BASH进阶

|  |  |
| --- | --- |
| cat | 用cat -A可以发现在DOS中使用的断行符号位^M$, 我们称为CR和LF两个符号；在Linux下面，则是仅有LF($)这个断行符号。 |
| echo | 特殊参数：  **-n** do not output the trailing newline  -e enable interpretation of backslash escapes  -E disable interpretation of backslash escapes (default) |
| **sort** | -b, --ignore-leading-blanks ignore leading blanks  -d, --dictionary-order consider only blanks and alphanumeric characters  -f, --ignore-case fold lower case to upper case characters  -g, --general-numeric-sort compare according to general numerical value  -i, --ignore-nonprinting consider only printable characters  -M, --month-sort compare (unknown) < `JAN' < ... < `DEC'  -h, --human-numeric-sort compare human readable numbers (e.g., 2K 1G)  -n, --numeric-sort compare according to string numerical value (default is alphabetic sorting)  **-r**, --reverse reverse the result of comparisons  -V, --version-sort natural sort of (version) numbers within text  -t, --field-separator=SEP use SEP instead of non-blank to blank transition  -k, --key=POS1[,POS2] start a key at POS1 (origin 1), end it at POS2   (default end of line).  -u, --unique with -c, check for strict ordering;   **without -c, output only the first of an equal run**  **例子：**  cat /etc/passwd | sort -t ‘:’ -k 3 -n |
| type | 查看命令的类型（内置命令还是外部命令）  type -a ls类似于which ls |
| 反斜杠 “\” | 反斜杠 “\”作为命令连接符时，**后面必须紧接着[Enter]按键**，否则会出现奇怪的结果。 |
| 变量 | 1. 变量与变量内容以一个等号来连接；  2. 等号两边不能直接接空格符，若变量内容有空格符，可以使用单引号或双引号括起来，同时，若双引号内的特殊字符如$等，可以保持原本特性，而单引号内的特殊字符则仅为一般字符；  3. 可以使用转义字符 “\”, 将特殊符号变成一般字符；  4. 在一串命令中，若还需要其他的命令提供的信息，可以使用反单引号 **“`命令`” 或“$(命令)”**;  5. 取消变量的方法是使用 “unset 变量名称”；  6. 变量类型默认是字符串，所以若不指定变量类型，则1+2为一个字符串而不是一个计算式；  7. bash环境中的数值运算，默认最多仅能到达整数类型，所以1/3的结果是0. |
| PS1(命令提示符) | 常用的设置为PS1="[\u@\h \W]\$"  \u: 目前用户的账户名称，如 “root”;  \h: 仅取主机名在第一个小数点之前的名字，后面的省略；  \W: 利用basename函数取得工作目录名称，所以仅会列出最后一个目录名；  \$：提示符，如果是root，为#，否则就是$. |
| CPU类型 | 目前个人呢计算机的CPU主要分为32、64位，其中32位又可分为i386、i586、i686，而64位则成为x86\_64。较高级的硬件通常会向下兼容旧有的软件，但较高级的软件可能无法安装在旧机器上。因此可以在x86\_64的硬件上安装i386的Linux操作系统，但无法在i686的硬件上安装x86\_64的Linux操作系统。 |
| read | 用来读取键盘输入，用法：  read [-pt] variable  -p: 后面可以接提示符；  -t：后面可以接等待的秒数。  例子：  read -p “Please enter your name:” -t 30 name |
| declare | 用于声明变量的类型，用法：  declare [-aixr] variable  -a: 将变量定义为数组（array）类型；  -i: 将变量定义为整数数字（integer）类型；  -x: 将变量定义为环境变量；  -r: 将变量定义为只读类型，不能更改，不能重设。通常要注销再登录时才能复原改变量的类型。  declare -p variable，可以单独列出变量的类型。  若declare后面不接任何参数，则将所有变量名称和内容全部调出来，就像使用set一样。 |
| 数组 | 数组的设置方式是：  var[index]=content  例子：  var[1]=1  var[2]=2  echo ${var[1]}, ${var[2]} |
| history | history -n  列出最近的n条命令行 |
| export | 将变量设置为环境变量，用法：  export variable  若不接参数则打印所有的环境变量。 |
| source和小数点 | 读入配置文件的命令。 |
| 小括号()  大括号{} | 小括号(), 在中间为子shell的起始和结束  大括号，在中间为命令块的组合 |
| 标准输入输出 | 标准输入(stdin):代码为0，使用<或<<  标准输出(stdout):代码为1，使用>或>>  标准错误输出(stderr):代码为2，使用2>或2>>  黑洞设备：/dev/null  用法：  1.将标准输出和标准错误分别写入不同的文件：  find /home -name .bashrc > list\_right 2> list\_error  2.将标准错误忽略不输出：  find /home -name .bashrc 2> /dev/null  3.将标准输出和标准错误写入同一个文件：  find /home -name .bashrc > list 2>&1 或 find /home -name .bashrc &> list |
| cut | cut主要的用途在于将同一行里面的数据进行分解。  1.分隔字符：  **cut -d “分隔字符”-f fields**  例子：  echo $PATH | cut -d ‘:’ -f 5 (取出第5个路径)  echo $PATH | cut -d ‘:’ -f 3,5 (取出3和第5个路径)  2.用于排列整齐的信息：  cut -c 字符范围  例子：  export | cut -c 1-7 （打印第1-第7个字符）  **批量改名**：  for i in `ls Water\_100\*`; do name=`echo $i | cut -d "\_" -f 2`; mv $i water\_${name}; done  for i in `ls Water\_100\*`; do name=`echo $i | awk ‘BEGIN {FS=“\_”} {printf “%s.%s”,$1,$2}’`; mv $i $name; done |
| grep | 刚才的cut是在一行信息当中取出某部分我们想要的，而grep则是分析一行信息，若当中有我们所需要的信息，就将该行拿出来：  用法：  grep [-acinv] [--color=auto] ‘要查找的字符串’ filename  -a：将binary文件以text方式查找数据  -c：计算查找的次数  -i：忽略大小写  -n：顺便输出行号  -v:反向选择，即显示没有“要查找字符串”内容的行 |
| uniq | 这个命令用来将重复的行删除掉只显示一个，用法：  uniq -ic  -i: 忽略大小写  -c: 进行计数  last | cut -d ‘ ’ -f 1 | sort | uniq -c |
| tee | 双向重定向，tee会同时将数据流送与文件和屏幕，而输到屏幕的就是stdout，可以让下个命令继续处理：  last | tee -a last.list| cut -d ‘ ’ -f 1  -a: 以累加（append）的方式，将数据写入文件中。 |
| tr | tr可以用来删除一段信息当中的文字，或者进行文字信息的替换。  用法：  tr [-ds] SET1  -d: 删除信息当中SET1这个字符串；  -c：替换掉重复的字符  例子：  last | tr ‘[a-z]’ ‘[A-Z]’ 将小写字符变成大写字符  cat /ect/passwd | tr -d ‘:’ 将冒号删除 |
| col | 用法：  col [-xb]  -x: 将tab键转换成对等的空格键  -b: 在文字内有反斜杠时，仅保留反斜杠最后接的那个字符 |
| join | 将两个文件当中有相同数据的那一行加在一起，在使用join之前，所需要处理的文件应该要事先经过排序  用法：  join [-ti12] file1 file2  -t: join默认以空格符分隔数据，并且默认对比第一个字段的数据，如果两个文件相同，则将两条数据连成一行  -i: 忽略大小写  -1：数字1，代表第一个文件要用哪个字段来分析  -2：数字2，代表第二个文件要用哪个字段来分析  例子：  join -t ‘:’ /etc/passwd /etc/shadow  joint -t ‘:’ -1 4 /etc/passwd -2 3 /etc/shadow |
| paste | 相当于join必须要对比两个文件的数据相关性，paste直接将两行粘贴在一起，默认中间以tab键隔开，可以用-d指定分隔符  例子：  paste /etc/passwd /etc/passwd |
| split | 切割命令，可以将一个大文件依据文件大小或行数来切割成小文件，用法：  split [-bl] file PREFIX  -b：后面可以接切割成的文件大小，可以加单位，如b、k、m等  -l：代表用行数进行切割  -d: 使用数字后缀（默认是使用字母后缀）  -a: 后缀的长度（默认是2）  PREFIX：代表前导符，可以作为切割文件的前导文字。  例子：  split -l 10 -d -a 1 windows windows  ls -al . | split -l 10 - list  注：-这里代表标准输入 |
| **xargs** | 参数替代符：  xargs可以读入stdin的数据，并且以空格符或断行字符进行分辨，将stdin的数据分隔成参数。  用法：  | xargs -i 命令 {}  可以用-p参数进行交互式操作。 |
| 减号（-） | 某些命令（如tar）需要文件名来进行处理时，该stdin与stdout可以用减号来替代： 例子：  **tar -cvzf - /home | tar -xvf -**  这个例子将/home里面的文件打包，但打包的数据不是记录到文件，而是传送到stdout；经过管道后，将tar -cvf - /home传送到后面的tar -xvf -，后面这个-则是取前一个命令的stdout，因此就不需要使用文件了。 |
| sed | sed可以将数据进行替换、删除、新增、选取特定行等的功能，用法：  sed [-nefr] **‘**动作**’**  **-n:** 使用安静模式，在一般的sed用法中，所有来自STDIN的数据一般都会被打印到屏幕上。但如果加上-n参数后，则只有经过sed特殊处理的那些行才会被列出。  -e：直接在命令行模式进行sed的动作编辑，若sed后面只有一个动作可以省略，若有两个以上动作，则必须加上-e；  -f: 后面若借存有命令行的文件，可执行文件里的命令指定的动作。  -r: sed的动作支持的是扩展的正则表达式。  -i：直接修改文件的内容，而不是打印到屏幕。  **‘**动作**’**:  [n1[,n2]] function  n1,n2: 不见得会存在，代表进行动作的行数，**$代表最后一行**。  function有下面的参数：  a: 新增，参数a后面接的字符串会在当前行的下一行出现（插入新行）；  i：插入，参数i后面接的字符串会在当前行的上一行出现（插入新行）；  c：替换，参数c后面接的字符串会替换所指定的行；  **d：**删除，删除所指定的行；  **p：**打印，将选择的行打印出来，通常动作参数p会合sed -n一起运行；  **s：**替换，对指定的行进行模式替换，如：1,20s/old/new/g，这里**g**表示对所有指定的行进行替换。  例子：  1.删除指定的行：  nl /etc/passwd | sed ‘2,5d’ （删除2-5行，nl输出文件并加行号）  2.打印指定的行：  nl /ect/passwd | sed -n ‘2,5p’  3.查找指定行的模式并进行替换：  sed ‘1,$s/old/new/g’ text-old > text-new |
| printf | 格式化打印，用法：  printf ‘打印格式’ 打印内容  参数：  1.常见的变量格式：  %ns: s代表string（字符串），n是数字，代表字符串的长度，不足以空格补齐，默认右对齐；  %ni: i代表integer（整数数字），n是数字，代表整数的长度，不足以空格补齐，默认右对齐，若n前面有前导0，则不足部分以0补充，见例子；  %N.nf: f代表float（浮点型数值），N带打印总长度（包括小数点），n代表小数位数。  2.常见的特殊样式：  \n: 输出新的一行；  \t: 水平的tab键  例子：  printf ‘%s’ abc →abc  printf ‘%5s’ abc → abc  printf ‘%i’ 1 →1  printf ‘%5i’ 1 → 1  printf ‘%05i’ 1 →00001  printf ‘%5.2f’ 1.1 → 1.10  注：  printf默认不输出行末回车，所以要手动加上 ‘\n’。 |
| **awk** | 相比于sed常常作用于一整行的处理，awk则比较倾向于将一行分成数个字段来处理，用法：  awk ‘条件类型1 {动作1} 条件类型2 {动作2}…’ filename  注：  1. awk后面接两个单引号并加上大括号{}来设置想要对数据进行的处理动作，只有满足了条件类型，才会执行大括号里的动作；  2. awk主要是处理每一行的字段内的数据，而默认的字段分隔符是空格键或tab键；  3. 每一行所分隔的每个字段都有变量名称，即$1, $2…，另外，特殊地，$0代表一整行数据；  4. 几个特殊的awk内置变量，这些变量用大写，而且不用使用$符号。  NF: 每一行的字段总数；  NR: 目前awk所处理的是“第几行”数据  FS: 目前的字段分隔符，默认是空白（包括空格和tab键）  5. 两个特殊的控制变量BEGIN和END（大写）, 表示读到文件的开始和结尾处。  6. awk内的打印使用print和printf命令，但二者的格式不太一样：使用print命令时，若要打印字符串或者特殊字符，需要将其用双引号括起来；使用printf时，打印的内容全部需用双引号括起来。  7. 与bash的变量不同，在awk当中，变量可以直接使用（一般用小写），不需加上$符号；  8. =用于变量赋值，而==用于逻辑运算；  9. 大括号内的多个动作命令，需用分号“;”或回车符分开；  例子：  1. 取出账号和IP，且账号和IP之间以tab隔开，只取前五个：  last | awk ‘NR<5 {print $1 “\t” $3}’  2. 将/etc/passwd的账号和UID打印，只打印UID小于10的行：  cat /etc/passwd | awk ‘BEGIN {FS= “:”} $3<10 {print $1 “\t” $3}’  默认FS是从第二行开始起作用，所以需用BEGIN指令明确指定从第一行开始使用FS。  3. 用awk进行求和，text第一列为行号，第二列是数值：  cat text | awk ‘BEGIN {sum=0} {sum+=$2} END {print “sum=” sum}’  4. 求平均值：  cat profile-complete.xvg | awk 'BEGIN {s=0;n=0} NR>158 {n+=1;s+=$2} END {mean=s/n; print "mean is:" mean}'  或者  cat profile-complete.xvg | awk 'BEGIN {s=0;n=0} $1>6.0 && NR>12 {n+=1;s+=$2} END {mean=s/n; print "mean is:" mean}'  5.求最小值  cat profile-complete.xvg | awk 'NR==14 {min=$2} NR>14 {if ($2<min) min=$2} END { print "minimum is:" min}'  特例，从PMF曲线求自由能：   |  | | --- | | ##Calculate the minimum:  cat profile.xvg | awk 'NR==14 {min=$2} NR>14 {if ($2<min) min=$2} END { printf "minimum:\t%f\n", min}' >> result.dat  ##Calculate the average maximum:  cat profile.xvg | awk 'BEGIN {s=0;n=0} $1>6.0 && NR>12 {n+=1;s+=$2} END {mean=s/n; printf "maximum:\t%f\n", mean}' >> result.dat  cat result.dat | awk 'NR==1 {Gmin=$2\*1.0} NR==2 {Gmax=$2\*1.0} END {deltaG=Gmin-Gmax; printf "deltaG:\t%f\n", deltaG}' >> result.dat | |
| BASH脚本相关 | 1. #!/bin/bash声明这个script使用的是bash shell，除了这里的 “#!”组合是用来声明shell之外，其他的#都是批注的用途；  2. 命令、参数间的多个空白会被忽略掉；  3. 空白行也将被忽略掉，tab键所得的空白也视为空格键；  4. 如果读取到一个Enter（CR）符号，就尝试开始执行命令；  5. 如果一行的内容太多，则使用“\[ENTER]”来扩展到下一行，注意，反斜杠和回车直接没有空格；  6. 使用exit 0可以回传脚本的执行结果给系统，接着执行echo $?则可以得到script的执行结果（这里是0）。  7. 脚本执行方式：  （1）用sh或bash进行执行，这时脚本在子进程中执行，当子进程完成后，子进程内的各项变量和操作将会结束而不会传回到父进程中；  （2）用source或点号.进行执行，这时脚本在父进程中执行，各项变量和操作在父进程忠仍然有效，因此若要读取文件中的环境变量，必须用source或点号进行读取。  8. Shell脚本的默认变量：  $0: 代表脚本文件名称；  $1, $2…: 代表第1、2…个命令行参数；  $#: 代表后接的参数的个数；  $@: 代表 “$1”, “$2”, “$3”…，每个变量是独立的（用双引号括起来）；  $\*：代表 “$1c$2c$3c…”，其中c为分隔字符，默认为空格键。$@更常用。  另外，与之相关的**shift**命令，是拿掉最前面的几个参数。  8. 使用sh进行脚本测试：  格式：  sh [-nvx] script.sh  -n: 不执行脚本，仅查询语法问题，若脚本没有问题，则不会显示任何信息；  -v: 在执行脚本之前，先将脚本内容输出到屏幕；  **-x：**将命令执行过程显示出来，如此用户可以判定程序代码执行到哪一段会出现相关的信息。 |
| 数值运算 | $((计算式))，注意这里使用两个圆括号。  例子：  echo $((1+1)) →2  对比：  $(命令)，是执行命令，相当于`命令`,`是反括号。  echo $(ls) →打印当前目录的东西 |
| bc | **bc - An arbitrary precision calculator language**  **scale means decimal places**  echo "2/3" | bc  echo "scale=2; 2/3" | bc  echo "(2/3)+(7/8)" | bc  echo "scale=2;(2/3)+(7/8)" | bc  p\_step=`echo "$step\*0.001" | bc -l`  i=0.5  while [ $(echo "$i<5.0" | bc) -eq 1 ]; do echo $i; i=$(echo "$i+0.5" | bc);done |
| test判断式 | 常用参数：test -e filename  **-e：**该文件名是否存在；  **-f：**该文件名是否存在且为文件；  **-d：**该文件名是否存在且为目录；  -L：该文件名是否存在且为连接文件；  -s：该文件名是否存在且为非空白文件；  整数之间的比较：test n1 -eq n2  -eq：两数值相等；  -ne：两数值不相等；  -gt：n1大于n2；-lt：n1小于n2；-ge：n1大于等于n2；-le：n1小于等于n2  字符串的比较：  test -z string: 判断字符串是否为0，若string为空字符串，则返回true；  test -n string：判断字符串sf不为空，如string不为空字符串，则返回true，-n可省略；  test s1==s2: 判断s1是否等于s2；  test s1!=s2: 判断s1是否不等于s2 |
| 判断符号[] | 中括号可以作为test的替换，注意**中括号的两端需要有空格符来分隔**，中括号内的变量变量最好都以双引号括起来，中括号内的常量最好也以单引号或双引号括起来。  例子：  [ “$a” == “abc” ] 等价于test “$a” = “abc”  注：  在判断表达式中，=和==会得到相同的结果，但是为了清晰不混淆，一般使用=进行赋值而使用只使用==进行条件判断。 |
| 条件判断式 | 格式：  if [ 条件判断式1]**;** then  命令块1  **elif** [ 条件判断式2 ]**；**then  命令块2  else  命令块3  **fi**  在条件判断式中，&&代表AND, ||代表OR，分别相当于-a和-o，通常一个中括号只表示一个条件判断式，如  [ “$yn” == “Y” ] || [ “$yn” == “y”] |
| 自定义函数 | 格式：  function fname () {  命令块  }  注意括号和大括号是必需的。  由于shell的执行是从上到下，从左到右，因此shell脚本中function的设置一定要在程序的最前面。  注意，function也是有内置变量的，它的内置变量和shell script很相似，函数名称是$0，后续的变量是$1, $2…,这里的$0,$1,$2…是函数内部的局部变量，和脚本的相应变量是不同的，不会混淆。 |
| while循环 | 格式：  while [ 条件判断式 ]  do  命令块  done  例子：  i=0;s=0  while [ “$i” -lt 100 ]  do  i=$(($i+1))  s=$(($s+$i))  done |
| for循环 | **格式1**：  for var in con1 con2 con3 …  do  命令块  done  例子：  users=$(cut -d “:” -f 1 /etc/passwd)  for username in $users  do  echo $username  done  **格式2**：  for var in {num1..num2} →**注意要使用大括号和两个点号**  do  命令块  done  例子：  for i in {1..10}  do  echo $i  done  **格式3**：  for ((初始值；限制值；步长值)) →**注意要使用两个圆括号以及分号**  do  命令块  done  例子：  for (( c=1; c<=5; c++ )) →**注意这里变量不需要使用$符号**  do  echo "Welcome $c times" →注意这里变量要使用$符号  done |
| for 双变量循环 | 使用两个数组来完成：  例1：  A=({a..z}) B=({1..26})  for (( I = 0; I < 26; ++I )); do  echo "${A[I]} ${B[I]}” 或者echo "${A[$I]} ${B[$I]}”  done  注：  这里数组元素可加$符号，也可以不加。  例2：  names=(`cat names.dat`)  tprs=(`cat tpr-files.dat`)  xtcs=(`cat xtc-files.dat`)  for ((i=0;i<6;i++)); do  echo 21 | g\_density -s ${tprs[i]} -f ${xtcs[i]} -o ${names[i]}-Na.xvg -b 148000 -e 150000  echo 22 | g\_density -s ${tprs[i]} -f ${xtcs[i]} -o ${names[i]}-Cl.xvg -b 148000 -e 150000  done |
| BASH array | An entire array can be assigned by enclosing the array items in **parenthesis**:  arr=(Hello World)  Individual items can be assigned with the familiar array syntax (unless you're used to Basic or Fortran):  arr[0]=Hello  arr[1]=World  But it gets a bit ugly when you want to refer to an array item:  echo ${arr[0]} ${arr[1]} ***The braces are required to avoid conflicts with pathname expansion.***  In addition the following funky constructs are available:  ${arr[\*]} # All of the items in the array  ${!arr[\*]} # All of the indexes in the array  ${#arr[\*]} # Number of items in the array  ${#arr[0]} # Length of item zero  Note that the "@" sign can be used instead of the "\*" in constructs such as ${arr[\*]}, the result is the same except when expanding to the items of the array within a quoted string. In this case the behavior is the same as when expanding "$\*" and "$@" within quoted strings: "${arr[\*]}" returns all the items as a single *word*, whereas "${arr[@]}" returns each item as a separate word. |
|  |  |