Linux 操作系统

Linux Shell 介绍

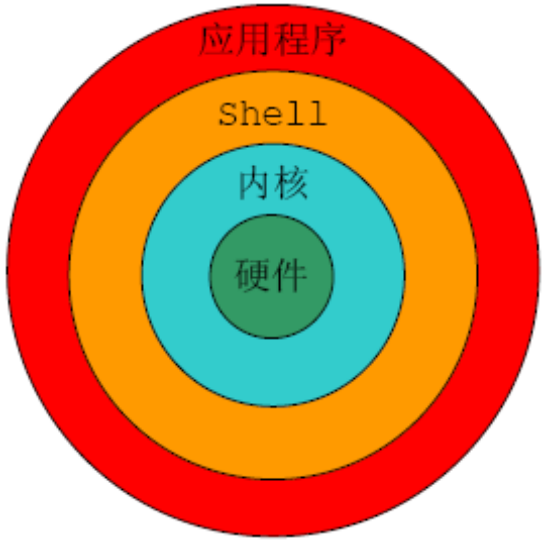
主要内容和学习要求

- 知道什么是 **shell** 和一些常见的 **shell**
- 掌握 **bash** 的基本功能（通配符、别名等）
- 了解 **bash** 的启动脚本
- 了解 **shell** 变量，学会查看和修改变量的值
- 理解如何定制 **bash**

Shell 简介

- **shell** 是系统的用户界面，它提供了用户和 **Linux**（内核）之间进行交互操作的一种接口。用户在命令行中输入的每个命令都由 **shell** 先解释，然后传给 **Linux** 内核去执行。

- 如果把 **Linux** 内核想象成一个球体的中心，**shell** 就是围绕内核的外层，从 **shell** 向 **Linux** 操作系统传递命令时，内核就会做出相应的反应。



Shell 简介

- shell 是一个命令语言解释器，拥有自己内建的 shell 命令集。此外，shell 也能被系统中其他应用程序所调用。

- shell 的另一个重要特性是它自身就是一个解释型的程序设计语言，shell 程序设计语言支持在高级语言里所能见到的绝大多数程序控制结构，比如循环，函数，变量和数组等。shell 编程语言简单易学，一旦掌握后它将成为你的得力工具。任何在命令行中能键入的命令也能放到一个可执行的 shell 程序里。

常用的 Shell

- 常用的 **shell** 有 **Bourne shell**, **C shell**, 和 **Korn shell**。

- 三种 **shell** 都有它们的优点和缺点。
  不同 **shell** 之间的转换非常方便。

- **Bourne shell (sh)**

作者是 **Steven Bourne**，它是 **UNIX** 最初使用的 **shell** 并且在每种 **UNIX** 上都可以使用。**Bourne shell** 在 **shell** 编程方面相当优秀，但在处理与用户的交互方面不如其他几种 shell。

常用的 Shell (续)

- **C shell (csh)**

C shell 由 Bill Joy 所写，它更多的考虑了用户界面的友好性。它支持象命令补齐等一些 Bourne shell 所不支持的特性。因为 C shell 的语法和 C 语言的很相似，C shell 被很多 C 程序员使用，这也是 C shell 名称的由来。

- **Korn shell (ksh)**

由 Dave Korn 所写。它集合了 C shell 和 Bourne shell 的优点并且和 Bourne shell 完全兼容。

常用的 Shell (续)

- 其它 shell

许多其它的 shell 基本上都是吸收了这些 shell 的优点扩展而成的 shell。常见的有 tcsh (csh 的扩展), Bourne Again shell(bash, sh 的扩展), 和 Public Domain Korn shell (pdksh, ksh 的扩展)。

- bash 是现在大多数 Linux 系统的缺省 shell

bash 与 Bourne shell 完全向后兼容，并且在 Bourne shell 的基础上增加和增强了很多特性。bash 也包含了很多 csh 和 ksh 里的优点。bash 有很灵活和强大的编程接口，同时又有很友好的用户界面。

Bash 的功能

- 命令行

当用户打开一个（虚拟）终端时，可以看到一个 **shell** 提示符，标识了命令行的开始。用户可以在提示符后面输入任何命令

```
command  [选项]  [参数]
```

例：`ls -l /home/jypan/linux/`

注意：命令行中选项先于参数输入

命令行特征

- 在一个命令行中可以输入多个命令，用分号将各个命令隔开。例如：

```
ls -F; cp -i mydata newdata
```

- 如果一个命令太长，无法在一行中显示，可以使用反斜杠 \ 来续行，在多个命令行上输入一个命令或多个命令。例如：

```
ls -F; \
cp -i mydata newdata
```

大多数 **shell** 在达到命令行行尾时都会自动断开长命令

命令行特征 (续)

- 命令行编辑

命令行实际上是可以编辑的一个文本缓冲区，在按回车之前，可以对输入的命令进行编辑。如用 **BACKSPACE** 键可以删除刚键入的字符，也可以进行整行删除，还可以插入字符等。

常用的快捷键和组合键

左/右箭头键	向左/向右移动一个字符
Ctrl+a	移动到当前行的行首
Ctrl+e	移动到当前行的行尾
Ctrl+f	向前移动一个字符
Ctrl+b	向后移动一个字符
Ctrl+k	从光标处删除到本行的行尾
Ctrl+u	从光标处删除到本行的行首
Ctrl+l	清屏
Alt+f	向前移动一个单词
Alt+b	向后移动一个单词

`stty -a` 可以看到更多的快捷键。

通 配 符

❑ 通配符

◆ **bash** 提供许多功能用来帮助用户节省输入命令的时间，其中最常用的一种方法就是使用通配符。

◆ 通配符就是一些特殊的字符，可以用来在引用文件名时简化命令的书写。用户在使用时可以用通配符来指定一种模式，即所谓的“模式串”(**pattern**)，然后 **shell** 将把那些与这种模式能够匹配的文件作为输入文件。

◆ 在 **bash** 中可以使用三种通配符：*****、**?**、**[]**。

通配符的含义

*	匹配 任意长度 的字符串（包括零个字符）
?	匹配任何 单个字符
[ ]	创建一个字符表列，方括号中的字符用来匹配或不匹配单个字符。如： **[xyz] 匹配 x、y 或 z，但不能匹配 xx，xy 或者其它任意组合。** 无论列表中有多少个字符，它只匹配一个字符。 [abcde] 可以简写为 [a-e] 。 另外，用感叹号作为列表的第一个字符可以起到反意作用，如： **[!xyz] 表示匹配 x、y、z 以外的任意一个字符。**

通配符举例

◆ 通配符 “*” 的常用方法就是查找具有相同扩展名的文件

```
ls   *.tar.gz
```

通配符 “*” 有时可以将几百的命令缩短成一个命令。假设当前目录下有许多文件，现在要删除扩展名为“.old”的文件，如果有几百个这样的文件，逐个删除显然很麻烦，这时可以使用通配符：

```
rm   *.old
```

◆ 问号通配符 “?” 必须匹配一个且只能匹配一个字符，通常用来查找比 * 更为精确的匹配。

```
ls   *.???
```

方括号通配符举例

◆ 方括号通配符使用括号内的字符作为被匹配的字符，且只能匹配其中的一个字符。如列出以 a、b、c 开头，且以 .dat 为扩展名的所有文件：

```
ls   [abc]*.dat
```

可以在方括号中使用连字符 - 来指定一个范围，如列出以字母开头，数字结尾的所有文件：

```
ls   [a-zA-Z]*[0-9]
```

通配符使用注意事项

- 文件名最前面的圆点“.”和路经名中的斜杠“/”必须显式匹配。例如“*”不能匹配“.bashrc”，而“.*”才可以匹配“.bashrc”。

- 连字符 - 仅在方括号内有效，表示字符范围。如果在方括号外面就成为普通字符了。而 * 和 ? 在方括号外面是通配符，若出现在方括号之内，它们也失去通配符的能力，成为普通字符了。

```
ls *
ls mem*
ls *x
```

```
ls *lax*
ls .*
ls mem?
```

```
ls mem?t
ls mem[1-9]
ls mem[*1-9]
```

别名

- 别名是 **bash** 中用来节省时间的另一项重要功能，它允许用户按照自己喜欢的方式对命令进行自定义。

- 别名的创建：创建别名的命令是 **alias**，例：

```
alias  lf='ls -F'
```

注：等号两边不能有空格！

- 别名的取消：所设置的别名在当前 **shell** 中一直有效，直到退出当前 **shell** 或用 **unalias** 取消别名，例：

```
unalias  lf
```

别名 (续)

◆ 查看已创建的别名：输入alias直接回车即可。

```
alias
```

◆ 如果需要一直使用某些别名，可以在 **bash** 启动脚本中添加设置别名的命令，这样每次打开一个（虚拟）终端时，系统就会自动设置别名。有关 **bash** 的启动脚本，我们将在后面讨论。

命令行自动补齐功能

❑ 命令行自动补齐功能

◆ 通常用户在 **bash** 下输入命令时不必把命令输全，**shell** 就能判断出你所要输入的命令。

◆ 该功能的核心思想是：**bash** 根据用户已输入的信息来查找以这些信息开头的命令，从而试图完成当前命令的输入工作。用来执行这项功能的键是 **Tab** 键，按下一次 **Tab** 键后，**bash** 就试图完成整个命令的输入，如果不成功，可以再按一次 **Tab** 键，这时 **bash** 将列出所有能够与当前输入字符相匹配的命令列表。

命令行自动补齐功能

例：查看用户的命令历史

```
his<Tab>
```

◆ 这项功能同样适用于文件名的自动补齐

例：要进入目录：

/etc/sysconfig/network-scripts/

```
cd /e<Tab>sys<Tab>c<Tab>ne<Tab>-<Tab>
```

管道

❑ 管道

◆ UNIX 系统的一个基本哲学是：一连串的小命令能够解决大问题。其中每个小命令都能够很好地完成一项单一的工作。现在需要有一些东西能够将这些简单的命令连接起来，这样管道就应运而生。

◆ 管道“|”的基本含义是：将前一个命令的输出作为后一个命令的输入。如：

```
ls /local | du -sh *
```

◆ 利用管道可以实现一些很强的功能。

管道举例

一个较复杂的例子：输出系统中用户名的一个排序列表。这里需要用到三个命令：**cat**、**awk**、**sort**，其中 **cat** 用来显示文件 **/etc/passwd** 的内容，**awk** 用来提取用户名，**sort** 用来排序。

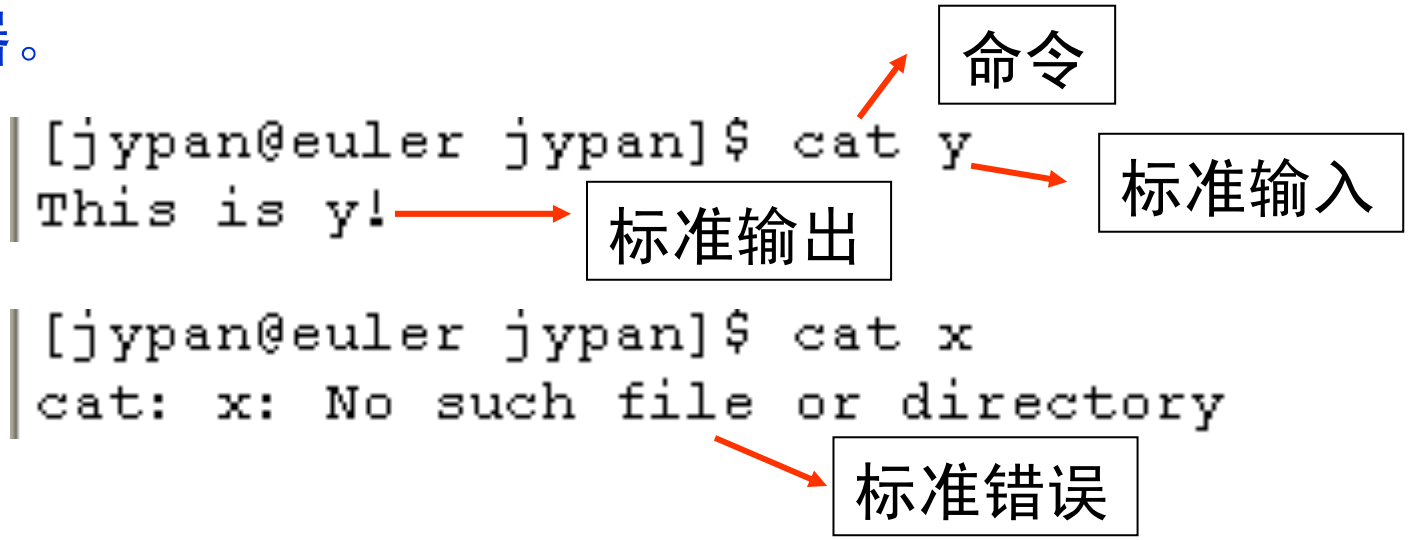
```
cat /etc/passwd | \
awk -F: '{print $1}' |\
sort
```

重定向

❑ 数据流

◆ **Linux** 中的数据流有三种：标准输入 (**STDIN**)、标准输出 (**STDOUT**) 和标准错误 (**STDERR**)。

◆ 标准输入通常来自键盘，标准输出是命令的结果，通常定向到显示器，标准错误是错误信息，通常也定向到显示器。



重定向

- 输入输出重定向
  - 输入重定向：“<”

    可以使用文件中的内容作为命令的输入。
  - 输出重定向：“>”

    允许将命令的输出结果保存到一个文件中。

```
ls > list
```

```
sort < list > sort_list
```

重定向

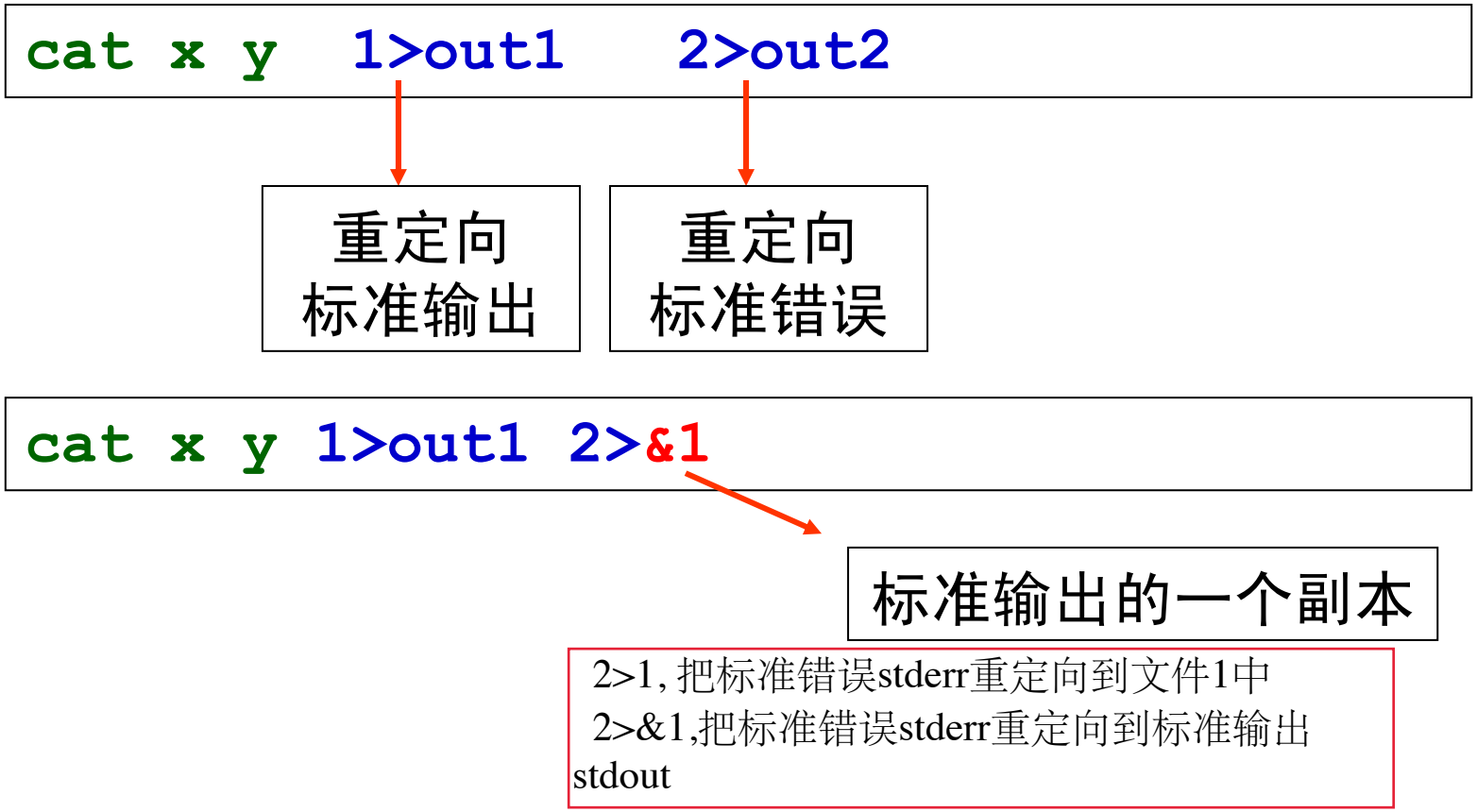
- 输入输出重定向
  - 在使用输出重定向时，如果输出文件已经存在，则原文件中的内容将被删除。
  - 如果希望保留原文件的内容，可以使用“>>”代替“>”，这样重定向输出的内容将添加到原文件的后面。

```
ls / > list
```

```
ls /home/ >> list
```

文件描述符

- **shell** 中进程处理文件时会建立一个文件描述符，标准文件描述符有三个：0、1、2，分别对应于标准输入、标准输出和标准错误。



命令历史记录

- 在命令行中输入的每个命令都被保存到一个称为 **history**（命令历史记录）的地方，在 **bash** 中，变量 **HISTSIZE** 用来指定存储在命令历史记录中的命令的最多个数。

- 查看命令历史记录：**history**

```
history
```

```
history 30
```

表示查看最近 30 个命令历史记录

命令历史记录

- 上下箭头键：除查看命令历史记录外，还可以利用上下箭头键在命令历史记录中移动。此外，还可以对所选的命令进行编辑。
- 感叹号的用法：
  - **!!**：执行最近一次使用的命令；
  - **!n**：其中 **n** 为一个具体的数字，表示执行在命令历史记录中的第 **n** 个命令；
  - **!s**：其中 **s** 为一个字符串，表示执行命令历史记录中以该字符串开头的最近的一个命令。
- **fc** 命令：

```
fc -l 30 50
```

列出命令历史记录中第30到第50之间的命令

引用

❑ 引用

◆ 在 **bash** 中，有些字符具有特殊含义，如果需要忽略这些字符的特殊含义，就必须使用引用技术。

◆ 引用可以通过下面三种方式实现：

- ✓ 使用转义字符：\
- ✓ 使用单引号：‘ ’
- ✓ 使用双引号：“ ”

◆ 转义字符的引用方法就是直接在字符前加反斜杠

例：\$，\`，\"，\\，\ ，\!

引用

◆ 单引号对中的字符都将作为普通字符，但不允许出现另外的单引号。

◆ 双引号对中的部分字符仍保留特殊含义，

如：$、\、‘、“、及换行符等。

◆ 单引号是强引用，而双引号是弱引用。

Shell 中的特殊字符

◆ 在 bash 中，有些字符具有特殊含义，通常称为特殊字符。

字符	含义	字符	含义
`	强引用	*、?、!	通配符
"	弱引用	<、>、>>	重定向
\	转义字符	-	选项标志
$	变量引用	#	注释符
;	命令分离符	空格、换行符	命令分隔符
`	命令替换：反引号中的字符串被 shell 解释为命令，在执行时，shell 首先执行该命令，并以它的标准输出取代整个反引号 (包括两个反引号) 部分		

Shell 变量

- **shell** 变量大致可以分为三类：内部变量、用户变量和环境变量。

  - 内部变量：由系统提供，用户不能修改。
  - 用户变量：由用户建立和修改，在 **shell** 脚本编写中会经常用到。
  - 环境变量：这些变量决定了用户工作的环境，它们不需要用户去定义，可以直接在 **shell** 中使用，其中某些变量用户可以修改。

常见的 Shell 变量

变量名	含义
HOME	用户主目录
LOGNAME	登录名
USER	用户名，与登录名相同
PWD	当前目录/工作目录名
MAIL	用户的邮箱路径名
HOSTNAME	计算机的主机名
INPUTRC	默认的键盘映像
SHELL	用户所使用的 shell 的路径名
LANG	默认语言
HISTSIZE	history 所能记住的命令的最多个数
PATH	shell 查找用户输入命令的路径 (目录列表)
PS1、PS2	shell 一级、二级命令提示符

Shell 变量

◆ PATH 变量是最重要的环境变量之一。当用户在命令行中输入命令时， **shell** 就会根据该变量定义的路径（目录）和顺序，查找并执行该命令。如果没有正确设置 PATH 变量，则必须输入完整的路径名来运行某个命令。

◆ 在 Linux 下输入命令的两种方式：

- ✓ 直接在命令行中输入命令：根据 PATH 查找该命令
- ✓ 输入完整的路径名

◆ 用户可以根据需要修改环境变量

如：HISTSIZE，PATH，PS1，PS2 等

Shell 变量查询

- 查询当前 **shell** 中的环境变量：**env**

```
env
```

- 查询某个变量的值：**echo**

```
echo ${变量名}
```

命令提示符

- 在 **bash** 中，有两个级别的命令输入提示：
  - ✓ 一级提示符是当 **bash** 等待输入命令时所出现的提示符，由环境变量 **PS1** 控制，缺省值为 “**$**”；
  - ✓ 二级提示符是在 **bash** 执行一个命令后，需要用户进一步输入才能完成次命令时，所出现的提示符，由环境变量 **PS2** 控制，缺省值为 “**>**”。

- 重设 **PS1** 和 **PS2** 的设置

```
export
使变量的值对当前shell及其所有子进程都可见
```

例：`export PS1="\t\w\$"`

命令提示符

- 在创建提示符时，可以使用下面的特殊字符：

\!	显示命令的历史编号	\h	显示机器的主机名
\#	显示命令的命令编号	\s	显示当前使用的 shell
\\	显示一个反斜杠	\u	显示用户名
\n	显示一个换行符	\W	显示当前目录名
\d	显示当前的日期	\w	显示当前目录完整路径名
\t	显示当前的时间		
\$	普通用户显示“$”，超级用户显示“#”	\nnn	显示与八进制 nnn 相对应的字符

bash 配置文件

❑ bash 配置文件

◆ 在命令行中设置和修改的变量值，只在当前的 shell 中有效。一旦用户退出 bash，所做的一切改变都会丢失。

◆ 在启动交互式会话过程中，在出现提示符前，系统会读取几个配置文件，并执行这些文件中的命令。所以这些文件可以用来定制 bash 环境。如：设置 shell 变量值或建立别名等。

◆ bash 配置文件：

```
/etc/profile
```

```
~/.bash_profile
~/.bash_login
~/.profile
```

```
~/.bashrc
```

bash 配置文件

- **/etc/profile**

  Linux 系统中的全局 bash 启动脚本，任何用户登录系统时 /etc/profile 都会被执行。通常用来设置标准 bash 环境，但修改该文件需 root 权限。

- 读取 /etc/profile 文件后，bash 将在用户主目录中按顺序查找以下文件，并执行第一个找到的文件：

```
~/.bash_profile
~/.bash_login
~/.profile
```

在这些文件中，用户可以定义自己的环境变量，而且能够覆盖在 /etc/profile 中定义的设置。

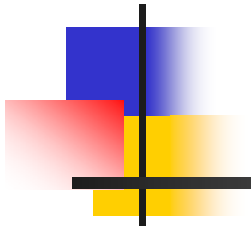
bash 配置文件

◆ **bash** 启动后，将读入配置文件 **~/.bashrc**，并执行这个文件中的所有内容。

◆ 另外，还可以从另一个 **shell** 或者 **bash** 自身启动一个新的 **bash**，这种过程称为非登录交互式，启动新 **bash** 的命令为 **bash**，此时所读入的唯一 bash 配置文件是 **~/.bashrc**

通常，个人**bash** 环境设置都定义在 **~/.bashrc** 文件里

Linux 操作系统



Linux 进程介绍

主要内容和学习要求

- 理解进程与多进程的概念
- 掌握如何运行后台进程
- 掌握如何进行进程控制
- 掌握相关命令的使用

进程

进程概念

- 正在运行的程序叫做进程（**process**）
  程序只有被系统载入内存并运行后才能称为进程。

- **Linux** 允许同时运行多个程序，为了区分每一个运行的程序，**Linux** 给每个进程都做了标号，称为进程号（**process ID**），每个进程的进程号是唯一的。

- 查看当前运行的程序及其进程号：`ps`

多进程

❑ 多进程

◆ **Linux** 是一个多用户的操作系统，当多个用户同时在一个系统上工作时，**Linux** 要能够同时满足用户们的要求，而且还要使用户感觉不到系统在同时为多个用户服务，就好像每一个用户都单独拥有整个系统一样。

◆ **Linux** 不可能在一个 **CPU** 上同时处理多个任务（作业）请求，而是采用“分时”技术来处理这些任务请求。

多进程

◆ 分时技术

所有的任务请求被排除一个队列，系统按顺序每次从这个队列中抽取一个任务来执行，这个任务执行很短的时间（几毫秒）后，系统就将它排到任务队列的末尾，然后读入队列中的下一个任务，以同样的方式执行。这样经过一段时间后，任务队列中的所有任务都被执行一次，然后又开始下一轮循环。

◆ 任务/作业

就是一个被用户指定运行的程序。如用户发出一个打印命令，就产生一个打印任务/作业，若打印成功，表示任务完成，没有成功表示任务没完成。

多进程

- **Linux** 是多用户系统，它必须协调各个用户。

Linux 给每个进程都打上了运行者的标志，用户可以控制自己的进程：给自己的进程分配不同的优先级，也可以随时终止自己的进程。

前台与后台

◆ 前台进程

指一个程序控制着标准输入/输出，在程序运行时，**shell** 被暂时挂起，直到该程序运行结束后，才退回到**shell**。在这个过程中，用户不能再执行其它程序。

◆ 后台进程

用户不必等待程序运行结束就可以执行其它程序。

◆ 在一个终端里只能同时存在一个前台任务，但可以有多个后台任务。

运行后台进程

❑ 运行后台进程

- 在命令最后加上“&”

例：`sleep 60 &`

- 如果程序已经在前台运行，需要将其改为后台运行，这时可以先安组合键 **Ctrl+z**，将任务挂起，然后利用 **bg** 命令将该程序转为后台运行
- 若要将一个后台进程转到前台运行，可以使用 **fg** 命令
- 相关命令：**jobs, bg, fg**

jobs 命令

- **jobs**：查看后台运行或被挂起的进程

例：

```
[jypan@server236 ~]$ jobs
[1]   Stopped                 sleep 111
[2]-  Stopped                 sleep 112
[3]+  Stopped                 sleep 113
```

- 第一列显示的是作业号
- “+”表示当前作业，“-”表示当前作业之后的作业

- 若加上选项 **-l**，则显示进程号

```
[jypan@server236 ~]$ jobs -l
[1]   16368 Stopped                 sleep 111
[2]-  16369 Stopped                 sleep 112
[3]+  16371 Stopped                 sleep 113
```

bg / fg 命令

- **bg**：将被挂起的进程转化到后台运行

```
bg jobnumber
```

  - **jobnumber** 是通过 **jobs** 查出来的作业号

例：

```
bg 2
```

```
bg 1 2
```

- **fg**：将后台进程转化到前台运行
  - 用法与 **bg** 类似

进程控制：ps

- 查看正在运行的程序：ps

```
ps [选项]
```

例：

```
ps
```

```
ps -u jypan
```

```
ps u -u jypan
```

```
ps u -u jypan --sort=cmd
```

```
ps -u jypan -o "%U %p %c %x %t"
```

ps 常用选项

-A, -e	显示所有进程
-u	查看指定用户的进程（用户名或用户ID）
-l	长格式显示，可查看各个进程的优先权值
-f	完全显示（显示完整的命令）
-C	列出指定命令名称的进程

u	增加用户名，起始时间，CPU和内存使用等信息
a	显示终端机下用户执行的进程，包含其它用户
f	显示进程树，等价于 --forest
r	显示正在运行的进程

-o	按指定的格式输出
--sort	按指定内容进行排序

ps 举例

```
ps -A
```

```
ps -u jypan
```

```
ps -u jypan -l  或 ps -l -u jypan
```

```
ps -u jypan -f
```

```
ps -C sleep -f
```

```
ps -C sleep u
```

```
ps af
```

```
ps r
```

常见列标志的含义

例：`ps -u jypan u`

PID	进程 ID	**CMD**	命令名（COMMAND）
UID	用户 ID	**START**	进程启动时间
USER	用户名	**%CPU**	进程所用CPU时间百分比
TIME	执行时间	**%MEM**	进程所有MEM百分比
STAT	进程状态	**NI**	优先权值 / nice 值

TTY	启动进程的终端	**RSS**	进程所用内存块数
PGID	进程组 ID	**VSZ**	所用虚拟内存块数

- 更多列标志见 **man ps**

进程状态

R	正在运行或处在运行队列中
S	休眠（等待）
T	停止或被追踪
D	不可中断的睡眠，通常指 I/O
Z	僵尸进程（已结束但未被父进程收回）
X	已死进程 （这个状态不会出现）

<	具有最高优先权
N	具有较低的优先权
s	is a session leader
l	is multi-threaded
+	is in the foreground process group

指定输出格式

例：
```
ps -u jypan -o "%U %p %c %x %t"
```

- 输出格式中的常用字段

%U	用户名
%u	用户名
%G	用户组
%g	用户组
%C	CPU
%P	父进程

%c	命令名
%a	命令名（含选项与参数）
%p	进程号
%x	运行时间
%t	Elapsed time
%n	nice 值（代表优先权）

```
%r  PGID -- ID of the process group ( leader )
```

例：
```
ps -u jypan -o %c%p%r%n
```

指定输出格式

- 另一种使用方式

```
ps -u jypan -o user,pid,pcpu,time,etime
```

- 字段对应表

%U	用户名	user	%c	命令名	comm
%u	用户名	ruser	%a	命令名	args
%G	用户组	group	%p	进程号	pid
%g	用户组	rgroup	%x	运行时间	time
%C	CPU	pcpu	%t	Elapsed time	etime
%P	父进程	ppid	%n	nice 值	nice
%r	进程组	pgid		用户ID	uid

进程排序

例：

```
ps au --sort=uid,-pid
```

uid	用户 ID
user	用户名
cmd	命令名
pid	进程 ID
ppid	父进程 ID
pgrp	用户组 ID
size	内存大小
pcpu	CPU
utime	用户时间
start_time	起始时间

- 更多选项见 `man ps`

nohup 命令

- 用户退出系统后能继续运行的进程
  - 通常当用户退出系统后，所有属于该用户的进程将全部被终止。如果希望程序在退出系统后仍然能够继续运行，需使用 **nohup** 命令后台启动该进程

```
nohup 命令 [选项] [参数] &
```

  - 若有输出，则通常输出到指定的文件中

进程的优先权

- 进程的 **nice值** 和 **优先权**
  - 在任务队列中的进程并不享有同等的优先权，每个进程都有一个指定的 **nice值**（优先权值），从 **-20** 到 **19**
  - **nice值**为 **-20** 的进程具有最大优先权
- 进程的缺省 **nice值** 为 0

例：查看缺省的 nice值： `nice`

- 查看进程的 **nice值**

```
ps -l 进程号    % NI 的值
```

进程的优先权

- 调整进程的 **nice值**
  - 可以在进程启动时指定，也可以在启动后修改
  - 在启动进程时就指定优先级： **nice**

```
nice -n 命令 &
```

n 是指优先级的增量

- 若为正，表示增加 **nice值**，即降低进程优先权
- 若为负，表示减小 **nice值**，即提高优先权
- 若缺省，则默认为 10，即 **nice值** 增加 10

进程的优先级

例：

```
nice -5 sleep 60 &
```

- 普通用户只能增加 **nice值**
- 只有系统管理员才能降低一个进程的 **nice值**

例：

```
sudo nice --5 sleep 60 &
```

- 使用 nice 同样可以增加前台任务的 **nice值**

例：

```
nice -5 sleep 60
```

进程的优先级

- 进程运行后调整 **nice值**： **renice**

进程已经运行，此时又有许多用户登录，他们使得各个进程分得的 CPU 时间下降。此时，root 可以提高进程的优先权，但普通用户没这个权限，在系统资源紧张时，只能通过降低其它不着急的进程的优先权，从而使得急用的进程能分得更多的 CPU 时间。

renice n [-p pid] [-u user] [-g pgid]

- 增加指定进程的 **nice值**
- **n** 可以是正的，也可以是负数；
- 注意与 **nice** 命令的区别：没有减号
- **pgid** 是进程组的 ID

进程的优先级

例：

```
renice 5 2673    % 增加进程2673的nice值
                 % -p 可以省略
```

- 增加进程2673的 **nice值**
- **-p 可以省略**

例：

```
renice 5 -u jypan
```

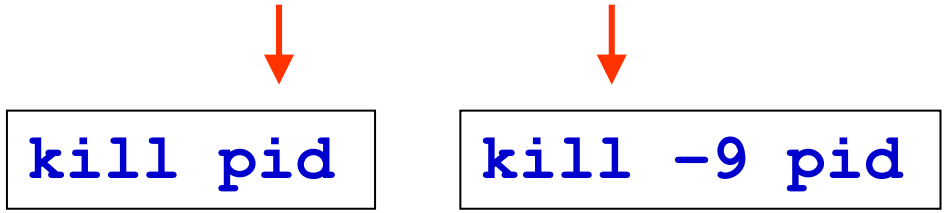
- 增加指定用户的所有进程的 **nice值**

注：普通用户一旦增加某个进程的 **nice值** (即降低优先权) 后，就无法再回复到原来的 **nice值**

终止进程

❑ 终止进程

- 终止前台进程使用：**Ctrl+c**
- 终止后台进程使用：**kill**
  - **kill** 有两种方法：正常结束和强制结束



注：(1) 使用 **kill** 前需要先用 **ps** 查看需要终止的进程的**pid**;

(2) **kill -9** 很霸道，它在杀死一个进程的同时，将杀死其所有子进程，使用时要谨慎。如错杀 login 进程或 shell 进程等。

常用 bash 内部命令

- 一些常用的 bash 内部命令

  - alias/unalias ：设置和取消 bash 别名。
  - bg: 使一个被挂起的进程在后台继续执行。
  - cd: 切换当前工作目录。
  - exit: 退出 shell。
  - export:使变量的值对当前shell的所有子进程都可见。
  - fc: 用来显示和编辑历史命令列表里的命令。
  - fg: 使一个被挂起的进程在前台继续执行。
  - help: 显示帮助信息。
  - kill: 终止某个进程。
  - pwd: 显示当前工作目录。

相关命令

- `id`: print real and effective UIDs and GIDs
- `who`: show who is logged on
- `whoami`: `id -un`
- `hostname`: show or set the system's host name
- `w`: show who is logged on and what they are doing
- `last`: show listing of last logged in users
- `finger`: displays information about the system users
- `top`: display Linux tasks（很有用的系统监控工具）

- 更多 **bash**内部命令见：`man bash --> 3370`
  或任一 **bash**内部命令的 manual，例： `man bg`

Linux 操作系统

正则表达式

正则表达式

- 当从一个文件或命令输出中抽取或过滤文本时，可以使用正则表达式（rexp，regular expressions）
- 正则表达式是由普通字符和特殊字符的集合
- 系统自带的所有大的文本过滤工具在某种模式下都支持正则表达式的使用，并且还包括一些扩展的元字符集
- 正则表达式广泛使用在 grep、sed 命令和 awk 语言中

基本元字符集及其含义

^	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 个或多个此单字符
[ ]	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
c\{n\}	匹配 字符 c 连续出现 n 次的情形
c\{n,\}	匹配 字符 c 至少连续出现 n 次的情形
c\{n,m\}	匹配 字符 c 连续出现次数在 n 与 m 之间的情形

注：字符 c 可以通过 [ ]，\ 或 . 来指定，但只能是单个字符。

如： [a-z]\{5\}，\$\{2,\}，.\{2,5\}

元字符集举例

- 使用句点“ . ”匹配单字符
  - 匹配任意单个ASCII 字符，可以为字母或数字

```
..XC..      可以匹配 deXC1t、23XCdf
.w..w..w.   可以匹配 rwxrw-rw-
```

- 在行首以“ ^ ”匹配字符串或字符序列
  - 在一行的开始匹配字符或单词

```
^d      可以匹配 drwxr-xr-x、drw-r--r—
^.01    可以匹配 0011cx4、c01sdf
```

元字符集举例

- 在行尾以“ $ ”匹配字符串
  - 在行尾匹配字符串或字符，$ 符号放在匹配单词后面

```
trouble$  匹配以单词 trouble 结尾的所有行
^$        匹配所有空行
```

- 使用“ * ”匹配单个字符或其重复序列
  - 一个单字符后紧跟 *，表示匹配 0 个或多个此字符

```
compu*t  可以匹配  comput、compuuut
1013*    可以匹配  1013、101333、101
```

  - 注：星号必须跟其前面的字符结合才有意义

元字符集举例

- 使用“ \ ”屏蔽一个特殊字符的含义
  - 用来屏蔽一个元字符的特殊含义

```
\*\.pas$      匹配以 *.pas 结尾的所有行
```

- 使用“ [] ” 匹配一个字符范围或集合
  - 匹配“ [] ”内的字符，可以是单个字符，或字符序列，可以使用 - 表示一个字符序列范围，如 [A-Za-z0-9]
  - 当 [后面紧跟 ^ 符号时，表示不匹配方括号里内容

```
[Cc]omputer  匹配 Computer 和 computer
[^a-zA-Z]    匹配任一个非字母型字符
```

元字符集举例

- 使用“\{ \}”匹配模式出现的次数
  - c\{n\}：匹配 字符 c 连续出现 n 次的情形
  - c\{n,\}：匹配 字符 c 至少连续出现 n 次的情形
  - c\{n,m\}：匹配 字符 c 连续出现次数在 n 与 m 之间

A\{2\}B	只能匹配 AAB
A\{2,\}B	可以匹配 AAB 或 AAAAAB，但不能匹配 AB
A\{2,4\}B	匹配 AAB、AAAB、AAAAB 但不能匹配 AB 或 AAAAAB 等

- 实际上真正的格式是 {n}、{n,}、{n,m}，只不过对“{”和“}”用了 Esacpe 字符“\”

常用的正则表达式举例

[Ss]igna[lL]	匹配 signal、signaL、Signal、SignaL
[Ss]igna[lL]\.	同上，但后面加一句点
^USER$	只包含 USER 的行
\.	带句点的行
^d..x..x..x	用户、同组用户及其他用户都有可执行权限的目录
^[^s]	不以 s 开始的行
[yYnN]	大写或小写的 y 或 n
.*	匹配任意多个字符
^.*$	匹配任意行
^......$	只包含 6 个字符的行

常用的正则表达式举例

[a-zA-Z]	任意单个字母
[^a-zA-Z0-9]	非字母或数字
[^0-9\$]	非数字或美元符号
[123]	1 到 3 中一个数字
\^q	包含 ^q 的行
^.$	仅有一个字符的行
^\.[0-9][0-9]	以一个句点和两个数字开始的行

[0-9]\{2\}-[0-9]\{2\}-[0-9]\{4\} 日期格式 dd-mm-yyyy
[0-9]\{3\}\.[0-9]\{3\}\.[0-9]\{3\}\.[0-9]\{3\} 类 IP 地址格式 nnn.nnn.nnn.nnn

在 **shell** 编程中，一段好的脚本与完美的脚本之间的差别之一，就是要熟知正则表达式并学会使用它们。有很多可以处理文本的程序，比如 **grep**、**awk**、**sed** 等都使用正则表达式。

Linux 操作系统

文本过滤

主要内容和学习要求

- 能够熟练运用 **grep** 命令
- 掌握 **sed** 流编辑器
- 学会使用 **awk** 编程

grep 家族

- **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** 命令不会对输入文件进行任何修改或影响

- **grep** 家族由 **grep**、**egrep** 和 **fgrep** 组成：
  - **grep**：标准 **grep** 命令，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**），也称快速 **grep**（**fast 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。这里的“快速”并不是指速度快。

grep 的使用

- **grep 命令的一般形式**

```
grep [选项] 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 ...
```

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	显示匹配的行及行号
-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 'an*' datafile
```

grep 命令应用举例

- 查询多个文件,可以使用通配符“ * ”

```
grep "math2"   *.txt
```

```
grep "12"   *
```

- 反向匹配

```
ps aux | grep "ssh" | grep -v "grep"
```

- 匹配空行

```
grep -n '^$' datafile
```

```
grep -v '^$' datafile > datafile2
```

grep 命令应用举例

- **精确匹配单词:** `\<` 和 `\>`
  - 找出所有包含 以 north 开头 的单词的行

```
grep '\<north' datafile
```

  - 找出所有包含 以 west 结尾 的单词的行

```
grep 'west\>' datafile
```

  - 找出所有包含 north 单词的行

```
grep '\<north\>' datafile
```

grep 命令应用举例

- 递归搜索目录中的所有文件：-r

```
grep -r "north" datafile  ~/Teaching/linux/
```

- 关于某个字符连续出现次数的匹配

```
grep 'o\{2,\}'  helloworld
```

```
'o\{2,4\}' ,  'o\{2,4\}' ,  'lo\{2,4\}'
```

grep 命令应用举例

- 其它

```
grep '^n' datafile
```

```
grep 'y$' datafile
```

```
grep 'r\.' datafile
```

```
grep '^[we]' datafile
```

```
grep -i 'ss*' datafile
```

```
grep -n '[a-z]\{9\}' datafile
```

```
grep -c '\<[a-z].*n\>' datafile
```

grep 与管道

```
ls -l | grep '^d'
```

如果传递给 grep 的文件名参数中有目录的话，
需使用 “ -d ” 选项

```
grep -d [ACTION] directory_name
```

其中 ACTION 可以是
read：把目录文件当作普通文件来读取
skip：目录将被忽略而跳过
recurse：递归的方式读取目录下的每一个文件，可以用选项 "-r" 代替 "-d recurse"

```
grep -rl "eth0" /etc
```

egrep 命令

- 使用 egrep 的主要好处是，它在使用 grep 提供的正则表达式元字符基础上增加了更多的元字符，见下表，但不能使用 \{ \}。

在 Linux下：egrep = grep -E

- egrep 增加的元字符

+	匹配一个或多个前一字符
?	匹配零个或一个前一字符
str1|str2	匹配 str1 或 str2
( )	字符组

注意星号 * 和问号 ? 在 shell 通配符和正则表达式中的区别

egrep 举例与 fgrep

```
egrep 'WE+' datafile
```

```
egrep 'WE?' datafile
```

```
egrep 'S(h|u)' datafile
```

```
egrep 'Sh|u' datafile
```

- **fgrep 命令**

  fgrep 的使用方法与 grep 类似，但对正则表达式中的任何元字符都不做特殊处理。

```
fgrep '^n' datafile
```

流编辑器 sed

- **sed 是什么**

  sed 是一个精简的、非交互式的编辑器，它在命令行中输入编辑命令和指定文件名，然后在屏幕上查看输出。

- **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。

sed 的三种调用方式

- 在命令行中直接调用

```
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。

sed 的三种调用方式

- 将 sed 命令插入脚本文件，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`

定位方式

- 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

常用 sed 编辑命令

- 常用的 **sed_edit_cmd**
  - **p**：打印匹配行

```
sed -n '1,3p' datafile  // ('1,3!p')
```

```
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
```

常用 sed 编辑命令

- **a\ :** 在指定行后面追加一行或多行文本，并显示添加的新内容，该命令主要用于 sed 脚本中。

```
sed -n '/eastern/a\newline1\
                    newline2\
                    newlineN'   datafile
```

- **i\ :** 在指定行前追加一行或多行，并显示添加的新内容，使用格式同 a\
- **c\ :** 用新文本替换指定的行，使用格式同 a\
- **l :** 显示指定行中所有字符，包括控制字符(非打印字符)

```
sed -n '/west/l'   datafile
```

常用 sed 编辑命令

- **s**：替换命令，使用格式为：

```
[address]s/old/new/[gpw]
```

  - **address** ：如果省略，表示编辑所有的行。
  - **g** ：全局替换
  - **p** ：打印被修改后的行
  - **w fname** ：将被替换后的行内容写到指定的文件中

```
sed -n 's/west/east/gp'   datafile
```

```
sed -n 's/Aanny/Anndy/w newdata'   datafile
```

```
sed 's/[0-9][0-9]$/&.5/'   datafile
```

& 符号用在替换字符串中时，代表 被替换的字符串

常用 sed 编辑命令

- 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 newdata'  datafile
```

- w：写文件，将指定行写入到另外一个文件中。

```
sed -n '/west/w newdata'  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

常见的 sed 编辑命令小结

- 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

shell 变量的使用

- sed 支持 shell 变量的使用

在 sed_cmd 中可以使用 shell 变量，此时应使用 双引号

```
myvar= "west"
sed -n "/${myvar}/p" datafile
```

- 如何输入控制字符，如：回车、Esc、F1 等

以输入 回车（^M）为例：

先按 Ctrl+v，释放按下的两个键后，按下对应的功能键 (Enter 键) 即可。

一些 sed 行命令集

'/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 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)

- 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 如果输入是文件列表，**awk** 将按顺序处理列表中的每个文件。

⑦ **awk** 处理完所有的输入后，若存在 **END**，执行相应的 **actions**。

awk 举例

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

```
awk '{print $1,$3}' shipped
```

```
awk '/Mar/' shipped
```

```
awk 'BEGIN{print "Mon data"}/Mar/{print
$1,$3}' shipped
```

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

```
awk -F: -f awkfile1 employees2
```

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

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}' shipped
```

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

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}' shipped
```

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

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

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

注：表达式中有比较运算时，一般用圆括号括起来

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。

```
awk '{print $1}' datafile2
```

```
awk -F: '{print $1}' datafile2
```

```
awk -F'[ :]' '{print $1}' datafile2
```

❑ 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 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：
      +=， -=， *=， /=， %=， ^=， ++， --

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

流控制结构举例：**awkfile2**

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

```
awk '{$6=3*$2+$3; print}' shipped
```

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**)

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

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 shipped
```

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

AWK 内置函数

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

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("hello",/l[^l]/)}'
```

- **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)

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)}'
```

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

内置系统函数

③ 常见 awk 内置系统函数

- **close(filename)**
  将输入或输出的文件 **filename** 关闭。
- **system(command)**
  此函数允许调用操作系统的指令，执行完毕后返回 **awk**

```
awk 'BEGIN {system("ls")}'
```

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 \
}' shipped
```

AWK 中的数组

- 数组使用前，无需预先定义，也不必指定数组元素个数
- 访问数组的元素

  经常使用循环来访问数组元素

```
for (element in array_name ) print \
array_name[element]
```

```
awk 'BEGIN{print \
split("123#456#789",mya,"#"); \
for (i in mya) {print mya[i]}}'
```

字符串屏蔽

- 使用字符串或正则表达式时，有时需要在输出中加入一新行或一个特殊字符。这时就需要字符串屏蔽。
- **awk** 中常用的字符串屏蔽序列

\b	退格键	\t	tab 键
\f	走纸换页	\n	新行
\r	回车键	\ddd	八进制值 ASCII 码
\c	任意其他特殊字符。如: \\ 为反斜线符号		

```
awk 'BEGIN{print  \
 "\nMay\tDay\n\nMay\t\104\141\171"}'
```

AWK 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 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}'
```

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。

Linux 操作系统

Shell 的输入与输出

Shell 的输入与输出

- **shell** 的输入与输出主要包括：
  - `echo` 命令
  - `read` 命令
  - `tee` 命令
  - `cat` 命令
  - 管道
  - 重定向

echo 命令

❑ echo

- ◆ 使用 **echo** 命令可以显示文本行、字符串或变量的值
- ◆ **echo** 命令的一些细节在 System V、BSD 和 Linux 这三种UNIX-like 系统上会有所不同，这里以 Linux 为主。
- ◆ echo 命令的一般形式：

```
echo [-e] [-n]  string
```

其中：

string：字符串，可以含 **shell** 变量、转义符等，
一般用双引号括起来

-e：让 **echo** 解释 **string** 中的转义符

-n：禁止 **echo** 输出后输出 **NEWLINE**（换行）。

echo 命令

- echo 命令支持的转义符

\num	ASCII码为num（八进制）的字符		
\a	alert (bell) 响铃	\r	carriage return 回车
\b	backspace 退格	\t	horizontal tab 水平制表符
\f	form feed 换页		
\c	suppress trailing newline 不换行	\v	vertical tab 垂直制表符
\n	new line 换行	\\	backslash 反斜杠

```
echo -e  "Hello \bworld"
```

echo 命令举例

```
echo "your home directory is $HOME"
```

```
echo -n "your home directory is $HOME"
```

```
echo -e "your home directory is $HOME\c"
```

```
echo -e "User: $USER\tUID: $UID"
```

read 命令

- 从键盘或文件的某一行中读取输入，并将其赋给变量。
- **read** 命令的一般形式：

```
read  variable1 variable2 ...
```

```
read -p "提示信息"  var1 var2 ...
```

- 如果只指定了一个变量，**read** 将会把输入行的所有内容赋给该变量，直至遇到第一个文件结束符或回车。
- 如果指定了多个变量，**read** 用空格作为分隔符把输入行分成多个域，分别赋给各个变量。如果输入的文本域数量多于 **read** 给出的变量数，**read** 将所有的超长部分赋予最后一个变量。

read 命令举例

```
read name    // John Lemon Doe
```

```
read name subname    // John Lemon Doe
```

```
#!/bin/bash
echo -e "First name: \c"
read name
echo -e "Middle name: \c"
read middle
echo -e "surname: \c"
read surname
echo "the name is $name $middle $surname"
```

cat 命令

◆ **cat** 是一个简单而通用的命令，可以用它来显示文件内容，创建文件，还可以用它来显示控制字符。

◆ **cat** 命令的一般形式：

```
cat [-n][-b][-t][-e] file1 file2 ...
```

-n：显示行号

-b：显示行号（不含空行）

-t：显示制表符

-e：显示行结束符

◆ 创建文件：

```
cat file1 file2 > newfile
```

合并文件

```
cat > newfile
```

输入文本，按 **ctrl+d** 结束输入

管道

- |：把一个命令的输出传递给另一个命令作为输入。

```
comd1 | comd2
```

例： 显示当前目录下的所有子目录

```
ls -l | grep ^d
```

tee 命令

- 把输出的一个副本输送到标准输出，另一个副本拷贝到相应的文件中。
- tee 命令的一般形式：

```
tee [-a] filename
```

-a：追加到文件末尾

- tee 命令一般与管道结合使用
- 例：

```
ls  | tee list.out
```

标准输入、输出和错误

- 标准输入（**STDIN**），文件描述符为 **0**
- 标准输出（**STDOUT**），文件描述符为 **1**
- 标准错误（**STDERR**），文件描述符为 **2**

重定向

comd > filename	**comd >> filename**
comd 2 > filename	**comd 2 >> filename**
comd > filename 2>&1	**comd >> filename 2>&1**
comd < filename	
comd < filename >filename2	
comd << delimiter 从标准输入中读入，直至遇到 **delimiter** 分界符	
comd <&m 把文件描述符 **m** 作为标准输入	
comd >&m 把标准输出重定向到文件描述符 **m** 中	
comd >& filename 重定向标准输出和错误到指定文件	
comd <&- 关闭标准输入	

命令的执行顺序

- 在执行某个命令的时候，有时需要依赖于前一个命令是否执行成功。例如，假设你希望将一个目录中的文件全部拷贝到另外一个目录中后，然后删除源目录中的全部文件。在删除之前，你希望能够确信拷贝成功，否则就有可能丢失所有的文件。

- 如果希望在成功地执行一个命令之后再执行另一个命令，或者在一个命令失败后再执行另一个命令，**&&** 和 **||** 可以完成这样的功能。

命令的执行顺序

- 使用 &&

```
comd1 && comd2
```

&& 左边的命令（comd1）返回真（即返回 0，成功被执行）后，&& 右边的命令（comd2）才能够被执行。

- 使用 ||

```
comd1 || comd2
```

如果 || 左边的命令（comd1）未执行成功，那么就执行 || 右边的命令（comd2）。

( ) 和 { } 的使用

- 几个命令合在一起执行，可以使用下面两种方法

```
( comd1; comd2; ... )
```

```
{ comd1; comd2; ... }
```

- ( )、{ } 一般和 && 或 || 一起使用

```
cp file1 file2 || \
( echo "cp failed" | mail jypan; exit; )
```

- 在编写 shell 脚本时，使用 && 和 ||，可根据前面命令的返回值来控制其后面命令的执行，对构造判断语句很有用。

Linux 操作系统

Shell 脚本编程

Shell 变量

主要内容和学习要求

- shell 变量的设置、查看和清除
- 局部变量与作用域
- 环境变量及其设置
- 内置命令 declare 和 printf
- 变量测试与赋值
- 位置变量与变量的间接引用
- 命令替换的两种方式
- 整型变量的算术运算和算术扩展
- 数组变量及其引用方法

变量

- 变量命名
  - 变量名必须以字母或下划线开头，后面可以跟字母、数字或下划线。任何其它字符都标志变量名的结束。
  - 变量名关于大小写敏感。
- 变量类型：
  - 根据变量的作用域，变量可以分为局部变量和环境变量
  - 局部变量只在创建它们的 shell 中可用。而环境变量则在所有用户进程中可用，通常也称为全局变量。
- 变量赋值：`variable=value`
  - 等号两边不能有空格
  - 如果要给变量赋空值，可以在等号后面跟一个换行符

变量

- 显示变量的值

```
echo $variable  或  echo ${variable}
```

- 清除变量

```
unset variable
```

- 显示所有变量

```
set
```

例：

```
myname=jypan
echo $myname
unset myname
echo $myname
```

变量举例

```
[jypan@qtm213 ~]$ round=world
[jypan@qtm213 ~]$ echo $round

[jypan@qtm213 ~]$ name=Peter Piper

[jypan@qtm213 ~]$ name="Peter Piper"
[jypan@qtm213 ~]$ echo $name

[jypan@qtm213 ~]$ x=
[jypan@qtm213 ~]$ echo $x

[jypan@qtm213 ~]$ file.bak="$HOME/junk"
```

局部变量和作用域

- **变量的作用域**

  是指变量在一个程序中那些地方可见。对于 **shell** 来说，局部变量的作用域限定在创建它们的 **shell** 中。

例：

```
[jypan@qtm213 ~]echo $$
5617
[jypan@qtm213 ~]round=world
[jypan@qtm213 ~]echo $round
```

$ 是特殊变量，用来存储当前运行进程的PID

```
[jypan@qtm213 ~]bash
[jypan@qtm213 ~]echo $$
5751
[jypan@qtm213 ~]echo $round

[jypan@qtm213 ~]exit
```

只读变量

- **只读变量**

  是指不能被清除或重新赋值的变量。

```
readonly variable
```

例：

```
[jypan@qtm213 ~]myname=jypan
[jypan@qtm213 ~]readonly myname
[jypan@qtm213 ~]unset myname
bash: unset: myname: cannot unset: readonly variable
```

```
[jypan@qtm213 ~]myname="Jianyu Pan"
bash: myname: readonly variable
```

环境变量

- 环境变量
  - 作用域包含创建它们的 shell，以及从该 shell 产生的任意子 shell 或进程。
  - 按照惯例，环境变量通常使用大写。
  - 环境变量是已经用 export 内置命令输出的变量。
- 变量被创建时所处的 shell 称为父 shell。如果在父 shell 中启动一个新的 shell（或进程），则该 shell (或进程) 被称为子shell (或子进程)。
  - 环境变量就象 DNA，可以从父亲传递给儿子，再到孙子，但不能从子进程传递给父进程。

环境变量举例

- 设置环境变量

```
export variable=value
```

```
variable=value; export variable
```

例：

```
[jypan@fish213 ~]$ echo $$
[jypan@fish213 ~]$ export round=world
[jypan@fish213 ~]$ bash
[jypan@fish213 ~]$ echo $$
15175
[jypan@fish213 ~]$ echo $round
```

```
export -n variable
```

将全局变量转换成局部变量

```
export -p
```

列出所有全局变量

内置命令 declare

- 内置命令 declare 可用来创建变量。

```
declare [选项] variable=value
```

declare 常用选项

选项	含义
-r	将变量设为只读 (*readonly*)
-x	将变量输出到子 shell 中（*export* 为全局变量）
-i	将变量设为整型 (*integer*)
-a	将变量设置为一个数组 (*array*)
-f	列出函数的名字和定义 (*function*)
-F	只列出函数名

declare 举例

例：

```
declare myname=jypan
```

```
declare -r myname=jypan
unset myname
declare myname="Jianyu Pan"
```

```
declare -x myname2=pjy
```

```
myname2=pjy
declare -x myname2
```

```
declare
```

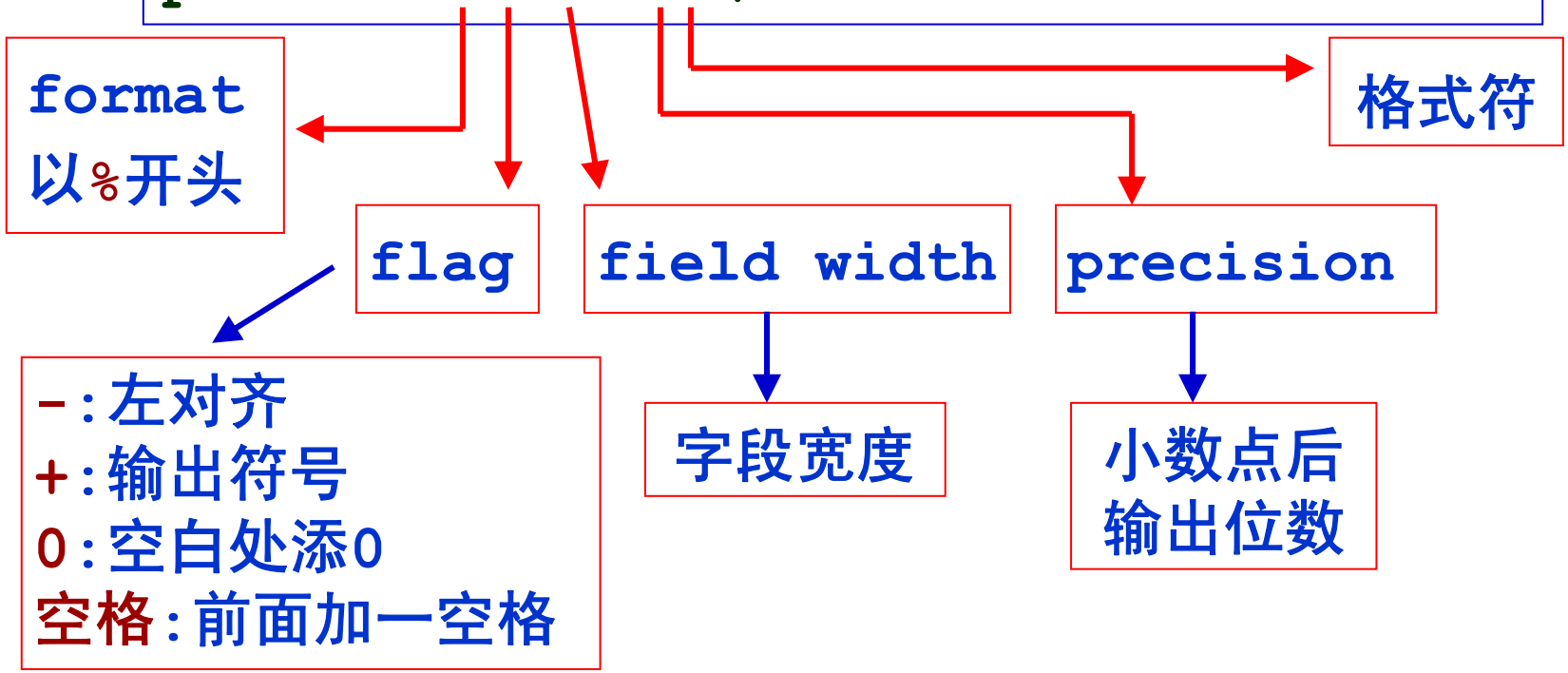
printf 命令

- printf 可用来按指定的格式输出变量

```
printf format 输出参数列表
```

printf 的打印格式与 C 语言中的 printf 相同

```
printf "%-12.5f\n"  123.456
```



printf 命令

printf 命令的格式说明符

c	字符型	g/G	浮点数（自动）
d	十进制整数	o	八进制
e/E	浮点数（科学计数法）	s	字符串
f	浮点数（小数形式）	x/X	十六进制

format 中还可以使用

\a	警铃	\t	水平制表符
\b	退后一格	\v	垂直制表符
\n	换行	\\	反斜杠
\f	换页	\"	双引号
\r	回车	%%	百分号

printf 命令举例

例：

```
printf "The number is: %.2f\n" 100
```

```
printf "%-20s|%12.5f|\n" "Joy" 10
```

```
printf "%-10d%010o%+10x\n" 20 20 20
```

```
printf "%6d\t%6o\"%6x\"\n" 20 20 20
```

变量测试

- shell 提供一些专用的修饰符来检查某个变量是否已被设置，然后根据测试结果指定变量的值，也称变量置换

${var:-word}	如果var存在且非空，则表达式${var:-word}的值为$var；如果var未定义或为空值，则表达式的值为word，但var的值不变。
${var:=word}	如果var存在且非空，则表达式${var:=word}的值为$var；如果var未定义或为空值，则表达式的值为word，且var被赋值word。
${var:?word}	如果var存在且非空，则表达式的值为$var；如果var未定义或为空值，则输出信息word，并终止脚本。
${var:+word}	如果var存在且非空，则表达式的值为word；否则返回空值，但var的值不变。

变量测试举例

例：

```
color=blue
newcolor=${color:-grey}
```

```
unset color
echo "The sky is ${color:-grey} today"
echo $color
```

```
echo "The sky is ${color:=grey} today"
echo $color
```

```
echo "The sky is ${color:?error} today"
echo $color
```

```
echo "The sky is ${color:+blue} today"
echo $color
```

位置参量（命令行参数）

- 位置参量是一组特殊的内置变量，通常被 shell 脚本用来从命令行接受参数，或被函数用来保存传递给它的参数。
- 执行 shell 脚本时，用户可以通过命令行向脚本传递信息，跟在脚本名后面的用空格隔开的每个字符串都称为位置参量。
- 在脚本中使用这些参数时，需通过位置参量来引用。例如： $1 表示第一个参数，$2 表示第二个参数，以此类推。 $9 以后需要用花括号把数字括起来，如第 10 个位置参量以 ${10} 的方式来访问。

位置参量列表

$0	当前脚本的文件名
$1-$9	第 1 个到第 9 个位置参量
${10}	第 10 个位置参量，类似地，有 ${11}，...
$#	位置参量的个数
$*	以单字符串显示所有位置参量
$@	未加双引号时与 $* 含义相同，加双引号时有区别
$$	脚本运行的当前进程号
$!	最后一个后台运行的进程的进程号
$?	显示前面最后一个命令的退出状态。 0 表示没有错误，其他任何值表示有错误。
$-	显示当前 shell 使用的选项

位置参量举例

例1： 显示位置参量的值。

(shprg1.sh)

例2： $* 与 $@ 的区别：二者仅在被双引号括起来时有区别，此时前者将所有位置参量看成一个字符串，而后者将每个位置参量看成单独的字符串。

(shprg2.sh)

- 如果位置参量中含有空格，则需要使用双引号

```
args2.sh  This is "Peter Piper"
```

basename

- **basename**

  返回不含路径的文件名或目录名

```
basename /home/jypan/linux
```

```
basename ~
```

```
shprg3.sh
```

变量的间接引用

```
str1="Hello World"
str2=str1
echo $str2
```

- 如何通过 str2 的值来引用 str1 的值？(间接引用)

```
echo $$str2 ?
echo ${$str2} ?
```

```
eval newstr=\$$str2
echo $newstr
```

```
newstr=${!str2}   # bash2.0以上才支持
echo $newstr      # echo ${!str2}
```

脚本范例：args3.sh Hello world!

eval

```
eval arg1 [arg2] ... [argN]
```

- 将所有的参数连接成一个表达式，并计算或执行该表达式，参数中的任何变量都将被展开。

```
listpage="ls -l | more"
$listpage
```

```
listpage="ls -l | more"
eval $listpage
```

```
eval newstr=\$$str2
```

命令替换

- 命令替换的用处是将命令的输出结果赋给一个变量，或者用命令的输出结果代入命令所处的位置。
- 所有的 shell 都支持使用反引号来执行命令替换。

```
echo "The hostname is `hostname`"
```

- Bash 除了使用反引号来执行命令替换外，还有另外一种替换方法：将命令放在前有美元符的一对圆括号内。

```
echo "The hostname is $(hostname)"
```

- 命令替换可以嵌套使用。

  如果使用反引号，则内部的反引号必须用反斜杠来转义。

```
echo `basename \`pwd\``
echo $(basename $(pwd))   #see shprg4.sh
```

算术运算

- Bash 变量是没有严格的类型定义，本质上 Bash 变量都是字符串，但 Bash 也允许定义整型变量，可以参加运算与比较。
- 可以用 declare 命令定义整型变量。

```
declare -i  num
num=1; echo $num
num=$num+3; echo $num   #num=num+3
num2=$num+3; echo $num2
```

```
declare -i   #列出所有整型变量
```

```
num2=1; echo $num2
num2=$num2+1; echo $num2
```

未被定义为整型的变量不能直接参加算术运算！

整数运算

- declare 定义的整型变量可以直接进行算术运算。
- 未被定义为整型的变量，可用内置命令 let 进行算术运算。

```
num2=1; echo $num2
let num2=4+1; echo $num2
let num2=$num2+1; echo $num2
```

- 赋值符号和运算符两边不能留空格！
- 如果将字符串赋值给一个整型变量时，则变量的值为 0
- 如果变量的值是字符串，则进行算术运算时设为 0

```
let num2=4 + 1
let "num2=4 + 1" # 用引号忽略空格的特殊含义
```

用 let 命令进行算术运算时，最好加双引号。

let 命令运算操作符

let 命令操作符

+、 -、 *、 /	（四则运算）
**、 %	（幂运算 和 模运算，取余数）
<<、 >>	（按位左移 和 按位右移）
&、 ^、 |	（按位与 、按位异或 和 按位 或）
=、 +=、 -= 、 *=、 /= 、 %= <<= 、 >>= 、 &=、 ^=、 |=	（赋值运算）
<、 >、 <=、 >=、 ==、 !=	（比较操作符）
&&、 ||	（逻辑与 和 逻辑 或）

注：按位运算是以二进制形式进行的。

```
a=2; let "a<<=2"    #用引号忽略 << 的特殊含义
```

浮点数运算

- Bash 只支持整数运算，但可以通过使用 bc 和 awk 工具来处理更复杂的运算。

```
n=$(echo "scale=3; 13/2" | bc )
echo $n
```

```
m=`awk 'BEGIN{x=2.45;y=3.123; \
   printf "%.3f\n", x*y}'`
echo $m
```

算术扩展

- 除了使用 let 命令外，Shell 可以通过下面两种方式对一个算术表达式进行求值。

```
$[expression]
$((expression))
```

例：

```
num2=$[4 + 1]; echo $num2
num2=$(($num2*2-3)); echo $num2
```

用 let 命令和 $[…]，$((…)) 进行整数运算时，美元符号 $ 可以省略，但最好写上。

注意 ${…}，$(…)，$[…]，$((…)) 的不同作用

数组变量

- Bash 2.x 以上支持一维数组，下标从 0 开始。
- 数组可以用 declare 命令创建， 或直接给变量名加下标来创建。

```
declare -a variable
variable=(item1 item2 item2 ... )
```

```
variable=(item1 item2 item2 ... )
variable[n]=value
```

- 数组的引用

```
${variable[n]}
```

数组变量举例

```
declare -a stu
stu=(math1101 math1102 math1103)
echo ${stu[0]}   # 列出stu的第一个元素
echo ${stu[*]}   # 列出stu的所有元素
echo ${#stu[*]} # 给出数组stu中元素的个数
```

- 数组与数组元素的删除

```
unset stu[1]     # 删除stu的第二个元素
unset stu        # 删除整个数组
```

- 数组赋值时无须按顺序赋值

```
x[3]=100; echo ${x[*]}
state=(ME [3]=CA [2]=NT); echo ${state[*]}
```

相关命令小结

```
echo $variable  或  echo ${variable}
```

```
unset variable
```

```
set
```

```
readonly variable
```

```
export variable=value
export -n variable
export -p
```

```
declare [选项] variable=value
```

```
printf format 输出参数列表
```

相关命令小结

basename

let

$[expression] 、$((expression))

eval newstr=\$$str2 、newstr=${!str2}

`hostname` 、$(hostname)

${var:-word}、${var:=word}、${var:?word}、${var:+word}

$0、$1-$9、${n}、$#、$*、$@、$$、$!、$?、$-

Linux 操作系统

Shell 脚本编程

主要内容和学习要求

- 掌握创建 **shell** 脚本的基本步骤
- 学会使用条件测试
- 掌握 **if** 条件结构与 **case** 选择结构
- 掌握 **for** 循环、**while** 循环和 **until** 循环结构
- 学会 **shift** 命令的使用
- 学会 **shell** 脚本的调试

Shell 脚本

Shell 脚本

当命令不在命令行中执行，而是从一个文件中执行时，该文件就称为 **shell 脚本**。**shell 脚本**按行解释。

Shell 脚本的编写

- **Shell 脚本**是纯文本文件，可以使用任何文本编辑器编写
- **Shell 脚本**通常是以 .sh 作为后缀名

Shell 脚本的执行

```
chmod +x script_name
./script_name
```

```
sh script_name
```

Shell 脚本

Shell 脚本的格式

- 第一行：指定用哪个程序来编译和执行脚本。

```
#!/bin/bash
```

```
#!/bin/sh
```

```
#!/bin/csh
```

- 可执行语句和 shell 控制结构

  一个 shell 脚本通常由一组 Linux 命令、shell 命令、控制结构和注释语句构成。

- 注释：以“#”开头，可独占一行，或跟在语句的后面。

在脚本中多写注释语句是一个很好的编程习惯

Shell 脚本举例

```
#!/bin/bash
# This is the first Bash shell program
# Scriptname: greetings.sh
echo
echo -e "Hello $LOGNAME, \c"
echo    "it's nice talking to you."
echo -e "Your present working directory is:"
pwd # Show the name of present directory
echo
echo  "The time is `date +%T`!. \nBye"
echo
```

```
sh greetings
```

```
chmod +x greetings
greetings
```

Shell 脚本举例

```
#!/bin/bash
# This script is to test the usage of read
# Scriptname: ex4read.sh
echo "=== examples for testing read ==="
echo -e "What is your name? \c"
read name
echo "Hello $name"
echo
echo -n "Where do you work? "
read
echo "I guess $REPLY keeps you busy!"
echo
read -p "Enter your job title: "
echo "I thought you might be an $REPLY."
echo
echo "=== End of the script ==="
```

条件测试

- 条件测试可以根据某个特定条件是否满足，来选择执行相应的任务。
- **Bash** 中允许测试两种类型的条件：

  命令成功或失败，表达式为真或假

- 任何一种测试中，都要有退出状态（返回值），退出状态为 0 表示命令成功或表达式为真，非0 则表示命令失败或表达式为假。
- 状态变量 **$?** 中保存命令退出状态的值

```
grep $USER /etc/passwd
echo $?
grep hello /etc/passwd; echo $?
```

测试表达式的值

- 表达式测试包括字符串测试、整数测试和文件测试。
- 内置测试命令 **test**
  - 通常用 **test** 命令来测试表达式的值

```
x=5; y=10
test $x -gt $y
echo $?
```

  - **test** 命令可以用 ***方括号*** 来代替

```
x=5; y=10
[ $x -gt $y ]
echo $?
```

方括号前后要留空格!

测试表达式的值

- 2.x 版本以上的 Bash 中可以用双方括号来测试表达式的值，此时可以使用通配符进行模式匹配。

```
name=Tom
[ $name = [Tt]?? ]
echo $?
```

```
[[ $name = [Tt]?? ]]
echo $?
```

字符串测试

❑ 字符串测试

操作符两边必须留空格!

`[ -z str ]`	如果字符串 `str` 长度为 `0`，返回真
`[ -n str ]`	如果字符串 `str` 长度不为 `0`，返回真
`[ str1 = str2 ]`	两字符串相等
`[ str1 != str2 ]`	两字符串不等

```
name=Tom; [ -z $name ]; echo $?
```

```
name2=Andy; [ $name = $name2 ] ; echo $?
```

整数测试

- 整数测试，即比较大小

操作符两边必须留空格！

[int1 -eq int2]	int1 等于 int2
[int1 -ne int2]	int1 不等于 int2
[int1 -gt int2]	int1 大于 int2
[int1 -ge int2]	int1 大于或等于 int2
[int1 -lt int2]	int1 小于 int2
[int1 -le int2]	int1 小于或等于 int2

```
x=1; [ $x -eq 1 ]; echo $?
```

```
x=a; [ $x -eq 1 ]; echo $?
```

整数测试

- 整数测试也可以使用 `let` 命令或双圆括号
  - 相应的操作符为：

只能用于整数测试!

```
==  、 !=  、 >  、 >=  、 <  、 <=
```

  - 例：

```
x=1; let "$x == 1"; echo $?
```

```
x=1; (($x+1>= 2 )); echo $?
```

- 两种测试方法的区别
  - 使用的操作符不同
  - `let` 和 双圆括号中可以使用算术表达式，而中括号不能
  - `let` 和 双圆括号中，操作符两边可以不留空格

逻辑测试

逻辑测试

`[ expr1 -a expr2 ]`	逻辑与，都为真时，结果为真
`[ expr1 -o expr2 ]`	逻辑或，有一个为真时，结果为真
`[ ! expr ]`	逻辑非

```
x=1; name=Tom;
[ $x -eq 1 -a -n $name ]; echo $?
```

注：不能随便添加括号

```
[ ( $x -eq 1 ) -a ( -n $name ) ]; echo $?
```

X

逻辑测试

- 可以使用模式的逻辑测试

`[[ pattern1 && pattern2 ]]`	逻辑与
`[[ pattern1 || pattern2 ]]`	逻辑或
`[[ ! pattern ]]`	逻辑非

```
x=1; name=Tom;
[[ $x -eq 1 && $name = To? ]]; echo $?
```

文件测试

- **文件测试：文件是否存在，文件属性，访问权限等。**

常见的文件测试操作符

-f fname	**fname** 存在且是普通文件时，返回真 (即返回 0)
-L fname	**fname** 存在且是链接文件时，返回真
-d fname	**fname** 存在且是一个目录时，返回真
-e fname	**fname**（文件或目录）存在时，返回真
-s fname	**fname** 存在且大小大于 0 时，返回真
-r fname	**fname**（文件或目录）存在且可读时，返回真
-w fname	**fname**（文件或目录）存在且可写时，返回真
-x fname	**fname**（文件或目录）存在且可执行时，返回真

- 更多文件测试符参见 **test** 的在线帮助 **man test**

检查空值

- 检查空值

```
[ "$name" = "" ]
```

```
[ ! "$name" ]
```

```
[ "X${name}" != "X" ]
```

if 条件语句

- 语法结构

```
if expr1          # 如果expr1 为真(返回值为0)
then              # 那么
    commands1     # 执行语句块 commands1
elif expr2        # 若expr1 不真，而expr2 为真
then              # 那么
    commands2     # 执行语句块 commands2
 ... ...          # 可以有多个 elif 语句
else              # else 最多只能有一个
    commands4     # 执行语句块 commands4
fi                # if 语句必须以单词 fi 终止
```

几点说明

- **elif** 可以有任意多个（**0** 个或多个）
- **else** 最多只能有一个（**0** 个或 **1** 个）
- **if** 语句必须以 **fi** 表示结束
- **expr** 通常为条件测试表达式；也可以是多个命令，以最后一个命令的退出状态为条件值。
- **commands** 为可执行语句块，如果为空，需使用 **shell** 提供的空命令“ **:** ”，即冒号。该命令不做任何事情，只返回一个退出状态 **0**
- **if** 语句可以嵌套使用

ex4if.sh，**chkperm.sh**，**chkperm2.sh**，
name_grep，**tellme**，**tellme2**，**idcheck.sh**

case 选择语句

❑ 语法结构

```
case expr in  # expr 为表达式，关键词 in 不要忘！
  pattern1)   # 若 expr 与 pattern1 匹配，注意括号
    commands1 # 执行语句块 commands1
    ;;        # 跳出 case 结构
  pattern2)   # 若 expr 与 pattern2 匹配
    commands2 # 执行语句块 commands2
    ;;        # 跳出 case 结构
  ... ...     # 可以有任意多个模式匹配
  *)          # 若 expr 与上面的模式都不匹配
    commands  # 执行语句块 commands
    ;;        # 跳出 case 结构
esac          # case 语句必须以 esac 终止
```

几点说明

- 表达式 **expr** 按顺序匹配每个模式，一旦有一个模式匹配成功，则执行该模式后面的所有命令，然后退出 **case**。

- 如果 **expr** 没有找到匹配的模式，则执行缺省值“ ***)** ”后面的命令块 (类似于 **if** 中的 **else**)；“ ***)** ”可以不出现。

- 所给的匹配模式 **pattern** 中可以含有通配符和“ **|** ”。

- 每个命令块的最后必须有一个双分号，可以独占一行，或放在最后一个命令的后面。

- **case** 语句举例：**yes_no.sh**

for 循环语句

❑ 语法结构

```
for variable in list
# 每一次循环，依次把列表 list 中的一个值赋给循环变量
do                  # 循环开始的标志
   commands         # 循环变量每取一次值，循环体就执行一遍
done                # 循环结束的标志
```

❑ 几点说明

- 列表 **list** 可以是命令替换、变量名替换、字符串和文件名列表 (可包含通配符)
- **for** 循环执行的次数取决于列表 **list** 中单词的个数
- **for** 循环体中一般要出现循环变量，但也可以不出现

for 循环执行过程

- 循环执行过程

执行第一轮循环时，将 **list** 中的第一个词赋给循环变量，并把该词从 **list** 中删除，然后进入循环体，执行 **do** 和 **done** 之间的命令。下一次进入循环体时，则将第二个词赋给循环变量，并把该词从 **list** 中删除，再往后的循环也以此类推。当 **list** 中的词全部被移走后，循环就结束了。

forloop.sh，mybackup.sh

- 位置参量的使用： **$*** 与 **$@** **greet.sh**
- 可以省略 **in list**，此时使用位置参量

permx.sh tellme greet.sh / permx.sh *

while 循环语句

❑ 语法结构

```
while expr    # 执行 expr
do # 若 expr 的退出状态为0，进入循环，否则退出while
   commands   # 循环体
done          # 循环结束标志，返回循环顶部
```

❑ 执行过程

先执行 expr，如果其退出状态为 0，就执行循环体。执行到关键字 done 后，回到循环的顶部，while 命令再次检查 expr 的退出状态。以此类推，循环将一直继续下去，直到 expr 的退出状态非 0 为止。

until 循环语句

❑ 语法结构

```
until expr    # 执行 expr
do # 若expr的退出状态非0，进入循环，否则退出until
   commands   # 循环体
done          # 循环结束标志，返回循环顶部
```

❑ 执行过程

与 **while** 循环类似，只是当 **expr** 退出状态非 **0** 时才执行循环体，直到 **expr** 为 **0** 时退出循环。

break 和 continue

```
break [n]
```

- 用于强行退出当前循环。
- 如果是嵌套循环，则 **break** 命令后面可以跟一数字 **n**，表示退出第 **n** 重循环（最里面的为第一重循环）。

```
continue [n]
```

- 用于忽略本次循环的剩余部分，回到循环的顶部，继续下一次循环。
- 如果是嵌套循环，**continue** 命令后面也可跟一数字 **n**，表示回到第 **n** 重循环的顶部。

例：**months.sh**

exit 和 sleep

exit 命令

```
exit n
```

exit 命令用于退出脚本或当前进程。n 是一个从 0 到 255 的整数，0 表示成功退出，非零表示遇到某种失败而非正常退出。该整数被保存在状态变量 $? 中。

sleep 命令

```
sleep n
```

暂停 n 秒钟

select 循环与菜单

❑ 语法结构

```
select variable in list
do                  # 循环开始的标志
   commands         # 循环变量每取一次值，循环体就执行一遍
done                # 循环结束的标志
```

❑ 说明

- **select** 循环主要用于创建菜单，按数字顺序排列的菜单项将显示在标准错误上，并显示 **PS3** 提示符，等待用户输入
- 用户输入菜单列表中的某个数字，执行相应的命令
- 用户输入被保存在内置变量 **REPLY** 中。

例：**runit.sh**

select 与 case

- select 是个无限循环，因此要记住用 break 命令退出循环，或用 exit 命令终止脚本。也可以按 ctrl+c 退出循环。
- select 经常和 case 联合使用

例：goodboy.sh

- 与 for 循环类似，可以省略 in list，此时使用位置参量

循环控制 shift 命令

```
shift [n]
```

- 用于将参量列表 list 左移指定次数，缺省为左移一次。
- 参量列表 list 一旦被移动，最左端的那个参数就从列表中删除。while 循环遍历位置参量列表时，常用到 shift。

例：

```
doit.sh a b c d e f g h
```

```
shft.sh a b c d e f g h
```

随机数和 expr 命令

- 生成随机数的特殊变量

```
echo $RANDOM
```

- **expr：** 通用的表达式计算命令

表达式中参数与操作符必须以空格分开，表达式中的运算可以是算术运算，比较运算，字符串运算和逻辑运算。

```
expr 5 % 3
```

```
expr 5 \* 3   # 乘法符号必须被转义
```

字符串操作

❑ 字符串操作

m 的取值从 0 到 ${#var}-1

${#var}	返回字符串变量 var 的长度
${var:m}	返回${var}中从第m个字符到最后的部分
${var:m:len}	返回${var}中从第m个字符开始，长度为len的部分
${var#pattern}	删除${var}中开头部分与pattern匹配的最小部分
${var##pattern}	删除${var}中开头部分与pattern匹配的最大部分
${var%pattern}	删除${var}中结尾部分与pattern匹配的最小部分
${var%%pattern}	删除${var}中结尾部分与pattern匹配的最大部分
${var/old/new}	用new替换${var}中第一次出现的old
${var//old/new}	用new替换${var}中所有的old(全局替换)

注：pattern，old 中可以使用通配符。

例：ex4str

脚本调试

```
sh -x 脚本名
```

该选项可以使用户跟踪脚本的执行，此时 **shell** 对脚本中每条命令的处理过程为：先执行替换，然后显示，再执行它。**shell** 显示脚本中的行时，会在行首添加一个加号“ **+** ”。

```
sh -v 脚本名
```

在执行脚本之前，按输入的原样打印脚本中的各行。

```
sh -n 脚本名
```

对脚本进行语法检查，但不执行脚本。如果存在语法错误，**shell** 会报错，如果没有错误，则不显示任何内容。

编程小结：变量

- 局部变量、环境变量（export、declare -x）
- 只读变量、整型变量

```
例：declare -i x; x="hello"; echo $x
```

```
0
```

- 位置参量（$0,$1,...,$*,$@,$#,$$,$?）
- 变量的间接引用（eval，${!str}）

```
例：name="hello"; x="name"; echo ${!x}
```

```
hello
```

- 命令替换（`cmd`、$(cmd)）
- 整数运算

declare 定义的整型变量可以直接进行运算，

否则需用 let 命令或 $[...]、$((...)) 进行整数运算。

编程小结：输入输出

- 输入：read

```
read var1 var2 ...
```

```
read
```
→ REPLY

```
read -p "提示"
```
→ REPLY

- 输出：printf

输出参数用空格隔开

```
printf "%-12.5f \t %d \n" 123.45 8
```

format
以%开头

flag
- -：左对齐
- +：输出符号
- 0：空白处添0
- 空格：前面加一空格

field width
字段宽度

precision
小数点后
输出位数

格式符
- c d e f g s o x

\b \n \r \t \v \\ \" %%

编程小结：条件测试

- 字符串测试

操作符两边必须留空格！

[-z string]	如果字符串string长度为0，返回真
[-n string]	如果字符串string长度不为0，返回真
[str1 = str2]	两字符串相等（也可以使用 == ）
[str1 != str2]	两字符串不等

如果使用双方括号，可以使用 通配符 进行模式匹配。

[[str1 = str2]]	两字符串相等（也可以使用 == ）
[[str1 != str2]]	两字符串不等
[[str1 > str2]]	str1大于str2,按ASCII码比较
[[str1 < str2]]	str1小于str2,按ASCII码比较

例：`name=Tom; [[ $name > Tom ]]; echo $?`

编程小结：条件测试

- 整数测试

注意这两种方法的区别！

[int1 -eq int2]	int1 等于 int2
[int1 -ne int2]	int1 不等于 int2
[int1 -gt int2]	int1 大于 int2
[int1 -ge int2]	int1 大于或等于 int2
[int1 -lt int2]	int1 小于 int2
[int1 -le int2]	int1 小于或等于 int2

((int1 == int2))	int1 等于 int2
((int1 != int2))	int1 不等于 int2
((int1 > int2))	int1 大于 int2
((int1 >= int2))	int1 大于或等于 int2
((int1 < int2))	int1 小于 int2
((int1 <= int2))	int1 小于或等于 int2

编程小结：条件测试

- 逻辑测试

`[ expr1 -a expr2 ]`	逻辑与，都为真时，结果为真
`[ expr1 -o expr2 ]`	逻辑或，有一个为真时，结果为真
`[ ! expr ]`	逻辑非

如果使用双方括号，可以使用 通配符 进行模式匹配。

`[[ pattern1 && pattern2 ]]`	逻辑与
`[[ pattern1 || pattern2 ]]`	逻辑或
`[[ ! pattern ]]`	逻辑非

编程小结：条件测试

- 文件测试

-f fname	fname 存在且是普通文件时，返回真 (即返回 0)
-L fname	fname 存在且是链接文件时，返回真
-d fname	fname 存在且是一个目录时，返回真
-e fname	fname（文件或目录）存在时，返回真
-s fname	fname 存在且大小大于 0 时，返回真
-r fname	fname（文件或目录）存在且可读时，返回真
-w fname	fname（文件或目录）存在且可写时，返回真
-x fname	fname（文件或目录）存在且可执行时，返回真

编程小结：控制结构

- if 条件语句
- case 选择语句
- for 循环语句
- while 循环语句
- until 循环语句
- break、continue、sleep 命令
- select 循环与菜单
- shift 命令

- 各种括号的作用
  - ${...}, $(...), $[...], $((...))
  - [...], [[...]], ((...))

函数

- 和其它编程语言一样， Bash 也可以定义函数。
- 一个函数就是一个子程序，用于完成特定的任务，当有重复代码，或者一个任务只需要很少的修改就被重复几次执行时, 这时你应考虑使用函数。
- 函数的一般格式

```
function function_name {
    commands
}
```

```
function_name () {
    commands
}
```

函数举例

```
#!/bin/bash

fun1 () { echo "This is a function"; echo; }
# 一个函数可以写成一行，但命令之间必须用分号隔开
# 特别注意，最后一个命令后面也必须加分号

fun2 ()
{
    echo "This is fun2."
    echo "Now exiting fun2."
}
```

函数的调用

- 只需输入函数名即可调用该函数。
- 函数必须在调用之前定义

```
#!/bin/bash

fun2 ()
{
    echo "This is fun2."
    echo "Now exiting fun2."
}

fun2 # 调用函数 fun2
```

例：ex4fun2.sh，ex4fun3.sh

函数的调用

- 向函数传递参数

例：ex4fun4.sh

- 函数与命令行参数

例：ex4fun5.sh

- return 与 exit

例：ex4fun6.sh