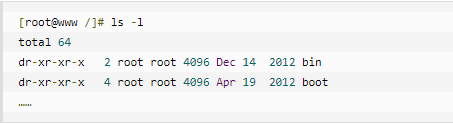
linux学习文档: https://linuxtools-rst.readthedocs.io/zh\_CN/latest/base/index.html

### 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)。



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 |

* 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：

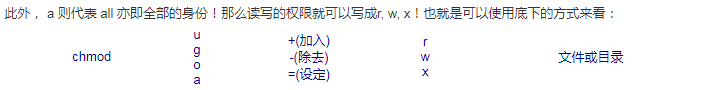
9个权限3个一组，分别是owner（user）/group/others,用每组权限的数字之和表示

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

#### 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)

#### 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

somecommand | xargs -item command

**捕获一个命令的输出，然后传递给另外一个命令**。因为有些命令不支持管道 | 来传递参数；

若没有command，默认是echo, 默认包含空白和换行；

参数：

-n[num]，一次用的argument的个数，默认是用所有的。

-d[x]，修改分隔符，默认的xargs分隔符是\n；

实例：

find . -type f -name "\*.log" **-print0** | xargs **-0** rm -f

### 文件内容查看

#### cat

从第一行开始显示文件内容

|  |
| --- |
| cat [-AbEnTv] |

-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

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；

[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；

#### head

取出前面几行

#### 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

### 文本处理

#### sed 文本替换

常用：

sed 's/text/replace\_text/' file //替换每一行的第一处匹配的text

sed 's/text/replace\_text/g' file // 全局替换

默认替换后，输出替换后的内容，如果需要直接替换原文件,使用-i:

sed -i 's/text/repalce\_text/g' file

sed -i '.bak' 's/text/repalce\_text/g' file //给源文件添加备份 .bak是备份后缀名

在linux下，不加备份后缀会直接覆盖原文件，但mac下必须加空字符串 ''才能实现不备份直接辅导原文件

sed '/^$/d' file //移除空白行

详细：

sed [options] 'command' file(s) //利用script处理文本文件

options:

-e:用指定的scripit处理文本文件(一般用于多点编辑)

-i: 直接替换源文件

command中的参数：

a\ ：在当前行下面插入文本

i\ ：在当前行上面插入文本

c\ : 把选定的行改为新的文本

d : 删除选择的行

p ：打印，亦即将某个选择的数据印出。通常 p 会与参数 sed -n 一起运行～

s ：取代，可以直接进行取代的工作！通常这个 s 的动作可以搭配正则表示法！例如 ：s/old/new/g 就是啦！

#### grep

搜索文件的内容，而不是搜索文件名

|  |
| --- |
| usage: grep [-abcDEFGHhIiJLlmnOoqRSsUVvwxZ] [-A num] [-B num] [-C[num]]  [-e pattern] [-f file] [--binary-files=value] [--color=when]  [--context[=num]] [--directories=action] [--label] [--line-buffered]  [--null] [pattern] [file ...] |

注意：最后一个参数是文件或文件的集合，不能是文件夹！！

参数：

-e: 指定字符串做为查找文件内容的样式(可指定多个样式)

grep -e 'hello' -e 'world' test.txt

-o: 只输出匹配的文本行

-v: 只输出没有匹配的文本行

-c: 统计文件中包含文本的行数

-n: 打印匹配的行号

-i: 搜索时忽略大小写

-l: 只打印文件名（一般搜索范围会是多个文件）

-r: 递归查询

grep "class" . -r -n //在多级目录递归搜索(在当前文件夹下)

pattern正则表达式主要参数：

\： 忽略正则表达式中特殊字符的原有含义。

^：匹配正则表达式的开始行。

$: 匹配正则表达式的结束行。

\<：从匹配正则表达 式的行开始。

\>：到匹配正则表达式的行结束。

[ ]：单个字符，如[A]即A符合要求 。

[ - ]：范围，如[A-Z]，即A、B、C一直到Z都符合要求 。

.：所有的单个字符。

\* ：有字符，长度可以为0。

#### 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

#### 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。

#### 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。

### 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。

#### shell变量

|  |
| --- |
| variable=value  variable='value'  variable="value" |

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**)** |

#### 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：

your\_name="qinjx"

greeting="hello, "$your\_name" !"

greeting\_1="hello, ${your\_name} !"

echo $greeting $greeting\_1

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`

#### 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@来获取数组里的所有元素：

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]

#### 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：





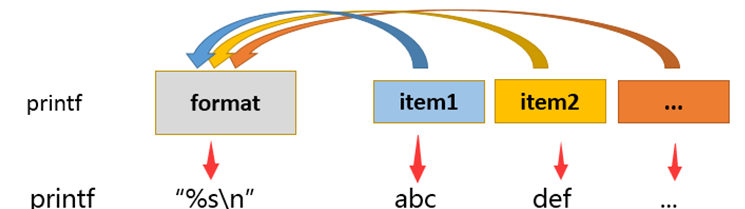
#### 数学计算命令 (())

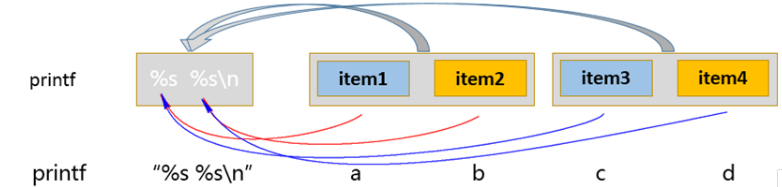
* + 1. （（表达式）） 整数运算

|  |  |
| --- | --- |
| 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。 |
| 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。 |

#### printf

printf [format] [文本1] [文本2]





|  |  |
| --- | --- |
| %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

#### echo命令

* 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

#### 输入/输出重定向

|  |  |
| --- | --- |
| 命令 | 说明 |
| command > file | 将输出重定向到 file。 |
| command < file | 将输入重定向到 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)  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

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

#### 流程控制

本质上，if 检测的是命令的退出状态

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 |

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

#### for 循环

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

#### for in

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

value\_list的取值：

* 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 |

* 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

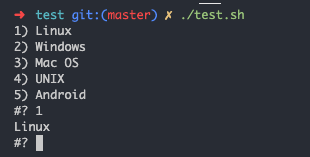
|  |
| --- |
| select variable in value\_list  do  statements  done |

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]  } |

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

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

#### $?

* 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" |

使用 'read' 命令读取用户键盘输入内容，并将其赋值给一个名为 PERSON 的变量

#### 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，否则为非 0 值。

* 检测文件
* 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 |
| -z str | 判断字符串 str 是否为空。 |
| -n str | 判断宇符串 str 是否为非空。 |
| str1 = str2 str1 == str2 | =和==是等价的，都用来判断 str1 是否和 str2 相等。 |
| str1 != str2 | 判断 str1 是否和 str2 不相等。 |
| 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。 |
| 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。 |

test奇葩用法：

* + 比较数字必须用-eq、-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 <=

#### [[]] 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 |

### 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

#### \*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 |

### awk

#### 基本用法

|  |
| --- |
| # 格式  $ awk 动作 文件名  # 示例  $ awk '{print $0}' demo.txt |

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 |