小

.[!.]\* 是正则式, 意思是第一位是点, 第二位是除了点以外的字符, 第三位是任意字符或者不存在

du -h /data :查看/data各个子目录，子目录的子目录的大小

du -h --max-depth=1 /文件夹 逐层查找文件

#### du与df不一致

当出现不一致时，是进行了文件删除操作。当一个文件删除后，文件系统目录中已经不存在，du命令将不再统计。此时若还有进程持有该文件的句柄，那么该文件没有真正从磁盘中删除，仍然占用内存空间。df命令仍然会统计这个文件。

解决：

当由于文件删除，发现空间没有被释放，即df和du命令不一致时：

使用**lsof -n | grep deleted**命令查看处于deleted状态的文件，即表示文件被删除，但仍有进程使用这个文件，导致内存空间没有被释放。

找到占用该文件的进程id,删除该pid,即可释放内存空间。

##或者重启电脑，关闭所有进程，也能释放资源。

#### 清缓存

sync

echo 1 > /proc/sys/vm/drop\_caches

echo 2 > /proc/sys/vm/drop\_caches

echo 3 > /proc/sys/vm/drop\_caches

#### 查看系统版本

cat /etc/redhat-release

#### 查看端口使用情况

netstat -ntlp

-n: 拒绝显示别名，能显示数字的全部转化为数字

-t: 仅显示tcp相关选项

-l: 仅列出在Listen(监听)的服务状态

-p: 显示程序名

### shell脚本

#!/bin/bash 指明解释此脚本的shell程序

chmod +x ./test.sh #使脚本具有执行权限

./test.sh #执行脚本

注:一定要写成 ./test.sh ,而不是test.sh, 运行其它二进制的程序也一样，直接写 test.sh，linux 系统会去 PATH 里寻找有没有叫 test.sh 的，而只有 /bin, /sbin, /usr/bin，/usr/sbin 等在 PATH 里，你的当前目录通常不在 PATH 里，所以写成 test.sh 是会找不到命令的，要用 ./test.sh 告诉系统说，就在当前目录找。

脚本开头加：

set -e // 如果任何语句的执行结果不是true则应该退出

-c:

/bin/bash + 执行的脚本

/bin/bash test.sh

/bin/bash -c ‘执行的命令’

/bin/bash -c ‘ls’

#### shell变量

|  |
| --- |
| variable=value  variable='value'  variable="value" // **=两边不能有空格** |

如果 value 不包含任何空白符（例如空格、Tab 缩进等），那么可以不使用引号；如果 value 包含了空白符，那么就必须使用引号包围起来。

单引号会原样输出

双引号会先解析里面的变量和命令

所以建议：如果变量的内容是数字，那么可以不加引号；如果真的需要原样输出就加单引号；其他没有特别要求的字符串等最好都加上**双引号**

在 Bash Shell 中，如果不特别指明，每一个变量的值都是字符串，无论你给变量赋值时有没有使用引号，值都会以**字符串**的形式存储。

your\_name="runoob.com" 无$符号 ,定义时不加$

echo $your\_name 使用变量 ,使用时加$

或echo ${your\_name}

只读变量：readonly your\_name

只读变量不能再次给他赋值了

删除变量：unset your\_name

将命令的执行结果赋值给变量：

两种方式：

|  |
| --- |
| variable=**`**command**`**  variable=**$(** command ）// 小括号 |

#### 字符串

单引号字符串的限制：

单引号里的任何字符都会原样输出，单引号字符串中的变量是无效的；

单引号字串中不能出现单引号（对单引号使用转义符后也不行）。

双引号的优点：

双引号里可以有变量

双引号里可以出现转义字符

拼接字符串：

name="Shell"

url="http://c.biancheng.net/shell/"

str1=$name$url #中间不能有空格

str2="$name $url" #如果被双引号包围，那么中间可以有空格

## $name 和 $url 之间之所以可以出现空格，是因为当字符串不被任何一种引号包围时，遇到空格就认为字符串结束了，空格后边的内容会作为其他变量或者命令解析

str3=$name": "$url #中间可以出现别的字符串

str4="$name: $url" #这样写也可以

str5="${name}Script: ${url}index.html" #这个时候需要给变量名加上大括号，若不加大括号的话会把nameScript作为一个变量

获取字符串长度：

echo ${#your\_name}

提取子字符串：

echo ${your\_name:1:4} //从字符串第 **2** 个字符开始截取 **4** 个字符：

查找字符串：

echo `epr index “${your\_name}” y`

截取字符串

# 截取右边字符

${string#\*chars} // \*chars 忽略左边的所有字符，直到遇见 chars, 开始截取chars右边的字符，返回值不包含chars

${string##\*chars} // 直到最后一个指定字符（子字符串）再匹配结束

% 截取左边字符

${string%chars\*} 同上面类似

#### shell数组

数组名=(值1 值2 ... 值n)

读取数组元素值:

${数组名[下标]}

用@或\*来获取数组里的所有元素：(输出的是个字符串！)

echo ${array\_name[@]} 或echo ${array\_name[\*]}

获取数组长度：

echo ${#array\_name[@]} 或 ${#array[\*]}

数组合并：

array\_new=(${array1[@]} ${array2[@]})

删除数组元素：

unset array\_name[index] // 注意此时变量没有$

往数组添加元素：

arr+=(v1...vN)

截取部分长度

${arr[\*]:start:length} // start为起始索引，length为截取长度

替换数组元素

${arr[\*]/element/replacement} // 将数组里的element 替换为replacement

#### shell传递参数

#!/bin/bash

echo '文件名：'$0

echo '第一个参数：'$1

echo '第二个参数：'$2

echo '第三个参数：'$3

运行shell脚本时，./test.sh 1 2 3 会将三个参数传入脚本中

$0表示脚本文件名，$1为传入的第一个参数，以此类推

另外：





在不加双引号时，$\* 和$@ 都是参数的数组，但是加了双引号，”$\*”就成了字符串，”$@”还是数组

#### 数学计算命令 (())

* + 1. （（表达式）） 整数运算

|  |  |
| --- | --- |
| 运算操作符/运算命令 | 说明 |
| ((a=10+66) ((b=a-15)) ((c=a+b)) | 这种写法可以在计算完成后给变量赋值。以 ((b=a-15)) 为例，即将 a-15 的运算结果赋值给变量 b。  注意，**使用变量时不用加$前缀，(( )) 会自动解析变量名**。 |
| a=$((10+66) b=$((a-15)) c=$((a+b)) | 可以在 (( )) 前面加上$符号获取 (( )) 命令的执行结果，也即获取整个表达式的值。以 c=$((a+b)) 为例，即将 a+b 这个表达式的运算结果赋值给变量 c。  注意，类似 c=((a+b)) 这样的写法是错误的，不加$就不能取得表达式的结果。 |
| ((a>7 && b==c)) | (( )) 也可以进行逻辑运算，在 if 语句中常会使用逻辑运算。  可以用**> ,>=**等 |
| echo $((a+10)) | 需要立即输出表达式的运算结果时，可以在 (( )) 前面加$符号。 |
| ((a=3+5, b=a+10)) | 对多个表达式同时进行计算。 |

* + 1. Shell 不能直接进行算数运算，必须使用数学计算命令

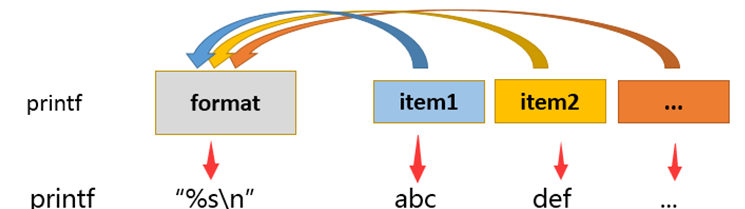
echo 2+8 // 2+8

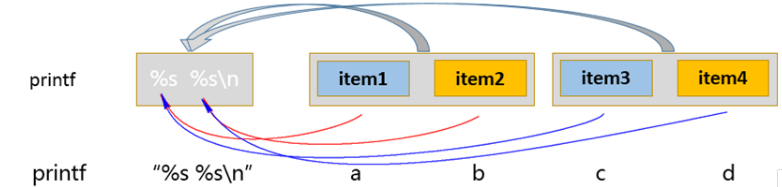
shell会把+两边的数据（数值或者变量）当做字符串，最终拼接成新字符串

**这是因为，在 Bash Shell 中，如果不特别指明，每一个变量的值都是字符串，无论你给变量赋值时有没有使用引号，值都会以字符串的形式存储。**

#### printf

printf [format] [文本1] [文本2]





item 按顺序，依次去填占位符

|  |  |
| --- | --- |
| %s | 字符串 |
| %f | 浮点格式 |
| %c | ASCII字符，即显示对应参数的第一个字符 |
| %d,%i | 十进制整数 |
| %o | 八进制值 |
| %u | 不带正负号的十进制值 |
| %x | 十六进制值（a-f） |
| %X | 十六进制值（A-F） |
| %% | 表示%本身 |

\r 表示回到行首

\n 表示换行，但不回到行首

所以**\n\r**才代表回车

**format**: 指定格式（**%后第一个数字代表宽度，.后的数字表示保留几位小数**）

case 见印象笔记printf

#### echo命令

* 用于字符串的输出
* 原样输出 用单引号
* 输出变量的值 用双引号
* 显示命令执行结果

echo `date` (反引号)

Thu Jul 24 10:08:46 CST 2014

* 换行(shell默认是不会开启转义的)

echo -e "OK! \n" # -e 开启转义

* 不换行

echo -e "OK! \c" # -e 开启转义 \c 不换行（下面输出会与OK在同一行）

* 将结果输出到文件

echo "It is a test" > myfile

echo -n “sth” // echo 命令输出结束后默认会换行, -n 可以不换行

#### 输入/输出重定向

|  |  |
| --- | --- |
| 命令 | 说明 |
| command > file | 将输出重定向到 file。全称：**command 1>file** |
| command < file | 将输入重定向到 file。全称：**command 0<file** |
| command >> file | 将输出以追加的方式重定向到 file。 |
| n > file | 将文件描述符为 n 的文件重定向到 file。 |
| n >> file | 将文件描述符为 n 的文件以追加的方式重定向到 file。 |
| n >& m | 将输出文件 m 和 n 合并。 |
| n <& m | 将输入文件 m 和 n 合并。 |
| << tag | 将开始标记 tag 和结束标记 tag 之间的内容作为输入。 |

一般情况下，每个 Unix/Linux 命令运行时都会打开三个文件：

* 标准输入文件(stdin)：stdin的文件描述符为0，Unix程序默认从stdin读取数据。
* 标准输出文件(stdout)：stdout 的文件描述符为1，Unix程序默认向stdout输出数据。
* 标准错误文件(stderr)：stderr的文件描述符为2，Unix程序会向stderr流中写入错误信息。

so：

* 如果希望 stderr 重定向到 file，可以这样写：

command 2 >> file

* 如果希望将 stdout 和 stderr 合并后重定向到 file，可以这样写：

command >> file 2>&1

从左到右是有顺序的：

|  |
| --- |
| Command >log 2>&1  本来1----->屏幕 （1指向屏幕）  执行>log后， 1----->log (1指向log)  执行2>&1后， 2----->1 (2指向1，而1指向log,因此2也指向了log)  若换成command 2>&1 >log  本来1----->屏幕 （1指向屏幕）  执行2>&1后， 2----->1 (2指向1，而1指向屏幕,因此2也指向了屏幕)  执行>log后， 1----->log (1指向log，2还是指向屏幕)  注意：>log 永远是将stdout指向log，即1 > log  所以这就不是我们想要的结果。 |

每次写>log 2>&1太麻烦：可以简写：

|  |
| --- |
| &>log 一般用这个  或  >&log |

* 若不希望在屏幕显示结果，可以将输出重定向到 /dev/null

command > /dev/null

注：/dev/null 是一个特殊的文件，写入到它的内容都会被丢弃；如果尝试从该文件读取内容，那么什么也读不到。但是 /dev/null 文件非常有用，将命令的输出重定向到它，会起到"禁止输出"的效果。

如果希望屏蔽 stdout 和 stderr，可以这样写：

command > /dev/null 2>&1

* **输出**重定向的完整写法其实是fd>file或者fd>>file，其中 fd 表示文件描述符，如果不写，默认为 1，也就是标准输出文件。

当文件描述符为 1 时，一般都省略不写，但其他描述符不能省略

command >file 实际上是command 1>file

* **输入**重定向的完整写法是fd<file，其中 fd 表示文件描述符，如果不写，默认为 0，也就是标准输入文件。

command <file 实际上是 command 0<file

* **代码块重定向**

代码块是由多条语句组成的一个整体，for、while、until循环或者if…else、case…in选择结构，或者由{ }包围起来的命令都可以称为代码块。

**逐行读取文件内容，read每次读取一行数据**

|  |
| --- |
| while read str; do  echo $str  done <readme.txt |

|  |
| --- |
| {  read name;  read url;  read age  } <log.txt |

* Here Document

|  |
| --- |
| cat << EOF  欢迎来到  菜鸟教程  www.runoob.com  EOF |

将EOF之间的内容，重定向给stdin文件，cat 读取stdin文件

关于Curl的输出：若要屏蔽掉执行时间等参数，可以加-s(静默模式)参数即可

#### 文件描述符

当一个应用程序刚刚启动的时候，0是标准输入，1是标准输出，2是标准错误。如果此时去打开一个新的文件，它的文件描述符会是3。POSIX标准要求每次打开文件时（含socket）必须使用当前进程中最小可用的文件描述符号

Linux 中一切皆文件，包括标准输入设备（键盘）和标准输出设备（显示器）在内的所有计算机硬件都是文件。

为了表示和区分已经打开的文件，Linux 会给每个文件分配一个 ID，这个 ID 就是一个整数，被称为文件描述符（File Descriptor）。

|  |  |  |  |
| --- | --- | --- | --- |
| 文件描述符 | 文件名 | 类型 | 硬件 |
| 0 | stdin | 标准输入文件 | 键盘 |
| 1 | stdout | 标准输出文件 | 显示器 |
| 2 | stderr | 标准错误输出文件 | 显示器 |

stdin stdout stderr是三个文件，默认打开的，可以直接使用

#### 流程控制

本质上，if 检测的是命令的退出状态；0成功，1失败

if:

|  |
| --- |
| **if** condition  then  command1  command2  ...  commandN  fi  或：  if condition**;** then  statement(s)  fi |

写成一行，需要加分号：

|  |
| --- |
| if [ $(ps -ef | grep -c "ssh") -gt 1 ]; then echo "true"; fi |

if else:

|  |
| --- |
| if condition  then  command1  command2  ...  commandN  else  command  fi |

if elif else

|  |
| --- |
| if condition1  then  command1  elif condition2  then  command2  else  commandN  fi |

for循环：

|  |
| --- |
| for var in item1 item2 ... itemN  do  command1  command2  ...  commandN  done |

while语句：

|  |
| --- |
| while condition  do  command  done |

case:

|  |
| --- |
| case 值 in  模式1) # 要跟)  command1  command2  ...  commandN  **;;** # ;;表示break  模式2**）**  command1  command2  ...  commandN  ;;  \*) #若以上都不匹配，则执行\*好的命令  Command1  ;;  esac |

;;表示break

;;& 在匹配一个条件之后，并没有退出case结构，而是继续判断下一个条件。(Bash 4.0后)

case可以使用各种通配符

* a)：匹配a。
* a|b)：匹配a或b。
* [[:alpha:]])：匹配单个字母。
* ???)：匹配3个字符的单词。
* \*.txt)：匹配.txt结尾。
* \*)：匹配任意输入，通过作为case结构的最后一个模式。

跳出循环：

break n ： 跳出并结束循环

continue n ：跳出当前循环，进入下一轮循环

n表示跳出循环层数,不传表示跳出当前层的循环

#### 普通命令的逻辑运算

// 先执行command1，只有command1执行成功后， 才会执行command2

command1 && command2

// 先执行command1，只有command1执行失败后， 才会执行command2

command1 || command2

例：

[ -d temp ] || mkdir temp 若temp不存在，则创建temp

[ ! -d temp ] && exit 1 若temp 不存在，退出返回1

If 和 &&结合使用：

if command1 && command2;then

...

fi

例如：

if grep $word1 $filename && grep $word2 $filename

then

echo "$word1 and $word2 are both in $filename."

fi

若在文件里同时搜出来$word1和$word2才算满足条件。

#### for 循环

|  |
| --- |
| for**((**exp1; exp2; exp3**))**  do  statements  done |

并行运行：用“{}”将主执行程序变为一个块，用&放入后台。最后wait指令，等待所有后台进程执行结束，再进行下一步。

|  |
| --- |
| for**((**exp1; exp2; exp3**))**  do  **{**  statements  **}&**  done  **wait** |

#### for in

|  |
| --- |
| for variable in value\_list  do  statements  done |

value\_list的取值：

value\_list 值为**字符串**，以空格分隔

* 直接列出具体值，以空格分隔

|  |
| --- |
| for n in 1 2 3 4 5 6  do  echo $n  done |

* 数组

|  |
| --- |
| for n in ${arr[\*]}  do  echo $n  done |

* 一个取值范围

|  |
| --- |
| {start**..**end} |

start 表示起始值，end 表示终止值；注意中间用两个点号相连，而不是三个点号；

这种形式只支持数字和字母；

* 使用命令的执行结果，使用反引号``或者$()都可以取得命令的执行结果

|  |
| --- |
| for file in $(ls)  do  echo $file  done |

* shell通配符

|  |
| --- |
| for filename in \*.sh  do  echo $filename  done |

* shell特殊变量

|  |
| --- |
| function func(){  for str in $@  do  echo $str  done  } |

$@ ：传递给脚本或函数的所有参数

#### select in

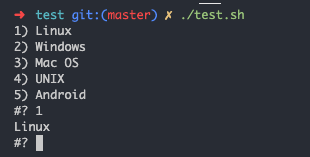
|  |
| --- |
| select variable in value\_list  do  statements  done |

value\_list 中的内容会以菜单的形式显示出来，用户输入菜单编号，就表示选中了某个值，这个值就会赋给变量 variable，然后再执行循环体中的 statements

注意：select 是无限循环（死循环），输入空值，或者输入的值无效，都不会结束循环，只有遇到 **break** 语句，或者按下 Ctrl+D 组合键才能结束循环。

例子：

|  |
| --- |
| select name in "Linux" "Windows" "Mac OS" "UNIX" "Android"  do  echo $name  done |



结合case in 的完整实例：

|  |
| --- |
| echo "What is your favourite OS?"  select name in "Linux" "Windows" "Mac OS" "UNIX" "Android"  do  case $name in  "Linux")  echo "Linux是一个类UNIX操作系统，它开源免费，运行在各种服务器设备和嵌入式设备。"  ;;  "Windows")  echo "Windows是微软开发的个人电脑操作系统，它是闭源收费的。"  ;;  "Mac OS")  echo "Mac OS是苹果公司基于UNIX开发的一款图形界面操作系统，只能运行与苹果提供的硬件之上。"  ;;  "UNIX")  echo "UNIX是操作系统的开山鼻祖，现在已经逐渐退出历史舞台，只应用在特殊场合。"  ;;  "Android")  echo "Android是由Google开发的手机操作系统，目前已经占据了70%的市场份额。"  ;;  \*)  echo "输入错误，请重新输入"  esac  done |

#### 函数

|  |
| --- |
| function name() {  statements  [return value]  }  简化版：  name() {  statements  [return value]  }  function name {  statements  [return value]  }  调用：  name 1 2 3 // 注意不加括号 |

* function fun {}定义，也可以直接fun() {}定义,不能带任何参数。

但是不能直接fun {} 定义

* 参数返回，可以显示加：return 返回，如果不加，将以最后一条命令运行结果，作为返回值。return后跟数值n(0-255)
* 所有函数在**使用前必须定义**；
* 调用函数仅使用其函数名即可。

函数参数：

定义的时候不能带参数，调用的时候可以传参数！

通过 $n 的形式来获取参数的值，例如，$1表示第一个参数，$2表示第二个参数...

函数返回值：

* return $? 只能返回0-255的数字，其他的不行
* echo + $() 将要返回的值echo出来，通过$() 获取
* 函数内部声明的变量是全局变量，要使用局部变量用local 声明

|  |
| --- |
| function test(){  foo=3  return $foo  }  test  echo $? // 3  echo $foo // 3  function test(){  local foo=3  return $foo  }  test  echo $? // 3  echo $foo // |

#### $?

* 获取上个命令的退出状态，0表示成功，1表示失败
* 获取函数返回值

#### read

|  |
| --- |
| echo "What is your name?"  **read** PERSON  echo "Hello, $PERSON" |

read 命令从标准输入读取**单行**数据，并将其赋值给变量

如果没有进行重定向，默认就是从键盘读取用户输入的数据；

如果进行了重定向，那么可以从文件中读取数据。

read [-options] [var1...varN]

options:

|  |  |
| --- | --- |
| 选项 | 说明 |
| -a array | 把读取的数据赋值给数组 array，从下标 0 开始。 |
| -n num | 读取 num 个字符，而不是整行字符，不用等待用户按下回车键 |
| -p prompt | 显示提示信息，提示内容为 prompt。 |
| -s | 静默模式（Silent mode），不会在屏幕上显示输入的字符 |
| -t seconds | 设置超时时间，单位为秒。 |

case1:

|  |
| --- |
| read -t 10 -sp "请输入密码：" pass1 && printf "\n" &&  read -t 10 -sp "请再次出入密码" pass2 && printf "\n" && [[ $pass1=$pass2 ]]  if [[ $? ]];then  echo '成功'  else  echo '失败'  fi |

case2:

用户每行输入一个数字，计算所有数字的和;

在终端中读取数据，可以等价为在文件中读取数据;

按下 Ctrl+D 组合键表示读取到文件流的末尾，此时 read 就会读取失败，得到一个**非 0 值**的退出状态，从而导致判断条件不成立，结束循环

|  |
| --- |
| sum=0  echo "请输入您要计算的数字，按 Ctrl+D 组合键结束读取"  while **read n**  do  ((sum += n))  done  echo "The sum is: $sum" |

case3:

// 可以同时给多个变量赋值，输入时多个值必须在一行，不能换行

read var1 var2 var3

#### exit

exit 退出状态只能是一个介于 0~255 之间的整数，其中只有 0 表示成功，其它值都表示失败。

#### test

检测某个条件是否成立,通常和if一起使用

|  |
| --- |
| test expression  或  [ expression ] 中括号里两边一定要留空格 |

当 test 判断 expression 成立时，退出状态为 0，返回true;否则为非 0 值，返回false;

这一点跟(()) 刚好相反，((0)) 返回false, ((其他))返回true

* 检测文件

|  |  |
| --- | --- |
| 选 项 | 作 用 |
| -e filename | 判断文件是否存在。 |
| -f filename | 判断文件是否存在，井且是否为普通文件。 |
| -d filename | 判断文件是否存在，并且是否为目录文件。 |

* 数值比较

|  |  |
| --- | --- |
| 选 项 | 作 用 |
| num1 -eq num2 | 判断 num1 是否和 num2 相等。 |
| num1 -ne num2 | 判断 num1 是否和 num2 不相等。 |
| num1 -gt num2 | 判断 num1 是否大于 num2 。 |
| num1 -lt num2 | 判断 num1 是否小于 num2。 |
| num1 -ge num2 | 判断 num1 是否大于等于 num2。 |
| num1 -le num2 | 判断 num1 是否小于等于 num2。 |

test只能用来比较整数，小数还得用bc命令。

* 字符串比较

|  |  |
| --- | --- |
| 选 项 | 作 用 |
| [ str ] | 字符串长度大于0，返回true |
| [ -n str ] | 字符串长度大于0，返回true |
| [ -z str ] | 字符串长度为0，返回true |
| [ str1 = str2 ] [ str1 == str2 ] | =和==是等价的，都用来判断 str1 是否和 str2 相等。=号两端也要留空格 |
| [ str1 != str2 ] | 判断 str1 是否和 str2 不相等。 |
| [ str1 \> str2 ]  [ str1 ‘>’ str2] | 判断 str1 是否大于 str2。\>是>的转义字符，这样写是为了防止>被误认为成重定向运算符。  或者用引号引起来 |
| [ str1 \< str2 ]  [ str1 ‘<’ str2 ] | 判断 str1 是否小于 str2。同样，\<也是转义字符。 |

注意：==、>、< 在 Shell 中，只能用来比较字符串，不能比较数字！！

不管是比较数字还是字符串，Shell 都不支持 >= 和 <= 运算符

* 逻辑运算

|  |  |
| --- | --- |
| 选 项 | 作 用 |
| expression1 -a expression | 逻辑与，表达式 expression1 和 expression2 都成立，最终的结果才是成立的。 |
| expression1 -o expression2 | 逻辑或，表达式 expression1 和 expression2 有一个成立，最终的结果就成立。 |
| ! expression | 逻辑非，对 expression 进行取反。(**!后有空格**) |

test奇葩用法：

* + 比较数字必须用-gt 等选项
  + >、<、== 只能用来比较字符串，不能用来比较数字
  + 不管是比较字符串还是数字，test 都不支持 >= 和 <=
  + 强烈建议将变量用双引号""包围起来，这样能避免变量为空值时导致的很多奇葩问题

|  |
| --- |
| file=""  [ -f $file ] // [ -f ] 会返回真  echo $? // 返回0  file=""  [ -f "$file" ] // [ -f “” ] 会返回假  echo $? // 返回1  **但是使用[[]] ，就不用管这些细节，不用加双括号** |

#### [[]] 检测某个条件是否成立

[[ expression ]]

两种写法的比较：

|  |  |  |  |
| --- | --- | --- | --- |
| test 或 [] | | [[ ]] | |
| [ -z "$str1" ] || [ -z "$str2" ] | √ | [[ -z $str1 ]] || [[ -z $str2 ]] | √ |
| [ -z "$str1" -o -z "$str2" ] | √ | [[ -z $str1 -o -z $str2 ]] | × |
| [ -z $str1 || -z $str2 ] | × | [[ -z $str1 || -z $str2 ]] | √ |

[[]]支持正则

|  |
| --- |
| [[ str =**~** regex ]] |

总结：[[]]比test更友好，推荐使用，但是对于数字比较仍就不友好；

所以，用 (()) 来处理整型数字，用 [[ ]] 来处理字符串或者文件

#### 注释

单行注释 #

多行注释

|  |
| --- |
| #!/bin/bash  **: '**  下面是两个数的相乘的积  这里是7乘9  **'**  ((area=7\*9))  echo $area |

#### 内建命令

alias 别名=’原命令 -选项/参数’ // 为指定命令定义一个别名

dirs 当前文件所在目录的路径

eval 将指定的参数拼接成一个命令，然后执行该命令

export 设置子 shell 进程可用的变量

exec 用指定命令替换 shell 进程

exit 强制 shell 以指定的退出状态码退出

#### eval

首先扫描命令行进行所有的置换，然后再执行该命令,两次扫描

eval “cat file” // 会执行cat file，拿出文件内容

### 通配符

#### ?字符

?字符代表单个字符

|  |
| --- |
| # 存在文件 a.txt 和 b.txt  $ ls ?.txt  a.txt b.txt |

？不能匹配空字符

两个?? 表示两个字符

#### \*字符

\*代表任意数量的字符

|  |
| --- |
| # 存在文件 a.txt、b.txt 和 ab.txt  $ ls \*.txt  a.txt b.txt ab.txt |

\*可以匹配空字符

#### […]模式

[...]匹配方括号之中的任意一个字符

[start-end]表示一个连续的范围

|  |
| --- |
| # 存在文件 a.txt 和 b.txt  $ ls [ab].txt  a.txt b.txt  # 存在文件 a.txt、b.txt 和 c.txt  $ ls [a-c].txt  a.txt b.txt c.txt |

#### 7.4 其他

[:alpha:] 匹配单个字母

[:lower:] 匹配单个小写字母

[:upper:] 匹配单个大写字母

[:digit:] 匹配单个数字

[:alnum:] 匹配单个数字或字母

[:punct:] 匹配单个符号

[:space:] 匹配单个空格

例如：

以大写字母开头的文档

[[:upper:]]\* 外层[] 即7.3 匹配内部任意一个字符，内层[:upper:]匹配单个小写字母，外面的\*，匹配后面的所有其他字符

### awk

见印象笔记

http://c.biancheng.net/view/4097.html

#### 基本用法

|  |
| --- |
| # 格式  $ awk 动作 文件名  # 示例  $ awk **'**{print $0}**'** demo.txt  **{} 要用单引号引上** |

花括号里是demo.txt文件里**每一行**的处理动作；

$0: 当前行

awk会根据空格和制表符，将每一行分成若干字段，依次用$1、$2、$3代表第一个字段、第二个字段、第三个字段…

**-F** 指定分隔符，按分隔符将每一行分割开来

#### 变量

NF：当前行有多少个字段（字段相当于单词）

$NF：代表最后一个字段

$(NF-1)：代表倒数第二个字段

NR：表示当前处理的是第几行

其他：

|  |
| --- |
| FILENAME：当前文件名  FS：字段分隔符，默认是空格和制表符。  RS：行分隔符，用于分割每一行，默认是换行符。  OFS：输出字段的分隔符，用于打印时分隔字段，默认为空格。  ORS：输出记录的分隔符，用于打印时分隔记录，默认为换行符。 |

print命令里面，如果原样输出字符，要放在双引号里面

#### 函数

|  |
| --- |
| tolower()：字符转为小写。  length()：返回字符串长度。  substr()：返回子字符串。  sin()：正弦。  cos()：余弦。  sqrt()：平方根。  rand()：随机数。 |

#### 条件

|  |
| --- |
| awk '条件 动作' 文件名 |

例如：

|  |
| --- |
| awk -F ':' '/usr/ {print $1}' demo.txt |

print命令前面是一个正则表达式，只输出包含usr的行。

#### if

|  |
| --- |
| awk -F ':' '**{**if ($1 > "m") print $1**}**' demo.txt |

if语句要放到花括号里

if…else

|  |
| --- |
| awk -F ':' '{if ($1 > "m") print $1; else print "---"}' demo.txt |

### 终端控制符

作用：控制终端的显示效果

例如：\x1b[0K

\x1b 表示16进制码1b，1b对应ascii码是ESC，可以表示\033（8进制）

[是一个CSI(Control sequence introducer),转义序列作用由最后一个字符决定，即K（清除行）；0是参数，表恢复默认属性。

该语句的含义是：清除光标右边文本。

设置终端字体颜色和背景颜色：

echo -e "\033[背景颜色号;字体颜色m字符串\033[0m"

字体颜色：

|  |  |
| --- | --- |
| 颜色 | 代码 |
| 黑色前景 | 30 |
| 红色前景 | 31 |
| 绿色前景 | 32 |
| 棕色前景 | 33 |
| 蓝色前景 | 34 |
| 紫色前景 | 35 |
| 青色前景 | 36 |
| 白色前景 | 37 |

背景颜色：

|  |  |
| --- | --- |
| 颜色 | 代码 |
| 黑色背景 | 40 |
| 红色背景 | 41 |
| 绿色背景 | 42 |
| 棕色背景 | 43 |
| 蓝色背景 | 44 |
| 紫色背景 | 45 |
| 青色背景 | 46 |
| 白色背景 | 47 |

### export

修改环境变量

#### export PATH=/usr/local/mongodb/bin:$PATH

通过echo $PATH查看

生效方法：立即生效

有效期限：临时改变，只能在当前的终端窗口中有效，当前窗口关闭后就会恢复原有的path配置

用户局限：仅对当前用户

#### 修改~/.bashrc文件

vim ~/.bashrc

//在最后一行添上：

export PATH=/usr/local/mongodb/bin:$PATH

生效方法：（有以下两种）

1、关闭当前终端窗口，重新打开一个新终端窗口就能生效

2、输入“source ~/.bashrc”命令，立即生效

有效期限：永久有效

用户局限：仅对当前用户（~/.bashrc保存的是当前用户的自定义配置）

#### 修改 /etc/profile

vim /etc/profile

找到设置PATH的行，添加:

export PATH=/usr/local/mongodb/bin:$PATH

生效方法：系统重启

有效期限：永久有效

用户局限：对所有用户

#### 修改environment文件

vim /etc/environment

在PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games"中加入“:/usr/local/mongodb/bin”

生效方法：系统重启

有效期限：永久有效

用户局限：对所有用户

* 1. $PATH

PATH=/bin:/usr/bin:/usr/local/bin:/usr/local/jdk/bin

PATH 以**：**号分割不同的目录

当接到用户送入的命令时，依次检索“/bin”、“/usr/bin”、“/usr/local/bin”、“/usr/local/jdk/bin”等目录，直到找到用户输入的命令

添加PATH环境变量：

export PATH=$PATH:<newPath>

即在旧$PATH后面拼上新路径，最后再重新赋值给PATH并导出

### profile、bashrc、~/.bash\_profile、~/.bashrc、~/.bash\_profile 区别

============

/etc/profile

============

此文件为系统的每个用户设置环境信息,当用户第一次登录时,该文件被执行.

并从/etc/profile.d目录的配置文件中搜集shell的设置.

===========

/etc/bashrc

===========

为每一个运行bash shell的用户执行此文件.当bash shell被打开时,该文件被读取.

===============

~/.bash\_profile

===============

每个用户都可使用该文件输入专用于自己使用的shell信息,当用户登录时,该

文件仅仅执行一次!默认情况下,他设置一些环境变量,执行用户的.bashrc文件.

=========

~/.bashrc

=========

该文件包含专用于你的bash shell的bash信息,当登录时以及每次打开新的shell时,该文件被读取.

==========

~/.profile

==========

在Debian中使用.profile文件代 替.bash\_profile文件

.profile(由Bourne Shell和Korn Shell使用)和.login(由C Shell使用)两个文件是.bash\_profile的同义词，目的是为了兼容其它Shell。在Debian中使用.profile文件代 替.bash\_profile文件。

==============

~/.bash\_logout

==============当每次退出系统(退出bash shell)时,执行该文件.

**执行顺序：**

①在 刚登录Linux时，

首先启动 /etc/profile 文件，

然后再启动用户目录下的 ~/.bash\_profile、 ~/.bash\_login或 ~/.profile文件中

的其中一个，执行的顺序为：~/.bash\_profile、 ~/.bash\_login、 ~/.profile

以上两个文件会在用户登录时执行

②下面开始执行用户的bash设置

如果 ~/.bash\_profile文件存在的话，一般会以这样的方式执行用户的 ~/.bashrc文件。

在 ~/.bash\_profile文件中一般会有下面的代码：

|  |
| --- |
| # if running bash  if [ -n "$BASH\_VERSION" ]; then  # include .bashrc if it exists  if [ -f "$HOME/.bashrc" ]; then  . "$HOME/.bashrc"  fi  fi |

同样~/.bashrc中，一般还会在文件的前面有以下代码，来执行/etc/bashrc

|  |
| --- |
| if [ -f /etc/bashrc ] ; then  　. /etc/bashrc  fi |

所以，~/.bashrc会调用 /etc/bashrc文件。最后，在退出shell时，还会执行 ~/.bash\_logout文件。

所以，总体执行顺序是：

|  |
| --- |
| /etc/profile  ~/.bash\_profile | ~/.bash\_login | ~/.profile  ~/.bashrc  /etc/bashrc  ~/.bash\_logout |

### expect

expect是个处理交互的命令，可以将交互过程写在脚本里，使之自动化完成。

相当于模拟了用户和命令行的交互操作。

|  |
| --- |
| spawn启动指定进程 -> expect获取指定关键字 -> send想指定进程发送指定指令 -> 执行完成, 退出. |

常用命令：

spawn：启动新的进程，后面跟命令或者指定程序

expect：从进程接收信息，如果匹配成功, 就执行expect后的动作

send：向进程发送字符串，注意加 \n 回车

interact：保持用户的交互状态，否则执行完就退出了

exit： 退出expect脚本

expect eof ：expect执行结束, 退出

send “str**\n**” 将该参数发送到进程,一般后面加换行符，表示回车，即**运行**

expect “str” 等待进程的输入, 若于str匹配，则执行后面的send

三种常用模式：

单一分支模式语法：

expect "hi" {send "You said hi"}

多分支模式语法：

|  |
| --- |
| expect "hi" { send "You said hi\n" } \  "hello" { send "Hello yourself\n" } \  "bye" { send "That was unexpected\n" } |

|  |
| --- |
| expect {  "hi" { send "You said hi\n"}  "hello" { send "Hello yourself\n"}  "bye" { send "That was unexpected\n"}  } |

spawn: 启动新的进程

spawn后的send和expect命令都是和spawn打开的进程进行交互的

结束符：

expect eof ：等待执行结束。由spawn启动的命令在结束时会产生一个eof标记，expect eof 即在等待这个标记

interact ： 执行完成后保持交互状态, 这时可以手动输入信息

用“-c”选项： 从命令行执行expect脚本,默认expect是**交互**地执行的

|  |
| --- |
| $ expect -c ‘expect “\n” {send “pressed enter\n”}’ |

**在shell脚本执行expect**

|  |
| --- |
| #!/bin/bash  /usr/bin/expect **-c** '  spawn ssh jump  send "ssh work@172.17.1.111\n"  interact  '  /usr/bin/expect <<EOF  spawn ssh jump  send "ssh work@172.17.1.111\n"  interact  EOF // EOF后不能有空格  注意： 对于expect << EOF，expect的stdin是here-doc，而不是tty。  但是interact命令只有在expect的stdin是tty时才起作用，所以用interact的话，要使用-c |

使用实例：

**执行xxx.exp脚本**

|  |
| --- |
| #!/usr/bin/expect  spawn ssh jump  send "ssh [work@172.17.1.110\n"](mailto:work@172.17.1.110/n\")  send “command”  interact |

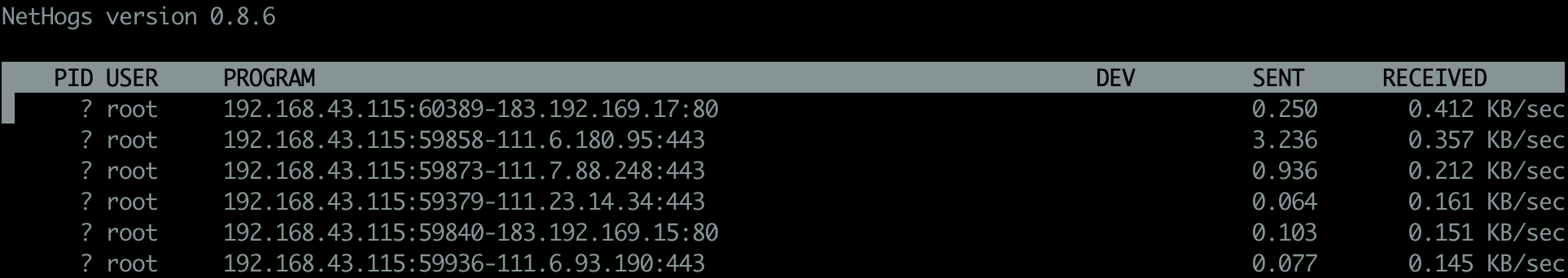
使用./xxx.exp执行

### 三方工具

#### nethogs

监控进程流量

命令：sudo nethogs

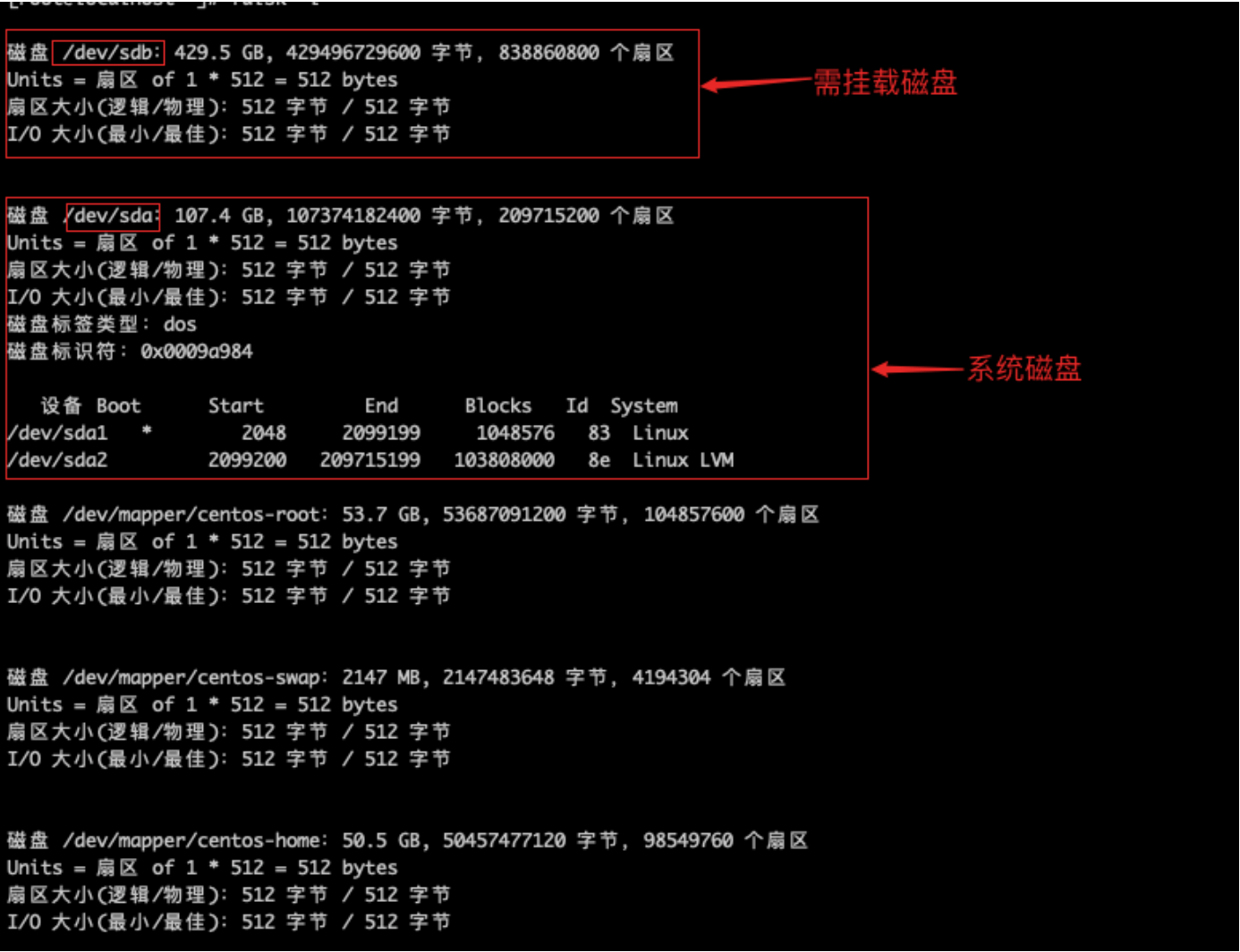


#### iptables

<https://blog.csdn.net/u011537073/article/details/82685586>

### 磁盘挂载

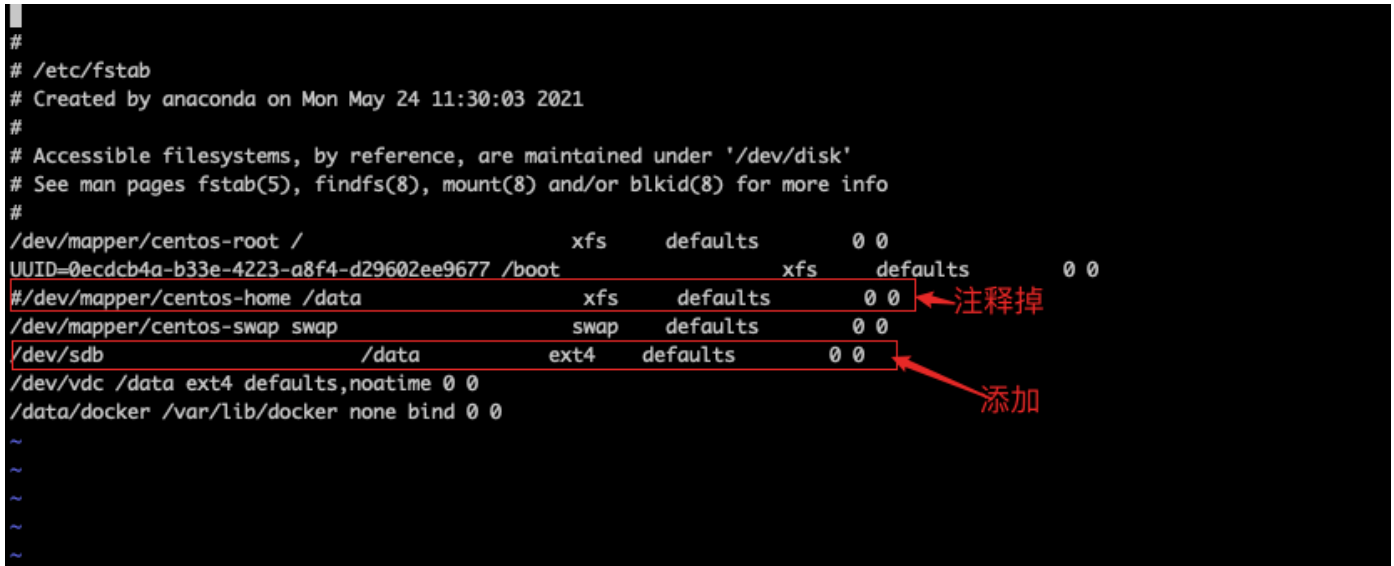
#### 判断需要挂载磁盘的逻辑名称



备注：在本次实例中需挂载磁盘的逻辑名为**/dev/sdb**

#### 挂载磁盘

|  |
| --- |
| mkfs.ext4 -T largefile /dev/sdb //格式化磁盘  mkdir /data //创建挂载⽬录  mount /dev/sdb /data //磁盘挂载  vi /etc/fstab //开启磁盘开机⾃动挂载  ##########################################################  把之前挂载在/data下的注释掉  /dev/sdb /data ext4 defaults 0 0  ########################################################## |



#### 若环境要求node值⼤于等于10000000（可选项,是否操作根据实际情况⽽定）

|  |
| --- |
| umount /dev/sdb  mkfs.ext4 -N 10000000 /dev/sdb  mkdir -pv /data  mount /dev/sdb /data  echo "/dev/vdc /data ext4 defaults,noatime 0 0">> /etc/fstab |

### 切换用户

#### sudo

sudo：暂时切换到超级用户模式以执行超级用户权限，提示输入密码时该密码为当前用户的密码，而不是超级账户的密码，有时间限制，一般15mim

具体命令：

sudo -i 加载用户变量，并跳转到目标用户home目录

sudo -s 不加载用户变量，不跳转目录

#### su

切换到某某用户模式，提示输入密码时该密码为***切换后账户的密码***

没有时间限制

用法：

su commonuser 切换到commonuser账户,要commonuser的密码

su su后不加账户，默认切换到root用户,要root的密码

**su -** 也是切换到root账户，完全等同于root登录

**su和su- 区别：**

su 切换到root后，仍然保持旧的环境变量

su - 切换后，创建一个新的环境（由 root 用户 ~/.bashrc 文件所设置的环境），相当于使用 root 用户正常登录（从登录屏幕登录）

#### sudo su

因为su 命令直接以 root 用户身份登录有风险，一些 Linux 发行版（如 Ubuntu）默认禁用 root 用户帐户，鼓励用户在需要 root 权限时使用 sudo 命令。

**sudo su**

由于你使用 sudo 运行命令，你只需要输入当前用户的密码。所以，一旦完成操作，su 命令将会以 root 用户身份运行，这意味着它不会再要求输入任何密码。

#### 创建普通用户

创建新用户commonuser：

adduser commonuser

初始化密码：

passwd commonuser

Changing password for user lims.

New password: # 输入密码

Retype new password: # 再次输入密码

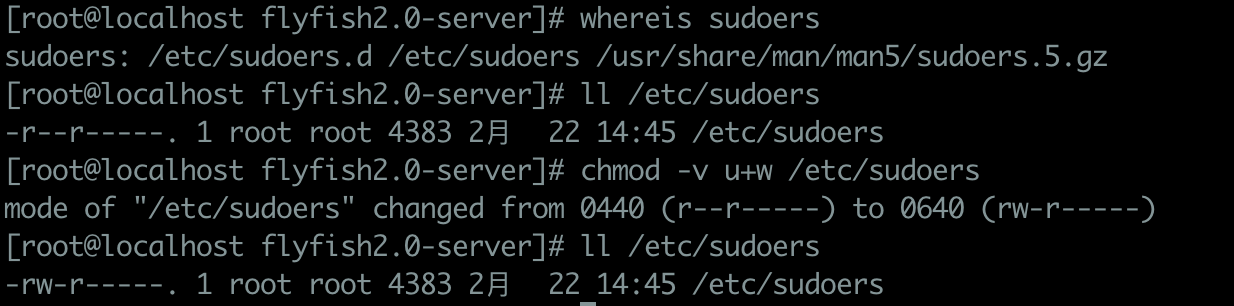
passwd: all authentication tokens updated successfully.

#### 授权：

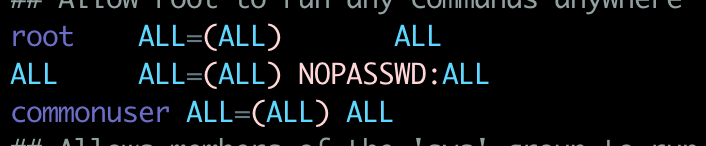
个人用户的权限只可以在本home下有完整权限，其他目录需要别人授权。经常需要root用户的权限，可以通过修改sudoers文件来赋予权限。

新创建的用户并不能使用sudo命令，需要给他添加授权。

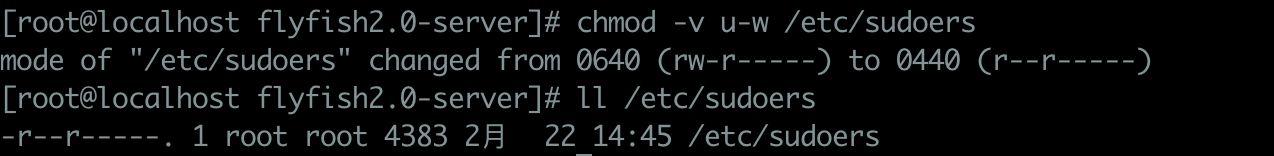
查找sudoers文件路径并赋予权限：



修改sudoers文件：



收回权限：



新用户登录