# 第5章

# bash shell

# 第5章 bash shell

- 5.1 bash shell 介绍
- 5.2 shell变量
- 5.3 命令的别名与历史
- 5.4 bash shell的操作环境
- 5.5 输入输出重定向
- 5.6 多命令间的逻辑关系
- 5.7 管道命令

# 5.1 Bash shell介绍

# 什么是shell

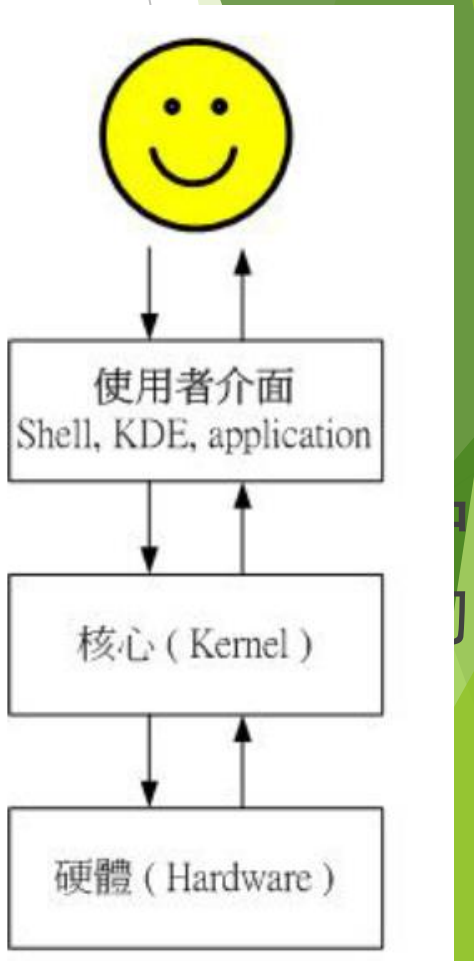
- 计算机实现指定功能，需要硬件、操作系统（核心管理）、应用程序，缺一不可。比如以听音乐为例，需要：

  硬件：能播放声音的硬件，声卡

  核心管理：操作系统的核心管理和驱动程序控制程

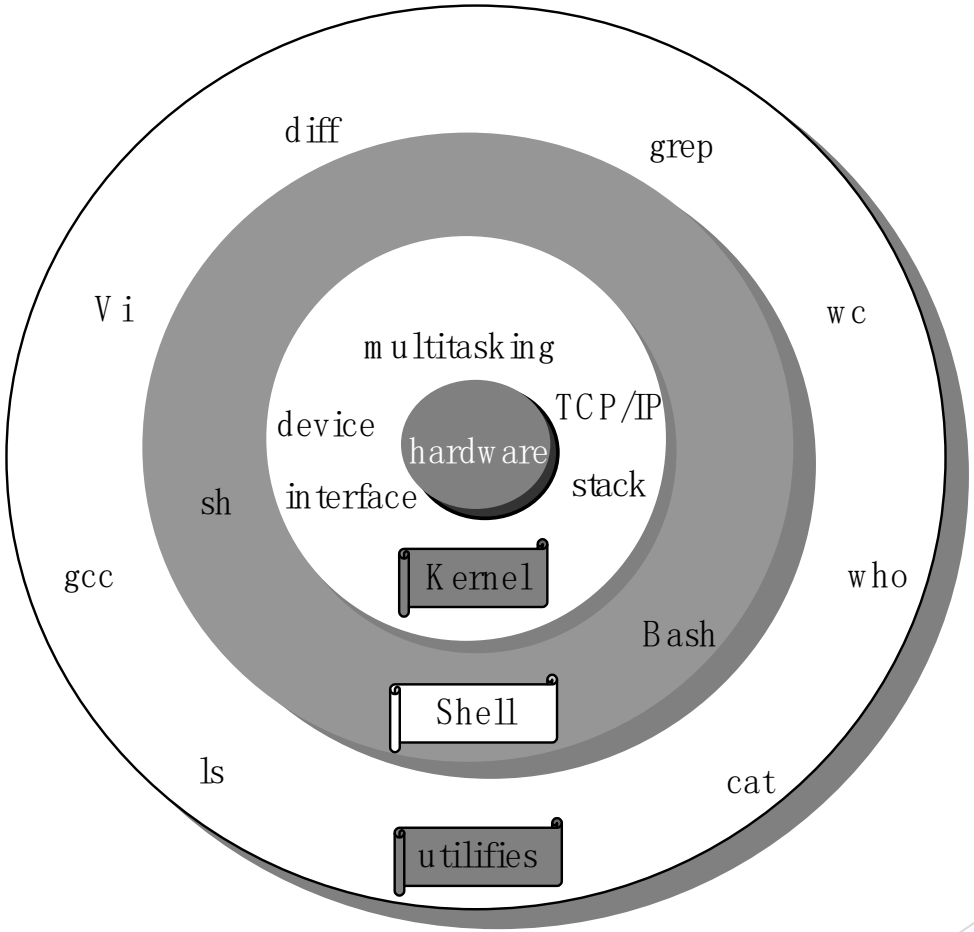
  应用程序：发出使用声卡的指令。

  在该过程中，系统核心要接收应用程序的指令，随意操作系统核心，所以需要利用shell实现输入指沟通，进而让Kernel可以控制硬件来工作。



# 什么是shell

- Shell是用户与操作系统内核之间的接口，起着协调用户与系统的一致性和在用户与系统之间进行交互的作用。像外壳（shell）一样围绕在外层，所以称为shell程序。



# 什么是shell

- 文字形式shell接口的优势

1. 功能完善，且各发行版都使用相同的bash
2. 远程管理，文字接口传输速度更快
3. 更好的管理主机

# 什么是shell

- 系统合法的shell与/etc/shell的功能

  由于早年Unix发展者众多，Linux系统中有多个shell程序，包括：

  1. Bourne Shell （sh）
  2. SUN: C Shell(csh)；
  3. K SHell；
  4. TCSH；
  5. Bourne Again SHell(bash).

  可以通过/etc/shells文件查看当前系统支持的shell

# 什么是shell

- 思考：当前使用者会使用哪个shell

# Bash shell的功能

- bash是GNU计划中的重要工具软件之一，是Linux预设的标准shell。bash兼容sh，并依据需求加强而产生，不论哪个发行版都需要学习bash。
- 主要功能（优势）：
  - 命令编修功能（history）
  - 命令与文件补全功能（tab）
  - 命令别名设定功能（alias）
  - 工作控制、前景背景控制（jobs）
  - 程序化脚本（shell scripts）
  - 通配符：（Wildcard）

# Bash shell的功能

- 为了方便shell的操作，bash内建了很多指令，比如cd、umask等等。
- 可以通过type指令查看指令类型

使用方式：type [-tpa] name

不加任何选项或参数时，type会显示出name是外部命令还是bash内建命令；

-t：显示命令的意义；

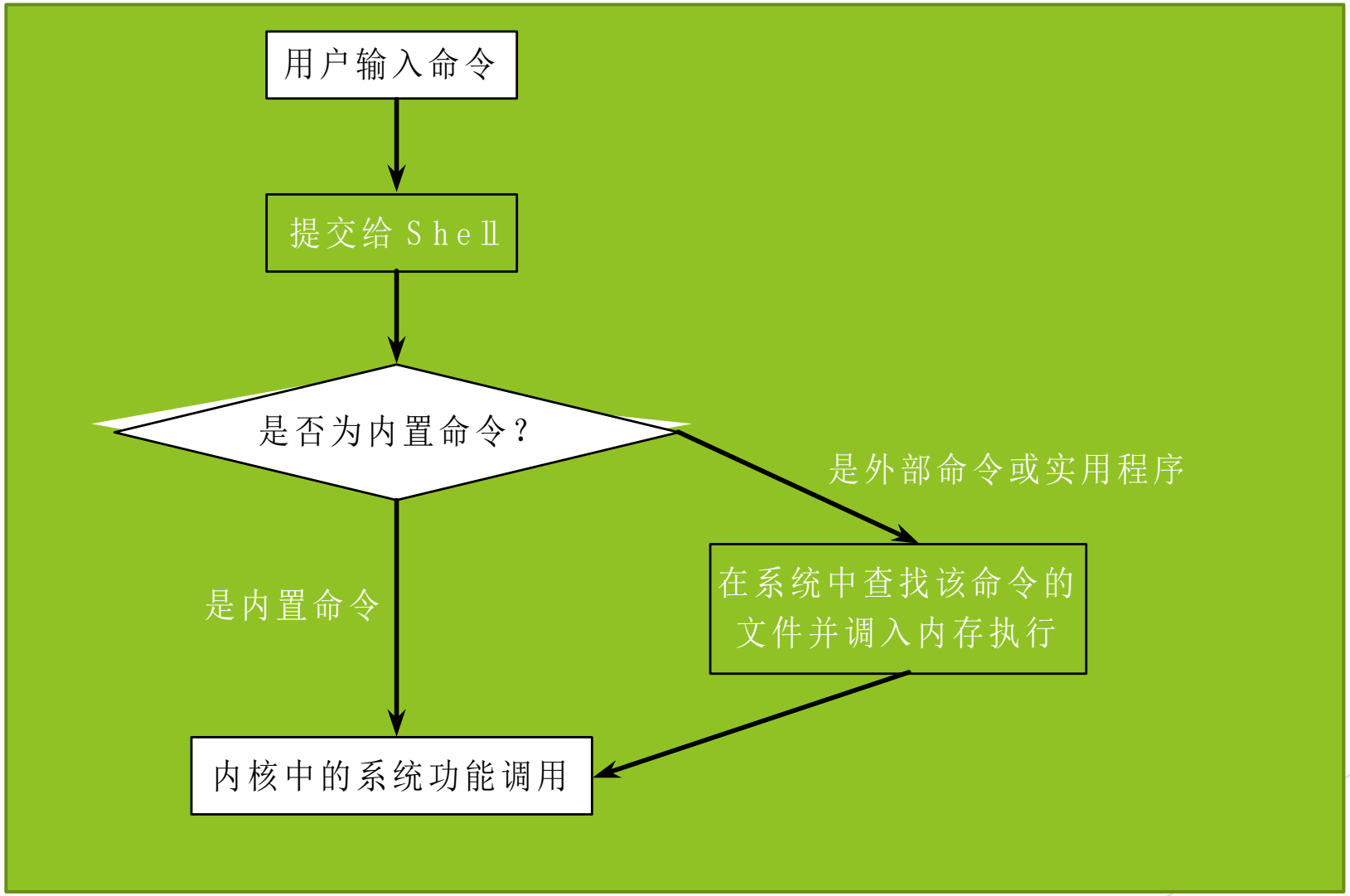
-p：如果name为外部命令，则显示完整文件名；

-a：根据PATH变量，将所有含有name的命令都进行罗列，包括别名。

file表示为外部命令；alias表示该命令为命令别名所配置的名称；builtin表示该命令为bash内建命令。

```
[liu@bogon maxproject]$ type ls
ls 是 `ls --color=auto' 的别名
[liu@bogon maxproject]$ type cd
cd 是 shell 内嵌
[liu@bogon maxproject]$ type python
python 是 /usr/bin/python
```

# Bash shell的功能



# 5.2 shell变量

# shell变量

- 什么是变量

用某一个特定字符串代表比较复杂或容易变动的内容，方便使用。

比如：y=ax+b

# shell变量

- shell变量的好处

1. 影响bash环境的变量

用户成功登录，使用shell，获得bash运行程序；系统通过一些变量来提供数据的存取，或一些环境的配置参数值；如PATH、HOME、MAIL等

2. 脚本程序中的变量

大型script中，某些变量常被使用，可以进行变量的声明，改变一行就能应用到整个脚本中。

# shell变量

- 变量的取用：echo

  利用echo读取变量，需要在名称前面加上$,或者以${变量}的方式来引用。

  例：echo ${variable}、echo $variable

  echo ${HOME}、echo $HOME

- 变量的设定：=

  利用等号（=）进行变量值的设定和修改

  例：var=abc

  **bash中，当一个变量尚未被设定时，也可访问，预设的内容是“空”的。

# shell变量

- 变量设定的规则

1. 变量与变量内容以一个等号”=”进行连接：

```
myname=teacher
```

2. 等号两边不能有空格符，错误示例：

```
myname = teacher
```

3. 变量名称只能是英文字母与数字，但是开始的字符不能是数字，错误示例：

```
2myname=teacher
```

# shell变量

4. 变量内容中如果有空格符，可以使用双引号”或者单引号’，将变量内容结合起来，但两者存在区别：

双引号内的特殊字符，如$等，可以保持原有的特性：

```
var="lang is $LANG”; echo $var
lang is zh_CN.UTF-8
```

单引号内的特殊字符

```
var='lang is $LANG'; echo $var
lang is $LANG
```

# shell变量

5. 可用斜杠”\”，将特殊符号（如$、空格符、’ 等）变成一般字符：

6. 其他命令的返回值作为变量值的情况，可以使用`cmd`或$(cmd)，如：

```
version=$(uname -r); echo $version
3.10.0-327.4.5.el7.x86_64
```

7. 如果需要增加变量的内容，则可以使用$var或${var}累加内容，如：

```
PATH=“$PATH”:/home/bin
```

# shell变量

8. 如果该变量需要运行与其他子程序，则需要以export来使变量成为环境变量：

```
export PATH
```

9. 通常大写字符为系统默认变量，自行配置的变量尽量使用小写字符，方便判断，非强制；

10. 取消变量使用unset，unset var，如：

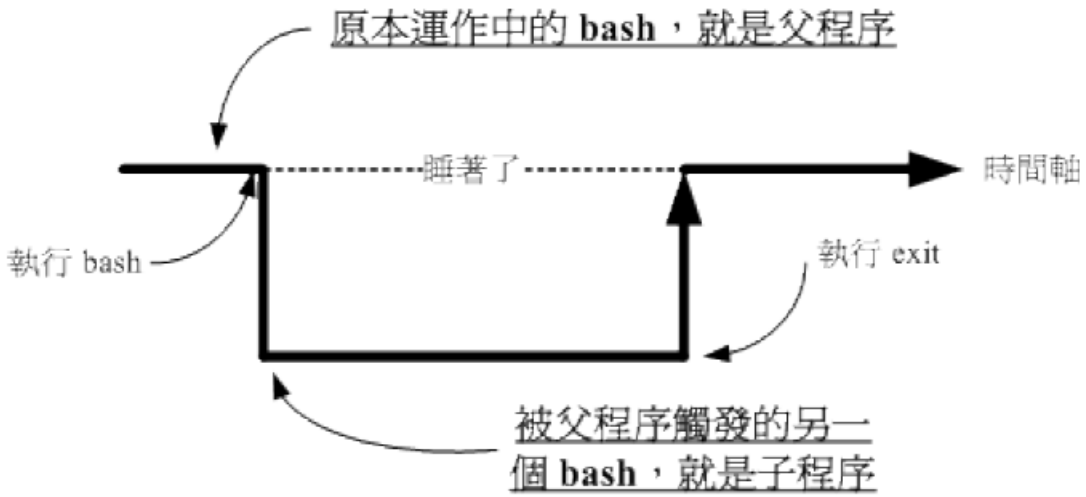
```
unset myname
```

# shell变量

- shell中的环境变量
  - 环境变量是指由Shell定义和赋初值的Shell变量，**能够被子程序所引用**
  - Shell用环境变量来确定查找路径、注册目录、终端类型、终端名称、用户名等。
  - 所有环境变量都是全局变量，并可以由用户重新设置。
  - 可以用env指令查看当前shell环境中的所有环境变量
  - 可以用export将自定义变量转换成环境变量

# shell变量

## 子程序的概念




如上所示，在原本的bash 底下执行另一个bash ，结果操作的环境接口会跑到第二个bash 去(就是子程序)，那原本的 bash 就会在暂停的情况。 整个指令运作的环境是实线的部分!若要回到原本的bash 去，就只有将第二个 bash 结束掉(下达exit 或logout) 才行。

# shell变量

- 常用环境变量

  PATH 决定了shell将到哪些目录中寻找命令或程序

  HOME 当前用户主目录

  HISTSIZE 历史记录数

  LOGNAME 当前用户的登录名

  HOSTNAME 指主机的名称

  SHELL 当前用户Shell类型

  LANGUGE 语言相关的环境变量，多语言可以修改此环境变量

  MAIL 当前用户的邮件存放目录

# shell变量

- 环境变量与自定义变量的差异

  环境变量会在子进程中可见，而自定义变量则不可见。

  将自定义变量变为环境变量

  export　变量

# shell变量

- 重要系统变量

  set命令查看所有变量，重要的系统变量包括：

  PS1：　基本提示符，对于root用户是#，对于普通用户是$

  $:当前shell的进程id

  ？：上一个指令所回传的值，指令执行成功回传0，发生错误回传非0错误代码。

\d ：可显示出『星期 月 日』的日期格式，如："Mon Feb 2"
\H ：完整的主机名。举例来说，鸟哥的练习机为『study.centos.vbird』
\h ：仅取主机名在第一个小数点之前的名字，如鸟哥主机则为『study』后面省略
\t ：显示时间，为 24 小时格式的『HH:MM:SS』
\T ：显示时间，为 12 小时格式的『HH:MM:SS』
\A ：显示时间，为 24 小时格式的『HH:MM』
\@ ：显示时间，为 12 小时格式的『am/pm』样式
\u ：目前使用者的账号名称，如『dmtsai』;
\v ：BASH 的版本信息，如鸟哥的测试主机版本为 4.2.46(1)-release，仅取『4.2』显示
\w ：完整的工作目录名称，由根目录写起的目录名称。但家目录会以 ~ 取代;
\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。
\# ：下达的第几个指令。
\$ ：提示字符，如果是 root 时，提示字符为 # ，否则就是 $ 啰~

# shell变量

- 读取来自键盘输入的变量read
  - 用法： read［选项］ 变量名

  选项：-p：后面可以接提示字符

  -t:接等待的秒数

```
[liu@bogon repo_git]$ read -p "input a number:"  num
input a number:12
[liu@bogon repo_git]$ echo $num
12
```

# shell变量

- 定义变量类型 declare

  用法： declare［选项］ 变量名

  选项：-a:将变量定义成数组

  -i:将变量定义成为整数数字

  -x:将变量变为环境变量

  -r：将变量设定为readonly

```
[liu@bogon repo_git]$ sum=1+2+3
[liu@bogon repo_git]$ echo $sum
1+2+3
[liu@bogon repo_git]$ declare
gdb/          make/         multifile/    test.c
.git/         mfile/        standard_dir/
[liu@bogon repo_git]$ declare
gdb/          make/         multifile/    test.c
.git/         mfile/        standard_dir/
[liu@bogon repo_git]$ declare -i sum=1+2+3
[liu@bogon repo_git]$ echo $sum
6
```

# 5.3 命令的别名与历史命令

# 命令的别名与历史命令

- 别名的意义：
  1. 简化比较长的惯用命令；
  2. 限制导致严重后果命令的执行；
  3. 支持用户使用习惯。

```
[root@thispc make]# alias
alias cp='cp -i'
alias grep='grep --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias mv='mv -i'
alias rm='rm -i'
```

# 命令的别名与历史命令

- 别名的使用：
  - 为命令添加别名：

    alias 别名=‘指令 选项’
  - 查看有哪些命令别名：

    alias
  - 取消命令别名

    unalias 别名

# 命令的别名与历史命令

- 查看历史命令history：

  用法：

  history [选项]

  参数/选项：

  n：数字，列出最近n条历史命令；

  -c：将目前shell的所有历史命令清除；

  -a：将目前新增的history命令追加到histfiles中；如果没有指定histfiles，默认写入~/.bash_history；

  -r：将histfiles中的内容读到目前shell的histroy中；

  -w：将目前history所记录的内容写入histfiles。

# 命令的别名与历史命令

- 历史命令读取和记录的过程:
  - 历史命令读取和记录的过程
    - 登陆，从~/.bash_history读取;
    - 数量: HISTSIZE;
    - 登出: 更新~/.bash_history;
    - history -w 可强制写入。

# Bash shell的操作环境

- 利用”!”重复执行历史命令:

  用法:

  !number：执行历史命令中的第number调指令

  !command：由最近指令向前搜索，找到开头为“command的指令”，并执行

  !!: 执行上一条指令

# 5.4 bash shell的操作环境

# Bash shell的工作环境

- 路径与命令搜索顺序：

  在bash shell环境中，下达指令后的搜索顺序为：

  （1）以相对/绝对路径执行指令，例如 bin/ls 或 ./ls

  （2）由alias找到该指令来执行

  （3）由bash内建的指令来执行

  （4）透过$PATH这个变量的顺序搜寻到的第一个指令来执行

```
[root@thispc ~]# type -a ls
ls 是 `ls --color=auto' 的别名
ls 是 /usr/bin/ls
ls 是 /bin/ls
```

# Bash shell的操作环境

- bash进站欢迎信息的设置：

  更改配置文件/etc/issue，显示在登录之前

  ① \d 本地端时间的日期；

  ② \l 显示第几个终端机接又；

  ③ \m 显示硬件的等级（i386/i486/i586/i686..）；

  ④ \n 显示主机的网络名称；

  ⑤ \o 显示 domain name；

  ⑥ \r 操作系统的版本（相当于 uname -r）

  ⑦ \t 显示本地端时间的时间；

  ⑧ \s 操作系统的名称；

  ⑨ \v 操作系统的版本

  更改配置文件/etc/motd，成功登录后显示信息

# Bash shell的操作环境

- bash的环境配置文件：

  ①login与non-login shell

  login shell：取得bash时需要完整的登陆流程的，就称为loginshell。举例来说，要由 tty1 ~ tty6 登陆，需要输入用户的账号与密码，此时取得的 bash 就称为『 login shell 』；

  non-login shell：取得 bash 接又的方法不需要重复登陆的举动，举例来说，(1)以图形化操作界面登陆Linux后， 再以图形化接又启动终端，此时那个终端接又并没有需要再次的输入账号与密码，那个bash的环境就称为non-login shell。(2)你在原本的bash环境下再次下达 bash 这个命令，同样的也没有输入账号密码， 那第二个bash(子程序) 也是non-login shell。

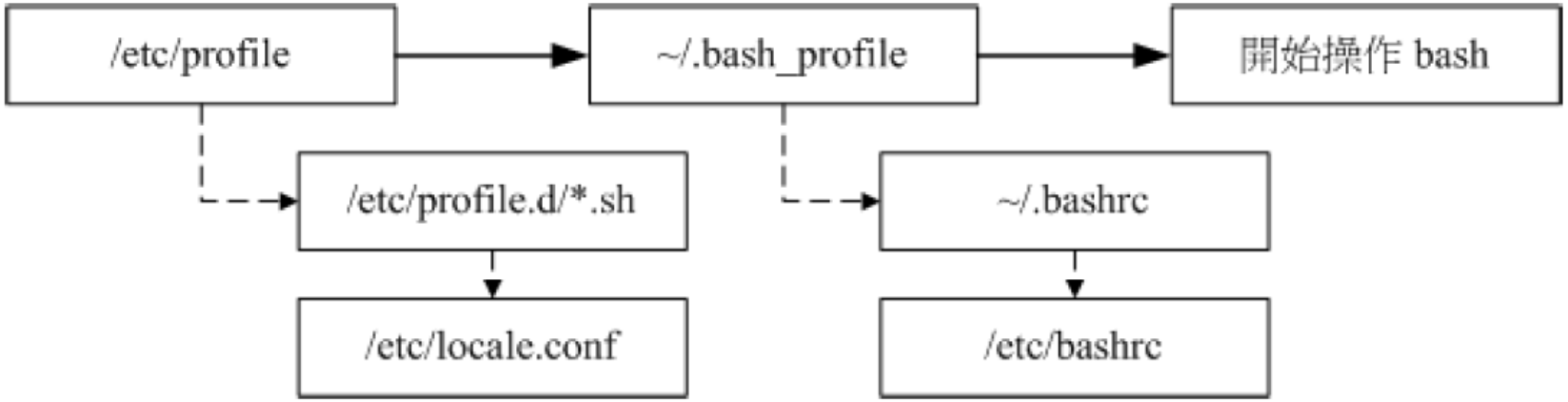
# Bash shell的操作环境

- login shell读取的配置文件

（1）/etc/profile:用于系统的整体设定，不建议修改

（2）~/.bash_profile或~/.bash_login或~/.profile：用于个人设定，可以修改自己的配置信息。

登录后，先读整体配置文件/etc/profile, 然后依序选择读取一个个人配置文件。



# Bash shell的操作环境

- 读入环境配置文件

/etc/profile， ~/.bash_profile 都是在取得login shell的时候才会读取的配置文件，所以添加后需要注销登录才能生效。

可用source 命令或 ‘.’ ，使配置文件立即生效。

用法： source 或 . 配置文件名

例: source ~/.bashrc

. ~/.bashrc

# Bash shell的操作环境

- non-login shell读取的配置文件
  - （1）~/.bashrc：会调用/etc/bashrc
  - （2）/etc/bashrc

# Bash shell的操作环境

- 终端环境配置

在登录终端后，可以使用退格删除、ctr+c强行终止命令运行，可以通过stty查看当前使用环境中的按键设置。

命令: stty [-a]

功能：列出目前所有的按键与内容

[root@thispc ~]# stty -a

speed 38400 baud; rows 24; columns 80; line = 0;

intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = M-^?; eol2 = M-^?;

# Bash shell的操作环境

- 终端环境配置

set除显示变量外，还可以通过set设定整个指令的输入/输入环境。

命令: set [-uvCHhmBx]

参数：

-u:启用后，当使用未设定变量是，会显示错误信息；

-v:启用后，在信息被输出前，会先显示信息的原始内容；

-x：启用后，指令被执行前，会先显示指令内容

例： [root@thispc ~]# set -u

[root@thispc ~]# echo $abc

bash: abc: 为绑定变量

# Bash shell的操作环境

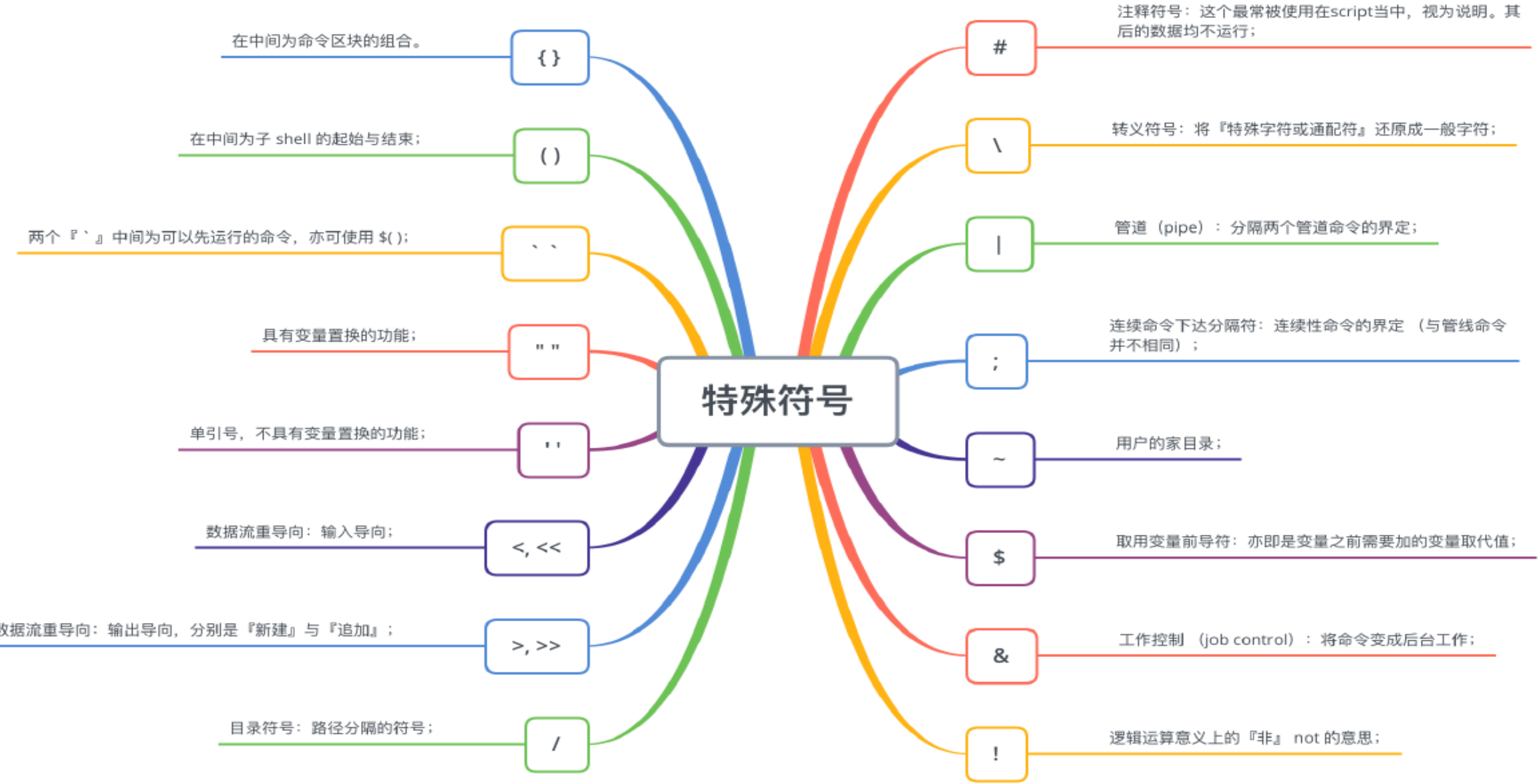
- 通配符与特殊符号

| | 运行结果 |
|---|---|
| Ctrl+C | 终止当前的命令； |
| Ctrl+D | 输入结束（EOF），例如邮件结束的时候； |
| Ctrl+M | 相当于Enter； |
| Ctrl+S | 暂停屏幕的输出； |
| Ctrl+Q | 恢复屏幕的输出； |
| Ctrl+U | 在提示字符下，将整列命令删除； |
| Ctrl+Z | 暂停目前的命令。 |

# Bash shell的操作环境

- 通配符与特殊符号

| | 意义 |
|---|---|
| * | 代表“0到无穷多个”任意字符； |
| ? | 代表“一定有一个”任意字符； |
| [] | “一定有一个括号内”的字符（非任意）。例如：[abc]表示一定有一个字符，可能是a、b、c中任何一个； |
| [-] | 如果有减号在中括号中，表示“在编码顺序内的所有字符”。例如：[0-9]代表0到9之间的所有数字； |
| [^] | 如果中括号内的第一个字符为指数符号（^），则代表“反向选择”，例如：[^abc]表示一定有一个字符，是a、b、c之外的其它字符。 |


特殊符号
{ }
在中间为命令区块的组合。
( )
在中间为子 shell 的起始与结束；
` `
两个『 ` 』中间为可以先运行的命令，亦可使用 $( )；
" "
具有变量置换的功能；
' '
单引号，不具有变量置换的功能；
<, <<
数据流重导向：输入导向；
>, >>
数据流重导向：输出导向，分别是『新建』与『追加』；
/
目录符号：路径分隔的符号；
#
注释符号：这个最常被使用在script当中，视为说明。其后的数据均不运行；
\
转义符号：将『特殊字符或通配符』还原成一般字符；
|
管道（pipe）：分隔两个管道命令的界定；
;
连续命令下达分隔符：连续性命令的界定 （与管线命令并不相同）；
~
用户的家目录；
$
取用变量前导符：亦即是变量之前需要加的变量取代值；
&
工作控制 （job control）：将命令变成后台工作；
!
逻辑运算意义上的『非』 not 的意思；

# 实践：安装Java

1.下载JDK

https://www.oracle.com/java/technologies/downloads/#jdk18-linux

命 令 ： wget https://download.oracle.com/java/18/latest/jdk-18_linux-x64_bin.tar.gz

2.创建安装目录 并解压到该目录

/usr/local/java

3.设置环境变量

vim /etc/profile

export JAVA_HOME=/usr/local/java/jdk1.8.0_171

export JRE_HOME=${JAVA_HOME}/jre

export CLASSPATH=.:${JAVA_HOME}/lib:${JRE_HOME}/lib

export PATH=${JAVA_HOME}/bin:$PATH

# 5.5 输入输出重定向

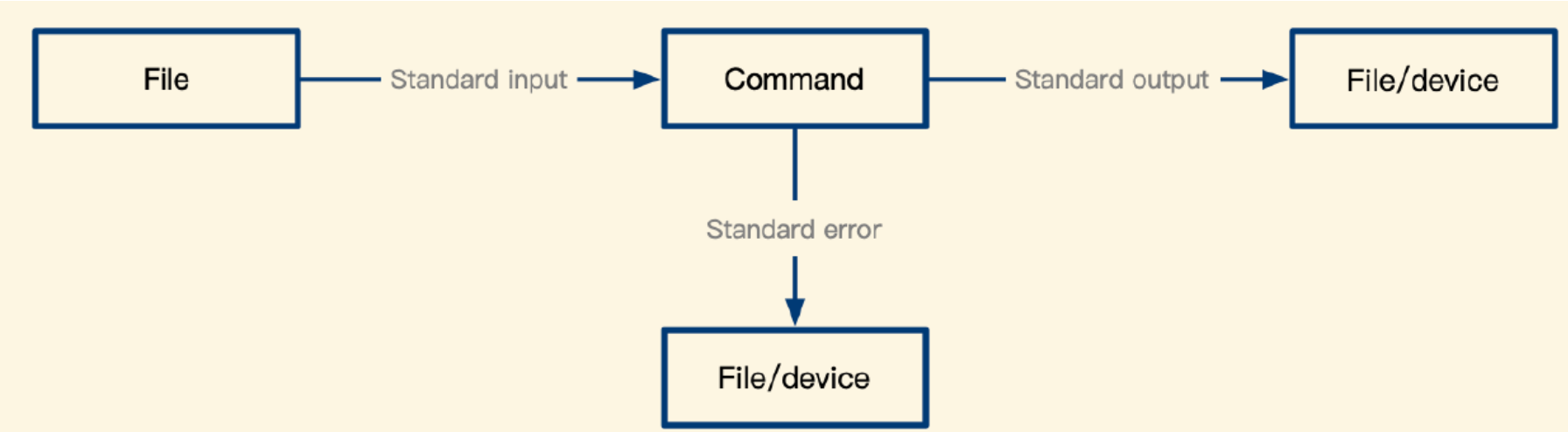
# 输入输出重定向

- 输入输出重定向

所谓重定向，就是不使用系统的标准输入端口、标准输出端口或标准错误端口，而进行重新的指定。

输入重定向：重新指定设备\文件来代替键盘作为新的输入；

输出重定向：重新指定设备\文件来代替显示器作为新的输出。



# 输入输出重定向

- 输出重定向实现

  标准输出（stdout）：命令运行所返回的正确信息；代码为1，使用>或>>;

  标准错误输出（stderr）：命令运行失败后，所返回的错误信息。代码为2，使用2>或2>>;

# 输入输出重定向

- 命令 > 文件：将命令执行的标准输出结果重定向到指定的文件中，如果文件已包含数据，会清空原有数据，再写入新数据。
- 命令 >> 文件:将命令执行的标准输出结果重定向到指定的文件中，如果该文件已包含数据，新数据将写入到原有内容的后面。
- 命令 2> 文件:将命令执行的错误输出结果重定向到指定的文件中，如果文件中已包含数据，会清空原有数据，再写入新数据。
- 命令 2>> 文件：将命令执行的错误输出结果重定向到指定的文件中，如果该文件中已包含数据，新数据将写入到原有内容的后面。
- 命令 >> 文件 2>&1或者 命令 &>> 文件：将标准输出或者错误输出写入到指定文件，如果该文件中已包含数据，新数据将写入到原有内容的后面。注意，第一种格式中，最后的 2>&1 是一体的，可以认为是固定写法。
- 特殊的输出重定向位置/dev/null

  不想显示也不想存储的信息，可以重定向到/dev/null，可以看作回收站。

# 输入输出重定向

- 输入重定向实现

  标准输入(stdin): 代码为0，使用<或<<;

  命令 < 文件: 将指定文件作为命令的输入设备

  命令 << 分界符: 表示从标准输入设备（键盘）中读入，直到遇到分界符才停止（读入的数据不包括分界符），这里的分界符其实就是自定义的字符串

  命令 < 文件 1 > 文件 2: 将文件 1 作为命令的输入设备，该命令的执行结果输出到文件 2 中。

  常用cat：cat >> new.txt < result.txt

  cat >> result.txt << “eof”

# 输入输出重定向

- 数据流重定向举例

① 将ls命令生成的/tmp目录的一个清单存到当前目录中的dir文件中。

$ ls –l  /tmp >dir

② 将ls命令生成的/tmp目录的一个清单以追加的方式存到当前目录中的dir文件中。

$ ls –l /tmp >>dir

③ 将passwd文件的内容作为wc命令的输入。

$ wc</etc/passwd

④ 将命令myprogram的错误信息保存在当前目录下的err_file文件中。

$ myprogram 2> err_file

⑤ 将命令myprogram的输出信息和错误信息保存在当前目录下的output_file文件中。

$ myprogram &>output_file

⑥ 将命令ls的错误信息保存在当前目录下的err_file文件中。

$ ls –l  2>err_file

# 用途

- 屏幕输出的信息很重要，而且我们需要将他存下来的时候；
- 背景执行中的程序，不希望他干扰屏幕正常的输出结果时；
- 一些系统的例行命令(例如写在/etc/crontab 中的文件)的执行结果，希望他可以存下来时；
- 一些执行命令的可能已知错误讯息时，想以『2>/dev/null 」将他丢掉时；
- 错误信息与正确信息需要分别输出时。

# 5.6 多命令间的逻辑关系

# 命令执行的判断依据

- 某些情况下，需要一次执行多条命令；
- 常见：sync；sync；shutdown -h

# 命令执行的判断依据

- 根据命令间的逻辑关系有以下几种形式：

  （1）cmd;cmd;　　指令间不存在相关性

  （2）cmd1 && cmd2 cmd1执行正确则执行cmd2，否则不执行cmd2

  （3）cmd1 || cmd2 cmd1执行错误则执行cmd2，否则不执行cmd2

  判断命令执行正确与否，依据变量$?的值，为0表示正确，否则表示不正确。

# 命令执行的判断依据

例1：[root@thispc ~]# ls /tmp/abc && touch /tmp/abc/hehe

ls: 无法访问/tmp/abc: 没有那个文件或目录

例2：[root@thispc ~]# ls /tmp/abc || mkdir /tmp/abc

ls: 无法访问/tmp/abc: 没有那个文件或目录

[root@thispc ~]# ls /tmp

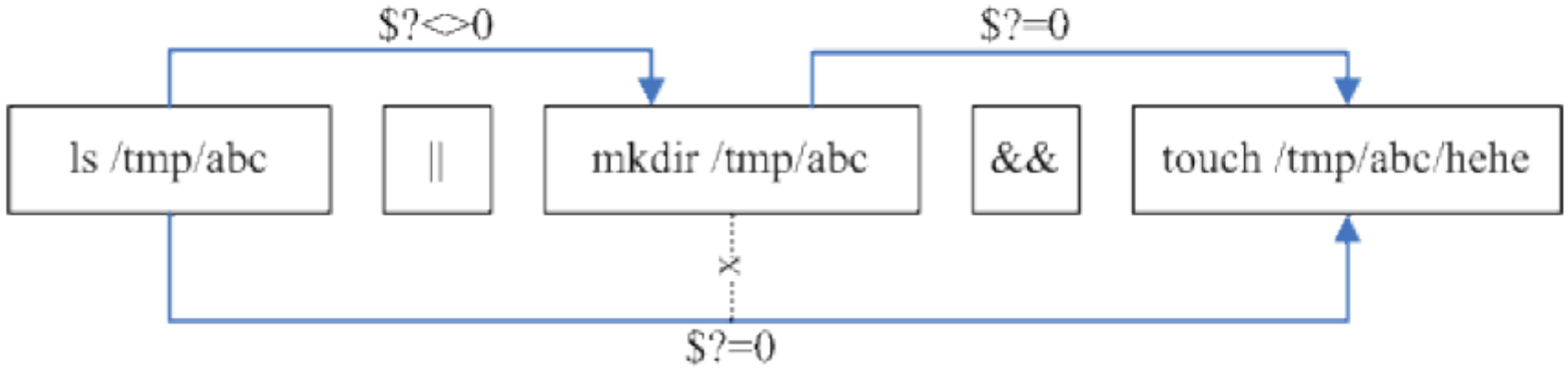
abc

例3： [root@thispc ~]# ls /tmp/abc && touch /tmp/abc/hehe

ls: 无法访问/tmp/abc: 没有那个文件或目录

# 命令执行的判断依据

- ls /tmp/abc || mkdir /tmp/abc && touch /tmp/abc/hehe




例：测试/tmp/test 是否存在，若存在则显示“exist”，不存在则显示“not exist”

# 5.7 管道命令

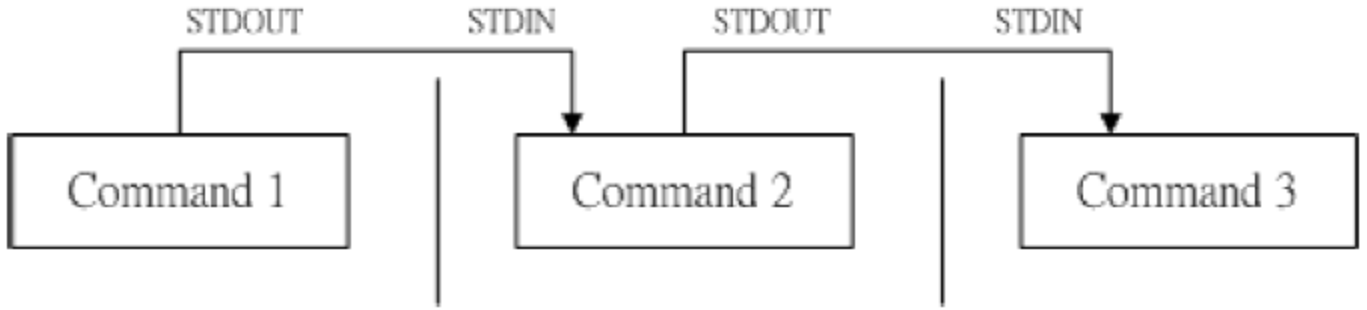
# 管道命令

- 什么是管道命令

许多Linux命令具有过滤特性，即一条命令执行后产生的结果数据可通过标准输出端口送给后一条命令执行，作为该命令的输入数据，经过连续多次处理得到结果。

Shell提供管道命令“|”将这些命令前后衔接在一起，形成一个管道线。格式为：命令1|命令2|……|命令n

例：ls /etc的输出内容很多，可以使用 ls /etc | more，分屏显示ls /etc的结果。



# 管道命令

- 管道命令的限制
  - 管道命令仅处理标准输出，忽略标准错误输出；
  - 管道命令必须能够接收来自另一个指令的数据作为其标准输入继续处理才可以。

常用的管道命令：more、less、cut、sort、grep、，而ls、 cp、mv等不能接收前一个指令的数据，不是管道命令

# 管道命令

- 常用管道命令（截取）
- 1. cut命令

语法：cut [选项] [file 或 标准输入]

说明：用于将同一行的数据进行切分

参数：-d ： 自定义分隔符，默认为制表符。

-f ： 与-d一起使用，指定显示哪个区域，逗号分隔

-c ： 以字符为单位进行分割， 5-10

```
[root@thispc ~]# echo ${PATH} | cut -d ':' -f 3
/usr/bin
```

# 管道命令

- 常用管道命令（截取）
- 2. grep命令

  语法：grep［选项］查找模式［文件名1或标准输入］

  说明：grep 指令用于查找内容包含指定的范本样式的文件，如果发现某文件的内容符合所指定的范本样式，预设 grep 指令会把含有范本样式的那一列显示出来。

  选项： -v：列出不匹配的行。

  -c：对匹配的行计数。

  -l：只显示包含匹配模式的文件名。

  -h：抑制包含匹配模式的文件名的显示。

  -n：每个匹配行只按照相对的行号显示。

  -i：对匹配模式不区分大小写。

```
[root@thispc ~]#ls  -al /etc | grep ssh
drwxr-xr-x.   2 root root 225 1月 11 2020 ssh
```

# 管道命令

- 常用管道命令（排序）
- 1. sort命令

  语法：sort［选项］ ［文件或标准输入］

  说明：Linux sort命令用于将文本文件内容加以排序。针对文本文件的内容，以行为单位来排序。

# 管道命令

- 1. sort命令

  常用参数：

  参数： -b 忽略每行前面开始出的空格字符。

  -f 排序时，将小写字母视为大写字母。

  -M 将前面3个字母依照月份的缩写进行排序。

  -n 依照数值的大小排序。

  -u 意味着是唯一的(unique)，输出的结果是去完重了的。

  -o<输出文件> 将排序后的结果存入指定的文件。

  -r 以相反的顺序来排序。

  -t<分隔字符> 指定排序时所用的栏位分隔字符。

# 管道命令

- 1. sort命令

范例1：将/etc/passwd中的内容以：来分隔，并按照第3栏排序

```
cat /etc/passwd | sort -t ':' -k 3
```

范例2：将/etc/passwd中的内容以：来分隔，并按照第3栏数字大小排序

```
cat /etc/passwd | sort -t ':' -nk 3
```

# 管道命令

- 常用管道命令（排序）
- 2.uniq命令

语法：uniq [-ic]

说明：uniq 命令用于检查及删除文本文件中重复出现的行列，一般与 sort 命令结合使用。

参数：-i：忽略大小写字符的不同；

-c：进行计数；

例：Last账号列出，取出账号栏，进行排序后进去出一位

last | cut -d ‘’ -f1 | sort | uniq

# 管道命令

- 常用管道命令（双向重导向）

输出从定向”>”将输出重定向到文件，如果想要保存到文件的同时，输出执行结果，可以用双向重导向。

- tee命令

语法：tee [-a] file

说明：tee指令可以前条指令的结果输出到标准输出设备，同时保存成文件。

参数：-a：以累加的方式，将数据加入到file中

例： ls -l / | tee -a myfile | more

# 管道命令

- 常用管道命令（字符串转换）
- 1.tr命令

语法：tr [-ds] 字符串

说明：tr命令可以读取命令输出，经过字符串转译后，将结果输出到标准输出设备。

选项：

-d：删除信息中的字符串

-s:取代重复的字符

例： last | tr ‘[a-z]’ ‘[A-Z]’ # 替换last输出信息中的

cat /etc/psswd | tr -d ‘:’ # 删除输出内容中的“：”

# 管道命令

- 常用管道命令（字符串转换）
- 2.col命令

  语法：col [-xb]

  说明：col命令可以用来过滤控制字符，通常用来将【tab】转成空白。

  选项：

  -x：将tab键转换成对等的空格键

例：　cat -A /etc/man_db.conf # 显示所有特殊按键

　　cat /etc/man_db.conf | col -x | cat -A | more # 替换tab

# 管道命令

- 常用管道命令（分区命令）
- split

用法：split [-bl] file PREFIX

说明：切分大文件

参数：

-b:后面可接欲区分成的文件大小，可加单位，如b，k，m等

-l:以行数来进行分区

PREFIX:代表前导符的意思，可作为分区文件的前导文字

例：/etc/service文件分成300k一个

```
cd /tmp; split  -b 300k /etc/services services
```

合并：cat services*  >> serviceback

```
ls -al /etc | split -l 10 - etc  #每10条记录一个文件
```

# 管道命令

- 常用管道命令（分区命令）
- xargs

用法：xargs [-0epn] command

说明：产生某个指令的参数，读入stdin的数据，以空格字符或断行字符将文件分割成argument。

参数：

-0:将stdin的特殊字符还原成一般字符；

-e:end of file，后面可以接字符串，xargs分析到这个字符串时，会停止工作。

-p：在执行每个指令的argument时，都会询问使用者

-n:后面接次数，每次command指令执行时，需要使用几个参数的意思

# 管道命令

1. 每次执行id，传入一个参数

```
cut -d ':' -f 1 /etc/passwd | head -n 3 | xargs -n 1 id
```

2. 查到sync就指令结束

```
cut -d ':' -f 1 /etc/passwd | xargs -e 'sync' -n 1 id
```

3. 找到后，列出详细信息，ls不支持管道，可以用xargs

```
find /usr/sbin -perm /700 | xargs ls -l
```

# 练习

- 1. 取PATH环境变量的第三个字段
- 2. 取PATH环境变量的第三个字段，并查看该目录下文件的详细信息

# 5.7 正则表达式

# 正则表达式

- 正则表达式是处理字符串的方法，他是以行为单位来进行字符串处理的行为，通过一些特殊符号的辅助，可以让使用者轻易的达到搜索/删除/取代某特定字符串的处理程序。
- 正则表达是一种表示方法，工具程序支持这种表示方法，则可以用正则表达式来进行字符串的处理。例如：vi、grep、awk、sed等

# 正则表达式

- 用途：
- 分析日志、简单的垃圾邮件过滤、软件（系统）配置等等

# 正则表达式

- 正则表达式拥有不同的规范，POSIX规范，Perl规范，Python规范等，Shell中的grep、egrep都使用POSIX规范。POSIX规范包括：

1. 基本的正则表达式（Basic Regular Expression 又叫Basic RegEx 简称BREs）

2. 扩展的正则表达式（Extended Regular Expression 又叫Extended RegEx 简称EREs）

# 基本正则表达式

- 基本正则表达式常用字符

| 正则表达式 | 描述 |
| --- | --- |
| \ | 转义符号，将特殊字符转义 |
| ^ | 匹配行首 |
| $ | 匹配行尾 |
| * | 匹配0到无穷多个任意字符 |
| . | 匹配除换行符\n之外的任意单个字符 |
| [] | 匹配包含在[字符]之中的任意一个字符 |
| [^] | 匹配[^字符]之外的任意一个字符 |
| [-] | 匹配[]中指定范围内的任意一个字符，要写成递增 |
| \{n,m\} | 匹配 连续n到m个前一个 字符 |

# 基本正则表达式

- POSIX字符

POSIX字符类是一个形如[:...:]的特殊元序列（meta sequence），他可以用于匹配特定的字符范围。

[:alnum:] 匹配任意一个字母或数字字符
[:alpha:] 匹配任意一个字母字符（包括大小写字母）
[:blank:] 空格与制表符（横向和纵向）
[:digit:] 匹配任意一个数字字符
[:lower:] 匹配小写字母
[:upper:] 匹配大写字母
[:punct:] 匹配标点符号
[:space:] 匹配一个包括换行符、回车等在内的所有空白符
[:graph:] 匹配任何一个可以看得见的且可以打印的字符
[:xdigit:]任何一个十六进制数（即：0-9，a-f，A-F）
[:cntrl:] 任何一个控制字符（ASCII字符集中的前32个字符）
[:print:] 任何一个可以打印的字符

# 基本正则表达式

- 搜索指定字符串（以grep命令，搜索regular.txt文件为例）

grep -n ‘the’ man_db.conf

```
[liu@bogon ~]$ grep -n the man_db.conf
3:# This file is used by the man-db package to configure the man and cat paths.
5:# their PATH environment variable. For details see the manpath(5) man page.
16:# every automatically generated MANPATH includes these fields
45:# an optional extra string indicating the catpath associated with the
46:# manpath. If no catpath string is used, the catpath will default to the
51:# they exist, otherwise the permissions of the user running man/mandb will
```

"Open Source" is a good mechanism to develop programs.
apple is my favorite food.
Football game is not use feet only.
this dress doesn't fit me.
However, this dress is about $ 3183 dollars.
GNU is free air not free beer.
Her hair is very beauty.
I can't finish the test.
Oh! The soup taste good.
motorcycle is cheap than car.
This window is clear.
the symbol '*' is represented as start.
Oh!  My god!
The gd software is a library for drafting programs.
You are the best is mean you are the no. 1.
The world <Happy> is the same with "glad".
I like dog.
google is the best tools for search keyword.
goooooogle yes!
go! go! Let's go.
# I am VBird

# 基本正则表达式

- 含有元字符的搜索 （以regular.txt为例）

1. grep -n 't[ae]st' regular.txt
2. grep -n '[^g]oo' regular.txt
3. grep -n '[^a-z]oo' regular.txt
4. grep -n '[^[:lower:]]oo' regular.txt
5. grep -n '[^[:digit:]]' regular.txt

# 基本正则表达式

- 指定开始、结尾字符串的搜索 （以grep命令，搜索regular.txt文件为例）

1. grep -n ‘^the’ regular.txt
2. grep -n ‘^[a-z]’ regular.txt
3. grep -n ‘^[[:lower:]]’ regular.txt
4. grep -n ‘^[[:alpha:]]’ regular.txt
5. grep -n ‘\.$’ regular.txt
6. grep -n ‘^$’ regular.txt
7. grep -v ‘^$’ regular.txt | grep -v ‘^#’

# 基本正则表达式

- 含有任意、重复字符的字符串搜索 （以grep命令，搜索regular.txt文件为例）

  .:一定有一个任意字符

  *：重复前一个字符，0到多次

例：

```
grep -n 'g..d' regular.txt
grep -n 'o*' regular.txt
grep -n 'oo*' regular.txt
grep -n 'g*g' regular.txt
grep -n 'g.*g' regular.txt
```

# 基本正则表达式

- 给定字符范围的字符串搜索 （以grep命令，搜索regular.txt文件为例）

‘{}’能够指定字符个数，但‘{}’在shell中有特殊意义，因此需要使用转移字符‘\’。

```
grep -n ‘o\{2\}’ regular.txt
grep -n ‘o\{2,5\}’ regular.txt
```

# 正则表达式与通配符的区别

通配符是bash 操作接口的一个功能

正则表达式是一种字符串处理的表示方式。

例如：

通配符中，*代表0~无限多个字符， 而正则中*标识重复前一个字符多次

通配符中,?表示任意一个字符， 正则中则用’.’ 表示

# 总结

| RE 字符 | 意义与范例 |
| --- | --- |
| ^word | 意义：待搜寻的字符串 (word) 在行首！<br>范例：搜寻行首为 # 开始的那一行，并列出行号<br>**grep -n '^#' regular_express.txt** |
| word$ | 意义：待搜寻的字符串 (word) 在行尾！<br>范例：将行尾为 ！ 的那一行打印出来，并列出行号<br>**grep -n '!$' regular_express.txt** |
| . | 意义：代表『一定有一个任意字符』的字符！<br>范例：搜寻的字符串可以是 (eve) (eae) (eee) (e e)， 但不能仅有 (ee) ！亦即 e 与 e 中间『一定』仅有一个字符，而空格符也是字符！<br>**grep -n 'e.e' regular_express.txt** |
| \ | 意义：跳脱字符，将特殊符号的特殊意义去除！<br>范例：搜寻含有单引号 ' 的那一行！<br>**grep -n \' regular_express.txt** |
| * | 意义：重复零个到无穷多个的前一个 RE 字符<br>范例：找出含有 (es) (ess) (esss) 等等的字符串，注意，因为 * 可以是 0 个，所以 es 也是符合带搜寻字符串。另外，因为 * 为重复『前一个 RE 字符』的符号， 因此，在 * 之前必须要紧接着一个 RE 字符喔！例如任意字符则为 『.*』 ！<br>**grep -n 'ess*' regular_express.txt** |

# 总结

| | |
|---|---|
| [list] | 意义：字符集合的 RE 字符，里面列出想要撷取的字符！<br>范例：搜寻含有 (gl) 或 (gd) 的那一行，需要特别留意的是，在 [] 当中『谨代表一个待搜寻的字符』，例如『 a[afl]y 』代表搜寻的字符串可以是 aay, afy, aly 即 [afl] 代表 a 或 f 或 l 的意思！<br>**grep -n 'g[ld]' regular_express.txt** |
| [n1-n2] | 意义：字符集合的 RE 字符，里面列出想要撷取的字符范围！<br>范例：搜寻含有任意数字的那一行！需特别留意，在字符集合 [] 中的减号 - 是有特殊意义的，他代表两个字符之间的所有连续字符！但这个连续与否与 ASCII 编码有关，因此，你的编码需要设定正确(在 bash 当中，需要确定 LANG 与 LANGUAGE 的变量是否正确！) 例如所有大写字符则为 [A-Z]<br>**grep -n '[A-Z]' regular_express.txt** |
| [^list] | 意义：字符集合的 RE 字符，里面列出不要的字符串或范围！<br>范例：搜寻的字符串可以是 (oog) (ood) 但不能是 (oot) ，那个 ^ 在 [] 内时，代表的意 |
| | 义是『反向选择』的意思。例如，我不要大写字符，则为 [^A-Z]。但是，需要特别注意的是，如果以 grep -n [^A-Z] regular_express.txt 来搜寻，却发现该文件内的所有行都被列出，为什么？因为这个 [^A-Z] 是『非大写字符』的意思，因为每一行均有非大写字符，例如第一行的 "Open Source" 就有 p,e,n,o.... 等等的小写字<br>**grep -n 'oo[^t]' regular_express.txt** |

# 总结

| | |
|---|---|
| | |
| \{n,m\} | 意义：连续 n 到 m 个的『前一个 RE 字符』<br>意义：若为 \{n\} 则是连续 n 个的前一个 RE 字符，<br>意义：若是 \{n,\} 则是连续 n 个以上的前一个 RE 字符！ 范例：在 g 与 g 之间有 2 个到 3 个的 o 存在的字符串，亦即 (goog)(gooog)<br>**grep -n 'go\{2,3\}g' regular_express.txt** |

# 扩展正则表达式

- grep 仅支持基础正则表达式，如果要使用扩展正则表达式，一般使用egrep，或 grep -E。

| 正则表达式 | 描述 |
| --- | --- |
| + | 匹配之前的项1次或者多次； |
| ? | 匹配之前的项1次或者0次； |
| () | 匹配表达式，创建一个用于匹配的子串； 例如：mat（tri）？ |
| \| | 交替匹配\|两边的任意一项（或的 方式） |
| ()+ | 匹配多个重复 字符串 |

# 扩展正则表达式

- 例如：

标准正则：

```
grep -v '^$'  regular_express.txt | grep -v  '^#'
```

用扩展正则表达式替换为：

```
egrep -v '^$|^#' regular_express.txt
```

# 总结

| RE 字符 | 意义与范例 |
| --- | --- |
| + | 意义：重复『一个或一个以上』的前一个 RE 字符<br>范例：搜寻 (god) (good) (goood)... 等等的字符串。 那个 o+ 代表『一个以上的 o 』所以，底下的执行成果会将第 1, 9, 13 行列出来。<br>**egrep -n 'go+d' regular_express.txt** |
| ? | 意义：『零个或一个』的前一个 RE 字符<br>范例：搜寻 (gd) (god) 这两个字符串。 那个 o? 代表『空的或 1 个 o 』所以，上面的执行成果会将第 13,14 行列出来。 有没有发现到，这两个案例('go+d' 与 'go?d')的结果集合与 'go*d' 相同？想想看，这是为什么喔！ ^_^<br>**egrep -n 'go?d' regular_express.txt** |
| \| | 意义：用或( or )的方式找出数个字符串<br>范例：搜寻 gd 或 good 这两个字符串，注意，是『或』！ 所以，第 1,9,14 这三行都可以被打印出来喔！那如果还想要找出 dog 呢？<br>**egrep -n 'gd\|good' regular_express.txt** |

# 总结

| | |
|---|---|
| () | 意义：找出『群组』字符串<br>范例：搜寻 (glad) 或 (good) 这两个字符串，因为 g 与 d 是重复的，所以，我就可以将 la 与 oo 列于 ( ) 当中，并以 | 来分隔开来，就可以啦！<br>**egrep -n 'g(la\|oo)d' regular_express.txt** |
| ()+ | 意义：多个重复群组的判别<br>范例：将『AxyzxyzxyzxyzC』用 echo 叫出，然后再使用如下的方法搜寻一下！<br>**echo 'AxyzxyzxyzxyzC' \| egrep 'A(xyz)+C'**<br>上面的例子意思是说，我要找开头是 A 结尾是 C ，中间有一个以上的 "xyz" 字符串的意思～ |

# 总结

| | |
|---|---|
| () | 意义：找出『群组』字符串<br>范例：搜寻 (glad) 或 (good) 这两个字符串，因为 g 与 d 是重复的，所以，我就可以将 la 与 oo 列于 ( ) 当中，并以 | 来分隔开来，就可以啦！<br>**egrep -n 'g(la\|oo)d' regular_express.txt** |
| ()+ | 意义：多个重复群组的判别<br>范例：将『AxyzxyzxyzxyzC』用 echo 叫出，然后再使用如下的方法搜寻一下！<br>**echo 'AxyzxyzxyzxyzC' \| egrep 'A(xyz)+C'**<br>上面的例子意思是说，我要找开头是 A 结尾是 C ，中间有一个以上的 "xyz" 字符串的意思～ |

# 附：Perl正则

| 正则表达式 | 描述 | 示例 | |
|---|---|---|---|
| \b | 单词边界 | \bcool\b 匹配cool，不匹配coolant | |
| \B | 非单词边界 | cool\B 匹配coolant，不匹配cool | |
| \d | 单个数字字符 | b\db 匹配b2b，不匹配bcb | |
| \D | 单个非数字字符 | b\Db 匹配bcb，不匹配b2b | |
| \w | 单个单词字符（字母、数字与_） | \w 匹配1或a，不匹配& | |
| \W | 单个非单词字符 | \W 匹配&，不匹配1或a | |
| \n | 换行符 | \n 匹配一个新行 | |
| \s | 单个空白字符 | x\sx 匹配x x，不匹配xx | |
| \S | 单个非空白字符 | x\S\x 匹配xkx，不匹配xx | |
| \r | 回车 | \r 匹配回车 | |
| \t | 横向制表符 | \t 匹配一个横向制表符 | |
| \v | 垂直制表符 | \v 匹配一个垂直制表符 | |
| \f | 换页符 | \f 匹配一个换页符 | |

# sed工具

- sed 命令是利用脚本来处理文本文件。
- sed 可依照脚本的指令来处理、编辑文本文件。
- sed 主要用来自动编辑一个或多个文件、简化对文件的反复操作、编写转换程序等。

# sed工具

- 用法：sed [-hnV][-e<script>][-f<script文件>][文本文件]

参数：

-e<script>或--expression=<script> 以选项中指定的script来处理输入的文本文件。

-f<script文件>或--file=<script文件> 以选项中指定的script文件来处理输入的文本文件。

-h或--help 显示帮助。

-n或--quiet或--silent 仅显示script处理后的结果。

-V或--version 显示版本信息。

script用单引号（‘’）括起来

# sed工具

- 动作说明

a ：新增， a 的后面可以接字串，而这些字串会在新的一行出现(目前的下一行)～

c ：取代， c 的后面可以接字串，这些字串可以取代 n1,n2 之间的行！

d ：删除， d 后面通常不接任何东西；

i ：插入， i 的后面可以接字串，而这些字串会在新的一行出现(目前的上一行)；

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达式！例如 1,20s/old/new/g ！

# sed工具

- 应用举例

以行为单位新增/删除

```
[root@www ~]# nl  passwd | sed '2,5d'
1 root:x:0:0:root:/root:/bin/bash
6 sync:x:5:0:sync:/sbin:/bin/sync
7 shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
```

# sed工具

- 应用举例

以行为单位新增/删除

```
nl  passwd | sed '2,5d'
nl  passwd | sed '2d'
nl  passwd | sed '3,$d'

nl  passwd | sed  ‘2a hello world’    #第2行后
nl /etc/passwd | sed  ‘2i drink tea’   #第2行前
nl /etc/passwd | sed '2a Drink tea or ......\
> drink beer ?’      # 多行添加
```

# sed工具

- 应用举例

以行为单位显示和替换

```
nl   passwd | sed  ‘2,5c No 2-5 number’    #替换
nl   passwd | sed  ‘5,7p No 2-5 number’    #替换
```

# sed工具

- 应用举例

以行为单位搜索并显示

```
nl /etc/passwd | sed '/root/p'
1   root:x:0:0:root:/root:/bin/bash
2   daemon:x:1:1:daemon:/usr/sbin:/bin/
```

# sed工具

- 应用举例

数据的搜索并删除

```
nl /etc/passwd | sed  '/root/d'
2  daemon:x:1:1:daemon:/usr/sbin:/bin/sh
3  bin:x:2:2:bin:/bin:/bin/sh
```

# sed工具

- 应用举例

数据的搜索并执行命令

```
nl /etc/passwd | sed -n '/root/{s/bash/blueshell/;p;q}'
1  root:x:0:0:root:/root:/bin/blueshell
```

# sed工具

- 应用举例

数据的搜索并替换

```
nl /etc/passwd | sed -n  ‘s/root/sroot/g'
```

# sed工具

- 应用举例

直接修改内容（危险）

```
sed -i 's/\.$/\!/g' regular.txt
```

# awk工具

awk也是管道命令，sed通常处理一行数据，awk处理一行中的每个字段，默认以空格或tab键分隔。

使用方法：

awk ‘条件类型1 {动作1} 条件类型2 {动作2}…’ filename

注意*：条件类型和动作一定用单引号括起来。

# awk工具

- 处理流程：

1. 读入第一行，并将第一行的资料填入$0(整行），$1, $2, …
2. 依据“条件类型的限制”，判断是否需要进行后面的动作
3. 完成所有动作
4. 对后续行重复上面的步骤1~3，直到所有的数据都读完为止

# awk工具

- 常用内建变量：

NF：每一行（$0）拥有的字段总数

NR：目前awk所处理的是第几行

FS：目前的分隔字符，默认是空格

```
[liu@bogon ~]$ last -n 5 | awk '{print $1 "\t lines: " NR "\t columns" NF}'
liu      lines: 1        columns10
root     lines: 2        columns9
root     lines: 3        columns9
liu      lines: 4        columns10
liu      lines: 5        columns10
         lines: 6        columns0
wtmp     lines: 7        columns7
```

# awk工具

- 逻辑运算字符：
- >、<、>=、<=、==、！=

```
[liu@bogon ~]$ cat /etc/passwd | awk '{FS=":"} $3<10 {print $1 "\t" $3}'
root:x:0:0:root:/root:/bin/bash
bin     1
daemon  2
adm     3
lp      4
sync    5
shutdown        6
halt    7
mail    8
```

# awk工具

awk中的begin、end

- BEGIN{ 执行前的语句 }
- END { 处理完所有的行后要执行的语句 }
- 例：
- 对比：
  - cat /etc/passwd | awk '{FS=":"} $3<10 {print $1 "\t" $3}'
  - cat /etc/passwd | awk ‘BEGIN {FS=":"} $3<10 {print $1 "\t" $3}'

# awk工具

## awk中的条件与循环语句

IF语句：

```
if (condition)
{
    action-1
    action-1
    .
    .
    action-n
}
```

```
$ awk 'BEGIN {
    num = 11;
    if (num % 2 == 0) printf "%d 是偶数\n", num;
    else printf "%d 是奇数\n", num
}'
```

# awk工具

awk中的条件与循环语句

for语句：

```
for （initialisation; condition; increment/decrement)
    action
```

例：

```
awk 'BEGIN { for (i = 1; i <= 5; ++i) print i }'
```

# awk工具

awk中的条件与循环语句

While语句：

```
while (condition)
    action

awk 'BEGIN {i = 1; while (i < 6) { print i; ++i } }'
```

# awk工具

awk中的条件与循环语句

break 和 continue语句：

break结束循环，continue结束本次循环

```
$ awk 'BEGIN {
      sum = 0; for (i = 0; i < 20; ++i) {
      sum += i; if (sum > 50) break; else print "Sum = ", sum
   }
}'
awk 'BEGIN {for (i = 1; i <= 20; ++i) {if (i % 2 == 0) print i ; else continue} }'
```

# awk工具

## awk中的条件与循环语句

### exit

结束脚本程序的执行，该函数接受一个整数作为参数表示 AWK 进程结束状态。 如果没有提供该参数，其默认状态为 0。

```
$ awk 'BEGIN {
        sum = 0; for (i = 0; i < 20; ++i) {
        sum += i; if (sum > 50) exit(10); else print "Sum =", sum
    }
}'
```