超算培训

linux基础

常用命令介绍

```
(1) ctrl c: 取消命令，并且换行
(2) ctrl u: 清空本行命令
```

(3) tab键：可以补全命令和文件名，如果补全不了快速按两下tab键，可以显示备选选项
(4) ls：列出当前目录下所有文件，蓝色的是文件夹，白色的是普通文件，绿色的是可执行文件
(5) pwd：显示当前路径
(6) cd XXX：进入XXX目录下，cd .. 返回上层目录
(7) cp XXX YYY：将XXX文件复制成YYY，XXX和YYY可以是一个路径，比如../dir_c/a.txt，表示上层目录下的dir_c文件夹下的文件a.txt
(8) mkdir XXX：创建目录XXX
(9) rm XXX：删除普通文件； rm XXX -r：删除文件夹
(10) mv XXX YYY：将XXX文件移动到YYY，和cp命令一样，XXX和YYY可以是一个路径；重命名也是用这个命令
(11) touch XXX：创建一个文件
(12) cat XXX：展示文件XXX中的内容
(13) 复制文本
windows/Linux下：Ctrl + insert，Mac下：command + c
(14) 粘贴文本
windows/Linux下：Shift + insert，Mac下：command + v

tmux教程

功能：
(1) 分屏。
(2) 允许断开Terminal连接后，继续运行进程。
结构：
一个tmux可以包含多个session，一个session可以包含多个window，一个window可以包含多个pane。
实例：

```
tmux:
    session 0:
        window 0:
            pane 0
            pane 1
            pane 2
            ...
        window 1
        window 2
        ...
    session 1
    session 2
    ...
```

操作：
(1) tmux：新建一个session，其中包含一个window，window中包含一个pane，pane里打开了一个shell对话框。
(2) 按下Ctrl + a后手指松开，然后按%：将当前pane左右平分成两个pane。
(3) 按下Ctrl + a后手指松开，然后按"（注意是双引号"）：将当前pane上下平分成两个pane。
(4) Ctrl + d：关闭当前pane；如果当前window的所有pane均已关闭，则自动关闭window；如果当前session的所有window均已关闭，则自动关闭session。
(5) 鼠标点击可以选pane。
(6) 按下ctrl + a后手指松开，然后按方向键：选择相邻的pane。
(7) 鼠标拖动pane之间的分割线，可以调整分割线的位置。
(8) 按住ctrl + a的同时按方向键，可以调整pane之间分割线的位置。
(9) 按下ctrl + a后手指松开，然后按z：将当前pane全屏/取消全屏。
(10) 按下ctrl + a后手指松开，然后按d：挂起当前session。
(11) tmux a：打开之前挂起的session。
(12) 按下ctrl + a后手指松开，然后按s：选择其它session。
方向键 —— 上：选择上一项 session/window/pane

```
        方向键 -- 下：选择下一项 session/window/pane
        方向键 -- 右：展开当前项 session/window
        方向键 -- 左：闭合当前项 session/window
    (13) 按下Ctrl + a后手指松开，然后按c：在当前session中创建一个新的window。
    (14) 按下Ctrl + a后手指松开，然后按w：选择其他window，操作方法与(12)完全相同。
    (15) 按下Ctrl + a后手指松开，然后按PageUp：翻阅当前pane内的内容。
    (16) 鼠标滚轮：翻阅当前pane内的内容。
    (17) 在tmux中选中文本时，需要按住shift键。（仅支持Windows和Linux，不支持Mac，不过该操作并不是必须的，因此影响不大）
    (18) tmux中复制/粘贴文本的通用方式：
        (1) 按下Ctrl + a后松开手指，然后按[
        (2) 用鼠标选中文本，被选中的文本会被自动复制到tmux的剪贴板
        (3) 按下Ctrl + a后松开手指，然后按]，会将剪贴板中的内容粘贴到光标处
```

vim教程

```
功能：
    (1) 命令行模式下的文本编辑器。
    (2) 根据文件扩展名自动判别编程语言。支持代码缩进、代码高亮等功能。
    (3) 使用方式：vim filename
        如果已有该文件，则打开它。
        如果没有该文件，则打开个一个新的文件，并命名为filename
模式：
    (1) 一般命令模式
        默认模式。命令输入方式：类似于打游戏放技能，按不同字符，即可进行不同操作。可以复制、粘贴、删除文本等。
    (2) 编辑模式
        在一般命令模式里按下i，会进入编辑模式。
        按下ESC会退出编辑模式，返回到一般命令模式。
    (3) 命令行模式
        在一般命令模式里按下:/?三个字母中的任意一个，会进入命令行模式。命令行在最下面。
        可以查找、替换、保存、退出、配置编辑器等。
操作：
    (1) i：进入编辑模式
    (2) ESC：进入一般命令模式
    (3) h 或 左箭头键：光标向左移动一个字符
    (4) j 或 向下箭头：光标向下移动一个字符
    (5) k 或 向上箭头：光标向上移动一个字符
    (6) l 或 向右箭头：光标向右移动一个字符
    (7) n<Space>：n表示数字，按下数字后再按空格，光标会向右移动这一行的n个字符
    (8) 0 或 功能键[Home]：光标移动到本行开头
    (9) $ 或 功能键[End]：光标移动到本行末尾
    (10) G：光标移动到最后一行
    (11) :n 或 nG：n为数字，光标移动到第n行
    (12) gg：光标移动到第一行，相当于1G
    (13) n<Enter>：n为数字，光标向下移动n行
    (14) /word：向光标之下寻找第一个值为word的字符串。
    (15) ?word：向光标之上寻找第一个值为word的字符串。
    (16) n：重复前一个查找操作
    (17) N：反向重复前一个查找操作
    (18) :n1,n2s/word1/word2/g：n1与n2为数字，在第n1行与n2行之间寻找word1这个字符串，并将该字符串替换为word2
    (19) :1,$s/word1/word2/g：将全文的word1替换为word2
    (20) :1,$s/word1/word2/gc：将全文的word1替换为word2，且在替换前要求用户确认。
    (21) v：选中文本
```

(22) d：删除选中的文本
(23) dd：删除当前行
(24) y：复制选中的文本
(25) yy：复制当前行
(26) p：将复制的数据在光标的下一行/下一个位置粘贴
(27) u：撤销
(28) Ctrl + r：取消撤销
(29) 大于号 >：将选中的文本整体向右缩进一次
(30) 小于号 <：将选中的文本整体向左缩进一次
(31) :w 保存
(32) :w! 强制保存
(33) :q 退出
(34) :q! 强制退出
(35) :wq 保存并退出
(36) :set paste 设置成粘贴模式，取消代码自动缩进
(37) :set nopaste 取消粘贴模式，开启代码自动缩进
(38) :set nu 显示行号
(39) :set nonu 隐藏行号
(40) gg=G：将全文代码格式化
(41) :noh 关闭查找关键词高亮
(42) Ctrl + q：当vim卡死时，可以取消当前正在执行的命令

异常处理：
每次用vim编辑文件时，会自动创建一个.filename.swp的临时文件。
如果打开某个文件时，该文件的swp文件已存在，则会报错。此时解决办法有两种：
(1) 找到正在打开该文件的程序，并退出
(2) 直接删掉该swp文件即可

shell基础

概论

shell是我们通过命令行与操作系统沟通的语言。

shell脚本可以直接在命令行中执行，也可以将一套逻辑组织成一个文件，方便复用。
Terminal中的命令行可以看成是一个“shell脚本在逐行执行”。

Linux中常见的shell脚本有很多种，常见的有：

Bourne Shell(/usr/bin/sh或/bin/sh)
Bourne Again Shell(/bin/bash)
C Shell(/usr/bin/csh)
K Shell(/usr/bin/ksh)
zsh
…

Linux系统中一般默认使用bash，所以接下来讲解bash中的语法。
文件开头需要写#! /bin/bash，指明bash为脚本解释器。

学习技巧

不要死记硬背，遇到含糊不清的地方，可以在Terminal里实际运行一遍

脚本示例

新建一个test.sh文件，内容如下：

```
#! /bin/bash
echo "Hello World!"
```

运行方式

作为可执行文件

```
acs@9e0ebfcd82d7:~$ chmod +x test.sh  # 使脚本具有可执行权限
acs@9e0ebfcd82d7:~$ ./test.sh  # 当前路径下执行
Hello World!  # 脚本输出
acs@9e0ebfcd82d7:~$ /home/acs/test.sh  # 绝对路径下执行
Hello World!  # 脚本输出
acs@9e0ebfcd82d7:~$ ~/test.sh  # 家目录路径下执行
Hello World!  # 脚本输出
```

用解释器执行

```
acs@9e0ebfcd82d7:~$ bash test.sh
Hello World!  # 脚本输出
```

注释

单行注释

每行中#之后的内容均是注释。

```
# 这是一行注释

echo 'Hello World'  #  这也是注释
```

多行注释

格式:

```
:<<EOF
第一行注释
第二行注释
第三行注释
EOF
```

其中**EOF**可以换成其它任意字符串。例如：

```
:<<abc
第一行注释
第二行注释
第三行注释
abc

:<<!
第一行注释
第二行注释
第三行注释
!
```

变量

定义变量

定义变量，不需要加**$**符号，例如：

```
name1='yxc'  # 单引号定义字符串
name2="yxc"  # 双引号定义字符串
name3=yxc    # 也可以不加引号，同样表示字符串
```

使用变量

使用变量，需要加上**$**符号，或者**${}**符号。花括号是可选的，主要为了帮助解释器识别变量边界。

```
name=yxc
echo $name  # 输出yxc
echo ${name}  # 输出yxc
echo ${name}acwing  # 输出yxcacwing
```

只读变量

使用**readonly**或者**declare**可以将变量变为只读。

```
name=yxc
readonly name
declare -r name  # 两种写法均可

name=abc  # 会报错，因为此时name只读
```

删除变量

unset可以删除变量

```
name=yxc
unset name
echo $name  # 输出空行
```

变量类型

1.自定义变量（局部变量）
  子进程不能访问的变量
2.环境变量（全局变量）
  子进程可以访问的变量

自定义变量改成环境变量：

```
acs@9e0ebfcd82d7:~$ name=yxc  # 定义变量
acs@9e0ebfcd82d7:~$ export name  # 第一种方法
acs@9e0ebfcd82d7:~$ declare -x name  # 第二种方法
```

环境变量改为自定义变量：

```
acs@9e0ebfcd82d7:~$ export name=yxc  # 定义环境变量
acs@9e0ebfcd82d7:~$ declare +x name  # 改为自定义变量
```

字符串

字符串可以用单引号，也可以用双引号，也可以不用引号。

单引号与双引号的区别：

- 单引号中的内容会原样输出，不会执行、不会取变量；
- 双引号中的内容可以执行、可以取变量；

```
name=yxc  # 不用引号
echo 'hello, $name \"hh\"'  # 单引号字符串，输出 hello, $name \"hh\"
echo "hello, $name \"hh\""  # 双引号字符串，输出 hello, yxc "hh"
```

获取字符串长度

```
name="yxc"
echo ${#name}  # 输出3
```

提取子串

```
name="hello, yxc"
echo ${name:0:5}  # 提取从0开始的5个字符
```

笔记小结

```
1、定义变量时，等号两边不能有空格
2、定义变量的时候变量都是字符串，但当变量需要是整数时，会自动把变量转换成整数
3、type+命令可以解释该命令的来源（内嵌命令。第三方命令等）
如type readonly #readonly is a shell builtin(shell内部命令)
type ls # ls is aliased to 'ls -color+auto'
4、被声明为只读的变量无法被unset删除
5、bash可以用来开一个新的进程，exit或Ctrl+d退出新的bash
6、字符串中，不加引号和双引号效果相同
```

默认变量

文件参数变量

在执行shell脚本时，可以向脚本传递参数。**$1**是第一个参数，**$2**是第二个参数，以此类推。特殊的，**$0**是文件名（包含路径）。例如：

创建文件**test.sh**：

```
#! /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
echo "第四个参数："$4
```

然后执行该脚本：

```
acs@9e0ebfcd82d7:~$ chmod +x test.sh
acs@9e0ebfcd82d7:~$ ./test.sh 1 2 3 4
文件名：./test.sh
第一个参数：1
第二个参数：2
第三个参数：3
第四个参数：4
```

其它参数相关变量

参数	说明
`$#`	代表文件传入的参数个数，如上例中值为4
`$*`	由所有参数构成的用空格隔开的字符串，如上例中值为 `"$1 $2 $3 $4"`
`$@`	每个参数分别用双引号括起来的字符串，如上例中值为 `"$1" "$2" "$3" "$4"`
`$$`	脚本当前运行的进程ID
`$?`	上一条命令的退出状态（注意不是stdout，而是exit code）。0表示正常退出，其他值表示错误
`$(command)`	返回 `command` 这条命令的stdout（可嵌套）
`command`	返回 `command` 这条命令的stdout（不可嵌套）

数组

数组中可以存放多个不同类型的值，只支持一维数组，初始化时不需要指明数组大小。
数组下标从0开始。

定义

数组用小括号表示，元素之间用空格隔开。例如：

```
array=(1 abc "def" yxc)
```

也可以直接定义数组中某个元素的值：

```
array[0]=1
array[1]=abc
array[2]="def"
array[3]=yxc
```

读取数组中某个元素的值

格式：

```
${array[index]}
```

例如:

```
array=(1 abc "def" yxc)
echo ${array[0]}
echo ${array[1]}
echo ${array[2]}
echo ${array[3]}
```

读取整个数组

格式:

```
${array[@]}  # 第一种写法
${array[*]}  # 第二种写法
```

例如:

```
array=(1 abc "def" yxc)

echo ${array[@]}  # 第一种写法
echo ${array[*]}  # 第二种写法
```

数组长度

类似字符串

```
${#array[@]}  # 第一种写法
${#array[*]}  # 第二种写法
```

例如:

```
array=(1 abc "def" yxc)

echo ${#array[@]}  # 第一种写法
echo ${#array[*]}  # 第二种写法
```

expr命令

expr命令用于求表达式的值，格式为：

```
expr 表达式
```

表达式说明：

- 用空格隔开每一项
- 用反斜杠放在shell特定的字符前面（发现表达式运行错误时，可以试试转义）
- 对包含空格和其他特殊字符的字符串要用引号括起来
- expr会在stdout中输出结果。如果为逻辑关系表达式，则结果为真，stdout为1，否则为0。
- expr的exit code：如果为逻辑关系表达式，则结果为真，exit code为0，否则为1。

字符串表达式

- length STRING
  返回STRING的长度
- index STRING CHARSET
  CHARSET中任意单个字符在STRING中最前面的字符位置，下标从1开始。如果在STRING中完全不存在CHARSET中的字符，则返回0。
- substr STRING POSITION LENGTH
  返回STRING字符串中从POSITION开始，长度最大为LENGTH的子串。如果POSITION或LENGTH为负数，0或非数值，则返回空字符串。

示例:

```
str="Hello World!"

echo `expr length "$str"`  # ``不是单引号，表示执行该命令，输出12
echo `expr index "$str" aWd`  # 输出7，下标从1开始
echo `expr substr "$str" 2 3`  # 输出 ell
```

整数表达式

expr支持普通的算术操作，算术表达式优先级低于字符串表达式，高于逻辑关系表达式。

- -
  加减运算。两端参数会转换为整数，如果转换失败则报错。
- / %
  乘，除，取模运算。两端参数会转换为整数，如果转换失败则报错。
- () 可以该表优先级，但需要用反斜杠转义

```
a=3
b=4

echo `expr $a + $b`  # 输出7
echo `expr $a - $b`  # 输出-1
echo `expr $a \* $b`  # 输出12，*需要转义
echo `expr $a / $b`  # 输出0，整除
echo `expr $a % $b` # 输出3
echo `expr \( $a + 1 \) \* \( $b + 1 \)`  # 输出20，值为(a + 1) * (b + 1)
```

逻辑关系表达式

- |
  如果第一个参数非空且非0，则返回第一个参数的值，否则返回第二个参数的值，但要求第二个参数的值也是非空或非0，否则返回0。如果第一个参数是非空或非0时，不会计算第二个参数。
- &
  如果两个参数都非空且非0，则返回第一个参数，否则返回0。如果第一个参为0或为空，则不会计算第二个参数。
- < <= = == != >= >
  比较两端的参数，如果为true，则返回1，否则返回0。”==”是”=”的同义词。”expr”首先尝试将两端参数转换为整数，并做算术比较，如果转换失败，则按字符集排序规则做字符比较。
- () 可以该表优先级，但需要用反斜杠转义

示例:

```
a=3
b=4

echo `expr $a \> $b`  # 输出0，>需要转义
echo `expr $a '<' $b`  # 输出1，也可以将特殊字符用引号引起来
echo `expr $a '>=' $b`  # 输出0
echo `expr $a \<\= $b`  # 输出1

c=0
d=5

echo `expr $c \& $d`  # 输出0
echo `expr $a \& $b`  # 输出3
echo `expr $c \| $d`  # 输出5
echo `expr $a \| $b`  # 输出3
```

read命令

read命令用于从标准输入中读取单行数据。当读到文件结束符时，**exit code**为1，否则为0。

参数说明

- -p: 后面可以接提示信息
- -t：后面跟秒数，定义输入字符的等待时间，超过等待时间后会自动忽略此命令

实例:

```
acs@9e0ebfcd82d7:~$ read name  # 读入name的值
acwing yxc  # 标准输入
acs@9e0ebfcd82d7:~$ echo $name  # 输出name的值
acwing yxc  #标准输出
acs@9e0ebfcd82d7:~$ read -p "Please input your name: " -t 30 name  # 读入name的值，等待时间30秒
Please input your name: acwing yxc  # 标准输入
acs@9e0ebfcd82d7:~$ echo $name  # 输出name的值
acwing yxc  # 标准输出
```

echo命令

echo用于输出字符串。命令格式：

```
echo STRING
```

显示普通字符串

```
echo "Hello AC Terminal"
echo Hello AC Terminal  # 引号可以省略
```

显示转义字符

```
echo "\"Hello AC Terminal\""  # 注意只能使用双引号，如果使用单引号，则不转义
echo \"Hello AC Terminal\"  # 也可以省略双引号
```

显示变量

```
name=yxc
echo "My name is $name"  # 输出 My name is yxc
```

显示换行

```
echo -e "Hi\n"  # -e 开启转义
echo "acwing"
```

输出结果:

```
Hi

acwing
```

显示不换行

```
echo -e "Hi \c" # -e 开启转义 \c 不换行
echo "acwing"
```

输出结果:

```
Hi acwing
```

显示结果定向至文件

```
echo "Hello World" > output.txt  # 将内容以覆盖的方式输出到output.txt中
```

原样输出字符串，不进行转义或取变量(用单引号)

```
name=acwing
echo '$name\"'
```

输出结果:

```
$name\"
```

显示命令的执行结果

```
echo `date`
```

输出结果:

```
Wed Sep 1 11:45:33 CST 2021
```

printf命令

printf命令用于格式化输出，类似于C/C++中的printf函数。

默认**不会在字符串末尾添加换行符。**

命令格式:

```
printf format-string [arguments...]
```

用法示例

脚本内容:

```
printf "%10d.\n" 123  # 占10位，右对齐
printf "%-10.2f.\n" 123.123321  # 占10位，保留2位小数，左对齐
printf "My name is %s\n" "yxc"  # 格式化输出字符串
printf "%d * %d = %d\n"  2 3 `expr 2 \* 3` # 表达式的值作为参数
```

输出结果:

```
       123.
123.12    .
My name is yxc
2 * 3 = 6
```

test命令与判断符号[]

逻辑运算符&&和||

- && 表示与，|| 表示或
- 二者具有短路原则：
  expr1 && expr2：当expr1为假时，直接忽略expr2
  expr1 || expr2：当expr1为真时，直接忽略expr2
- 表达式的exit code为0，表示真；为非零，表示假。（与C/C++中的定义相反）

test命令

在命令行中输入man test，可以查看test命令的用法。

test命令用于判断文件类型，以及对变量做比较。

test命令用exit code返回结果，而不是使用stdout。0表示真，非0表示假。

例如：

```
test 2 -lt 3  # 为真，返回值为0
echo $?  # 输出上个命令的返回值，输出0
```

```
acs@9e0ebfcd82d7:~$ ls  # 列出当前目录下的所有文件
homework  output.txt  test.sh  tmp
acs@9e0ebfcd82d7:~$ test -e test.sh && echo "exist" || echo "Not exist"
exist  # test.sh 文件存在
acs@9e0ebfcd82d7:~$ test -e test2.sh && echo "exist" || echo "Not exist"
Not exist  # testh2.sh 文件不存在
```

文件类型判断

命令格式:

```
test -e filename  # 判断文件是否存在
```

测试参数代表意义
-e 文件是否存在
-f 是否为文件
-d 是否为目录

文件权限判断

命令格式：

```
test -r filename  # 判断文件是否可读
```

测试参数代表意义
-r 文件是否可读
-w 文件是否可写
-x 文件是否可执行
-s 是否为非空文件

整数间的比较

命令格式：

```
test $a -eq $b  # a是否等于b
```

测试参数代表意义
-eq a是否等于b
-ne a是否不等于b
-gt a是否大于b
-lt a是否小于b
-ge a是否大于等于b
-le a是否小于等于b

字符串比较

测试参数代表意义
test -z STRING 判断STRING是否为空，如果为空，则返回true
test -n STRING 判断STRING是否非空，如果非空，则返回true（-n可以省略）
test str1 == str2 判断str1是否等于str2
test str1 != str2 判断str1是否不等于str2

多重条件判定

命令格式：

```
test -r filename -a -x filename
```

测试参数代表意义
-a 两条件是否同时成立
-o 两条件是否至少一个成立
! 取反。如 test ! -x file，当file不可执行时，返回true

判断符号[]

[]与test用法几乎一模一样，更常用于if语句中。另外[[]]是[]的加强版，支持的特性更多。

例如：

```
[ 2 -lt 3 ]  # 为真，返回值为0
echo $?  # 输出上个命令的返回值，输出0
```

```
acs@9e0ebfcd82d7:~$ ls  # 列出当前目录下的所有文件
homework  output.txt  test.sh  tmp
acs@9e0ebfcd82d7:~$ [ -e test.sh ] && echo "exist" || echo "Not exist"
exist  # test.sh 文件存在
acs@9e0ebfcd82d7:~$ [ -e test2.sh ] && echo "exist" || echo "Not exist"
Not exist  # testh2.sh 文件不存在
```

注意：

- []内的每一项都要用空格隔开
- 中括号内的变量，最好用双引号括起来
- 中括号内的常数，最好用单或双引号括起来

例如：

```
name="acwing yxc"
[ $name == "acwing yxc" ]  # 错误，等价于 [ acwing yxc == "acwing yxc" ]，参数太多
[ "$name" == "acwing yxc" ]  # 正确
```

判断语句

if...then形式
类似于C/C++中的if-else语句。

单层if

命令格式：

```
if condition
then
    语句1
    语句2
    ...
fi
```

示例：

```
a=3
b=4

if [ "$a" -lt "$b" ] && [ "$a" -gt 2 ]
then
    echo ${a}在范围内
fi
```

输出结果：

```
3在范围内
```

单层if-else

命令格式

```
if condition
then
    语句1
    语句2
    ...
else
    语句1
    语句2
    ...
fi
```

示例:

```
a=3
b=4

if ! [ "$a" -lt "$b" ]
then
    echo ${a}不小于${b}
else
    echo ${a}小于${b}
fi
```

输出结果：

```
3小于4
```

多层if-elif-elif-else

命令格式

```
if condition
then
    语句1
    语句2
```

```
    ...
elif condition
then
    语句1
    语句2
    ...
elif condition
then
    语句1
    语句2
else
    语句1
    语句2
    ...
fi
```

示例：

```
a=4

if [ $a -eq 1 ]
then
    echo ${a}等于1
elif [ $a -eq 2 ]
then
    echo ${a}等于2
elif [ $a -eq 3 ]
then
    echo ${a}等于3
else
    echo 其他
fi
```

输出结果：

```
其他
```

case...esac形式

类似于C/C++中的switch语句。

命令格式

```
case $变量名称 in
    值1)
        语句1
        语句2
        ...
        ;;  # 类似于C/C++中的break
    值2)
        语句1
        语句2
        ...
        ;;
    *)  # 类似于C/C++中的default
```

```
        语句1
        语句2
        ...
        ;;
esac
```

示例：

```
其他
```

循环语句

for...in...do...done

命令格式:

```
for var in val1 val2 val3
do
    语句1
    语句2
    ...
done
```

示例1，输出a 2 cc，每个元素一行：

```
for i in a 2 cc
do
    echo $i
done
```

示例2，输出当前路径下的所有文件名，每个文件名一行：

```
for file in `ls`
do
    echo $file
done
```

示例3，输出1-10

```
for i in $(seq 1 10)
do
    echo $i
done
```

示例4，使用{1..10} 或者 {a..z}

```
for i in {a..z}
do
    echo $i
done
```

for ((...;...;...)) do...done

命令格式:

```
for ((expression; condition; expression))
do
    语句1
    语句2
done
```

示例，输出1-10，每个数占一行:

```
for ((i=1; i<=10; i++))
do
    echo $i
done
```

while...do...done循环

命令格式:

```
while condition
do
    语句1
    语句2
    ...
done
```

示例，文件结束符为Ctrl+d，输入文件结束符后read指令返回false。

```
while read name
do
    echo $name
done
```

until...do...done循环

当条件为真时结束。

命令格式:

```
until condition
do
    语句1
    语句2
    ...
done
```

示例，当用户输入yes或者YES时结束，否则一直等待读入。

```
until [ "${word}" == "yes" ] || [ "${word}" == "YES" ]
do
    read -p "Please input yes/YES to stop this program: " word
done
```

break命令

跳出当前一层循环，注意与C/C++不同的是：break不能跳出case语句。

示例

```
while read name
do
    for ((i=1;i<=10;i++))
    do
        case $i in
            8)
                break
                ;;
            *)
                echo $i
                ;;
        esac
    done
done
```

该示例每读入非EOF的字符串，会输出一遍1-7。
该程序可以输入Ctrl+d文件结束符来结束，也可以直接用Ctrl+c杀掉该进程。

continue命令

跳出当前循环。

示例：

```
for ((i=1;i<=10;i++))
do
    if [ `expr $i % 2` -eq 0 ]
    then
        continue
    fi
    echo $i
done
```

该程序输出1-10中的所有奇数。

死循环的处理方式

如果Terminal可以打开该程序，则输入Ctrl+c即可。

否则可以直接关闭进程：

1.使用top命令找到进程的PID
2.输入kill -9 PID即可关掉此进程

函数

bash中的函数类似于C/C++中的函数，但return的返回值与C/C++不同，返回的是exit code，取值为0-255，0表示正常结束。

如果想获取函数的输出结果，可以通过echo输出到stdout中，然后通过$(function_name)来获取stdout中的结果。

函数的return值可以通过$?来获取。

命令格式：

```
[function] func_name() {  # function关键字可以省略
    语句1
    语句2
    ...
}
```

不获取 return值和stdout值

示例

```
func() {
    name=yxc
    echo "Hello $name"
}

func
```

输出结果：

```
Hello yxc
```

获取 return值和stdout值

不写return时，默认return 0。

示例

```
func() {
    name=yxc
    echo "Hello $name"

    return 123
}

output=$(func)
ret=$?

echo "output = $output"
echo "return = $ret"
```

输出结果：

```
output = Hello yxc
return = 123
```

函数的输入参数

在函数内，**$1**表示第一个输入参数，**$2**表示第二个输入参数，依此类推。

注意：函数内的**$0**仍然是文件名，而不是函数名。

示例：

```
func() {  # 递归计算 $1 + ($1 - 1) + ($1 - 2) + ... + 0
    word=""
    while [ "${word}" != 'y' ] && [ "${word}" != 'n' ]
    do
        read -p "要进入func($1)函数吗？请输入y/n：" word
    done

    if [ "$word" == 'n' ]
    then
        echo 0
        return 0
    fi

    if [ $1 -le 0 ]
    then
        echo 0
        return 0
    fi

    sum=$(func $(expr $1 - 1))
    echo $(expr $sum + $1)
}

echo $(func 10)
```

输出结果：

```
55
```

函数内的局部变量

可以在函数内定义局部变量，作用范围仅在当前函数内。

可以在递归函数中定义局部变量。

命令格式：

```
local 变量名=变量值
```

例如：

```
#! /bin/bash

func() {
    local name=yxc
    echo $name
}
func

echo $name
```

输出结果：

```
yxc
```

第一行为函数内的name变量，第二行为函数外调用name变量，会发现此时该变量不存在。

exit命令

exit命令用来退出当前shell进程，并返回一个退出状态；使用$?可以接收这个退出状态。

exit命令可以接受一个整数值作为参数，代表退出状态。如果不指定，默认状态值是 0。

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。

示例：

创建脚本test.sh，内容如下：

```
#! /bin/bash

if [ $# -ne 1 ]  # 如果传入参数个数等于1，则正常退出；否则非正常退出。
then
    echo "arguments not valid"
    exit 1
else
    echo "arguments valid"
    exit 0
fi
```

执行该脚本：

```
acs@9e0ebfcd82d7:~$ chmod +x test.sh
acs@9e0ebfcd82d7:~$ ./test.sh acwing
arguments valid
acs@9e0ebfcd82d7:~$ echo $?  # 传入一个参数，则正常退出，exit code为0
0
acs@9e0ebfcd82d7:~$ ./test.sh
arguments not valid
acs@9e0ebfcd82d7:~$ echo $?  # 传入参数个数不是1，则非正常退出，exit code为1
1
```

文件重定向

每个进程默认打开3个文件描述符：

- stdin标准输入，从命令行读取数据，文件描述符为0
- stdout标准输出，向命令行输出数据，文件描述符为1
- stderr标准错误输出，向命令行输出数据，文件描述符为2

可以用文件重定向将这三个文件重定向到其他文件中。

重定向命令列表

命令	说明
`command > file`	将 `stdout` 重定向到 `file` 中
`command < file`	将 `stdin` 重定向到 `file` 中
`command >> file`	将 `stdout` 以追加方式重定向到 `file` 中
`command n> file`	将文件描述符 `n` 重定向到 `file` 中
`command n>> file`	将文件描述符 `n` 以追加方式重定向到 `file` 中

输入和输出重定向

```
echo -e "Hello \c" > output.txt  # 将stdout重定向到output.txt中
echo "World" >> output.txt  # 将字符串追加到output.txt中

read str < output.txt  # 从output.txt中读取字符串

echo $str  # 输出结果：Hello World
```

同时重定向stdin和stdout

创建bash脚本：

```
#! /bin/bash

read a
read b

echo $(expr "$a" + "$b")
```

创建input.txt，里面的内容为：

```
3
4
```

执行命令：

```
acs@9e0ebfcd82d7:~$ chmod +x test.sh  # 添加可执行权限
acs@9e0ebfcd82d7:~$ ./test.sh < input.txt > output.txt  # 从input.txt中读取内容，将输出写入output.txt中
acs@9e0ebfcd82d7:~$ cat output.txt  # 查看output.txt中的内容
7
```

引入外部脚本

类似于C/C++中的include操作，bash也可以引入其他文件中的代码。

语法格式：

```
. filename  # 注意点和文件名之间有一个空格

或

source filename
```

示例
创建test1.sh，内容为：

```
#! /bin/bash

name=yxc  # 定义变量name
```

然后创建test2.sh，内容为：

```
#! /bin/bash

source test1.sh # 或 . test1.sh

echo My name is: $name  # 可以使用test1.sh中的变量
```

执行命令：

```
acs@9e0ebfcd82d7:~$ chmod +x test2.sh
acs@9e0ebfcd82d7:~$ ./test2.sh
My name is: yxc
```

补充内容：linux数据处理

简述

1. grep 命令：筛选数据
2. sort 命令：为文件排序
3. wc 命令：文件的统计
4. uniq 命令：删除文件中的重复内容
5. cut 命令：剪切文件的一部分内容
6.

1. grep 命令：筛选数据

grep 是 Globally search a Regular Expression and Print 的缩写，意思是"全局搜索一个正则表达式，并且打印"。

grep 命令的功能简单说来是在文件中查找关键字，并且显示关键字所在的行。

grep 命令极为强大，也是 Linux 中使用最多的命令之一。它的强大之处在于它不仅可以实现简单的查找，而且可以配合 正则表达式 来实现比较复杂的查找。

grep 的简单用法

grep 的使用方法有很多种，我们一开始先学习最基本的用法：

```
grep text file
```

可以看到，上面就是 grep 命令的最基本用法。

text 代表要搜索的文本，file 代表供搜索的文件。

我们用实际的例子来学习：比如我要在用户的家目录的 .bashrc 文件中搜索 alias 这个文本，而且显示所有包含alias的行。

```
grep alias .bashrc
```

如果我们要用 grep 命令在一个文件中查找用空格隔开的文本，那么就要加上双引号，例如：

```
grep "Hello World" file2
```

如果我们要用 grep 命令在一个文件中查找用空格隔开的文本，那么就要加上双引号，例如：

```
grep "Hello World" file2
```

-i 参数：忽略大小写

默认的情况下，grep 命令是区分大小写的，也就是说搜索的文本将严格按照大小写来搜索。比如我搜索的文本是 text，那么就不会搜出 Text，tExt，TEXT 等等文本。

但是我们可以给 grep 加上 -i 参数，使得 grep 可以忽略大小写。i 是英语 ignore 的缩写，表示“忽略”。

例如：

```
grep -i alias .bashrc
```

-n 参数：显示行号

-n 参数的作用很简单，就是显示搜索到的文本所在的行号。n 是英语 number 的缩写，表示“数字，编号”。

```
grep -n alias .bashrc
```

-v 参数：只显示文本不在的行

-v 参数很有意思，v 是 invert 的缩写，表示“颠倒，倒置”。-v 参数的作用与正常 grep 的作用正好颠倒，就是只显示搜索的文本不在的那些行。

-r 参数：在所有子目录和子文件中查找

如果你不知道你要找的文本在哪个文件里，你可以用强大的 -r 参数。

r 是英语 recursive 的缩写，表示“递归”。

如果用了 -r 参数，那么 grep 命令使用时的最后一个参数（grep text file 这个模式中的 file）需要换成 directory，也就是必须是一个目录。因为 -r 参数是让 grep 命令能够在指定目录的所有子目录和子文件中查找文本。

例如：

```
grep -r "Hello World" folder/
```

表示在 folder 这个目录的所有子目录和子文件中查找 Hello World 这个文本。当然了，以上例子中，folder 后面的斜杠（/）不是必须的，这里只是为了清楚表明 folder 是一个目录。只要 folder 是一个目录，Linux 系统是不会搞错的。

Linux 中还有一个 rgrep 的命令，它的作用相当于 grep -r 。

grep 的高级用法：配合正则表达式

正则表达式使用单个字符串来描述、匹配一系列符合某个句法规则的字符串。

grep 配合正则表达式就可以实现比较高级的搜索了。

我们首先来看一眼以下的这个表格，表格中列出了最常用的一些正则表达式的字符以及其含义：

特殊字符	含义
.	匹配除 “\n” 之外的任何单个字符
^	行首（匹配输入字符串的开始位置）
$	行尾（匹配输入字符串的结束位置）
[]	在中括号中的任意一个字符
?	问号前面的元素出现零次或一次
*	星号前面的元素可能出现零次、一次或多次
+	加号前面的元素必须出现一次以上（包含一次）
|	逻辑或
()	表达式的分组（表示范围和优先度）

当然了，上表没有列出所有的正则表达式的字符。

首先，为了让 grep 命令知道我们要使用正则表达式，须要加上 -E 参数（E 是 extended regular expression 的第一个字母，表示“扩展的正则表达式”）。例如：

```
grep -E Alias .bashrc
```

当然了，Linux 也有一个命令 egrep，其效果等同 grep -E 。

到此为止，没什么新鲜的。我们用正则表达式只是和之前的搜索类似。接下来，我们才真的要用到正则表达式的特殊字符了。

首先来看这个例子：

```
grep -E ^alias .bashrc
```

这个例子中，我们用到了 ^ 这个特殊符号，上面的表格里对于 ^ 已经做了说明：行首（匹配输入字符串的开始位置）。也就是说，^ 后面的字符须要出现在一行的开始。

再来举几个例子：

```
grep -E [Aa]lias .bashrc
```

[] 的作用，是将 [] 中的字符任取其一，因此 [Aa]lias 的意思就是既可以是 Alias，又可以是 alias。因此 grep 的搜索结果把包含 Alias 和 alias 的行都列出来了。

再比如：

```
grep -E [0-4] .bashrc
```

用于搜索包含 0 至 4 的任一数字的行。

```
grep -E [a-zA-Z] .bashrc
```

用于搜索包含在 a 至 z 之间的任意字母或者 A 至 Z 之间的任意字母的行。

其他正则表达式还有很多例子。就不一一列举了。

2. sort 命令：为文件排序

我们用 sort 命令来举个例子：

```
sort name.txt
```

sort 命令将 name.txt 文件中的行按照首字母的英文字典顺序进行了排列。

sort 命令并不区分大小写，小写字母开头的 jude 还是排在 John 之后。

-o 参数：将排序后的内容写入新文件

如果你打开 name.txt 文件，你会发现，经过了 sort 命令的“洗礼”，name.txt 中的内容还是维持原来的顺序。

单独使用 sort 命令是不会真正改变文件内容的，只是把排序结果显示在终端上。

那我们要存储排序结果到新的文件怎么办呢？可以用 -o 参数。

o 是 output 的首字母，表示“输出”，就是将排序结果输出到文件中。

```
sort -o name_sorted.txt name.txt
```

name.txt 经过 sort 命令排序之后的内容被储存在了新的文件 name_sorted.txt 中，而 name.txt 的内容是不变的。

-r 参数：倒序排列

-r 参数中的 r 是 reverse 的缩写，是“相反”的意思，与普通的仅用 sort 命令正好相反。

```
sort -r name.txt
```

-R 参数：随机排序

R 是英语 random 的首字母，表示“随机的，任意的”。

-R 参数比较“无厘头”，因为它会让 sort 命令的排序变为随机，就是任意排序，也许每次都不一样。

但在有些时候，-R 参数还是很有用的。

```
sort -R name.txt
```

-n 参数：对数字排序

对数字的排序有点特殊。默认仅用 sort 命令的时候，是不区分字符是否是数字的，会把这些数字看成字符串，按照 1-9 的顺序来排序。例如 138 会排在 25 前面，因为 1 排在 2 的前面。

那如果我们要 sort 命令识别整个数字，比如按照整个数值的大小顺序来说，25 应该排在 138 前面，那该怎么办呢？

就可以请出我们的 -n 参数了。n 是 number 的缩写。是英语“数字”的意思。-n 参数用于对数字进行排序，按从小到大排序。

为了演示，我们再用文本编辑器来创建一个文件，就叫 number.txt 好了。

里面随便填一些数字，每行一个：

```
12
9
216
28
174
35
68
```

然后用 sort 不加 -n 参数和加上 -n 参数分别测试：

```
12
174
216
28
35
68
9
--------------------------分割线
9
12
28
35
68
174
216
```

可以看到，不加 -n 参数时，sort 就会把这些数字看成字符串，按字符依次来排序，按照 1-9 的顺序。

加上 -n 参数，就会把各行的数字看成一个整体，按照大小从小到大来排序了。

3. wc 命令：文件的统计

wc 是 word count 的缩写

wc 命令看起来是用来统计单词数目的，但其实 wc 的功能不仅止于此。wc 命令还可以用来统计行数，字符数，字节数等。

跟前面的命令一样，wc 命令的用法也是后接文件名。wc 命令很有用，应该会成为你常用的命令之一。

如果不加选项参数，那么 wc 命令的返回值会有些特殊，有点晦涩难懂。

例如：

```
wc name.txt
```

可以看到返回值是

```
9 9 50 name.txt
```

最后的 name.txt 只是表示文件名，不需考虑。

那么这三个数字：9，9，和 50 分别表示什么呢?

这三个数字，按顺序，分别表示：

- 行数（newline counts）：newline 是英语“换行、换行符”的意思。统计行数其实就是统计换行符的数目。
- 单词数（word counts）
- 字节数（byte counts）：byte 是英语“字节”的意思，等于 8 个二进制位（bit）。

可以用 man wc 查看 wc 的命令手册得知：

wc 的命令描述是“print newline, word, and byte counts for each file”，翻成中文就是“对每个文件，打印其行数，单词数和字节数”。

因为我们之前创建 name.txt 时，每一行只有一个单词（英语名字），所以这里统计的行数和单词数都是 9。

50 代表字节数。我数了一下，name.txt 里的 9 个英语单词一共包含 41 个英语字母（也就是 41 个英语字符），占用 41 个字节。再加上每行结尾的换行符（Linux 中换行符是 ‘\n’），共有 9 个换行符，占用 9 个字节。41 + 9 = 50，正好是 50 个字节。

我们稍微讲一下字符和字节的一些联系和区别：

- 字节（Byte 或 Octet）是计量单位，表示数据量多少，是计算机存储容量的计量单位。一个字节等于 8 位（Bit，比特位，是计算机最小的存储单位。就是 0 或 1 这样的二进制位）。
- 字符（Character）是计算机中使用的文字和符号，比如 “a”、“B”、“7”、“&”、“%”等。不同语言有不同的字符，一般我们中国人接触比较多的是英语和中文的字符。

字符在不同的编码中所占字节数是不一样的。字符的编码和标准有不少，这里我们就不深入展开了，大家可以看这个链接来深入了解：字符集 。

-l 参数：统计行数

为了只统计行数，我们可以加上 -l 参数。l 是 line 的缩写，表示“行”。

```
wc -l name.txt
```

-w 参数：统计单词数

w 是word 的缩写，表示“单词”。因此 -w 参数用于统计单词。

```
wc -w name.txt
```

-c 参数：统计字节数

不知道为什么是 c，因为 byte 或者 octet（都表示“字节”）的首字母都不是 c 啊。也许 c 是 character（英语“字符”的意思）的缩写吧。

```
wc -c name.txt
```

-m 参数：统计字符数

不知道为什么是 m，因为 character（英语“字符”）的首字母不是 m：

```
wc -m name.txt
```

为了加深理解，我们来测试一下。创建一个只包含中文字符的文本文件，可以起名叫 chinese.txt（chinese 是“中文”的意思）。在里面写入：

```
你好吗
我很好
```

这 6 个汉字。

我们用 wc 命令来统计一下 chinese.txt 的字节数和字符数：

```
wc -c chinese.txt
wc -m chinese.txt
```

chinese.txt 包含的字节数是 20，字符数是 8

其实这是因为使用的是 Unicode 标准 的 UTF-8 编码方式。中文字符占 3 个字节，一共有 6 个中文字符，6 * 3 = 18，再加上 2 个换行符占 2 个字节，18 + 2 = 20。

字符数则是 6 + 2 = 8 个

我们可以用 file 命令来确定文件的类型，运行：

```
file chinese.txt
file name.txt
```

chinese.txt 的编码是 UTF-8 Unicode，name.txt 的编码是 ASCII

4. uniq 命令：删除文件中的重复内容

有时候，文件中包含重复的行，我们想要将重复的内容删除，

这时，uniq 命令就显得很有用了。

uniq 是英语 unique 的缩写，表示“独一无二的”。

为了演示，我们创建一个文件 repeat.txt（repeat 是英语“重复”的意思），里面写入如下排序好的内容（因为 uniq 命令有点“呆”，只能将连续的重复行变为一行）：

```
Albert
China
France
France
France
John
Matthew
Matthew
patrick
Steve
Vincent
```

可以看到，有三个 France 连在一起，两个 Matthew 连在一起。

我们用 uniq 命令来处理看看：

```
uniq repeat.txt
```

可以看到，三个连续的 France 只剩下一个了，两个连续的 Matthew 也只剩一个了。

和 sort 命令类似，uniq 命令并不会改变原文件的内容，只会把处理后的内容显示出来。

如果想将处理后的内容储存到一个新文件中，可以使用如下的方法：

```
uniq repeat.txt unique.txt
```

-c 参数：统计重复的行数

-c 参数用于显示重复的行数，如果是独一无二的行，那么数目就是 1。c 是 count 的缩写，表示“统计，计数”。

```
uniq -c repeat.txt
```

-d 参数：只显示重复行的值

-d 参数只显示重复的行的值。d 是 duplicated 的缩写，表示“重复的”。

```
uniq -d repeat.txt
```

5. cut 命令：剪切文件的一部分内容

cut 是英语“剪切”的意思。大家平时肯定有剪切文本内容的经历吧，一般剪切之后还会把剪切的内容粘贴到某处。

cut 命令用于对文件的每一行进行剪切处理。

-c 参数：根据字符数来剪切

c 是 character 的缩写，表示“字符”。

比如，我们要 name.txt 的每一行只保留第 2 至第 4 个字符。可以这样做：

```
cut -c 2-4 name.txt
```

小结

1. grep 命令应该算是在文件中查找关键字最常用的工具了。
2. grep 命令可以通过正则表达式来查找。一开始正则表达式会比较难记，但是功能很强大。我们可以调用 egrep 命令，其等价于 grep -E。
3. sort 命令用于为文件中的行按字母顺序排序。使用 -n 参数可以按照数字顺序排序。
4. wc 命令可以统计文件中行数，单词数或者字节数。
5. uniq 命令可以用于删除文件中重复的内容。
6. cut 命令用于剪切文件的一部分内容。