Shell程序设计

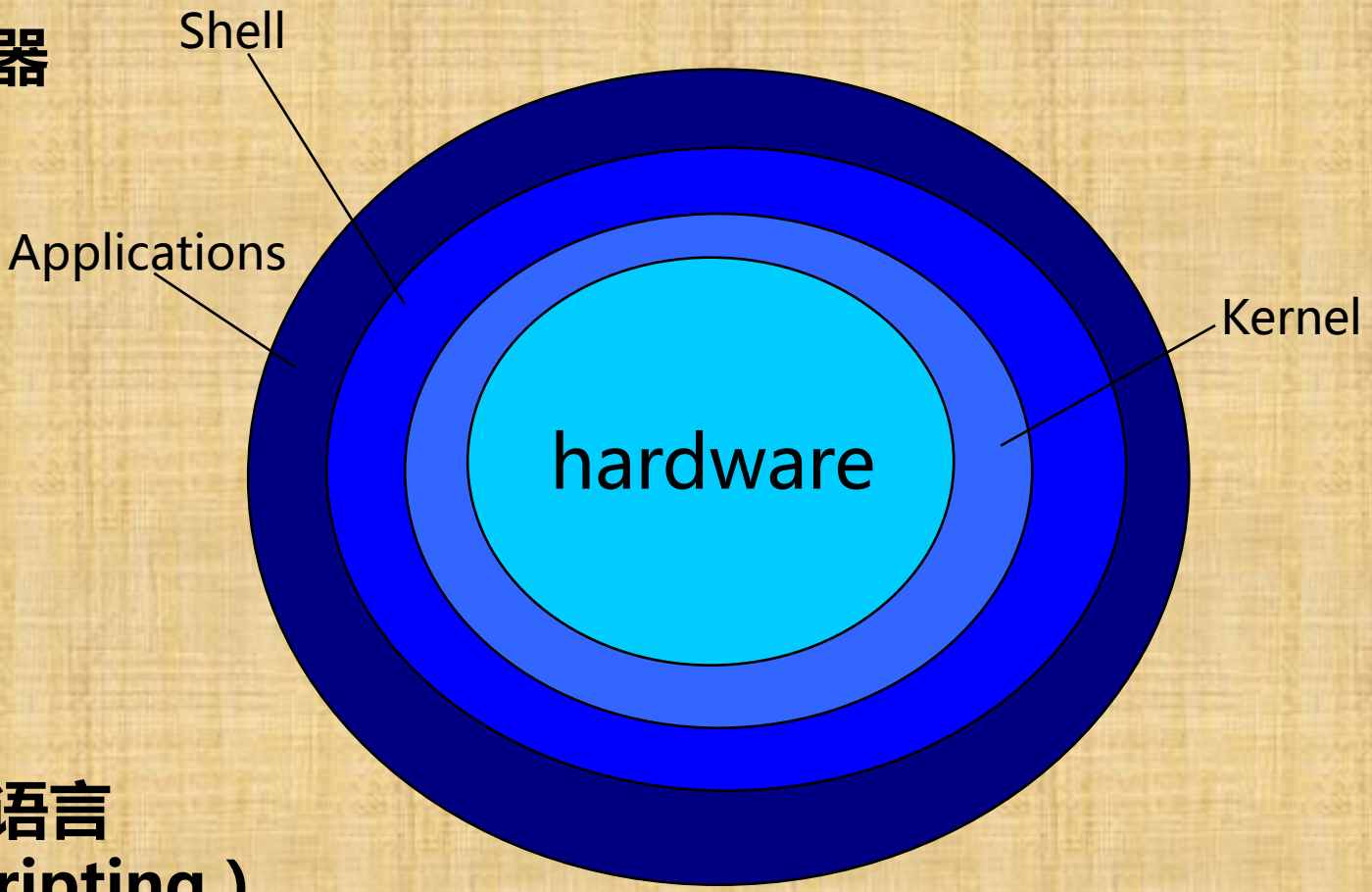
杨鑫

提纲

1. Shell简介
2. Bash编程基础
3. 常用字符处理工具介绍

什么是Shell？

1. 命令解释器



2. 程序设计语言
   脚本（Scripting）

为什么要学习Shell？

- Shell是Linux的用户接口
- 简单易用的命令组合完成复杂任务
- 强大的综合处理功能
- 一生受用好工具
- …

Shell有哪些种类？

- /bin/csh (c-like shell)
- /bin/ksh (kornshell, by AT&T Bell LAB)
- /bin/tcsh (enhanced csh, FreeBSD default login shell)
- /bin/bash (bourne-again shell)
- /bin/dash
- ...

- 当前系统安装的Shell可以通过 cat /etc/shells 查看

- 用户的默认登录Shell可以通过 cat /etc/passwd 查看
  tom:x:500:500:tom:/home/tom:/bin/bash

或者：echo $SHELL

提纲

1. Shell简介
2. Bash编程基础
3. 常用字符处理工具介绍

Bash Shell

发展历史：

AT & T 设计Unix 时设计了Bourne Shell；

Bash: Bourne Again Shell, GNU Project
大多数Linux发行版的默认Shell；

Bash的内部命令

Bash集成了一些内部命令，组成基本操作环境

常用的内部命令：
:(true) .(source) alias bg bind builtin cd declare dirs disown echo enable eval exec exit export fc fg getopts hash help history jobs kill let local logout popd pushd pwd read readonly return set shift stop suspend test times trap typest ultimit umask unalias unset wait

循环分支控制相关：
if else elif fi for do done case while until continue break

Bash的初始化脚本

通常在这些脚本中进行自定义设置，比如一些程序的环境变量等

/etc/profile
/etc/bashrc
~/.bash_profile
~/.bash_logout
~/.bashrc
~/.profile

Bash的执行方式(交互式)

Linux终端登录后，即进入交互式的执行环境
可以执行内部及外部命令

foo@localhost> ls
a.txt b.txt c.txt
foo@localhost> hostname
localhost

后台执行: command &
foo@localhost> sleep 300 &

Bash的执行方式(脚本方式)

脚本执行方式，将Bash语句写在文本文件中，批量执行
#开头表示注释

不开启子Shell，在当前Shell中运行：
foo@localhost> source test.sh
foo@localhost> . ./test.sh
foo@localhost> eval `cat test.sh`

在新的子Shell中运行：
foo@localhost> bash -c 'commands'
foo@localhost> bash test.sh
foo@localhost> ./test.sh
(脚本文件以#!/bin/bash开头，且具有可执行权限)

Hello world!

Bash的Hello world语句：

```
foo@localhost>  echo Hello World!
Hello World!

foo@localhost>  echo 'Hello World!'
Hello World!

foo@localhost>  echo "Hello World!"
Hello World!
```

Bash命令的组合方式

不同的命令可分行放置，也可以用分号隔开

```
#!/bin/bash
cd subdir1
rm a.txt
```

```
foo@localhost>  cd /home/foo/subdir2; rm b.txt
```

Bash命令的组合方式(逻辑组合)

管道线分隔符
command1 && command2 (前者执行成功才执行后者)
command1 || command2 (前者执行失败才执行后者)

foo@localhost> cd /home/foo && rm b.txt

foo@localhost> cd /home/foo | | echo "dir does not exist!"

Bash的别名设置（alias）

可以定义一些命令的别名，方便使用，别名设置可以放在Bash的初始化脚本中自动加载

```
foo@localhost> alias ll='ls -l --color=tty'
foo@localhost> alias ll
alias ll='ls -l --color=tty'

foo@localhost> alias rm='rm -i'

foo@localhost> unalias ll
```

取消别名

Bash的变量

变量申明、查看、清除

var=value　　#注意=号前后无空格

echo $var

unset var

```
foo@localhost> var1=abc
foo@localhost> echo $var1
abc
foo@localhost> unset var1
foo@localhost> echo $var1
```

（无输出）

Bash的变量(系统变量)

常用系统变量：

HOSTNAME
TERM
PATH
HOME
PWD
SHELL
…

```
foo@localhost> echo $SHELL
/bin/bash
foo@localhost> echo $HOME
/home/foo
```

Bash的变量(全局变量)

全局变量（子Shell会继承）
export var=value

```
foo@localhost> export var1=abc
foo@localhost> var2=123
foo@localhost> export var2

foo@localhost> bash -c 'echo $var1 $var2'
abc 123
```

Bash的变量(特殊变量)

$?	前一个命令的退出状态，正常退出返回0，异常退出返回非0值
$#	脚本或函数位置参数的个数
$0	脚本或函数的名称
$1, $2, ...	传递给脚本或函数的位置参数
$*	以”$1 $2...”的形式保存所有输入的命令行参数
$@	以“$1” “$2”...的形式保存所有输入的命令行参数

```
foo@localhost> cat test.sh
#!/bin/bash
echo "The command is $0 $1 $2"

foo@localhost> ./test.sh arg1 arg2
The command is test.sh arg1 arg2
```

Bash的变量(特殊变量)

```
foo@localhost> ls
a.txt b.txt
foo@localhost> echo $?
0

foo@localhost> ls c.txt
ls: cannot access c.txt: No such file or directory
foo@localhost> echo $?
2
```

Bash的通配符

通配符

*	配任何字符串，包括空字符串
？	匹配任何单个字符
[]	按照字符范围或列表匹配
{...,...}	按照字符串列表匹配
\	转意符，使元字符失去其特殊的含义

```
foo@localhost> ls *.txt
foo@localhost> ls A0?.txt
foo@localhost> ls [A-Z,a-z][0-9].txt
foo@localhost> ls [!a-z]*.txt
foo@localhost> ls abc.{txt,tar,dat}
foo@localhost> echo \*
```

Bash的命令替换

命令替换：把命令的输出结果字符串直接替换进来
`commands`　　注意是反引号，不是单引号
$(commands)

```
foo@localhost> echo `date`
Sun Jan 9 15:38:00  2011

foo@localhost> files=`ls`

foo@localhost> currentdir=$(pwd)
```

Bash的变量字符串操作(子串提取)

${#string}	返回字符串的长度
${string:position}	提取从指定位置开始的子串
${string:position:length}	提取从指定位置开始的定长子串

```
foo@localhost> string=1234567890
foo@localhost> echo ${#string}
10
foo@localhost> echo ${string:2}
34567890
foo@localhost> echo ${string:2:3}
345
```

Bash的变量字符串操作(子串剪裁)

${varible##*string}	从左向右截取最后一个string后的字符串
${varible#*string}	从左向右截取第一个string后的字符串
${varible%%string*}	从右向左截取最后一个string后的字符串
${varible%string*}	从右向左截取第一个string后的字符串

“*”只是一个通配符可以不要

```
foo@localhost> MYVAR=foodforthought.jpg
foo@localhost> echo ${MYVAR##*fo}
rthought.jpg
foo@localhost> echo ${MYVAR#*fo}
odforthought.jpg
```

Bash的变量字符串操作(子串替换)

${string/substring/replace}　　替换第一处匹配
${string//substring/replace}　　替换所有匹配
如果子串不匹配，不做裁剪操作

```
foo@localhost> string=black-black
foo@localhost> echo ${string/black/white}
white-black
foo@localhost> echo ${string//black/white}
white-white
```

Bash的变量数组(Array)

赋值：
array[0]=value1
array[1]=value2
...
或者一次性赋值：
array=(value1 value2 ...)

元素引用：
${#array[*]} 变量元素个数
${array[0]} 某个变量值
${array[*]} 全部变量值

```
foo@localhost> array=(dog fish cat)
foo@localhost> echo ${#array[*]}
3
foo@localhost> echo ${array[*]}
dog fish cat
foo@localhost> echo ${array[0]}
dog
```

Bash的终端控制（输入输出重定向）

Linux 默认的三个I/O 通道：
stdin（标准输入，文件描述符：0）– 默认是键盘
stdout（标准输出，文件描述符：1）– 默认是终端
stderr（标准错误，文件描述符：2）– 默认是终端

<	重定向stdin到文件
>	重定向stdout到文件（新建或覆盖）
>>	重定向stdout到文件（追加）
2>	重定向stderr到文件（新建或覆盖）
2>>	重定向stderr到文件（追加）
2>&1	重定向stderr到stdout
>&	重定向stdout和stderr到文件

Bash的终端控制（输出重定向）

```
foo@localhost> ls *.{txt,dat}
ls: cannot access *.txt: No such file or directory
a.dat b.dat

foo@localhost> ls *.{txt,dat} >/tmp/ls.out
ls: cannot access *.txt: No such file or directory

foo@localhost> cat /tmp/ls.out
a.dat b.dat

foo@localhost> ls *.{txt,dat}  >/tmp/ls.out  2>/tmp/err.out
foo@localhost> cat /tmp/err.out
ls: cannot access *.txt: No such file or directory
```

Bash的终端控制（输入重定向）

命令的输入定向到文件，而不是从键盘输入

foo@localhost> tr 'a-z' 'A-Z' </etc/hosts

foo@localhost> tr 'a-z' 'A-Z' </etc/hosts >result.log

Bash的终端控制(重定向到特殊文件)

可以重定向到特殊字符设备文件

e.g.

foo@localhost> ls >result.log 2>/dev/null

注：Linux下常用的特殊字符设备文件：

/dev/null ——“空文件”或“黑洞文件”，丢弃一切写入的内容

/dev/zero —— 永远提供NULL空字符

Bash的终端控制(管道符)

管道符pipe(|)，将命令的输出定向到下一条命令的输入

```
foo@localhost> ls /usr/lib | more

foo@localhost> cut -f1 -d: /etc/passwd | sort -r | more

foo@localhost> cut -f1 -d: /etc/passwd | sort -r > result.log
```

Bash的终端控制（调用tee）

如果既要在屏幕输出，又想重定向到文件，可以调用“tee”

```
foo@localhost> ls *.dat | tee /tmp/ls.out
a.dat b.dat

foo@localhost> cat /tmp/ls.out
a.dat b.dat

# tee -a 表示追加
foo@localhost> ls *.dat | tee -a /tmp/ls.out
a.dat b.dat

foo@localhost> cat /tmp/ls.out
a.dat b.dat
a.dat b.dat
```

Bash读入变量（read）

使用内部命令read，可以从终端读入Shell变量
与C语言的scanf()类似
格式：read [-p 提示字符] var1 [var2]

```
foo@localhost> cat read_test.sh
#!/bin/bash
read name age
echo $name is $age years old.

foo@localhost> ./read_test.sh
输入Bob 17，按Ctrl-d
Bob is 17 years old.

foo@localhost> echo Bob 17 | ./read_test.sh
Bob is 17 years old.
```

Bash中的引号

单引号屏蔽所有特殊字符

```
foo@localhost> echo '$HOME'
$HOME

foo@localhost> echo "$HOME"
/home/foo

foo@localhost> echo " '$HOME' "
'/home/foo'

foo@localhost> echo " \"$HOME\" "
"/home/foo"
```

Bash中的数学运算(内部命令)

内置命名 let 或 ((...))，只能用于整型运算

```
foo@localhost> echo $(( (1+1)*(2*2) ))
8

foo@localhost> a=1
foo@localhost> ((a+=1))
foo@localhost> echo $a
2

foo@localhost> a=1
foo@localhost> let a+=1
foo@localhost> echo $a
2
```

Bash中的数学运算(外部命令)

调用外部命令expr，只能用于整型运算

```
foo@localhost> expr \( 1 + 1 \) \* \(2 \* 2 \)
8

foo@localhost> a=`expr 1 + 1`
foo@localhost> echo $a
2
```

注意括号和运算符前后有空格

Bash中的数学运算(外部命令)

调用外部命令bc，可用于浮点运算，功能强大

```
foo@localhost> echo '1.2*1.2' | bc -l
1.44

foo@localhost> echo "ibase=10; obase=2; 10" | bc
1010
```

Bash中的逻辑判断

内部命令test，可用于数字比较、字符串比较、文件状态检查、逻辑判断等。语法：

test condition

或

[condition] **#注意[]前后的空格**

如果condition成立，返回0，不成立返回非0

```
foo@localhost> test 1 == 1; echo $?
0
foo@localhost> [ 1 == 2 ]; echo $?
1
```

Bash中的逻辑判断(字符串比较)

=或==	两个字符串相等
!=	两个字符串不等
-z	空字符串
-n	非空字符串
=~	与正则表达式regex进行比较

```
foo@localhost> str="abc"
foo@localhost> [ "$str" = "abc" ] && echo "true"
true
foo@localhost> [ -z "$str" ] && echo "true"
无输出
```

Bash中的逻辑判断(整数比较)

-eq或==	两个整数相等
-ne或!=	两个整数不等
-gt或>	前者大于后者
-ge或>=	前者大于等于后者
-lt或<	前者小于后者
-le或<=	前者小于等于后者

```
foo@localhost> var=1
foo@localhost> [ $var >= 1 ] && echo "true"
true
foo@localhost> [ $var < 0 ] && echo "true"
```
无输出

Bash中的逻辑判断(文件状态判断)

-d 所判断文件是一个目录
-f 正规文件（Regular file）
-L 符号链接
-r Readable（文件、目录可读）
-w Writable（文件、目录可写）
-x Executable（文件可执行、目录可浏览）
-u 文件有suid 权限标志
-s 文件长度大于0，非空
-z 文件长度等于0

```
foo@localhost> file=/etc/hosts
foo@localhost> [ -r $file ] && echo "read permission OK"
read permission OK
```

Bash中的逻辑判断(逻辑操作符)

condition1 -a condition2　**逻辑与**
condition1 -o condition2　**逻辑或**
! condition　**逻辑取非**

```
foo@localhost> a=2; b=3
foo@localhost> [ $a > 1 -a $b > 1  ] && echo "true"
true
foo@localhost> [ ! $a > 1 ] && echo "true"
无输出
```

Bash中的分支判断（if语句）

```
if condition_judgement
then
        ...
fi
```

```
if condition_judgement
then
        ...
else
        ...
fi
```

```
if condition_judgement1
then
        ...
elif
condition_judgement2
then
        ...
else
        ...
fi
```

Bash中的分支判断（if语句）

举例：

```
#!/bin/bash
if  [ $1 > 1 ]
then
    echo "Greater than 1"
elif [ $1 < 1 ]
then
    echo "Less than 1"
else
    echo “Equals to 1"
fi
```

```
#!/bin/bash
if  cd /tmp/test &>/dev/null
then
    echo “Entering dir succeeds"
fi
```

条件判断也可以是命令，执行成功表示真

Bash中的分支判断（case语句）

```
case expression in
pattern1)
        ...;;
pattern2)
        ...;;
...
patternn)
        ...;;
esac
```

```
case expression in
pattern1)
        ...;;
pattern2)
        ...;;
...
patternn)
        ...;;
*)
        ...;;
esac
```

Bash中的分支判断（case语句）

举例：

```
#!/bin/bash
case $1 in
apple)
        echo "You choose apple";;
grape)
        echo "You choose grape";;
*)
        echo “I have only apple and grape";;
esac
```

Bash中的循环控制（for循环）

```
for var in value_list
do
        …
done
```

```
for ((var=0; var<=10; var++))
do
        …
done
```

Bash中的循环控制（for循环）

举例：

```
for name in tom bob cindy
do
        echo $name
done
```

```
for i in `seq 1 10`
do
        ssh node$i date
done
```

变量取值也可以来自其它命令的输出

```
for ((i=0; i<=10; i++))
do
        echo $i
done
```

Bash中的循环控制（while循环）

```
while condition_judgement
do
        …
done
```

条件判断可以是：

- test 或 [..] 语句
- **其它命令，判断其执行成功与否**
- true 或：(Bash**的内部命令，代表真，使得**while**成为无限循环**)

Bash中的循环控制（while循环）

举例一：

```
COUNTER=0
# If COUNTER < 5, program will print the value of it.

while [ "$COUNTER" < 5 ]
do
        COUNTER=$((COUNTER+1))
        echo $COUNTER
done
```

Bash中的循环控制（while循环）

举例二：不停地从终端读入一个整数，输入9则退出

```
NUMBER=0

while [ "$NUMBER" != "9" ]; do
        read -p 'Enter a integer, 9 to exit' NUMBER
        echo You typed is $NUMBER
done
```

Bash中的循环控制（while循环）

举例三：逐行读取文件的内容，进行部分内容替换

```
while read LINE
do
        echo $LINE | sed -e "s/node/blade/"
done </etc/hosts
```

Bash中的循环控制（while循环）

举例四：无限循环及跳出

```
target_pid=`123 `
while true
do
        free >/tmp/memory_use.log
        sleep 10
        if  ! ps $target_pid >&/dev/null
        then
                break
        fi
done
```

break语句用于跳出循环
break <n>，可以跳出多重循环

Bash中的循环控制（until循环）

```
until condition_judgement
do
        …
done
```

与while循环语法一致，但意思“相反”，没有while循环常用
until循环是先执行循环，直到满足判断条件后跳出

Bash中的函数

在Shell 中，函数就是一组实现一定功能的命令集合；
函数由两部分组成：
函数名（在一个脚本中必须唯一）；
函数体（命令集合）；

```
function_name()
{
        …
}

#先定义，后使用
function_name
```

Bash中的函数(函数参数)

与Bash脚本的位置参数一样，函数的位置参数也用$1, $2, ...表示

举例：交换两个文件的文件名

```
swap_filename()
{
        file1=$1
        file2=$2
        cp $file1 tmp
        mv $file2 $file1
        mv tmp $file2
}
```

Bash中的函数(函数的退出状态)

函数结尾可以调用return语句返回执行状态，0表示无错误

举例：

```
check_file_lines()
{
        file=$1
        lines=`wc -l $file`
        if [ $lines -gt 1000 ]; then
                return 0
        else
                return -1
        fi
}
check_file_lines result.log && echo "file is long enough"
```

提纲

1. Shell简介
2. Bash编程基础
3. 常用字符处理工具介绍

正则表达式regex

正则表达式：进行精确文字匹配的句法规则；比通配符强大无数倍

Regex的基本元字符集：

.	匹配任意单个字符
*	一个单字符后紧跟*，表示匹配该字符0次或任意多次
^	匹配行首
$	匹配行尾
[]	匹配序列中的任何一个，比如[abc123]、[a-z0-9] [^...]匹配非序列中的字符
\	转意特殊字符
pattern{n}	匹配规则重复n次
pattern{n,}	匹配规则重复至少n次
pattern{n,m}	匹配规则重复至少n次，至多m次

正则表达式regex（续1）

Regex的扩展元字符集：

+	一个单字符后紧跟+，表示匹配该字符1次或任意多次
？	一个单字符后紧跟？，表示匹配该字符0次或1次
(patterns)	匹配规则组合

Regex举例：

^The	匹配行首的The
.*	匹配任意字符或字符串
^$	匹配空行
^.*$	匹配任意行
A[0-9]\.txt	匹配A0.txt，A1.txt，…，A9.txt
[^0-9a-zA-Z]	匹配非数字或字母的字符
\([0-9]\{1,3\}\.\)\{3\}[0-9]\{1,3\}	匹配IPv4的IP地址

字符查找工具grep

grep 是 Linux 下使用最广泛的命令之一，其作用是在一个或多个文件中查找某个字符模式所在的行，并将结果输出到屏幕上

grep 以行为单位进行处理，不会对输入文件本身进行修改

- grep 家族由 grep、egrep 和 fgrep 组成：
  - grep: 标准 grep 命令，本文主要讨论此命令。
  - egrep: 扩展 grep，支持基本及扩展的正则表达式。
  - fgrep: 固定 grep (fixed grep)，按字面解释所有的字符，即正则表达式中的元字符不会被特殊处理。

字符查找工具grep(续1)

- grep 命令的一般形式：

```
grep [option] pattern file1 file2 ...
```

  - pattern：可以是正则表达式（用单引号括起来）、或字符串（加双引号）、或一个单词。
  - file1 file2 ... ：文件名列表，作为 grep 命令的输入；grep 的输入也可以来自标准输入或管道；

- 可以把匹配模式写入到一个文件中，每行写一个，然后使用 -f 选项，将该匹配模式传递给 grep 命令

```
grep -f patternfile file1 file2 ...
```

字符查找工具grep (续2)

grep 常用选项

-c	只输出匹配的行的总数
-i	不区分大小写
-h	查询多个文件时，不显示文件名
-l	查询多个文件时，只输出包含匹配模式的文件的文件名
-n	显示匹配行的行号
-v	反向查找，即只显示不包含匹配模式的行
-s	不显示错误信息

```
grep -i  'apple' datafile
```

字符查找工具grep（举例）

查找包含字符串Hello的行
grep 'Hello' datafile

查找以#开头的行
grep '^#' datafile

查找非空行并另存
grep -v '^$' datafile >datefile2

与管道符连用
cat /etc/hosts | grep 'node' | wc -l

字符流编辑工具sed

- **sed** 是什么

  sed 是一个精简的、非交互式的编辑器

- **sed** 如何工作

  sed 逐行处理文件（或输入），并将输出结果发送到屏幕。
  即：**sed** 从输入（可以是文件或其它标准输入）中读取一行，将之拷贝到一个编辑缓冲区，按指定的 **sed** 编辑命令进行处理，编辑完后将其发送到屏幕上，然后把这行从编辑缓冲区中删除，读取下面一行。重复此过程直到全部处理结束。

sed 只是对文件在内存中的副本进行操作，所以 sed 不会修改输入文件的内容。sed 总是输出到标准输出，可以使用重定向将 sed 的输出保存到文件中。

字符流编辑工具sed(续1)

- sed的命令行调用方式

```
sed [-n][-e] 'sed_cmd' input_file
```

- **-n**：缺省情况下，**sed** 在将下一行读入缓冲区之前，自动输出行缓冲区中的内容。此选项可以关闭自动输出。
- **-e**：允许调用多条 **sed** 命令，如：

```
sed -e 'sed_cmd1' -e 'sed_cmd2'  input_file
```

- **sed_cmd**：使用格式: **[address]sed_edit_cmd** (通常用单引号括起来)，其中 **address** 为 **sed** 的**行定位模式**，用于指定将要被 **sed** 编辑的行。如果省略，**sed** 将编辑所有的行。**sed_edit_cmd** 为 **sed** 对被编辑行将要进行的**编辑操作**。
- **input_file**：**sed** 编辑的文件列表，若省略，**sed** 将从标准输入（重定向或管道）中读取输入。

字符流编辑工具sed (续2)

- 也可以将 **sed** 命令存在脚本文件，然后调用

```
sed [选项] -f sed_script_file input_file
```

```
例：sed -n -f sedfile1 datafile
```

- 将 **sed** 命令插入脚本文件，生成 **sed** 可执行脚本文件，在命令行中直接键入脚本文件名来执行。

```
#!/bin/sed -f
sed_cmd1
… …
```

```
例：./sedfile2.sed  -n  datafile
```

字符流编辑工具sed (续3)

- **sed_cmd** 中 **address** 的定位方式

n	表示第 n 行
$	表示最后一行
m,n	表示从第 m 行到第 n 行
/pattern/	查询包含指定模式的行。如 /disk/、/[a-z]/
/pattern/,n	表示从包含指定模式的行 到 第 n 行
n,/pattern/	表示从第 n 行 到 包含指定模式的行
/模式1/,/模式2/	表示从包含模式1 到 包含模式2的行
!	反向选择， 如 m,n ! 的结果与 m,n 相反

字符流编辑工具sed (续4)

- 常用的 sed_edit_cmd
  - p ：打印匹配行

```
sed -n '1,3p' datafile
```

```
sed -n '$p' datafile
```

```
sed -n '/north/p' datafile
```

  - = ：显示匹配行的行号

```
sed -n '/north/=' datafile
```

  - d ：删除匹配的行

```
sed -n '/north/d' datafile
```

字符流编辑工具sed (续5)

- r ：读文件，将另外一个文件中的内容附加到指定行后。

```
sed -n '$r newdatafile'  datafile
```

- w ：写文件，将指定行写入到另外一个文件中。

```
sed -n '/west/w newdatafile'  datafile
```

- n ：将指定行的下面一行读入编辑缓冲区。

```
sed -n '/eastern/{n;s/AM/Archie/;p}' datafile
```

对指定行同时使用多个 sed 编辑命令时，需用大括号 “ {} ” 括起来，命令之间用分号 “ ; ” 格开。注意与 -e 选项的区别

字符流编辑工具sed (续6)

- q ：退出，读取到指定行后退出 sed。

```
sed '/east/{s/east/west/;q}' datafile
```

常见的 sed 编辑命令小结

p	打印匹配行	s	替换命令
=	显示匹配行的行号	l	显示指定行中所有字符
d	删除匹配的行	r	读文件
a\	在**指定行**后面追加文本	w	写文件
i\	在**指定行**前面追加文本	n	读取指定行的下面一行
c\	用新文本**替换指定的行**	q	退出 **sed**

字符流编辑工具sed (举例)

'/north/p'	打印所有包含 north 的行
'/north/!p'	打印所有不包含 north 的行
's/\.$//g'	删除以句点结尾的行中末尾的句点
's/^ *//g'	删除行首空格（命令中 ^ * 之间有两个空格）
's/ */ /g'	将连续多个空格替换为一个空格 命令中 */ 前有三个空格，后面是一个空格
'/^$/d'	删除空行
's/^.//g'	删除每行的第一个字符，同 's/.//'
's/^/%/g'	在每行的最前面添加百分号 %
'3,5s/d/D/'	把第 3 行到第 5 行中每行的 第一个 d 改成 D

awk工具介绍

❑ awk 是什么

- **awk** 是一种用于处理数据和生成报告的编程语言
- **awk** 可以在命令行中进行一些简单的操作，也可以被写成脚本来处理较大的应用问题
- **awk** 与 **grep、sed** 结合使用，将使 shell 编程更加容易
- **Linux** 下使用的 **awk** 是 **gawk**

❑ awk 如何工作

awk 逐行扫描输入（可以是文件或管道等），按给定的模式查找出匹配的行，然后对这些行执行 **awk** 命令指定的操作。

❑ 与 sed 一样，awk 不会修改输入文件的内容。

可以使用重定向将 **awk** 的输出保存到文件中。

awk的三种调用方式

- 在命令行键入命令:

```
awk [-F 字段分隔符] 'awk_script' input_file
```

若不指定字段分隔符，则使用环境变量 IFS 的值（ 通常为空格或 Tab ）

- 将 **awk** 命令插入脚本文件 **awd_script**，然后调用：

```
awk -f awd_script  input_file
```

- 将 **awk** 命令插入脚本文件，生成 **awk** 可执行脚本文件，
  然后在命令行中直接键入脚本文件名来执行。

```
#!/bin/awk -f
awk_cmd1
… …
```

awk的三种调用方式

- awk_script 可以由一条或多条 awk_cmd 组成，每条 awk_cmd 各占一行。
- 每个 awk_cmd 由两部分组成：/pattern/{actions}
- awk_cmd 中的 /pattern/ 和 {actions} 可以省略，但不能同时省略；/pattern/ 省略时表示对所有的输入行执行指定的 actions；{actions} 省略时表示打印匹配行。
- awk 命令的一般形式:

```
awk 'BEGIN {actions}
     /pattern1/{actions}
      ......
     /patternN/{actions}
     END {actions}'  input_file
```

注意 BEGIN 和 END都是大写字母。

其中 BEGIN {actions} 和 END {actions} 是可选的

awk的执行过程

① 如果存在 **BEGIN** ，**awk** 首先执行它指定的 **actions**

② **awk** 从输入中读取一行，称为一条输入记录

③ **awk** 将读入的记录分割成数个字段，并将第一个字段放入变量 **$1** 中，第二个放入变量 **$2** 中，以此类推；**$0** 表示整条记录；字段分隔符可以通过选项 **-F** 指定，否则使用缺省的分隔符。

④ 把当前输入记录依次与每一个 **awk_cmd** 中 **pattern** 比较：
如果相匹配，就执行对应的 **actions**；
如果不匹配，就跳过对应的 **actions**，直到完成所有的 **awk_cmd**

⑤ 当一条输入记录处理完毕后，**awk** 读取输入的下一行，重复上面的处理过程，直到所有输入全部处理完毕。

⑥ **awk** 处理完所有的输入后，若存在 **END**，执行相应的 **actions**

⑦ 如果输入是文件列表，**awk** 将按顺序处理列表中的每个文件。

awk举例

```
awk '/Mar/{print $1,$3}' datafile
```

```
awk   ‘{print $1,$2,$1+$2}' datafile
```

```
awk '/Mar/' datafile
```

```
awk 'BEGIN{print "Name Data"}/Mar/{print $1,$3}' datafile
```

```
awk '/Mar/{print $1,$3} END{print "OK"}' datafile
```

```
awk -F: -f awkfile1 datafile
```

awk的模式匹配

- awk 中的模式（pattern）匹配

① 使用正则表达式：/rexp/，如 /^A/、/A[0-9]*/

awk 中正则表达式中常用到的元字符有:

^	只匹配行首（可以看成是行首的标志）
$	只匹配行尾（可以看成是行尾的标志）
*	一个单字符后紧跟 *，匹配 0个或多个此字符
[ ]	匹配 [] 内的任意一个字符（[^] 反向匹配）
\	用来屏蔽一个元字符的特殊含义
.	匹配任意单个字符
str1|str2	匹配 str1 或 str2
+	匹配一个或多个前一字符
?	匹配零个或一个前一字符
( )	字符组

awk的模式匹配

② 使用布尔（比较）表达式，表达式的值为真时执行相应的操作(**actions**)

- 表达式中可以使用变量（如字段变量 **$1,$2** 等）和 **/rexp/**
- 表达式中的运算符有
  - 关系运算符: < > <= >= == !=
  - 匹配运算符: ~ !~

    x ~ /rexp/ 如果 **x** 匹配 **/rexp/**，则返回真；

    x !~ /rexp/ 如果 **x** 不匹配 **/rexp/**，则返回真。

```
awk '$2 > 20 {print $0}' datafile
```

```
awk '$4 ~ /^6/ {print $0}' datafile
```

awk的模式匹配

- 复合表达式：&&（逻辑与）、||（逻辑或）、!（逻辑非）

expr1 && expr2 两个表达式的值都为真时，返回真
expr1 || expr2 两个表达式中有一个的值为真时，返回真
! expr 表达式的值为假时，返回真

```
awk '($2<20)&&($4~/^6/){print $0}' datafile
```

```
awk '($2<20)||($4~/^6/){print $0}' datafile
```

```
awk '!($4~/^6/){print $0}' datafile
```

```
awk '/^A/ && /0$/{print}' datafile
```

注：表达式中有比较运算时，一般用圆括号括起来

awk的字段分隔符、重定向和管道

❑ 字段分隔符

awk 中的字段分隔符可以用 -F 选项指定，缺省是空格或Tab。

```
awk '{print $1}' datafile
```

```
awk -F: '{print $1}' datafile
```

```
awk -F'+++' ' {print $1}' datafile                    #支持多字符分隔符
```

❑ 重定向与管道

```
awk '{print $1, $2 > "output"}' datafile2
```

```
awk 'BEGIN{"date" | getline d; print d}'
```

awk中的操作ACTIONS

- 操作由一条或多条语句或者命令组成，语句、命令之间用分号 “;” 隔开。操作中还可以使用流程控制结构的语句

- **awk命令**
  - **print** 输出列表：打印字符串、变量或表达式，输出列表中各参数之间用逗号隔开；若用空格隔开，打印时各输出之间没有空格
  - **printf**（[格式控制符], 输出列表）：格式化打印，语法与 **C**语言中的 **printf** 函数类似
  - **next**：停止处理当前记录, 开始读取和处理下一条记录
  - **nextfile**：强迫 **awk** 停止处理当前的输入文件而处理输入文件列表中的下一个文件
  - **exit**：使 **awk** 停止执行而跳出。若存在 **END** 语句，则执行 **END** 指定的 **actions**

awk的输出函数printf

- 基本上和 C 语言的语法类似

```
printf( [格式控制符], 参数列表 )
```

- 参数列表中可以有变量、数值数据或字符串，用逗号隔开
- 格式控制符：**%[-][w][.p]fmt**
  - **%**：标识一个格式控制符的开始，不可省略
  - **-**：表示参数输出时左对齐，可省略
  - **w**：一个数字，表示参数输出时占用域的宽度，可省略
  - **.p**：p是一个数值，表示最大字符串长度或小数位位数，可省略
  - **fmt**：一个小写字母，表示输出参数的数据类型，不可省略

awk的输出函数printf

- 常见的 fmt

c	ASCII 字符	d	整数
f	浮点数，如 12.3	e	浮点数，科学记数法
g	自动决定用 e 或 f	s	字符串
o	八进制数	x	十六进制数

```
echo "65" | awk '{ printf "%c\n",$0 }'
```

```
awk 'BEGIN{printf "%.4f\n",999}'
```

```
awk 'BEGIN{printf  "2 number:%8.4f %8.2f",999,888}'
```

awk中的变量

- 在 **awk_script** 中的表达式中要经常使用变量。**awk** 的变量基本可以分为：字段变量，内置变量和自定义变量。

- 字段变量：**$0, $1, $2, ...**
  - 在 **awk** 执行过程中，字段变量的值是动态变化的。
    但可以修改这些字段变量的值，被修改的字段值可以反映到 **awk** 的输出中。
  - 可以创建新的输出字段，比如：当前输入记录被分割为 **8**个字段，这时可以通过对变量 **$9**（或 **$9** 之后的字段变量）赋值而增加输出字段，**NF** 的值也将随之变化。
  - 字段变量支持变量名替换。如 **$NF** 表示最后一个字段

【举例】对于只有两个字段的文件，以下两条命令等价：

```
awk '{$3=$1+$2; print}' datafile
awk '{print $1,$2,$1+$2}' datafile
```

awk的内置变量

- 用于存储 awk 工作时的各种参数, 这些变量的值会随着 awk 程序的运行而动态的变化，常见的有：

- **ARGC**：命令行参数个数（实际就是输入文件的数目加 **1**）
- **ARGIND**：当前被处理的文件在数组 **ARGV** 内的索引
- **ARGV**：命令行参数数组
- **FILENAME**：当前输入文件的文件名
- **FNR**：已经被 **awk** 读取过的记录(行)的总数目
- **FS**：输入记录的字段分隔符 (缺省是空格和制表符）
- **NF**：**当前行或记录的字段数**
- **NR**：**对当前输入文件而言，已被 awk 读取过的记录（行）的数目**
- **OFMT**：数字的输出格式（缺省是 **%.6g**）
- **OFS**：输出记录的字段分隔符 (缺省是空格）
- **ORS**：输出记录间的分隔符 (缺省是 **NEWLINE**）
- **RS**：输入记录间的分隔符 (缺省是 **NEWLINE**）

awk的自定义变量

- 变量定义 **varname = value**
  - 变量名由字母、数字和下划线组成，但不能以数字开头
  - **awk** 变量无需声明，直接赋值即完成变量的定义和初始化
  - **awk** 变量可以是数值变量或字符串变量
  - **awk** 可以从表达式的上下文推导出变量的数据类型

- 在表达式中出现不带双引号的字符串都被视为变量

- 如果自定义变量在使用前没有被赋值，缺省值为 0 或 空字符串

awk变量赋值语句

- awk 语句：主要是赋值语句
  - 直接赋值：如果值是字符串，需加双引号。

```
awk 'BEGIN
    {x=1;y=x;z="OK";
     print "x=" x, "y=" y, "z=" z}'
```

  - 用表达式赋值：
    - 数值表达式: num1 operator num2
      其中 operator 可以是 +，-，*，/，%，^
      当 num1 或 num2 是字符串时，awk 视其值为 0
    - 条件表达式: A?B:C 当A为真时表达式的值为 B，否则为 C
    - awk 也支持以下赋值操作符：
      +=，-=，*=，/=，%=，^=，++，--

awk的变量传递

- 如何向命令行 awk 程序传递变量的值

```
awk 'awk_script' var1=val1  var2=val2 ... files
```

- var 可以是 awk 内置变量或自定义变量。
- var 的值在 awk 开始对输入的第一条记录应用 awk_script 前传入。如果在 awk_script 中已经对某个变量赋值，那么命令行上的赋值无效。
- 在 awk 脚本程序中不能直接使用 shell 的变量。
- 可以向 awk 可执行脚本传递变量的值，与命令行类似，即

```
awk_ex_script var1=val1 var2=val2 ... files
```

```
awk '{if ($3 < ARG) print}' ARG=30  datafile
```

```
cat /etc/passwd | awk 'BEGIN {FS=":"} {if ($1==user) {print}}'
user=$USER
```

awk的流控制

- awk 中的流控制结构（基本上是用 C 语言的语法）
  - **if (expr) {actions}**
    [else if {actions}] （可以有多个 **else if** 语句）
    [else {actions}]
  - **while (expr) {actions}**
  - **do {actions} while (expr)**
  - **for (init_val;test_cond;incr_val) {actions}**
  - **break**：跳出 **for**，**while** 和 **do-while** 循环
  - **continue**：跳过本次循环的剩余部分，直接进入下一轮循环

for循环举例，打印每行的最后3个字段：

```
awk '{for(i=NF-2;i<=NF;i++) printf("%s ",$i); print ""}'  datafile
```

awk的内置数值函数

① 常见 awk 内置数值函数

- **int(x)**：取整数部份，朝 **0** 的方向做舍去。
- **sqrt(x)**：正的平方根。
- **exp(x)**：以 **e** 为底的指数函数。
- **log(x)**：自然对数。
- **sin(x)、cos(x)**：正弦、余弦。
- **atan2(y,x)**：求 **y/x** 的 **arctan** 值，单位是弧度。
- **rand()**：得到一个随机数（平均分布在 **0** 和 **1** 之间）
- **srand(x)**：设定产生随机数的 **seed** 为 **x**

awk的内置字符串函数

② 常见 awk 内置字符串函数

- **index(str,substr)**：返回子串 **substr** 在字符串 **str** 中第一次出现的位置，若找不到，则返回值为 **0**

```
awk 'BEGIN{print index("peanut","an")}'
```

- **length(str)**：返回字符串 **str** 的字符个数

- **match(str,rexp)**: 返回模式 **rexp** 在字符串 **str** 中第一次出现的位置，如果 **str** 中不包含 **rexp**，则返回值 **0**

```
awk 'BEGIN{print match("arrange",/r[^r]/)}'
```

- **sprintf(format,exp1,...)**：返回一个指定格式的表达式，格式 **format** 与 **printf** 的打印格式类似（不在屏幕上输出）

awk的内置字符串函数

- **sub(rexp,sub_str,target)**：在目标串 **target** 中寻找第一个能够匹配正则表达式 **rexp** 的子串，并用字符串 **sub_str** 替换该子串。若没有指定目标串，则在整个记录中查找

```
awk 'BEGIN{str="water,water";sub(/at/,"ith",str);\
    print str}'
```

- **gsub(rexp,sub_str,target)**：与 **sub** 类似，但 **gsub** 替换所有匹配的子串，即全局替换。

- **substr(str,start,len)**：返回 **str** 的从指定位置 **start** 开始长度为 **len** 个字符的子串，如果 **len** 省略，则返回从 **start** 位置开始至结束位置的所有字符。

```
awk 'BEGIN{print substr("awk sed grep",5)}'
```

awk的内置字符串函数

- **split(str,array,fs)**：使用由 **fs** 指定的分隔符将字符串**str** 拆分成一个数组 **array**，并返回数组的下标数

```
awk 'BEGIN{split("11/15/2005",date,"/"); \
    print date[2]}'
```

- **tolower(str)**：将字符串 **str** 中的大写字母改为小写字母

```
awk 'BEGIN{print tolower("MiXeD CaSe 123")}'
```

- **toupper(str)**：将字符串 **str** 中的小写字母改为大写字母

awk的自定义函数

```
function fun_name (parameter_list) {
    body-of-function
   // 函数体，是 awk 语句块
}
```

- parameter_list 是以逗号分隔的参数列表
- 自定义函数可以在 awk 程序的任何地方定义
- 函数名可包括字母、数字、下标线，但**不可以数字开头**
- 调用自定义的函数与调用内置函数的方法一样

```
awk '{print "sum =", SquareSum($2,$3)} \
function SquareSum(x,y) { \
sum=x*x+y*y ; return sum \
}' datafile
```

使用awk的注意事项

- 为了避免碰到 **awk** 错误，要注意以下事项:
  - 确保整个 **awk_script** 用单引号括起来
  - 确保 **awk_script** 内所有引号都成对出现
  - 确保用花括号括起动作语句，用圆括号括起条件语句
  - 如果使用字符串，要保证字符串被双引号括起来(在模式中除外)

- **awk** 语言学起来可能有些复杂，但使用它来编写一行命令或小脚本并不太难。**awk** 是 **shell** 编程的一个重要工具。在**shell** 命令或编程中，可以使用 **awk** 强大的文本处理能力。

谢谢！