

# 《LINUX 与 UNIX SHELL 编程指南》

读书笔记

二次发布版

张启峰

Email: [zqf620@gmail.com](mailto:zqf620@gmail.com)

## 一些废话

这是笔者第一本发到网络上的 Linux 读书笔记，利用今年十一长假，重新编辑排版了一下，再次发到网上，姑且称为“二次发布版”吧！

关于《LINUX 与 UNIX SHELL 编程指南》这本书，我看的是 chinapub.com 的电子版，在很多网站都可以下载到。虽然看电子书很累人，但我还是建议不想掏钱买书的朋友看看（有钱也不一定买的到，反正我逛书店时就没有看到过有卖这本书的），它不愧为一本经典的讲解 shell 编程的书。

当初，写这本笔记时，花了很大的力气。参考了不少资料，在我认为书中某些讲的不详细的地方，在笔记中也记述的很详细。读者可以发现，这本笔记并不是简单的摘抄。

当然，记笔记的过程也是对我的一个提高，比如，awk 和脚本编程中可以遇到的 getopts，这两个东东我一直就没搞清楚，记笔记的过程中我就把它们搞明白了（我个人认为关键是要静下心来学）。希望这本笔记能对读者您有所帮助！

在笔记中给出了很多举例，这些例子都在 Red Hat Linux 9 和 Red Flag Advanced Server 4.1 中测试通过，大多数例子还给出了注释（使用C风格的双斜线“//”注释符号）。

好了，不再废话了。最后，给您推荐两本书，都是美国佬写的。一本是机械工业出版社出版的O'Reilly的《学习Bash（第二版）》，一本是人民邮电出版社出版的Sams的《精通Shell编程（第二版）》。



张启峰(zqf620@gmail.com)

2005 年 10 月 4 日

## 第一部分 SHELL

### 第 1 章 文件安全与权限

1. 一个文件一经创建，就具有三种基本访问方式：

- 1) 读(r): 可以显示该文件的内容。
- 2) 写(w): 可以编辑或删除它。
- 3) 执行(x): 如果该文件是一个 shell 脚本或程序的话。

2. 按照所针对的用户，文件的权限可分为三类：

- 1) 文件属主: 创建该文件的用户。
- 2) 同组用户: 拥有该文件的用户组中的任何用户。
- 3) 其他用户: 即不属于拥有该文件的用户组的某一用户。

3. ls -l (列出目录内容)命令的输出的分析

```
drwxr-xr-x  2 root  root    4096 Oct 14 20:18 bin
```

- 1) 第 1 个部分第 1 个字符: 表示文件的类型，详细说明见下。
- 2) 第 1 个部分第 2-10 个字符: 分为 3 组(triplet)，分别表示文件的属主、组用户和其它用户的权限。
- 3) 第 2 个部分: 表示该文件的硬链接的数目。
- 4) 第 3 个部分: 表示文件的属主。
- 5) 第 4 个部分: 文件的属主(root)所在的缺省组(也是 root 组)。
- 6) 第 5 个部分: 表示文件的长度(以字节 byte 为单位)。
- 7) 第 6-7 个部分: Oct 14 20:18 表示文件的更新时间。
- 8) 第 8 个部分: 该文件的文件名。

4. 文件的类型

- 1) d : 目录文件，目录也是一种文件 (directory)
- 2) l : 符号链接(指向另一个文件) (link)
- 3) b : 块设备文件 (block device)
- 4) c : 字符设备文件 (character device)
- 5) p : 命名管道文件 (named pipe)
- 6) s : 套接字文件 (socket)
- 7) - : 普通文件，或者更准确地说，不属于以上几种类型的文件

5. 文件的权限位

- 1) 一个文件的权限位由 9 个字符组成，分成 3 个 triplet，分别表示文件属主(owner)、文件属主所属缺省组的用户(group)、系统中其他用户(other)所拥有的权限。

2) 一个 triplet 由 3 个字符位组成(rwx)，分别表示可读、可写、可执行。如果对应位置的字符是"-"，表示不具有对应权限。

eg: rw- 表示可读写，不可执行      r-- 表示可读，不可写、执行

3) 在文件权限位的 owner triplet 第 3 个位置的字符如果设置为"s"，称为 SUID。当某用户执行该文件时，系统用文件 owner 的 uid 替代实际执行该文件的用户的 uid，文件会认为是 owner 在执行它。

4) 在文件权限位的 group triplet 第 3 个位置的字符如果设置为"s"，称为 SGID。当某用户执行该文件时，系统用文件组拥有者的 gid 替代实际执行该文件的用户的 gid。当在某个目录文件上设置 SGID 时候，则在该目录下的所有文件和子目录都会继承使用该目录的 GID 来代替实际执行者的 GID。

5) 文件设置了 SUID 或 SGID，一般来说文件的 owner 或 group 应该有执行(x)权限。如果没有 x 权限，设置 SUID 或 SGID 是没有意义的，triplet 上第 3 个字符就会是"S"(大写)而不是"s"(小写)。eg: rwSr-Sr--

6) 给可执行文件设置 SUID/SGID 可能带来安全风险，许多系统因而忽略 SUID/SGID 权限设置

7) 在某些目录(如/tmp)的 other triplet 的第 3 个字符设置为"t"。一般来说，在目录的 other triplet 上设置 w 和 x 权限，则任何用户都可以在该目录下执行、删除文件，而设置"t"的作用就是保证非目录的 owner 用户不能删除目录下的文件。

## 6.使用 chmod 命令改变权限位

1) 符号模式 Usage: chmod [who]<operator><perm> file\_name

① who: 可以是 u g o a，表示给文件的 owner、group、other 和 all 用户设置权限。省略时默认为 a。

② operator: 可以是+ - =，分别表示增加、取消、指定权限

③ perm: 可以是 r w x s(suid 和 guid) t(粘性位) l(给文件加锁，使其他用户无法访问)

④ 举例: chmod u+xw o-w myfile      // 给 myfile 的 owner 增加 x w 权限，去除 other 用户的 w 权限

2) 绝对模式 Usage: chmod mode file\_name

① mode: 4 位 8 进制数，每一位分别用于 SUID 和 SGID、属主权限、组权限、其他用户权限。当用户给出的 mod 值小于 4 位时，系统会在数字前面加 0 补齐。

② 权限 r w x 分别用数字 4 2 1 表示，suid guid 用 4 2 表示，各个 triplet 的数字相加可得 mod 值。

eg: rwxr-xr-- 为 754，rwsr-xr-x 为 4755，rwsr-sr-x 为 6755 -rwxrwSr-- 为 2764

## 7.目录文件的权限(r w x)的含义

1) r: 可以列出该目录中的内容。

2) w: 可以在目录中创建、修改文件。目录的 w 位不设置，即使拥有目录中某文件的 w 权限也不能写该文件。

3) x: 可以搜索和访问该目录。x 位不设置，不能访问目录下的任何文件，即使拥有文件的权限。

## 8.chown 和 chgrp 修改文件的拥有权

1) chown Usage: chown [选项] [owner][.group] file\_name

① 创建文件时，文件预设 owner 就是创建该文件的用户，预设所属组(group)就是 owner 所属的缺省组。只有文件的 owner 和系统管理员才可以改变文件的拥有权，改变一个文件的 owner 时，相应的 suid 也将被清除。

② chmod 的常用选项: -R 对指定目录所有文件和子目录递归式地进行同样的操作

-h 如果 file\_name 是符号链接，则只对符号链接本身进行操作

③ owner 和 .group 表示文件所有者名和文件所属组名，可以使用 uid/gid 表示。两者不能同时省略

④ 举例: chown http.http /etc/httpd // 将目录/etc/httpd 的属主修改为 http，组修改为 http

2) chgrp Usage: chgrp [选项] group file\_name

① chgrp 命令只能用来修改文件的所属组。chgrp 和 chown 用法类似，选项也一样。

② 举例: chown http /etc/httpd // 将目录/etc/httpd 的所属组修改为 http

3) 查询用户信息和用户所属组信息

id [user\_name] 或 groups [user\_name]

8.umask 确定系统创建文件时的缺省权限位。

1) Usage: umask [umask\_mod]

2) umask\_mod 的计算方法:  $\text{umask\_mod} = 777 - \text{文件缺省权限值}$ 。

3) 系统不允许你在创建一个文件时就赋予它执行权限，即 umask\_mod 为 002 时，创建文件的缺省权限值为 665，创建目录的缺省权限值为 775。

9.符号链接

1) Usage: ln -s source\_path target\_path

2) 存在 2 种类型的链接: 硬链接和软链接。软链接又称位符号链接，符号链接实际上是指向一个文件的指针，符号链接和 Windows OS 的快捷方式有点类似。

3) 不管是否在同一个文件系统中，都可以创建链接。在创建链接的时候，不要忘记在原有目录设置执行权限。链接一旦创建，链接目录将具有权限 777(rwxrwxrwx)，但是实际的原有文件的权限并未改变。

4) 举例: ln -s /var/tmp /home/zqf/tmp // 在 zqf 主目录创建一个名为 tmp，指向/var/tmp 的符号链接

## 第 2 章 使用 find 和 xargs

1.有时可能需要在系统中查找具有某一特征的文件(如文件权限、文件属主、文件长度、文件类型等)。find 是一个非常有效的工具，它可以递归式遍历当前目录甚至于整个文件系统(本地和网络文件系统)来查找某些文件或目录，只要你具有相应的文件或目录的权限。

2.find 命令的使用

1) Usage : find [path\_name ...] [expression]

2) path\_name : 用于指定要搜索的目录, 可以是以空格分隔的多个目录路径的列表。如果省略此参数, 默认使用当前目录。

3) expression : 用于指定搜索的方式、条件和要执行的操作等。如果 expression 省略, 则默认使用 -print 作为 expression。expression 可由 4 类项目组成:

- ① OPTION(选项) : 用于指定 find 命令的搜索方式, 一般把 OPTION 放在 expression 的开头。
- ② TEST(测试) : 用于指定 find 命令的检索条件, 只有符号条件的文件才会被指定的 ACTION 处理。
- ③ ACTION(操作) : 用于指定对 find 的搜索结果执行的操作。省略 ACTION 时预设为 -print。
- ④ OPERATOR(运算) : 对 TEST 或 ACTION 进行与、或、非等运算。

4) find 在执行时, 一般会将第 1 个 "-" 字符之后的命令行参数都看作 expression, 把之前的参数都看作为要搜索的路径。

### 3.常用的 OPTION 项目

0) OPTION 项目可以省略, find 默认从指定路径目录开始递归地向下层子目录搜索。

1) -depth : 在查找文件时, 首先查找当前目录中的文件, 然后再在其子目录中查找。

2) -maxdepth level : 进入指定的目录下层目录搜索时, 最深不超过 level(一个非负整数)层。

eg: find . -maxdepth 0 -name "1" // 只在当前目录而不向下层子目录搜索名为"1"的文件或目录

3) -follow : 如果 find 命令遇到符号链接文件, 就跟踪至链接所指向的文件。

4) -mount : 不搜索其它文件系统上的目录(不跨越文件系统 mount 点)。

5) -daystart : 从当日起始时开始而不是从 24 小时之前开始计算时间(如-amin, -atime, -cmin, -ctime, -mmin 和 -mtime)。

6) -noleaf : 不为"目录中子目录数量比硬链接数少 2"这种假设做优化。这个选项在搜索那些不遵循 UNIX 文件系统链接约定的文件系统(比如 CD-ROM, MS-DOS 文件系统或 AFS 卷的加载点)时使用。在普通的 UNIX 文件系统中, 每个目录至少有两个硬链接: 它的名字和它的 '.' 条目。另外, 它的每个子目录(假如有的话)还会各有一个 '.' 链接到它。在 find 检索一个目录时, 发现子目录数比它的链接数少二时, 它就知道目录中的其他条目并非目录(而是目录树中的叶('leaf')节点)。除非需要检索的是这个叶节点, 否则没必要去处理它。这样可以带来很大的搜索速度的提升。

### 4.常用的 TEST 项目

0) 在 TEST 项目的一些选项中有时会使用数字, 数字 N(非负整数)可以以 3 种形式给出:

+N 表示比 N 大, -N 表示比 N 小, N 表示正好是 N

1) -name PATTERN : 查找文件名符合模式 PATTERN(一般要加双引号)的文件, 也可直接使用文件名。

eg: find . -name "[a-z][a-z][0-9][0-9].txt" -print // 此命令可以返回名为 ax37.txt 的文件

2) -empty : 查找空白文件, 它可以是一般文件或目录。

3) -lname PATTERN: 只查找符合 PATTERN 的符号链接文件。

4) -iname 和 -ilname : 分别与 -name 和 -lname 类似, 只是不区分大小写。

5) -regex PATTERN : 查找完整文件名符合模式 PATTERN 的文件。

6) -type C : 查找某一类型的文件。C 可以是 b c d p l s f (块设备 字符设备 目录 命名管道 符号链接 socket 正规文件)。

eg: find ~ -type l // 此命令返回当前用户的主目录内所有的符号链接文件的文件名

7) -size N[bckw] : 查找使用 N 个单位空间的文件，可以使用 b(块,512 字节) c(字节) k(KB) w(2 字节)为单位，不带单位时预设为 b。

eg: find ~ -size +100k -size -1024k // 此命令返回文件尺寸大于 100k 小于 1M 的文件的文件名

8) -fstype FSTYPE : 查找位于某一类型文件系统上的文件，如 vfat ext3 nfs 等

9) -user USERNAME : 查找文件属主为 USERNAME 的文件。

-uid UID : 查找文件属主的 uid 为 UID 的文件。

10) -group GROUPNAME : 查找文件所属组为 GROUPNAME 的文件。

-gid GID : 查找文件所属组的 gid 为 GID 的文件。

11) -nouser : 查找无有效属主的文件，即该文件的属主在/etc/passwd 中不存在。

-nogroup : 查找无有效所属组的文件，即该文件所属的组在/etc/groups 中不存在。

12) -perm MODE : 查找文件的权限设置等于 MODE 的文件。MODE 前可加"+"或"-", 表示权限设置比 MODE 宽松)或更严格。

13) -anewer FILENAME : 查找其存取(access)时间比文件 FILENAME 的修改时间更接近现在的文件。

-cnewer FILENAME : 查找其状态改变(change)时间比文件 FILENAME 的修改时间更接近现在的文件。

-newer FILENAME : 查找其内容修改(modify)时间比文件 FILENAME 的修改时间更接近现在的文件。

-amin N : 查找在指定时间(N 为分钟)被存取过的文件。

-cmin N : 查找在指定时间(N 为分钟)更改过文件状态的文件。

-mmin N : 查找在指定时间(N 为分钟)修改过数据内容的文件。

-atime N : 查找在指定时间(N 为天)被存取过的文件。

-ctime N : 查找在指定时间(N 为天)更改过文件状态的文件。

-mtime N : 查找在指定时间(N 为天)修改过数据内容的文件。

注: 上面这些选项分别涉及到文件的 3 种时间: 文件存取时间(access)、文件状态改变时间(status change) 和文件内容修改时间(modify)，也就是在 ls -l 命令输出中显示的时间。

## 5.常用 ACTION

1) -print : 以完整文件路径名的形式将 find 的搜索结果显示到标准输出，以 NEWLINE 分隔各个文件名。

2) -ls : 以"ls -l"命令的格式将 find 的搜索结果显示到标准输出。

3) -exec CMD {} \; : 把 find 的搜索结果作为 shell 程序 CMD 的文件名参数，并执行 CMD 程序。

eg: find /logs -mtime +5 -exec rm -rf {} \; // 删除/logs 目录中更改时间在 5 日以前的文件

4) -ok CMD {} \; : 同上，是一种更安全的模式，在执行每个命令前，会提示用户来确定是否执行。

## 6.常用的 OPERATOR

1) `EXPR1 EXPR2` 或 `EXPR1 -a EXPR2` 或 `EXPR1 -and EXPR2` : 把 2 个 `EXPR` 相与。

eg: `find ~zqf -size +10k -name "*.c"` // 从 `zqf` 的主目录开始搜索大于 10kB 的 C 源程序文件

2) `EXPR1 -o EXPR2` 或 `EXPR1 -or EXPR2` : 把 2 个 `EXPR` 相或。

eg: `find . -name "*.c" -or -name "*.cpp"` // 从当前目录开始搜索 C 或 C++源程序文件

3) `! EXPR` 或 `-not EXPR` : 对 `EXPR` 取反。

eg: `find ~zqf ! -user zqf` // 查找 `zqf` 的主目录中属主不是 `zqf` 的文件或目录

4) `( EXPR )` : 当出现多个 OPERATOR 时，用来改变 OPERATOR 的运算顺序。

eg: `find . ! \( -user zqf -name "*.awk" \)`

## 7.xargs

1) Usage: `xargs CMD` (从管道中获取 `CMD` 命令的参数)

2) `xargs` 常常与 `find` 命令一起使用，用于取代 `find` 的 `-exec` 参数。有几个原因:

① `find` 的 `-exec` 参数有很多局限。比如，递给 `exec` 的命令长度有限制，在某些系统上 `-exec` 参数只能调用很少的 `shell` 命令等。

② 某些系统上 `-exec CMD` 选项会为 `find` 的搜索结果中的每一个文件名启动一个 `CMD` 进程，而不是把搜索结果作为一个参数文件列表整个传给 `CMD` 程序，当搜索结构很多时，会严重影响系统性能。`xargs CMD` 把从管道获取的参数作为一个参数列表一次传给 `CMD` 程序。

3) `find` 和 `xargs` 联合使用举例: `find . -perm -7 | xargs chmod o-w` // 在当前目录下查找所有用户具有读、写和执行权限的文件，并收回相应的写权限

# 第 3 章 后台执行命令

## 1.cron

1) `cron` 是系统主要的调度进程，可以在无需人工干预的情况下运行作业，与 Windows 的"计划任务"类似。

2) `cron` 守护进程支持 `crontab` 和 `at`，用户可以通过这 2 个程序来实现定时调度作业。

2.`crontab` 命令允许用户提交、编辑或删除相应的作业。

1) 要想让 `cron` 来定时调度执行用户指定的程序，需要 2 个步骤:

① 按照 `crontab` 文件的格式创建用户的 `crontab` 文件。

② 使用 `crontab` 命令向 `cron` 提交用户的 `crontab` 文件。

2) 系统管理员可以通过设置 `/etc` 目录下的 `cron.deny` 和 `cron.allow` 文件来禁止或允许用户使用 `crontab`。

3.创建用户的 `crontab` 文件:

1) `crontab` 文件由若干条记录组成，一条记录对应一个要运行的命令。注释行要在行首加 `#`。



2) crontab 文件记录格式: 分<>时<>日<>月<>星期<>要运行的 CMD

- ① 时间用数字表示，其中星期用 0~6 ( 0 表示星期天 )，小时用 1~23 ( 0 表示子夜 )
- ② <>表示空格，作为域分隔符，每一条记录必须含有 5 个时间域，而且每个域之间要用空格分隔。
- ③ 在这些表示时间的域中，可以用横杠"-"来表示一个时间范围，可以使用逗号","，使用星号\*来表示对某个表示时间的域没有特别的限定。例如，你希望星期一至星期五运行某个作业，那么可以在星期域使用"1-5"来表示。如果希望星期一和星期四运行某个作业，只需要使用"1,4"来表示。如果希望每天都运行某作业，应该在日域填入"\*"。
- ④ 举例: `echo "10 1 * * 6,0 /bin/find ~zqf -name \"core\" -exec rm {} \;" > zqf.cron`  
 // 上面的例子创建了一个用户 crontab 文件 zqf.cron，表示每周六、周日的 1:10 运行一个 find 命令。

4.用户提交用户 crontab 文件:

1) Usage: crontab user\_crontabfile

2) 用户提交了 crontab 文件后，cron 会把用户 crontab 文件中的内容添加到/var/spool/cron 目录下一个与用户名同名的文件中，用户第一次使用 cron 之前不存在/var/spool/cron 目录下的同名文件。

3) 举例: crontab zqf.cron

// 用户 zqf 执行此命令后，cron 将 zqf.cron 中的内容添加到文件/var/spool/cron/zqf 中

4)crontab Usage: crontab [-u USERNAME] [-e -l -r]

- ① -u USERNAME : 编辑指定用户名的 crontab 文件。
- ② -e : 直接编辑 crontab 文件/var/spool/cron/<user>。
- ③ -l : 列出 crontab 文件/var/spool/cron/<user>中的内容。
- ④ -r : 删除/var/spool/cron/<user>文件。

4.at 允许用户向 cron 守护进程提交作业，使其在指定的稍后时间运行。

1) 一旦一个作业被提交,at 命令将会保留所有当前的环境变量，包括路径，不象 crontab 只提供缺省的环境。该作业的所有输出都将以电子邮件的形式发送给用户，除非你对其输出进行了重定向，绝大多数情况下是重定向到某个文件中。

2) 和 crontab 一样，系统管理员可以通过/etc 目录下的 at.allow 和 at.deny 文件来控制哪些用户可以使用 at 命令。一般来说，对 at 命令的使用不如对 crontab 的使用限制那么严格。

3) 向 at 提交一个作业后，at 将为该作业分配一个唯一的作业号，进入 at 的队列，作业运行后退出队列。

4) 提交给 at 的作业，只能在指定的时间运行一次，不能象 crontab 那样周期性运行。

5.向 at 提交作业

1) 命令行方式: 一般在提交 shell 脚本时，使用命令行方式。

- ① Usage: at -f SCRIPT\_FILE [-m] TIME
- ② SCRIPT\_FILE : 是脚本文件名，可以把要提交给 at 的作业写到脚本文件中，然后提交给 at。
- ③ -m : 作业完成后给用户发邮件。

③ TIME : 指定作业将要执行的时间。TIME 的格式很灵活。

2) 交互方式: 在交互方式下, 要提交的作业直接从控制台输入。

① Usage: at [-m] TIME

② 在命令行下执行"at TIME"命令后, 就进入 at 命令提示符( at> )状态, 在 at 提示符状态下可以输入 shell 命令, 一行输入一条 shell 命令, 可以输入多行, 最后按"Ctrl+D"退出。

③ 举例: \$ at 21:10

```
at> find / -name "passwd" -print
```

```
at> <EOT>           // <EOF> 表示"Ctrl+D"
```

6.TIME : 作业被指定的时间, at 的时间格式很灵活, 时间粒度可以是时分、月日年。常见的格式有:

1) HH:MM : 这是最普遍的格式。比如 21:30、9:15 等

2) am 和 pm : 比如 10am、4pm、9:25pm 等

3) MMDDYY 和 MM/DD/YY: 比如 9:30pm 022005、11:50 02/20/2005 等

4) tomorrow 和 today : 比如 10pm today、21:30 tomorrow 等

5) now +N unit : unit 为时间单位, 可以是 minutes(min minute)、hours(hour) days(day)

比如 now+3min、now +2days、now +12hours 等

6) +N unit : 比如 9pm +2days、11:30 +4min 等

7.at 的其它操作

1) 查看中已提交到 at 队列等待运行的作业: at -l 或 atq

2) 取消还未运行的作业: at -r job\_number 或 atrm job\_number

8.&命令 把作业放到后台执行

1) 当在前台运行某作业时, 终端被该作业占据, 用户不能使用终端; 而把作业放在后台运行就不会占据终端。

2) 当在后台执行作业时, 用户可以继续使用终端做其他事情。但是作业在后台运行一样会将结果输出到屏幕上, 干扰你的工作。如果放在后台运行的作业会产生大量的输出, 最好使用下面的方法把它的输出重定向到某个文件中。比如 command >out\_file 2>&1 &

3) 适合后台运行的作业是那些非交互式的作业。需要用户交互的命令不要放在后台执行, 否则系统就会始终等待用户的输入。

9.向后台提交作业

1)Usage: CMD &

2)举例: find /etc -name "\*.conf" -print >find.dt 2>&1 &

10.nohup nohup 命令可以在你退出帐户之后继续运行相应的后台进程

1) Usage: nohup CMD &

2) 如果使用 nohup 命令提交后台作业, 那么在缺省情况下该作业的所有输出都被重定向到一个名为 nohup.out 的文件中, 除非另外指定了输出文件。

3) 举例: `nohup find / -name "*.tmp" -print > /root/tmp.fnd 2>&1 &`

## 第 4 章 文件名置换

1. 元字符 可在命令行上匹配文件名

1) 使用星号"\*"：可以匹配文件名中的任何字符串，包括空字符串。

eg: `ls app*` // 此命令可以列出文件 `app`、`appdva`、`appdva_SLA` 等

2) 使用问号"?"：可以匹配文件名中的任何单个字符。

eg: `ls conf.?.log` // 此命令可以列出文件 `conf.12.log`、`conf.25.log` 等

3) 使用"[...]"：可以匹配方括号[]中出现的任何单个字符。还可以使用"-"来连接两个字母或数字，以此来表示一个范围。

eg: `ls log.[0-5]*` // 此命令可以列出文件 `log.0321`、`log.2987`、`log.5367` 等

4) 使用"[!...]"：与"[...]"相反，匹配不属于方括号"[!...]"中出现字符的单字符。

eg: `ls log.[!0-9]` // 此命令可以列出文件 `log.sybase` 等

## 第 5 章 shell 输入与输出

1. echo 将一行字符串显示到标准输出

0) echo 命令的一些细节在 System V、BSD 和 Linux 这三种 Like-UNIX 系统上不同，这里以 Linux 系统为主。

1) Usage: `echo [-e] [-n] STRING`

① STRING：STRING 是要输出的字符串，其中可以包含 shell 变量名、转义符等，一般用双引号括起来。

② -e：Linux 的 echo 缺省不解释 STRING 中的转移符，除非加上此选项。

③ -n：echo 缺省在输出 STRING 后输出 NEWLINE(换行)，使用此选项 echo 将不输出 NEWLINE。

2) echo 支持的转移符：

`\NNN` ASCII 码为 NNN(8 进制)的字符，NNN 如果不是一个合理的值，将直接按照字面打印

<code>\a</code>	响铃	<code>\b</code>	退格	<code>\c</code>	去除结尾 NEWLINE 字符
<code>\f</code>	换页	<code>\n</code>	换行	<code>\r</code>	回车
<code>\t</code>	水平制表符	<code>\v</code>	垂直制表符	<code>\\</code>	反斜线

3) 举例: `echo -e "User: $USER\tUID: $UID"` // 显示 `User: zqf      UID: 500`

2. read 从键盘或文件的某一行文本中读入信息，并将其赋给变量。

1) Usage: read variable1 variable2 ...

- 2) 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。
- 3) 如果指定了多个变量，read 用空格(环境变量 IFS)作为分隔符把输入行分成多个域，分别赋给各个变量。
- 4) 输入文本分隔出的域数量多于 read 给出的变量数，read 将所有的超长部分赋予最后一个变量。

3.cat 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。

- 1) 显示文件内容: cat myfile | more // cat 命令不会在文件分页符处停下，它会一下显示完整个文件
- 2) 创建文件: cat file1 file2 file3 > bigfile // 创建一个名为 bigfile 的文件，包含三个文件的内容  
cat >myfile // 创建一个新文件，并向其中输入一些内容，输入完后按<CTRL+D>结束输入
- 3) 显示文件中控制字符: cat -v filename

4.管道 可以通过管道把一个命令的输出传递给另一个命令作为输入。管道用竖杠"|"表示。

- 1) Usage: command1 | command2
- 2) 举例: ls | grep "\*.c" // 将 ls 命令的输出作为 grep 命令的输入，即在当前目录下搜索 C 源程序文件
- 3) sed、awk 和 grep 等程序都很适合用管道，特别是在 shell 命令行下。

5.tee 它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

- 1)Usage: tee -a filename
  - ① -a : 表示追加到文件末尾。
  - ② tee 命令应该和管道结合使用
- 2) 举例: who | tee who.out // who 命令的输出不仅会输出到标准输出，还会输入到文件 who.out

6.标准输入、输出和错误

- 1) 文件描述符: 文件描述符是从 0 开始的整数，指向与进程相关的特定数据流。当进程启动时，通常打开三个文件描述符，分别对应三种标准的 I/O: 标准输入(文件描述符 0)，标准输出(文件描述符 1)，标准错误(文件描述符 2)。
- 2) 标准输入(STDIN): 它是命令的输入，缺省和终端的键盘关联。
- 3) 标准输出(STDOUT): 它是命令的输出，缺省和终端的屏幕关联。
- 4) 标准错误(STDERR): 它是命令的错误信息输出，缺省也和终端的屏幕关联。
- 5) 如果进程打开了额外的文件进行输入和输出，则其被设置为下一个可用的文件描述符，从 3 到 9。

7.文件重定向

- 1) 在执行命令时，命令的标准输入、输出和错误是和文件描述符 0、1、2 关联的，而文件描述符 0、1、2 缺省都和终端关联。如果希望命令从文件中读取标准输入或者希望命令的标准输出写到文件而不是屏幕，就需要使用文件重定向。

2) 重定向标准输出

- ① CMD > filename : 把 CMD 命令的标准输出重定向到一个文件中(如果文件存在，其内容将被覆盖)。
- ② CMD >> filename : 把 CMD 命令的标准输出重定向到一个文件中(追加文件尾部)。
- ③ > myfile : 创建一个长度为 0 的空文件，如果文件存在清空该文件。

- ④ 举例: `ls -l | grep ^d >>files.out` // 把当前目录下的子目录的列表写到文件 `files.out` 中  
`> zqf.log` // 清空日志文件 `zqf.log`
- ⑤ `CMD > filename` 实际上是和 `CMD 1> filename` 等效, `CMD >> filename` 和 `CMD 1>> filename` 等效。

### 3) 重定向标准输入

- ① `CMD < filename` : 以 `filename` 文件作为 `CMD` 命令的标准输入。
- ② `CMD << DELIMITER` : 从标准输入中读入输入, 直至遇到 `DELIMITER` 分界符。(here-document)
- ③ 举例: `sort < grade.txt` // 对文件 `grade.txt` 进行排序(sort)
- ④ `CMD < filename` 实际上是和 `CMD 0< filename` 等效, `CMD << filename` 和 `CMD 0<< filename` 等效。

### 4) 重定向标准错误

- ① `CMD 2 > filename` : 把 `CMD` 命令的标准错误重定向到一个文件中(如果文件存在, 其内容将被覆盖)。
- ② `CMD 2 >> filename` : 把 `CMD` 命令的标准错误重定向到一个文件中(追加文件尾部)。
- ③ 举例: `find / -name "*.tmp" -exec rm -rf {} \;` `2>/dev/null` // 把命令的错误消息输出丢弃

### 5) 结合使用标准输出和标准错误

- ① `CMD 1> file1 2> file2` : 将输出重定向到 `file1` 中, 并把标准错误重定向到 `file2` 中。
- ② `CMD < file1 > file2` : 以 `file1` 文件作为 `CMD` 命令的标准输入, 以 `file2` 文件作为标准输出。
- ③ 举例: `find / -name "*.tmp" -print 1>find.out 2>find.err` // 把 `find` 的搜索结果写到文件 `find.out` 中, 而把 `find` 命令的错误信息(比如没有足够权限搜索某些目录)写到文件 `find.err` 中  
`cat <1.txt >2.txt` // 实际是将 `1.txt` 的内容写到 `2.txt`, 等效于文件复制

### 6) 合并标准输出和标准错误

- ① `CMD > filename 2>&1` : 把标准输出和标准错误一起重定向到一个文件中
- ② `CMD >> filename 2>&1` : 把标准输出和标准错误一起重定向到一个文件中(追加)
- ③ 举例: `grep "standard" * > grep.out 2>&1` // 在当前目录下所有文本文件中搜索字符串"standard"
- ④ `CMD > filename 2>&1` 实际上可以看作 2 部分, "`> filename`"(重定向标准输出)和"`2>&1`"(把标准错误重定向到标准输出)。

7) 在使用一些接受文件名为参数的命令时, 有时命令会把文件描述符当成是文件名参数而报错。一般文件描述符和重定向符号之间留有不要有空格。

## 第 6 章 命令执行顺序

### 1.使用"&&"

- 1) 命令 1 && 命令 2
- 2) &&左边的命令 1 返回真(即返回 0, 成功被执行)后, &&右边的命令 2 才能够被执行
- 3) 举例: `cp /apps/bin /apps/dev/bin && rm -r /apps/bin` // 如果复制操作完成, 那么执行删除操作

## 2.使用"||"

- 1) 命令 1 || 命令 2
- 2) 如果||左边的命令 1 未执行成功，那么就执行||右边的命令 2
- 3) 举例: `cp file1 file2 || echo "if seeing this,cp failed."` // 如果复制失败，就打印信息

## 3.用"()"和"{}"将命令结合在一起

- 1) ( 命令 1;命令 2;... ) : 当前 shell 中执行一组命令
- 2) { 命令 1;命令 2;... } : 类上，相应的命令将在子 shell 而不是当前 shell 中作为一个整体被执行。只有在"{"中所有命令的输出作为一个整体被重定向时，其中的命令才被放到子 shell 中执行，否则仍然在当前 shell 执行。
- 3) ()、{} 一般和&&或||一起使用  
eg: `cp file1 file2 || (echo "cp failed" | mail zqf; exit)` // 当文件 file1 很大时，复制需要花费很多时间，如果复制过程中出错，将会发送一个邮件给用户，然后退出当前 shell
- 4) 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，对构造判断语句非常有用。

## 第二部 分文本过滤

### 第 7 章 正则表达式(RE)介绍

1.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(RE,regular expressions)，正则表达式是一些特殊或不很特殊的字符串模式的集合。正则表达式由一些特殊字符或进行模式匹配操作时使用的元字符组成，当然也可以使用规则字符。

2.使用句点"."匹配单字符

1) . : 匹配任意单 ASCII 字符,可以为字母，或为数字。

2) 举例: ..XC..匹配 deXC1t、23XCdf 等，.w..w..w.匹配 rwxrw-rw-

3.在行首以^匹配字符串或字符序列

1) ^ : 允许在一行的开始匹配字符或单词。

2) 举例: ^01 匹配 0011cx4、c01sdf 等，^d 匹配 drwxr-xr-x、drw-r--r--等

4.在行尾以\$匹配字符串或字符

1) \$ : 在行尾匹配字符串或字符，\$符号放在匹配单词后。

2) 举例: trouble\$ 匹配以单词 trouble 结尾的所有行

^\$匹配所有空行

5.使用\*匹配字符串中的单字符或其重复序列(与文件名置换中的"\*"不一样)

1) \* : 一个单字符后紧跟\*，匹配 0 个或多个此单字符。

2) 举例: compu\*t 将匹配字符 u 一次或多次，即匹配 computer computing compuuute 等

1033\* 可以匹配 101333 10133 1013444 等

3) 在正则表达式中使用""，有时会产生非预期的结果。

6.使用\屏蔽一个特殊字符的含义

1) \ : 用来屏蔽一个元字符的特殊含义。因为有时在 shell 中元字符有特殊含义。\\可以使其失去应有意义。

2) 举例: 在正则表达式中匹配以\*.pas 结尾的所有文件: \\*.pas\$

7.使用[]匹配属于一个范围或集合单个字符

1) [] : 匹配"[]"内的字符。可以是一个单字符，也可以是字符序列。可以使用"-表示括号"[]"内字符序列范围，如用[1-5]代替[12345]。可以用逗号","分隔括号"[]"内的字符。

2) 当"^"符号当直接靠着"[]"，意指否定或不匹配括号"[]"里内容

3) 举例: [0-9]匹配任意一个数字；[a-z]匹配任意一个小写字母；[0-9A-Za-z]匹配任意字母或数字；

[C,c]omputer 匹配 Computer 和 computer；[^a-zA-Z]匹配任一非字母型字符

## 8.使用"\{"匹配模式结果出现的次数

- 1) pattern{n} : 匹配模式 pattern 出现 n 次的情形。
- 2) pattern{n,} : 匹配模式 pattern 最少出现 n 次的情形。
- 3) pattern{,m} : 匹配模式 pattern 最多出现 m 次的情形。
- 4) pattern{n,m} : 匹配模式 pattern 出现次数在 n 与 m 之间的情形。

## 5) 举例: A{2}B 匹配的值为 AAB

A{2,}B 匹配的值可以是 AAB 或 AAAAB, 但不能匹配 AB

A{2,4}B 匹配的值可以是 AAB、AAAB、AAAAB, 但不能匹配 AB 或 AAAAB 等

[0-9]{4}CX[0-9]{4} 匹配数字出现 4 次后跟 CX, 最后是数字出现 4 次的情形

## 6) 实际上真正的格式是 {n} {n,} {,m} {n,m}, 只不过对 "{"和"}"应用了 Escape 字符"\".

## 9.经常使用的正则表达式举例

[Ss]igna[IL] 匹配单词 signal、signal、Signal、Signal

[Ss]igna[IL]\. 同上, 但加一句点

^USER\$ 只包含 USER 的行

\. 带句点的行

^d..x..x..x 对对用户、用户组及其他用户、组成员有可执行权限的目录

^[^I] 排除符号链接文件后的文件目录列表(即不是以"I"开始的行)

[yYnN] 大写或小写 y 或 n

^.\*\$ 匹配行中任意字符串

^.....\$ 包括 6 个字符的行

[a-zA-Z] 任意单个字母

[a-z]\* 至少一个小写字母

^[0-9\\$] 非数字或美元符号

[123] 1 到 3 中一个数字

^\q 以^q 开始行

^.\$ 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

. \* 匹配任意多个字符

10.在 shell 编程中, 一段好的脚本与完美的脚本间的差别之一, 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。

## 第 8 章 grep 家族



## 1.grep

1) grep 是使用最广泛的命令之一，用来对文本文件内容按行进行模式匹配查找。如果找到匹配模式的行，grep 将打印包含模式的行。

2) grep 有三种变形:

- ① Grep : 标准 grep 命令，主要讨论此格式。
- ② Egrep: 扩展 grep，支持基本及扩展的正则表达式。
- ③ Fgrep: 快速 grep，允许查找字符串而不是一个模式。这里的"快速"并不是指速度快。

## 2.grep 的用法

1) grep Usage: grep [OPTION] regular\_expressions [filename1 ...]

- ① regular\_expressions : 是正则表达式，一般用单引号把正则表达式括起来。当然，也可以不使用正则表达式而使用字符串，使用字符串时一般用双引号把字符串括起来。
- ② filename1 ... : 文件名列表，grep 将对这些指定的文件的内容进行匹配查找，如果文件名列表省略，grep 将从标准输入读取要匹配查找的内容。

2) grep 的常用选项:

- c 只输出匹配的行的总数(count)。
- i 不区分大小写(只适用于单字符)。
- 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。
- s 不显示不存在或无匹配文件等错误信息(silence)。
- v 只显示不包含匹配模式的行。

## 3.grep 应用举例

1) 查询多个文件,可以使用\*。比如:

```
grep "sort" *.doc // 在所有以 doc 为后缀的文件中查找字符串"sort"
grep "linux" * // 在当前目录下的所有文件中查找字符串"linux"
```

2) 精确匹配，可以在要匹配的字符串后加\>。

```
grep "48\>" data.f // 可以匹配 48、1248、c48 This 等而不能匹配 481、c480
```

3) 反转匹配

```
ps aux | grep "httpd" | grep -v "grep" // 查看正在运行的 httpd 进程
```

4) 匹配空行

```
grep -n '^$' myfile // 打印文件 myfile 中空行的行号
grep -v '^$' myfile // 去除文件 myfile 中的空行
```

4.grep 可以使用或匹配模式的类名形式

1) 常见的匹配模式的类名形式:

`[:upper:]` 等价于 `[A-Z]`

`[:lower:]` 等价于 `[a-z]`

`[:digit:]` 等价于 `[0-9]`

`[:alnum:]` 等价于 `[0-9a-zA-Z]`

`[:space:]` 等价于 空格或 tab 键

`[:alpha:]` 等价于 `[a-zA-Z]`

2) 举例: `grep '5[:upper:][:upper:]' data.f` // 可以匹配包括 5AP196、5DF540 的行

5.可以把要匹配的模式写到一个文件中，然后使用 `-f` 选项，将该文件传给 `grep`。

eg: `egrep -f grappats data.f` // 要匹配的模式存放在文件 `grappats` 中

6.可以在 `grep` 的模式字符串中使用 `()` 符号，意即“`|`”符号两边之一或全部,可以使用任意多“`|`”,可同时使用 `^` 符号排除字符串

eg: `who | grep '(zqf|zqc|zqx)'` // 查看是否 `zqf zqc zqx` 三者中是否有在线的

7.如果传递给 `grep` 的文件名参数是不是一个普通文件而是一个目录的话，要使用 `-d` 选项。

1) Usage: `grep -d [ACTION] directory_name`

2) ACTION : ACTION 用来指定对作为输入文件的目录文件的处理方式，ACTION 有 3 个可选值:

① `read` : 把目录文件当作普通文件来读取，是选项省略时的默认方式。

② `skip` : 目录将被忽略而跳过

③ `recurse` : `grep` 以递归的方式读取目录下的每一个文件。等同于选项 `-r`。

3) 举例: `grep -rl "eth0" /etc` // 查看/etc 目录中于"eth0"有关的文件的文件名

## 第 9 章 AWK 介绍

0.awk 有 3 个不同版本: `awk`、`nawk` 和 `gawk`，未作特别说明，一般指 `gawk`。

1.awk 语言的最基本功能是在文件或字符串中基于指定规则来分解抽取信息，也可以基于指定的规则来输出数据。完整的 `awk` 脚本通常用来格式化文本文件中的信息。

2.三种方式调用 `awk`

1) `awk [option] 'awk_script' input_file1 [input_file2 ...]`

`awk` 的常用选项 option 有 ;

① `-F fs` : 使用 `fs` 作为输入记录的字段分隔符，如果省略该选项，`wak` 使用环境变量 `IFS` 的值

② `-f filename` : 从文件 `filename` 中读取 `awk_script`

③ `-v var=value` : 为 `awk_script` 设置变量

2) 将 awk\_script 放入脚本文件并以 `#!/bin/awk -f` 作为首行，给予该脚本可执行权限，然后在 shell 下通过键入该脚本的脚本名调用之。

3) 将所有的 awk\_script 插入一个单独脚本文件，然后调用: `awk -f wak 脚本文件 input_file(s)`

### 3. awk 的运行过程

#### 1) awk\_script 的组成:

- ① awk\_script 可以由一条或多条 awk\_cmd 组成，两条 awk\_cmd 之间一般以 NEWLINE 分隔
- ② awk\_cmd 由两部分组成: `awk_pattern { actions }`
- ③ awk\_script 可以被分成多行书写，必须确保整个 awk\_script 被单引号括起来。

#### 2) awk 命令的一般形式:

```
awk ' BEGIN { actions }      \
      awk_pattern1 { actions } \
      .....                  > awk_script
      awk_patternN { actions } /
      END { actions }        /
' inputfile
```

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

#### 3) awk 的运行过程:

- ① 如果 BEGIN 区块存在，awk 执行它指定的 actions。
- ② awk 从输入文件中读取一行，称为一条输入记录。(如果输入文件省略，将从标准输入读取)
- ③ awk 将读入的记录分割成字段，将第 1 个字段放入变量 \$1 中，第 2 个字段放入 \$2，以此类推。\$0 表示整条记录。字段分隔符使用 shell 环境变量 IFS 或由参数指定。
- ④ 把当前输入记录依次与每一个 awk\_cmd 中 awk\_pattern 比较，看是否匹配，如果相匹配，就执行对应的 actions。如果不匹配，就跳过对应的 actions，直到比较完所有的 awk\_cmd。
- ⑤ 当一条输入记录比较了所有的 awk\_cmd 后，awk 读取输入的下一行，继续重复步骤③和④，这个过程一直持续，直到 awk 读取到文件尾。
- ⑥ 当 awk 读完所有的输入行后，如果存在 END，就执行相应的 actions。

4) input\_file 可以是多于一个文件的文件列表，awk 将按顺序处理列表中的每个文件。

5) 一条 awk\_cmd 的 awk\_pattern 可以省略，省略时不对输入记录进行匹配比较就执行相应的 actions。一条 awk\_cmd 的 actions 也可以省略，省略时默认的动作是打印当前输入记录(`print $0`)。一条 awk\_cmd 中的 awk\_pattern 和 actions 不能同时省略。

6) BEGIN 区块和 END 区块别位于 awk\_script 的开头和结尾。awk\_script 中只有 END 区块或者只有 BEGIN 区块是被允许的。如果 awk\_script 中只有 `BEGIN { actions }`，awk 不会读取 input\_file。

7) awk 把输入文件的数据读入内存，然后操作内存中的输入数据副本，awk 不会修改输入文件的内容。

8) awk 的总是输出到标准输出，如果想让 awk 输出到文件，可以使用重定向。

#### 4.awk\_pattern

awk\_pattern 模式部分决定 actions 动作部分何时触发及触发 actions。awk\_pattern 可以是以下几种类型：

1) 正则表达式用作 awk\_pattern: /regexp/

① awk 中正则表达式匹配操作中经常用到的字符：

\ ^ \$ . [] | () \* // 通用的 regexp 元字符

+: 匹配其前的单个字符一次以上，是 awk 自有的元字符，不适用于 grep 或 sed 等

?: 匹配其前的单个字符 1 次或 0 次，是 awk 自有的元字符，不适用于 grep 或 sed 等

② 举例：

```
awk '/ *$0\[0-9\][0-9].*/' input_file
```

2) 布尔表达式用作 awk\_pattern，表达式成立时，触发相应的 actions 执行。

① 表达式中可以使用变量(如字段变量\$1,\$2 等)和/regexp/

② 布尔表达式中的操作符：

关系操作符: < > <= >= == !=

匹配操作符: value ~ /regexp/ 如果 value 匹配/regexp/，则返回真

value !~ /regexp/ 如果 value 不匹配/regexp/，则返回真

举例: awk '\$2 > 10 {print "ok"}' input\_file

```
awk '$3 ~ /^d/ {print "ok"}' input_file
```

③ &&(与) 和 ||(或) 可以连接两个/regexp/或者布尔表达式，构成混合表达式。!(非) 可以用于布尔表达式或者/regexp/之前。

举例: awk '(\$1 < 10) && (\$2 > 10) {print "ok"}' input\_file

```
awk '/^d/ || /x$/ {print "ok"}' input_file
```

④ 其它表达式用作 awk\_script，如赋值表达式等

eg: awk '(tot+=\$6); END{print "total points : " tot}' input\_file // 分号不能省略

```
awk 'tot+=$6 {print $0} END{print "total points : " tot}' input_file // 与上面等效
```

#### 5.actions

actions 就是对 awk 读取的记录数据进行的操作。actions 由一条或多条语句或者命令组成，语句、命令之间用分号(;)分隔。actions 中还可以使用流程控制结构的语句。

1) awk 的命令：

① print 参数列表：print 可以打印字符串(加双引号)、变量和表达式，是 awk 最基本的命令。参数列表要用逗号(,)分隔，如果参数间用空格分隔，打印出时参数值之间不会有空格。

② printf ([格式控制符]，参数)：格式化打印命令(函数)，语法与 C 语言的 printf 函数类似。

③ next：强迫 awk 立刻停止处理当前的记录,而开始读取和处理下一条记录。

④ nextfile：强迫 awk 立刻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

⑤ exit：使 awk 停止执行而跳出。如果有 END 存在，awk 会去执行 END 的 actions。

2) awk 的语句: awk 的语句主要是赋值语句，用来给变量赋值。

① 把直接值或一个变量值赋值给变量。如果直接值是字符串要加双引号。

举例: `awk 'BEGIN {x=1 ; y=3 ; x=y ; print "x=" x " ; y=" y}'`

② 把一个表达式的值赋值给变量。表达式一般是数值表达式, 也可以是其它表达式。

数值表达式: `num1 operator num2`

operator 可以是: +(加) -(减) \*(乘) /(除) %(取模) ^(求幂)

当 num1 或者 num2 是字符串而不是数字时, 无论是否加有双引号, awk 都视其值为 0

条件选择表达式: `A?B:C` (A 为布尔表达式, B 和 C 可以是表达式或者直接值)

当布尔表达式 A 的值为真时, 整个表达式的值为 B, A 的值为假时, 整个表达式的值为 C

举例: `awk 'BEGIN {x=3 ; x+=2 ; y=x+2 ; print "x=" x " ; y=" y}'`

`awk 'BEGIN {x=3 ; y=x>4?"ok":4 ; print "x=" x " ; y=" y}'`

③ 为了方便书写, awk 也支持 C 语言语法的赋值操作符: `+= -= *= /= %= ^= ++ --`

### 3) 流程控制结构 (基本上是使用 C 语言的语法)

其中 condition 一般为布尔表达式, body 和 else-body 是 awk 语句块。

① `if (condition) {then-body} [else {else-body}]`

② `while (condition) {body}`

③ `do {body} while (condition)`

④ `for (initialization; condition; increment) {body}`

与 C 语言的 for 结构的语法相同。

⑤ `break` : 跳出包含它的 for、while、do-while 循环

⑥ `continue` : 跳过 for、while、do-while 循环的 body 的剩余部分, 而立刻进行下一次循环的执行。

## 6.awk 的变量

在 awk\_script 中的表达式中要经常使用变量。不要给变量加双引号, 那样做, awk 将视之为字符串。awk 的变量基本可以分为两类:

1) awk 内部变量: awk 的内部变量用于存储 awk 运行时的各种参数, 这些内部变量又可以分为:

① 自动内部变量: 这些变量的值会随着 awk 程序的运行而动态的变化, 在 awk\_script 中改变这些变量的值是没有意义的(即不应该被赋值)。常见的有:

NF : 当前输入字段的字段数

NR : 对当前输入文件而言, 已经被 awk 读取过的记录(行)的数目。

FNR : 已经被 awk 读取过的记录(行)的总数目。当输入文件只有一个时, FNR 和 NR 是一致的。

FILENAME : 当前输入文件的文件名。

ARGC : 命令行参数个数。(不包括选项和 awk\_script, 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(实际上 ARGV[1]就是第一个输入文件)

举例: `awk '{print NR,NF,$0} END {print FILENAME}' input_file`

② 字段变量(`$0 $1 $2 $3 ...`): 当 awk 把当前输入记录分段时, 会对这些字段变量赋值。和内部变量类似, 在 awk 运行过程中字段变量的值是动态变化的。不同的是, 修改这些字段变量的值是有意义的, 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, NR 的值也将随之变化。

字段变量支持变量名替换。

举例: `pwd | awk -F/ '{print $NF}'` // print \$NF 打印输入记录的最后一个字段

`awk '{x=2;print $x}' input_file` // 打印输入记录的第 2 个字段

③ 其它内部变量: 可以修改这些变量。常见的有:

FS : 输入记录的字段分隔符(默认是空格和制表符)

OFS : 输出记录的字段分隔符(默认是空格)

OFMT : 数字的输出格式(默认是 %.6g)

RS : 输入记录间的分隔符(默认是 NEWLINE)

ORS : 输出记录间的分隔符(默认是 NEWLINE)

ARGV : 命令行参数数组

ENVIRON : 存储系统当前环境变量值的数组, 它的每个成员的索引就是一个环境变量名, 而对应的值就是相应环境变量的值。可以通过给 ENVIRON 数组的成员赋值而改变环境变量的值, 但是新值只在 awk\_script 内有效。eg: `ENVIRON["HISTSIZE"]=500`

举例: `cat /etc/passwd | awk 'BEGIN { FS=":" } {print "User name: "$1,"UID: "$4}'`

## 2) 自定义变量

1) 定义变量: `varname=value` (自定义变量不需先声明后使用, 赋值语句同时完成变量定义和初始化)

2) 在表达式中出现不带双引号的字符串都被视为变量, 如果之前未被赋值, 默认值为 0 或空字符串。

## 3) 向命令行 awk 程序传递变量的值:

① Usage: `awk 'awk_script' awkvar1=value1 awkvar2=value2 .... input_file`

eg: `awk '{if ($5 < ARG) print $0 }' ARG=100 input_file`

② awkvar 可以是 awk 内置变量或自定义变量。

③ awkvar 的值将在 awk 开始对输入文件的第一条记录应用 awk\_script 前传入。如果在 awk\_script 中已经对某个变量赋值, 那么在命令行上传入到该变量的值就会无效(实际上是 awk\_script 中的赋值语句覆盖了从命令行上传入的值)。

④ 在 awk 脚本程序中不能直接使用 shell 的变量。通过使用下面的语法可达到这样的效果。

`awk 'awk_script' awkvar1=shellvar1 awkvar2=shellvar2 .... input_file`

eg: `awk '{if (v1 == "root") {print "User name is root!"}}' v1=$USER input_file`

⑤ 可以向 awk 脚本传递变量的值, 与上面的类似。

`awk_script_file awkvar1=value1 awkvar2=value2 ... input_file`

## 7. awk 的内置函数

可以在 awk\_script 的任何地方使用 awk 函数。和 awk 变量一样, awk 函数可以分为内置函数和自定义函数。

### 1) 常见 awk 内置数值函数

`int(x)` : 求出 x 的整数部份, 朝向 0 的方向做舍去。eg: `int(3.9)` 是 3, `int(-3.9)` 是 -3。

`sqrt(x)` : 求出 x 正的平方根值。eg: `sqrt(4)` = 2

`exp(x)` : 求出 x 的次方。eg: `exp(2)` 即是求  $e^2$ 。

`log(x)` : 求出 x 的自然对数。

`sin(x)` : 求出 x 的 sine 值, x 是弧度量。

`cos(x)` : 求出  $x$  的 cosine 值,  $x$  是弧度量。

`atan2(y,x)` : 求  $y/x$  的 arctangent 值, 所求出的值其单位是弧度量。

`rand()` : 得到一个随机数(平均分布在 0 和 1 之间)。每次执行 `gawk`, `rand` 从相同的 seed 生成值。

`srand(x)` : 设定产生随机数的 seed 为  $x$ 。如果在第二次运行 `awk` 程序时你设定相同的 seed 值, 你将再度得到相同序列的随机数。如果省略引数  $x$ , 例如 `srand()`, 则当前日期时间会被当成 seed。这个方法可使得随机数值是真正不可预测的。

`srand()` : 其值是当次 `awk_script` 运行过程中前次 `srand(x)` 的设定的 seed 值  $x$ 。

## 2) 常见 awk 内置字符串函数

`index(in, find)` : 返回字符串 `in` 中字符串 `find` 第一次出现的位置(索引从 1 开始), 如果在字符串 `in` 中找不到字符串 `find`, 则返回值为 0。eg: `print index("peanut","an")` 会印出 3。

`length(s)` : 求出字符串 `s` 的字符个数。eg: `length("abcde")` 是 5。

`match(s,r)` : 返回模式字符串 `r` 在字符串 `s` 的第一次出现的位置, 如果 `s` 不包含 `r`, 则返回值 0。

`sprintf(fmt,exp1,...)` : 和 `printf` 类似印出, 是 `sprintf` 不是打印而是返回经 `fmt` 格式化后的 `exp`。

eg: `sprintf("pi = %.2f (approx.)",22/7)` 传回的字符串为 "pi = 3.14 (approx.)"

`sub(p,r,t)` : 在字符串 `t` 中寻找符合模式字符串 `p` 的最靠前最长的位置, 并以字符串 `r` 代替最前的 `p`。

eg: `str = "water, water"sub(/at/, "ith",str)` 结果字符串 `str` 会变成 "withr, water"

`gsub(p,r,t)` : `gsub` 与 `sub` 类似。不过时在字符串 `t` 中以字符串 `r` 代替所有的 `p`。

eg: `str="water, water" ; gsub(/at/, "ith",str)` 结果字符串 `str` 会变成 "withr,withr"

`substr(str, st, len)` : 传回 `str` 的子字符串, 其长度为 `len` 字符, 从 `str` 的第 `st` 个位置开始。如果 `len` 没有出现, 则传回的子字符串是从第 `st` 个位置开始至结束。

eg: `substr("washington",5,3)` 传回值为 "ing"

`substr("washington",5)` 传回值为 "ington"

`split(s,a,fs)` : 在分隔符 `fs` 为分隔符将字符串 `s` 分隔成一个 `awk` 数组 `a`, 并返回 `a` 的下标数。

eg: `awk 'BEGIN{print split("123#456#789",myarray,"#")}'` 将打印 3。

`tolower(str)` : 将字符串 `str` 的大写字母改为小写字母。

eg: `tolower("MiXeD cAsE 123")` 传回值为 "mixed case 123"

`toupper(str)` : 将字符串 `string` 的小写字母改为大写字母。

eg: `toupper("MiXeD cAsE 123")` 传回值为 "MIXED CASE 123"

## 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, 执行完毕后将回到 `awk` 程序。

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

## 8 自定义函数

复杂的 `awk` 常常可以使用自己定义的函数来简化。调用自定义的函数与调用内置函数的方法一样。

1) 自定义函数定义的格式: 自定义函数可以在 `awk` 程序的任何地方定义。

`function fun_name (parameter_list) {` // `parameter_list` 是以逗号分隔的参数列表

`body-of-function` // 函数体, 是 `awk` 语句块

```
}
```

2) 举例:

```
awk '{ print "sum =",SquareSum($1,$2) }
    function SquareSum(x,y) { sum=x*x+y*y ; return sum } ' grade.txt
```

## 9.awk 的数组

数组使用前，不必预先定义，也不必指定数组元素个数。

1) 访问数组的元素。经常使用循环来访问数组元素，下面是一种循环类型的基本结构:

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

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

## 10.其他

1) 为了避免碰到 awk 错误，可以总结出以下规律:

- ① 确保整个 awk\_script 用单引号括起来。
- ② 确保 awk\_script 内所有引号成对出现。
- ③ 确保用花括号括起动作语句，用圆括号括起条件语句。
- ④ 可能忘记使用花括号，也许你认为没有必要，但 awk 不这样认为，将按之解释语法。
- ⑤ 如果使用字符串，一定要保证字符串被双引号括起来(在模式中除外)。

2) 在 awk 中，设置有意义的域名是一种好习惯，在进行模式匹配或关系操作时更容易理解。一般的变量名设置方式为 `name=$n`。(这里 name 为调用的域变量名，n 为实际域号。)

3) 通常在 BEGIN 部分给一些变量赋值是很有益的，这样可以在 awk 表达式进行改动时减少很多麻烦。

4) awk 的基本功能是根据指定规则抽取输入数据的部分内容并输出，另一个重要的功能是对输入数据进行分析运算得到新的数据并输出，这是通过在 awk\_script 中对字段变量(\$1、\$2、\$3...)从新赋值或使用更大的字段变量\$*n*(*n* 大于当前记录的 NF)而实现的。

5) 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查询一元字符。这时就需要字符串屏蔽序列。

awk 中经常使用的屏蔽序列有:

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

6) awk 的输出函数 `printf`，基本上和 C 语言的语法类似。

- ① 格式: `printf ("输出模板字符串",参数列表)`
- ② 参数列表是以逗号分隔的列表，参数可以是变量、数字值或字符串。
- ③ 输出模板字符串的字符串中必须包含格式控制符，有几个参数就要求有几个格式控制符。模板字符串中可以只有格式控制符而没有其它字符。
- ④ 格式控制符: `%[-][width][.prec]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。



width : 一个数字，表示参数输出时占用域的宽度，可省略。

.prec : prec 是一个数值，表示最大字符串长度或小数点右边的位数，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。

#### ⑤ 常见的 fmt : c ASCII 字符

d 整数

e 浮点数，科学记数法

f 浮点数，如 123.44

g 由 awk 决定使用哪种浮点数转换 e 或 f

o 八进制数

s 字符串

x 十六进制数

#### ⑥ 举例: echo "65" | awk '{ printf ("%c\n",\$0) }' // 将打印 A

```
awk 'BEGIN{printf "%.4f\n",999}' //将打印 999.0000
```

```
awk 'BEGIN{printf "2 number:%8.4f%8.2f",999,888}' // 将打印 2 number:999.0000 888.000
```

## 第 10 章 sed 用法介绍

### 1.sed 是一个非交互性文本流编辑器

1) sed 编辑文件或标准输入导出的文本拷贝。标准输入可能是来自键盘、文件重定向、字符串或变量，或者是一个管道的文本。可以在命令行输入 sed 命令，也可以在一个文件中写入命令，然后调用 sed。

2) sed 操作的只是一个输入文件在内存内的一个副本，对副本内容进行编辑改动，如果没有重定向到一个文件，将编辑修改的结果输出到屏幕。和 awk 一样，sed 不会修改输入文件的内容。

3) 因为 sed 是一个非交互性编辑器，sed 使用一种行定位模式来定位哪些文本行将要被编辑修改。

4) sed 从输入数据中读取一行，将之拷贝到一个编辑缓冲区，然后读命令行或脚本的第一条命令，并使用这些命令中指定的定位模式来确定是否编辑和如何编辑它。重复此过程直到命令结束。

### 2.调用 sed 的三种方式

1) 在命令行键入命令: sed [选项] 'sed\_cmd' input\_file

#### ① sed 常用选项:

-n : sed 默认在将下一行读入行缓冲区中之前，自动输出行缓冲区中的内容。此选项可以关闭自动输出。

-e : sed 如果要在命令行上调用多于一条的 sed\_cmd，必须在每条 sed\_cmd 前加"-e"选项

eg: sed -e 'sed\_cmd1' -e 'sed\_cmd2' -e 'sed\_cmd3' input\_file

#### ② sed\_cmd 的格式: '[address]sed\_edit\_cmd' (一般用单引号括起来)

address : sed 的行定位模式，用于指示将要被 sed 编辑的行。如果省略，sed 将编辑所有行。

sed\_edit\_cmd : sed 对被编辑行将要进行的编辑操作。

#### ③ input\_file : 指定 sed 的输入文件，可以是一个文件列表。如果省略，sed 将从标准输入读取输入。

2) 将 sed 命令插入脚本文件，然后调用 sed [选项] -f sed\_script\_file input\_file

3) 将 sed 命令插入脚本文件，使 sed 脚本文件可执行，在控制台下通过直接键入脚本文件名来执行。

sed 脚本文件格式:

```
#!/bin/sed -f
sed_cmd1
sed_cmd2
...
```

3.sed\_cmd 的 address 用于定位要编辑的行

x : x 为一行号。如 5

x,y : 表示行号范围从 x 到 y。如 2,5 表示从第 2 行到第 5 行

/pattern/ : 查询包含模式的行。如 /disk/或/[a-z]/

/pattern/pattern/ : 查询包含两个模式的行。例如 /disk/disks/

/pattern/,x : 在给定行号上查询包含模式的行。如 /ribbon/,3

x,/pattern/ : 通过行号和模式查询匹配行。如 3,/vdu/

x,y! : 查询不包含指定行号 x 和 y 的行。如 1,2!

4.常见的 sed\_edit\_cmd(sed 的行编辑命令)

1) p : 打印匹配行

eg: sed -n '1,3p' quote.txt // 只打印文件 quote.txt 的第 1-3 行

sed -n '\$p' quotel.txt // 只打印文件 quote.txt 的最后一行

2) = : 显示文件匹配行行号

eg: sed -n '/music/= ' quote.txt //打印文件 quote.txt 中包含"music"字符串的行的行号

3) a\ : 在指定行(不能是多行范围)后附加 N 行新文本并显示新文本，多用于脚本。

格式如: [address]a\ // 符号"\"是必须的

textline1\ // 如要附加多行，要在行尾加符号"\"，表示换行

textline2\ // 被附加的多行文本的最后一行不加"\"

..... // sed 会把它看作是附加命令的结尾

textlineN // 把其下一行当作一条新的 sed\_cmd

eg: \$ cat append.sed // 查看 sed 脚本文件 append.sed 的内容

#!/bin/sed -f // 调用脚本: ./append.sed quote.txt

/company/a\

The suddenly it happend.

4) i\ : 在指定行前插入新文本并显示新文本，格式同 a\

5) d : 删除被定位的行

eg: sed '3,\$d' quote.txt // 删除文件 quote.txt 的第 3 到最后一行

6) c\ : 用新文本替换被定位文本行并显示新文本,格式同 a\

7) s : 用模式 replacement 替换被定位行中的 oldpattern。

① 格式: [address]s/oldpattern/replacement/[gpw]

② 替换选项如下 :

g : 缺省情况下只替换行中第一次出现的 oldpattern , 使用 g 选项替换行全局所有出现的 oldpattern。

p : 显示被替换后的行 , 省略 p 并使用了 sed 的 "-n"选项 , 将看不到任何输出。

w filename : 将被替换后的行内容写到文件 filename 中。

eg: sed -n 's/night/NIGHT/gp' quote.txt // 只显示被替换过了的行

sed -n 's/night/NIGHT/' quote.txt // 在屏幕上没有任何显示

sed 's/night/NIGHT/' quote.txt // 显示 quote.txt 所有行(被替换和没有替换的行)

sed 's/night/NIGHT/w fdt' quote.txt // 同上 , 还把被替换的行内容写到文件 fdt 中

sed 's/night/NIGHT/' quote.txt > fdt // 把 quote.txt 所有行写入文件 fdt

③ 如果要附加或修改一个字符串 , 可以使用 & 命令 , & 命令保存模式 oldpattern 以便重新调用它 , 然后把它放在替换字符串里面。

eg: sed -n 's/nurse/Hello &p' quote.txt // 等效于在 nurse 前插入 "Hello "。

8) r : 从另一个文件中读文本附加到指定的行后 , 并显示读入的文本内容。

eg: sed '/company/r sedex.txt' quote.txt // 在显示 quote.txt 的内容中间显示 sedex.txt 的内容

sed -n '/company/r sedex.txt' quote.txt // 只显示 sedex.txt 的内容

9) w : 写文本到一个文件

eg: sed '1,2 w filedt' quote.txt // 将文件 quote.txt 的第 1-2 行写入到文件 filedt 中

10) q : 读取到被[address]定位的行后退出 sed , 以便执行其他处理脚本。

11) l : 显示被定位行中所有字符 , 包括控制字符(非打印字符)。

eg: sed '1,\$l' quote.txt // 显示文件 quote.txt 中所有字符。可以在每行行尾看到一个 "\$" 字符

12) {} : 在被定位的行上执行的命令组。

eg: sed -n '{s/night/Night;p}' quote.txt 与 sed '{s/night/Night/}' quote.txt 等效

13) n : 从另一个文件中读文本下一行 , 并附加在下一行

5.如何输入控制字符(如回车、Esc、F1 等) , 以输入回车(^M)为例:

1) 按下 Ctrl 键和 v 键

2) 释放 v 键 , 然后按下 ^ 键(此过程中 Ctrl 键保持按下不放)

3) 释放按下的两个键

4) 按下对应的功能键(Enter 键)即可

6.sed 支持 shell 变量名替换 , 即在 sed\_cmd 中可以使用 shell 变量。在 sed 命令中使用 shell 变量时,应使用双引号 , 否则不能得到期望的结果。

eg: REP="GO" ; echo "go home" | sed "s/go/\$REP/" // 此命令将显示 GO home

7. 一些 sed 一行命令集。( 表示空格, [ ]表示 tab 键 )

's/^\.\$//g' 删除以句点结尾行  
 '/abcd/d' 删除包含 abcd 的行  
 's/[ ]\*[ ]\*/[ ]/g' 删除一个以上空格, 用一个空格代替  
 's/^[ ]\*[ ]\*/[ ]/g' 删除行首空格  
 's/^\.[ ]\*[ ]\*/[ ]/g' 删除句点后跟两个或更多空格, 代之以一个空格  
 '/^\$/d' 删除空行  
 's/^\./[ ]/g' 删除第一个字符  
 's/^\./[ ]/g' 从路径中删除第一个/  
 's/[ ]/[ ]/[ ]/g' 删除所有空格并用 t a b 键替代  
 'S/^[ ]/[ ]/g' 删除行首所有 t a b 键

8. 如果使用 sed 对文件进行过滤, 最好将问题分成几步, 分步执行, 且边执行边测试结果, 这是执行一个复杂任务的最有效方式。

## 第 11 章 合并与分割

1.sort 用于文本文件数据内容排序

1) usage: sort [sort 选项] [input\_file ....]

2) sort 命令的操作可以分为 3 种模式:

- ① 排序模式: 对输入文件进行排序, 是默认的模式。
- ② 合并模式: 对两个已排序的文件进行合并。需要指定"-m"选项。
- ③ 检查模式: 测试给定的输入文件是否已排序。需要指定"-c"选项。

eg: sort -c video.txt // 如果 video.txt 已排序, 则无任何显示。如果未排序, 则会给出提示

3) sort 有许多不同的选项, 这些选项基本可以分为三类:

- ① sort 的操作模式选项: 就是用于指定 sort 工作模式的选项, 只有"-c"和"-m"两个。
- ② sort 的数据排序选项: 这些选项将影响输出行的排列顺序, 可以是针对整体或特殊键值字段设定的。
- ③ sort 的字段设定与输出选项: 与输出和字段有关的选项。

2.sort 的常用数据排序选项

- 1) -n: 当指定位置上是数字字符时, 按数值大小来排序, 而不是逐字符比较。
- 2) -b: 忽略前置空白。
- 3) -r: 颠倒输出排序的结果(即逆序输出)。
- 4) -d: 在排序时忽略所有除英文字母、数字及空白之外的字符。
- 5) -f: 在排序时将字母大小写视为相同。

- 6) -i : 在排序时忽略超过 ASCII 可打印范围(8 进制 040-0174)的字符。
- 7) -M : 对表示月份的三个大写字母进行比较, "无效名称" < "JAN" < "FEB" < ... < "DEC"。

### 3.sort 的常用字段设定与输出选项

如果没有使用任何字段设定选项, sort 默认对整行的内容做排序。如果希望针对行中某一特定的字段内容做排序, 就必须知道如何指定字段的分隔符以及指定适当的排序字段。

- 1) -o FILE : 指定排序结果的输出文件, 输出文件可以时输入文件之一。

eg: sort -o video.txt video.txt // 把 video.txt 的排序结果写入 video.txt

- 2) -t 分隔符 : 设定字段分隔符, 如果省略此选项, sort 使用 shell 环境变量 IFS。
- 3) -u : 检查指定域的唯一性(不重复)(检查排序模式), 或去除域重复的行(排序、合并模式下)。

- 4) -k pos1[,pos2] : 把 pos1 到 pos2 之间的内容当成一个字段来进行排序(域号 pos1<pos2,从 1 开始计算)

① 如果省略 pos2, 表示从 pos1 到行尾。

② k 选项的 pos 可以是"F[C]"格式, 即 F 指示使用第几个字段, C 指示从字段开头算起第几个字符。

eg: sort -t: -k2,3n video.txt // 从第二个字段的第三个字符开始排序

③ pos 后可以附加任何数据排序选项字符"Mbdfnr"。

- 5) +POS : 使用由 POS 指定的域开始排序。POS 和上面的 pos 用法类似, 只是 POS 从 0 开始计数。

eg: sort -t: +3 +2 employee // 先以第 4 个字段为第一排序字段, 以第 3 个字段第二排序字段

sort -t: -k4 -k3 employee // 与上面的等效

sort -t: +1,2n video.txt // 与使用"-k2,3n"选项等效

- 6) -POS : 排序时忽略由 POS 指定的域。不能单独使用, 只能放在+POS 选项之后使用。

eg: sort -t: +0 -2 +3 video.txt // 以第一个域开始排序, 忽略第三个域, 再使用第四个域排序

### 4.sort 用于合并 2 个排序文件(将文件合并前, 它们必须已被排序)

- 1) Usage : sort -m [-o 输出文件] [选项] file1 file2

- 2) sort 默认使用第一个字段来进行合并排序。

### 5.uniq

1) uniq 用来从一个文本文件中去除或禁止重复行。一般 uniq 假定输入文件已分类。但 uniq 并不强制要求如此, 也可以使用任何非排序文本, 甚至是无规律行。

2) uniq 不同于 sort 的-u 选项, uniq 认为持续不断重复出现的行(中间不包括其它文本)才是重复行。

3) uniq Usage: uniq [选项] [input\_file [output\_file]]

4) uniq 常用选项:

-u 只显示没有重复的行。不使用此选项时, uniq 会把连续重复行的内容显示一次。

- d 只显示有重复数据行，每种重复行只显示其中一行。
  - c 打印每一重复行出现次数。
  - i 忽略字母的大小写
  - fx x 为数字(x=1,2..)，先跳过 x 个域再开始比较，与"-x"等效。有的系统使用"-nx"选项。
  - sx 跳过 x 个字符后再开始比较(x=1,2..)，与"+x"等效。与"-fx"连用时，一般放在"-fx"之后。
- 5) 举例: `uniq -f2 -s4 parts.txt` // 从文件 parts.txt 行中的第 3 个字段第 5 字符开始执行

## 6.join 使用共同的字段连接文件

1) join Usage: `join [选项] file1 file2`

2) join 程序主要用来连接两个文件的数据行，join 在执行时会在输入文件中寻找具有相同 join 字段的输入行，并把连接的结果输出到标准输出。

3) join 的两个输入文件都应该已经对要 join 的字段作了递增排序，否则执行结果将会错误。

4) join 默认使用每个输入文件文件行的第一个字段为 join 字段。输出行的字段间以一个空格隔开，每一个输出行上包括 join 字段、file1 内其余字段以及 file2 内的其余字段。

## 7.join 的常用选项

- 1) -ax : 将文件编号为 x(x=1,2)的文件中未被匹配连接的行额外打印出来。
- 2) -o x.y[,x.y,...] : 在输出中只打印文件 x 的第 y(y=1,2...)个字段。
- 3) -j m : 指定两个文件都用第 m(m=1,2...)个字段作为 join 字段。
- 4) -j1 m 或 -1 m : 指定文件 1 使用第 m(m=1,2...)个字段作为 join 字段。
- 5) -j2 m 或 -2 m : 指定文件 2 使用第 m(m=1,2...)个字段作为 join 字段。
- 6) -t char : 指定 char 为输入输出字段的分隔符，省略时使用 shell 环境变量 IFS。
- 7) -i : 忽略字母的大小写。
- 8) -vx : 只打印文件 x(x=1,2)中未被匹配连接的行，而不打印连接的结果。

## 8.cut 打印行中被选取的部分

1) cut 命令主要用于选择性的打印输入文件行的部分内容。

2) cut usage: `cut [options] input_file(s)`

3) 输入文件可以是一个文件列表。如果没有指定输入文件，或设置为"-", 将会使用标准输入为输入。

## 4) cut 的常用选项:

- ① -d char : 指定 char 为输入行字段分隔符(预设使用 shell 环境变量 IFS)。输出使用相同的域分隔符。
- ② -f 字段列表 : 只打印在字段列表中的字段。用数字表示要打印的字段，列表可以使用"-"和","。数字都是从 1 开始编号。"-m"表示 1-m，"-n"表示从 n 到最末。  
eg: `cut -d: -f1,6 /etc/passwd` // 打印系统中的用户名及其主(home)目录(即第 1 和第 6 个域)
- ③ -c 字符列表 : 用数字列表指定要剪切的字符位置，列表格式同②。  
eg: `who -u |cut -c1-8` // 打印有哪些用户正在使用系统

## 9.paste

1) paste Usage: paste [options] [file1 file2]

2) paste 将两个输入文件的每一行连接成一个新行并输出。file1 的行内容位于 file2 之前。

3) 缺省情况下，paste 连接时，用空格或 tab 键分隔新行中不同文本，除非指定"-d"选项。

4) 常用选项:

① -d char : 指定 char 作为字段分隔符。例如用@分隔域，使用-d@。

② -s : 将每个文件的内容作为一行输出，有 n 个输入文件就输出 n 行。

5) 没有指定输入文件而用 "-" 代替时，表示使用标准输入作为输入源。

eg: ls | paste -d " " - - - - - // 以每 5 个文件名为一行显示文件和目录

6) 粘贴两个不同来源的数据时，首先需应该将其分类(即要求已经排序)，并确保两个文件行数相同。

## 10.split 将文件分段

1) split 用来将大文件分割成小文件。有时文件越来越大，传送这些文件时，首先将其分割可能更容易。使用 vi 或其他工具诸如 sort 时，如果文件对于工作缓冲区太大，也会存在一些问题。因此有时没有选择余地，必须将文件分割成更小的文件。

2) split Usage: split [选项] input-filename [输出文件前缀]

3) 常用选项: split 的选项一般用来指定输出文件的尺寸，split 预设 100 行生成一个新的文件。

① -x、-lx、--lines=x : 每 x 行生成一个新的输出文件

② -b n、--bytes=n : 每 n 字节生成一个新的输出文件，可用后缀 b k m 表示以 block KB MB 为单位。

③ --verbose : 每生成一个新的输出文件时，就打印一行信息到标准错误。

4) 输出文件前缀: 默认每个生成的输出文件的格式为 xaa xab..到..xzy xzz。文件前缀为 x，aa ab..为文件名后缀。可以指定文件名前缀。

5) 举例: split -3 video.txt // 将文件 video.txt(共 8 行)分割成 3 个小文件 xaa、xab、xac

split -3 video.txt vdo // 将文件 video.txt 分割成 3 个小文件 vdoaa、vdoab、vdoac

## 第 12 章 tr 用法

### 1.tr 转换或删除字符

1) tr 主要用来从标准输入中通过替换或删除操作进行字符转换,然后打印输出到标准输出。可以通过管道或重定向标准输入来获得 tr 的输入数据。

2) tr 对输入数据可以进行三种操作: 字符替换、压缩重复字符和删除字符。

3) tr Usage: tr [OPTION] String1 [String2]

4) tr 有 4 个常用选项(c d s t)，下面将作详细讲解。

2. 在 tr 中 string1 和 string2 用来指示一个字符集合范围。可以是下面的一些形式(一般要加双引号):

1) [a-d]: [a-d]表示 abcd，常用的有[a-z]、[A-Z]、[0-9]等，[bfgh]表示 bfgH。可以加上单或双引号

2) [C\*n]: 表示字符 C 重复出现指定次数 n。因此，[F\*3]表示 FFF。一般只能出现在 string2 中。

3) \nnn: 三位八进制数，对应有效的 ASCII 字符。一般用于表示特定的控制字符。

速记符	含义	八进制方式
\a	Ctrl-G 铃声	\007
\b	Ctrl-H 退格符	\010
\f	Ctrl-L 走行换页	\014
\n	Ctrl-J 新行	\012
\r	Ctrl-M 回车	\015
\t	Ctrl-I tab 键	\011
\v	Ctrl-X	\030

4) [:class\_name:]: tr 支持使用内建的字符类别。常见的字符类别有:

[:alnum:] 字母、数字(0-9,a-z,A-Z)

[:alpha:]字母(a-z,A-Z)

[:cntrl:] 控制字符

[:digital:] 数字(0-9)

[:graph:] 可打印的字符，不包含空格

[:lower:]小写字母(a-z)

[:print:] 可打印的字符，包含空格

[:punct:]标点符号

[:space:] 空格

[:upper:] 大写字母(A-Z)

[:xdigital:] 16 进制数字(0-9,a-f,A-F)

5) 普通字符串形式: 比如 aeiou、bdfgh 等。

3. 选项"-c": 表示用在输入数据中出现，但是不包含在 string1 字符范围内的字符组成的集合，代替原来的 string1。在 tr 的三类操作中都可以使用选项"-c"。

eg: echo "adcfghg" | tr -cd fgca // 将显示 acfgg 输入数据中不包含在 string1 中的字符范围  
// 是"dh"，所以，实际上只是删除了字符"dh"

4. tr 用于字符替换: 只需要同时给出 string1 和 string2，可以需要不指定特别的选项。

1) Usage: tr string1 string2



输入数据中属于 string1 字符范围内的字符都将被替换，string1 中的第 n 个字符被替换为 string2 中的第 n 个字符。

eg: echo "adcdfgh" | tr adcgw vbnle // 将显示 vbnflh (v 替换 a, b 退回 d, n 替换 c, l 替换 g...)

2) 一般 string1 和 string2 的字符个数应该相同。如果字符个数不同，分为两种情况：

① string1 的字符个数少于 string2 的字符个数: string2 种额外的字符将被忽略。

② string1 的字符个数多于 string2 的字符个数，又可分为两种情况：

·对于 BSD 系列的系统：tr 会重复 string2 中的最后一个字符，直到补齐到 string1 一样的长度。

·对于 System V 的系统：tr 将截去 string1 中超长的部分。

GUN tr 使用 BSD 方式，如果想使用 System V 方式，需要指定选项"-t"。

eg: echo "addcdfghg" | tr adcgw vbn // 将打印 vbbnfnhn

echo "addcdfghg" | tr -t adcgw vbn // 将打印 vbbnfghg

3) tr 的字符替换操作主要用途有 大小写转换等。

eg: tr [a-z] [A-Z] < video.txt // 将文件 video.txt 中的所有小写字母转换成大写字母

tr ":" "\011" < /etc/passwd // 将 passwd 文件的域分隔符改为"\011"即"TAB"

5.tr 用于删除字符: 使用选项"-d"。

1) Usage: tr -d string1

输入数据中所有在 string1 中出现过的字符都将被删除。

2) echo "This is a note !" | tr -d [:space:] // 将打印 Thisisanote 即删除所有空格

tr -cd "[a-z][A-Z][\n]" <diary.txt // 文件 diary.txt 中非字母或回车的字符都将被删除

6.tr 用于压缩重复字符: 使用选项"-s"。压缩操作可以单独进行，也可以在替换操作或删除操作之后进行，因而分成 3 种情况：

1) Usage: tr -s string1

输入数据中连续出现的字符，只有还在 string1 中出现过的，才会将被压缩成一个字符。

eg: echo "aaaccdefffgghhh" | tr -s adeg // 将打印 accdeffgghhh (string1 中无字符"cfh")

tr -s "\n" < plane.txt // 删除文件 plane.txt 中的空行 (即压缩回车符)

2) Usage: tr -s string1 string2

string1 和 string2 用于替换操作，string2 还用于压缩操作。

eg: tr -s "[\015\032]" "[\012\*]" <input\_file // 将文件 input\_file 从 dos 格式转换到 unix 格式

3) Usage: tr -ds string1 string2

string1 用于删除操作，string2 用于压缩操作。

eg: echo "aaaccdefffgghhh" | tr -ds adeg fh // 将打印 ccfh

echo "aaaccdefffgghhh" | tr -d adeg // 将打印 cffffhh

7.在 tr 的替换操作中，如果 string1 使用了中括号"[]"，那么 string2 也应该使用"[]"。

8.tr 主要用于字符转换或者抽取控制字符。tr 的大多数功能都可以用 sed 来完成，但有些人宁愿使用 tr，因为 tr 更加快捷、容易。

## 第三部分 登录环境

### 第 13 章 登录环境

#### 1. 登录过程

- 1) 键入用户名和密码
- 2) 系统检查是否为有效用户(查询/etc/passwd 文件)
- 3) 登录名/密码有效则系统执行环境设置文件(/etc/profile -> \$HOME/.profile)

#### 2. 密码文件/etc/passwd 的格式(7 个域):

用户名:加密的密码:uid:gid:用户全名:用户 home 目录(主目录):用户的 shell 路径

eg: zqf:x:500:500::/home/zqf:/bin/bash

#### 3. 系统全局环境设置文件/etc/profile，此文件的设置内容一般包含有:

- 1) 全局或局部环境变量: 设置环境变量便于用户及其进程和应用访问它。eg:export 指令等。
- 2) PATH 信息: 设置环境变量 PATH，定位包含可执行文件、库文件及一般文本文件的目录位置，便于用户快速访问。
- 3) 终端设置: 使系统获知用户终端的一般特性。
- 4) 安全命令: 包括文件创建模式(umask)或其他安全限定。
- 5) 日期信息或放弃操作信息等。

#### 4. 用户个人环境变量设置文件\$HOME/.profile

- 1) 不同的 shell 有不同的文件命名，bash 为.bash\_profile。
- 2) 在\$HOME/.profile 文件中可以通过设置相关条目以不同的值或使用 uset 命令来覆盖/etc/profile 文件中的设置。

#### 5. stty

- 1) stty 用于设置终端特性。

#### 2) Usage1: stty [option]

常用选项:

-a : 以用户看得懂的格式打印当前所有的设定。

-g : 以 stty 的格式打印当前设定。可以用于回存 stty 设置。

eg : SAVEDSTTY=`stty -a` // 保存 stty 设置

stty \$SAVEDSTTY // 还原 stty 的初始设置

#### 3) Usage2: stty [-]arg // 打开[关闭]参数 arg

stty name value //设置特殊字符

eg: stty erase '^H'

## 6.logout 文件

- 1) 此文件保存有执行 exit 命令时，在进程终止前执行的命令
- 2) Bourne shell 与其他 shell 不同，它没有.logout 文件。Bash 的 logout 文件名为\$HOME/.bash\_logout

## 第 14 章 环境和 shell 变量

### 1.四种 shell 变量:

- 1) 本地变量
- 2) 环境变量

还有两种变量被认为是特殊变量(只读):

- 3) 位置变量
- 4) 特定参数变量

### 2.本地变量

- 1) 在用户现在的 shell 生命期的脚本中使用，如果在 shell 中启动另一个进程或退出，此值将失效。
- 2) 设置变量: variable-name=value  
eg: name=zqf ; echo \$name // 定义一个变量 name，然后打印该变量，将输出 zqf
- 3) 显示变量: echo \$variable-name 或 echo \${variable-name}
- 4) 清除变量: unset variable-name  
eg: unset name ; echo \$name // 清除了变量 name，echo 的输出为空白
- 5) 显示所有本地 shell 变量: set

### 3.测试变量是否已经设置(变量置换)

- 1) \${var:-value} : 如果变量 var 未定义,返回一个默认值。

如果 var 存在且非空,则表达式\${var:-value}的值为\$var;如果 var 未定义,则表达式值\${var:-value}为 value

eg: name1="zqf" ; name2=\${name1:-no name} ; echo \$name1 \$name2 // 将打印 zqf zqf  
unset name1 ; name2=\${name1:-no name} ; echo \$name1 \$name2 // 将打印 no name

- 2) \${var:=value} : 如果变量 var 未定义,设置 var 的默认值为 value。

如果 var 存在且非空,则\${var:=value}的值为\$var;如果 var 为空或未定义,则 var 被赋值 value 且表达式值为 value

eg: name1="zqf" ; name2=\${name1:=no name} ; echo \$name1 \$name2 // 将打印 zqf zqf  
unset name1 ; name2=\${name1:=no name} ; echo \$name1 \$name2 // 将打印 no name no name

- 3) \${var:?mesg} : 捕获未定义变量导致的错误。

如果 var 存在且非空,则\${var:?value}的值为\$var;如果 var 为空或未定义,则打印 mesg 并终止脚本

eg: name1="zqf" ; name2=\${name1:?error,no value} ; echo \$name1 \$name2 // 将打印 zqf zqf

```
unset name1 ; name2=${name1:?error,no value} ; echo $name1 $name2
//将打印 -bash: name1: error,no value
unset name1 ; name2=${name1:?error,no value} ; echo $name1
//将打印 -bash: name1: error,no value
unset name1 ; name2=${name1:?error,no value} ; echo $name2
//将打印 -bash: name1: error,no value
```

#### 4) \${var:+mesg} : 测试一个变量的存在性。

如果 var 存在且非空,则\${var:+mesg}的返回值为 mesg;如果 var 为空或未定义,则返回 null

eg: name1="zqf";name2=\${name1:+value ok};echo \$name1 \$name2 // 将打印 zqf value ok

```
unset name1 ; name2=${name1:+value ok};echo $name1 $name2 // 将打印 null null(空行)
```

#### 4.设置只读变量 : varname=value;readonly varname

查看所有只读变量 : readonly 或 readonly -p

#### 5.环境变量

1) 登录进程称为父进程,shell 中执行的用户进程均称为子进程。环境变量可用于所有子进程,这包括编辑器、脚本和应用。环境变量最好在 profile 文件中定义。习惯上,所有环境变量均应该大写。

2) 设置环境变量: VARNAME=value; export VARNAME

3) 显示环境变量: echo \$VARNAME

4) 查看所有环境变量: env

5) 清除环境变量: unset VARNAME

#### 6.SHELL 预留的环境变量名

每一种 shell 有一些预留的环境变量名,这些变量名不能用作其他用途。

eg : Bourne Shell 的环境变量有 HOME,PATH,LANG,PWD,PS1,EDITOR...

#### 7.export 命令

1) export varname 命令可以把变量 varname 输出到子进程中,如果在子进程中修改了变量 varname 的值,退出子进程后,在子进程中被赋的值将不会传回到父进程。

2) 不可以将变量从子进程导出到父进程,然而通过重定向就可做到这一点

#### 8.位置参数变量 位置变量的数目有可以任意多,但一般只有\$0 - \$9 可以被访问,多用于脚本中。

\$0 \$1 \$2 \$3 \$4 \$5 \$6 \$7 \$8 \$9 (\$0 的值为脚本名)

eg: service httpd status // 对于脚本/sbin/service 来说,\$0 的值为 service,\$1 的值为 httpd,  
// \$2 的值为 status。可以在 service 脚本中使用这些变量来传递参数值

#### 9.特定参数变量(7 个)

\$#: 传递到脚本的参数的个数。

\$\*: 以一个单字符串显示所有向脚本传递的参数。\$\* 等价于"\$1 \$2 \$3 .. \$n"。

`$$` : 脚本运行的当前进程 ID 号。

`$!` : 最后一个后台运行的进程的进程 ID 号。

`$@` : 与 `$*` 类似，但是使用时加引号，并在引号中返回每个参数(返回一个参数列表)。`$*` 等价于 `"$1" "$2" .. "$n"`。

`$-` : 显示 shell 使用的当前选项。

`$?` : 显示前面最后一个命令的退出状态。0 表示没有错误，其他任何值表明有错误。

## 10.退出状态

1) `$?` 可以在任何命令或脚本中返回此变量以获得返回信息。检验脚本退出状态时，最好将返回值赋值给一个有意义的名字的变量，这样可以增加脚本的可读性。

2) 举例: `cp ok.txt /usr/local/app/def >/dev/null 2>&1`

```
cp_status=$?           // 保存上一条命令(cp)的退出状态
echo $cp_status         // 打印保存的状态值
```

## 第 15 章 引号

### 1.双引号(" ")

1) 使用双引号可引用除字符 `$`(美元符号)、```(反引号)、`\`(反斜线) 外的任意字符或字符串。双引号不会阻止 shell 对这 3 个字符作特殊处理(标示变量名、命令替换、反斜线转义)。

```
eg: name=zqf; echo "User name : $name" // 将打印 User name : zqf
    echo "The date is : `date +%d-%m-%Y`" // 将打印 The date is : 02-15-2005
    echo -e "$USER\t$UID" // 将打印 zqf      500
```

2) 如果要查询包含空格的字符串，经常会用到双引号。

### 2.单引号(' ')

1) 如果用单引号把字符串括起来，则单引号内字符串中的任何特殊字符的特殊含义均被屏蔽。

2) 举例: `echo -e '$USER\t$UID' // 将打印 $USER $UID` (没有屏蔽 `\t`，是因为选项 `-e` 的缘故)

```
echo '$USER\t$UID' // 将打印 $USER\t$UID
```

### 3.反引号(` `)

1) shell 将反引号中的内容作为一个系统命令，并执行其内容。使用这种方法可以替换输出为一个变量。

2) 举例: `a=`date +%d-%m-%Y`; echo $a // 将打印 16-02-2005`

### 4.反斜线(\)

1) 如果下一个字符有特殊含义，反斜线防止 shell 误解其含义，即屏蔽其特殊含义。

2) 下述字符包含有特殊意义: `& * + ^ $ ` " | ?`

3) 在打印字符串时要加入八进制字符(ASCII 相应字符)时，必须在前面加反斜线，否则 shell 作普通数字处理。

4) 举例: `bj=beijing ; echo "variable \${bj} = ${bj}"` // 将打印 `variable ${bj} = beijing`

## 第四部分 基础 shell 编程

### 第 16 章 shell 脚本介绍

1.脚本内容 shell 脚本不是复杂的程序，它是按行解释的。

1) shell 脚本第一行总是以#!/bin/sh 开始，这段脚本通知 shell 使用系统上的 Bourne shell 解释器。

2) shell 脚本加注释行要求该行的第一个字符为#，第二行注释中写入脚本名是一个好习惯。

3) shell 脚本从上到下执行。运行脚本前需要增加其执行权限。确保正确建立脚本路径。

### 第 17 章 条件测试

1.test 命令

1) test 命令用于测试字符串，文件状态和数字。test 一般有两种格式，即：

test condition 或 [ condition ] (使用方括号时，要注意在条件两边加上空格)

2) test 命令的退出状态: 0 表示成功，其他非 0 为失败

2.测试文件状态

1) 常见的文件状态测试:

-b filename : 当 filename 存在并且是块文件时返回真(返回 0)  
 -c filename : 当 filename 存在并且是字符文件时返回真  
 -d pathname : 当 pathname 存在并且是一个目录时返回真  
 -e pathname : 当由 pathname 指定的文件或目录存在时返回真  
 -f filename : 当 filename 存在并且是正规文件时返回真  
 -g pathname : 当由 pathname 指定的文件或目录存在并且设置了 SGID 位时返回真  
 -h filename : 当 filename 存在并且是符号链接文件时返回真 (或 -L filename)  
 -k pathname : 当由 pathname 指定的文件或目录存在并且设置了"粘滞"位时返回真  
 -p filename : 当 filename 存在并且是命名管道时返回真  
 -r pathname : 当由 pathname 指定的文件或目录存在并且可读时返回真  
 -s filename : 当 filename 存在并且文件大小大于 0 时返回真  
 -S filename : 当 filename 存在并且是 socket 时返回真  
 -t fd : 当 fd 是与终端设备相关联的文件描述符时返回真  
 -u pathname : 当由 pathname 指定的文件或目录存在并且设置了 SUID 位时返回真  
 -w pathname : 当由 pathname 指定的文件或目录存在并且可写时返回真  
 -x pathname : 当由 pathname 指定的文件或目录存在并且可执行时返回真  
 -O pathname : 当由 pathname 存在并且被当前进程的有效用户 id 的用户拥有时返回真(字母 O 大写)  
 -G pathname : 当由 pathname 存在并且属于当前进程的有效用户 id 的用户的用户组时返回真  
 file1 -nt file2 : file1 比 file2 新时返回真  
 file1 -ot file2 : file1 比 file2 旧时返回真



2) 举例: `if [ -b /dev/hda ] ;then echo "yes" ;else echo "no";fi` // 将打印 yes

`test -c /dev/hda ; echo $?` // 将打印 1 表示 test 命令的返回值为 1, /dev/hda 不是字符设备  
`[ -w /etc/passwd ]; echo $?` // 查看对当前用户而言, passwd 文件是否可写

3.测试时使用逻辑操作符(逻辑与、或、非) 用于[ ]操作符

1) -a 逻辑与, 操作符两边均为真, 结果为真, 否则为假。

2) -o 逻辑或, 操作符两边一边为真, 结果为真, 否则为假。

3) ! 逻辑否, 条件为假, 结果为真。

4) 举例: `[ -w result.txt -a -w score.txt ] ;echo $?` // 测试两个文件是否均可写

4.字符串测试 字符串测试是错误捕获很重要的一部分, 特别在测试用户输入或比较变量时尤为重要

1) 常见字符串测试

`-z string` : 字符串 string 为空串(长度为 0)时返回真

`-n string` : 字符串 string 为非空串时返回真

`str1 = str2` : 字符串 str1 和字符串 str2 相等时返回真

`str1 != str2` : 字符串 str1 和字符串 str2 不相等时返回真

`str1 < str2` : 按字典顺序排序, 字符串 str1 在字符串 str2 之前

`str1 > str2` : 按字典顺序排序, 字符串 str1 在字符串 str2 之后

2) 举例: `name="zqf"; [ $name = "zqf" ];echo $?` // 打印 0 表示变量 name 的值和字符串"zqf"相等

5.测试数值(比较整数)

1) 常见数值测试

`int1 -eq int2` : 如果 int1 等于 int2, 则返回真

`int1 -ne int2` : 如果 int1 不等于 int2, 则返回真

`int1 -lt int2` : 如果 int1 小于 int2, 则返回真

`int1 -le int2` : 如果 int1 小于等于 int2, 则返回真

`int1 -gt int2` : 如果 int1 大于 int2, 则返回真

`int1 -ge int2` : 如果 int1 大于等于 int2, 则返回真

2) 举例: `x=1; [ $x -eq 1 ]; echo $?` // 将打印 0 表示变量 x 的值等于数字 1

`x=a; [ $x -eq "1" ]` // shell 打印错误信息 [: a: integer expression expected

6.expr 用于数值和字符串运算

1) expr Usage: `expr expression`

2) expr 命令运算完毕后, 除了会返回表达式运算的结果外, 还会生成一个 expr 执行状态码表示 expr 的执行状态。

3) expr 的执行状态码:

0: 表达式结果不是 0 或 null

1: 表达式结果是 0 或 null

2: 表达式无效

```
eg: LOOP=10;expr $LOOP '-' 10 // 输出结果为 0 (表达式$LOOP '-' 10 的运算结果)
    echo $?                // 输出结果为 1 (expr 的执行状态码)
```

4) expression: 可以是一个数值表达式, 也可以是一个字符串表达式。可以进行数值或字符串运算。

```
eg: expr zqf // 将打印 zqf
```

## 7.expr 支持的运算符

0) expr 的 expression 中, 运算符前后都要留一个空格, 并且一般最好加上单引号。

1) expr 的 expression 中可使用加(+)减(-)乘(\*)整除(/)取余(%)运算符。

2) expr 的 expression 中还可以使用两类运算符:

① | : 如果第一个参数不是 null 也不是 0, 则使用第一个参数为算子, 否则使用第二个参数为算子。

& : 如果第一个参数不是 null 也不是 0, 则使用第一个参数为算子, 否则使用数值 0 为算子。

```
eg: a=0 ; expr $a '|' 4 + 5      // 将打印 9
    a=6 ; expr '(' $a '&' 4 ')' + 5 // 将打印 11
    a=0 ; expr '(' $a '&' 4 ')' + 5 // 将打印 5
```

② 比较运算符: <、<=、=、==、!=、>=、> (== 相当于 =)

比较两个参数的逻辑关系, 如果逻辑关系正确, 返回数值 1, 否则返回数值 0。如果两个参数都是数字则按算术大小比较, 否则按字典顺序比较。

```
eg: b=beijing ; expr $b = beijing // 将打印 1
    b="c" ; expr $b '<' d          // 将打印 1
```

## 8.expr 用于整数运算

1) expr 的 expression 中运算数只能是整数不能是小数。

2) 举例: expr '(' 2 '+' 3 ')' '\*' 2 // 将打印 10

```
LOOP=0 ; LOOP=`expr $LOOP + 1` // 在循环结构中, expr 可用于增量计数
```

9.expr 可以用来测试一个数, 如果试图计算非整数, expr 将返回错误信息。

```
eg: r=4 ; expr $r + 4 // 将打印 8
    r="a" ; expr $r + 4 // 将打印错误信息 expr: non-numeric argument
    expr 1.5 + 4        // 将打印错误信息 expr: non-numeric argument
```

## 10.expr 用于字符串运算

1) 常见字符串运算符:

expr length STRING : 返回字符串 STRING 的长度

expr index STRING CHAR : 返回字符 CHAR 在字符串 STRING 中第一次出现的位置(没有出现则返回 0)

expr substr STRING POS LEN : 返回 STRING 的子字符串(从位置 POS 开始长度为 LEN)

```
eg: expr length "Linux and Unix Shell Programming" // 返回字符串的长度, 即打印 22
    expr index "Linux and Unix Shell Programming" i // 将打印 2
    expr substr "Linux and Unix Shell Programming" 16 7 // 将打印 Shell P
```

2) 字符串模式匹配: 使用正则表达式 REGEXP 时，要用单引号括起来。

① `expr match STR REGEXP` : STR 是字符串，REGEXP 是正则表达式。返回 STR 中匹配 REGEXP 的字符个数。

`expr STR:REGEXP` : 相当于 `expr match STR REGEXP`。

eg: `expr accounts.doc : b` // 将打印 0

`expr accounts.doc : acc` // 将打印 3

`expr 1234dfgh : '[0-9]*'` // 将打印 4

`expr 1234dfgh : '.'` // 将打印 8

② 在 REGEXP 中使用小括号(要加上转义符"\")，则 `expr` 返回的是由这对括号所包括的内容。

eg: `expr abcdefgh : '...\(...\).'` // 将打印 def

`expr accounts.doc : '\(.*\)doc'` // 将打印 accounts

## 第 18 章 控制流结构

### 1.退出状态

1) 四种退出状态:

① 最后命令退出状态(用"\$?"指示)

② 控制次序命令"\$\$"和"||"

③ shell(脚本)退出状态

④ 函数返回码

2) `exit [n]` :退出并返回退出码 n，n 为整数

① 在 shell 命令行下使用 `exit [n]`，将退出当前 shell

② 在 shell 脚本中使用 `exit`，将退出脚本 shell 并返回 `exit` 上一条命令的退出状态码。

③ 在 shell 脚本中使用 `exit n`，将退出脚本 shell 并返回退出状态码 n

### 2.控制结构

1) 流控制：if-then-else 语句、case 语句

2) 循环：for 循环、until 循环、while 循环语句

### 3.if-then-else 语句

1) 语法结构:

`if condition1` // 如果 condition1 为真(返回值为 0),condition 一般是条件测试表达式

`then` // 那么

`statements1` // 执行语句块 1

`[elif condition2` // 如果(条件 1 不成立)而条件 2 成立 (If 语句可以有許多 elif 部分)

`then` // 那么

`statements2]` // 执行语句块 2

```
[else                // 如果条件 1[, 条件 2]均不成立
    statements3 ]    // 执行语句块 3
fi                  // 完成 (if 语句必须以单词 fi 终止。在 if 语句中漏写 fi 是最一般的错误)
```

2) if 语句的各个 condition 和 statements 部分不能留空。如果想留空的话, 必须使用 shell 提供的空命令 ":"(即冒号), 来占据要留空的位置。空命令表示永远为真。

3) 条件 condition 可以是命令语句列表, 以最后一个命令的退出状态用作条件值。

#### 4.case 语句

1) case 语句为多选择语句。一个值与多个模式匹配, 如果匹配成功, 执行相匹配的命令。

2) 语法结构:

```
case expression in    // expression 为一表达式 ( in 不能忘掉了 )
    pattern1 )         // 如果表达式 expression 匹配模式 pattern1
        statements1 ;;  // 则执行语句块 1,然后退出。如果不匹配, 则进行模式 2 的匹配
    pattern2 )         // 如果表达式 expression 匹配模式 pattern2
        statements2 ;;  // 则执行语句块 2,然后退出。如果不匹配, 则进行模式 3 的匹配
    .....              // .....
    patternN )         // 如果表达式 expression 匹配模式 patternN
        statementsN ;;  // 则执行语句块 N, 然后退出。
esac                  // case 语句结束标志
```

3) 表达式按顺序匹配每个模式。一旦有一个模式匹配成功, 则执行完该模式相应命令后退出不再继续匹配。

4) 模式部分可以包括元字符: \* 任意字符、? 任意单字符、[] 类或范围中任意字符。

5) 为了防止表达式未匹配到任何模式, 可以在最后一个模式中使用星号\*来表示匹配任意表达式 expression

6) 每个模式的语句块必须以 2 个分号(;;)结尾

7) 在模式 pattern 中可以使用或操作符"|"。

eg: vt100 | vt102 | vt 220 ) statements ;;

#### 5.for 循环

1) 语法结构:

```
for name [in list]    // 每一次循环, 依次把列表 list 中的一个值赋予变量 name
do                    // 循环体开始标志
    statements using $name // 变量 name 每取一次值时, 循环体执行一遍
done                  // for 循环体结束标志
```

2) list 应该是一系列由空格分隔的字符序列(单词), 省略 in list 时默认为\$@ (命令行的参数列表)

3) 列表 list 可以是命令替换、变量名替换、字符串和文件名列表(\*可表示当前目录下所有文件)

4) for 循环执行的次数取决于列表 list 中单词的个数

5) 在 for 循环体中一般应该要出现变量\$name, 但也可以不出现。

## 6.until 循环

1) 当条件为假时执行循环体(执行循环体直到条件为真时停止)

2) 语法结构:

```
until condition // 当条件 condition 不成立时
do              // 开始执行
  statements    // 循环体
done            // do 循环结束标志
```

3) until 释义: until prep. 在...以前,到...为止 ; conj. 在...以前, 到...为止, 直到...才

4) until 在每次循环开始前先检查条件 condition , 不同于 C 语言的"直到.."(先循环后检查条件)结构

5) condition 一般是 test 一类的表达式。也可是命令语句列表, 则以最后一个命令的退出状态作条件值。

## 7.while 循环

1) 当 condition 为真时执行循环体命令

2) 语法结构:

```
while condition // 执行 condition。(condition 可以是 cmd 或 test 一类表达式)
do              // 如果 cmd 的退出状态为 0, 则执行循环体。cmd 退出状态不为 0, 退出循环。
  statements    // 循环体
done            // 循环体结束, 返回第一步。
```

3) while 释义: while conj.当...的时候

4) condition 一般是 test 一类的表达式。也可是命令语句列表, 则以最后一个命令的退出状态作条件值。

5) while 循环常用于从一个文件中一次循环读取一行数据, 这时需要使用输入重定向。

eg : while read LINE ; do echo \$LINE ; done < data.txt

6) 在 5)的情况下, while 也可以一次把一行数据的各个域分别读入到不同的变量中。(注意设定 IFS 变量)

eg: IFS=: ; while read F1 F2 F3 ; do echo -e "\$F1\t \$F2\t \$F3\t" ; done < data.txt

7) 如果希望每次处理 2 个记录, 则可在 while 后放一个 read var1 语句, 再在循环体中放一个 read var2 语句

## 8.break 语句

1) break 语句允许退出循环或 case 语句。

2) Usage: break [n] // 跳出[n 层]循环

## 9.continue

1) continue 语句用于跳出当前本轮循环步, 重新开始新一轮的循环步。

2) Usage: continue

# 第 19 章 shell 函数

## 1.函数

### 1) 定义函数的语法:

```
函数名 ()      or      function 函数名 ()      // 小括号"()"不能掉
{
statements
}
{
statements
}
// "{" 和 statements 之间至少要有有一个空格
```

2) 可以在命令行或脚本中定义函数。所有函数必须先定义后使用，在脚本中函数一般在开始部分定义。

3) 调用函数仅使用其函数名即可，可以带参数。函数一旦定义了，可作为一个合法的命令，用在所有的后继 shell 中。

eg: funcname arg1 arg2 ...

4) 在一个脚本中定义的函数只能在那个脚本和所有由该脚本生成的子 shell 中。

5) 向函数传递参数就像在一般脚本中使用特殊变量\$1、\$2...\$9 和 @\$ 一样。

6) 可以在一个函数的定义内部调用另一个函数 -- 函数链接。

7) 可以再一个函数的定义内部调用这个函数本身 -- 函数递归。

### 2.从调用函数中返回，可以有两种处理方式:

1) 让函数正常执行到函数末尾，然后返回脚本中调用函数的控制部分(默认方式)。

2) 使用 return 返回脚本中函数调用的下一条语句，可以带返回值。0 为成功，非 0 为有错误。

## 3.在 shell 中使用函数

1) 将函数的定义写入函数文件 func\_file 中。

2) 将函数文件 func\_file 的内容载入 shell，在 bash 下使用命令: source func\_file

3) 当需要修改函数定义时，先删除函数( unset func\_name )，然后再修改文件中有关函数 func\_name 的内容。

4) 重新载入函数文件( source func\_file )，使函数修改生效。

## 4.echo 问题

echo 语句的使用类型依赖于使用的系统是 LINUX、BSD 还是 System V

1) 使提示符放在语句末尾，而不是在新行的行首

BSD/Linux :     echo -n "string"

System V :     echo "string\c"

2) LINUX 的 echo 语句必须额外使用 -e 选项用来显示反馈控制字符。

## 第 20 章 向脚本传递参数

1.任何 UNIX 或 LINUX 命令均接受的一般格式：命令 选项 文件

- 1) 选项部分最多可包含 12 个不同的值，如果必须控制不同的命令选项，就要加入大量脚本。
- 2) shell 提供 shift 命令以帮助偏移选项，使用 shift 可以去除只使用\$1 到\$ 9 传递参数的限制。

## 2.shift 命令

- 1) 功能: 它每次将参数位置向左偏移一位
- 2) shift Usage: shift N // N 为一个数字
- 3) 举例: shift `expr \$# - 1` // 使命令行的最后一个参数(通常为文件名)成为\$1  
// 也可以用 eval 命令来实现上述功能: eval echo \\$\$# (显示命令行最后的参数)

4)注意: 当脚本中使用了一次 shift 后，相应脚本命令行的 \$# 值会减少 1, \$\* 和\$@ 的值也相应的变化

## 3.getopts

- 1) getopts 可以编写脚本，使控制多个命令行参数更加容易。getopts 用于形成命令行处理标准形式。
- 2) getopts Usage: getopts option\_string variable\_name
- 3) getopts 命令中的 option\_string 字符串:
  - ① getopts 读取字符串 option\_string，获知脚本可以使用的有效选项(在命令行中选项应该以"-"开头)。
  - ② option\_string 由字母和冒号组成，每一个字母就是一个有效选项。  
如果 option\_string 中一个字母后跟一个":", 表示该字母选项后应该有一个参数。  
如果 option\_string 以":"开头，表示当命令行中出现了无效选项时，getopts 不打印错误信息。  
eg: getopts :ac:h optchar // 表示脚本文件在命令行下只可以接受选项-a -h -c(必须带参数)

## 4.getopts 的运行方式

- 1) getopts 语句一般和 while 循环、case 结构联合使用。

```
eg: while getopts :ac:h OPT
do
    case $OPT in
        a) statements ;; // 当出现选项-a 时，要执行的语句
        c) statements ;; // 因为选项-c 要带参数，该参数由环境变量$OPTARG 指示
        h) statements ;; // 当出现选项-h 时，要执行的语句
    esac
done
```

2) getopts 语句放在 while 循环结构中，getopts 语句每执行一次就依次从 option\_string(如上例的:ac:h)中取出一个字母(忽略":")，如果在命令行中有选项匹配该字母，则变量 variable\_name(如上例的 OPT)将被赋值为该选项字母，如果该选项带有一个参数(字母后带冒号的情形下)，该选项的参数将被保存到环境变量\$OPTARG 中。

3) 每次 getopts 执行时，如果在命令行的第 x(x=1...n)个参数中匹配到字母选项，则环境变量 OPTIND 的值为 x+1。在命令行脚本启动时，OPTIND 会被初始化为 1。

4) 当 getopt 匹配完 option\_string 中的字母选项后，getopt 将以大于 0 的返回值退出。

## 5. getopt 对错误的处理

### 1) 两类错误:

- ① 命令行中出现非法选项字母(没有被包含在 option\_string 中的)。
- ② 需要带参数的选项在命令行中没有给出参数(参数应该跟在该选项字母后)。

### 2) 错误处理方式:

- ① silent 方式 : option\_string 以冒号开头，即 getopt 将不打印错误信息。
- ② 非 silent 方式 : getopt 将打印错误信息。

当设置环境变量 OPTERR 为 0 时，即使在非 silent 方式下 getopt 也不会打印错误信息。

### 3) 错误处理:

- ① 当 getopt 遇到非法选项时，变量 variable\_name 将设置为"?"。
  - 在非 silent 方式下，变量 OPTARG 被清零，然后 getopt 打印错误信息。
  - 在 silent 方式下，该非法选项字母将放入变量 OPTARG，getopt 不打印错误信息。
- ② 当需要带参数的选项在命令行中没有给出参数时，变量 variable\_name 将赋值"?"。
  - 在非 silent 方式下，变量 OPTARG 被清零，然后 getopt 打印错误信息。
  - 在 silent 方式下，该非法选项字母将放入变量 OPTARG，getopt 不打印错误信息。
- ③ 可以看出，getopt 对两类错误并未作区分，采用了同样的处理。

## 6. Unix/Linux 命令行程序常用的选项字母

- a 扩展
- c 计数、拷贝
- d 目录、设备
- e 执行
- f 文件名、强制
- h 帮助
- i 忽略状态
- l 注册文件
- o 完整输出 t
- q 退出、安静模式
- p 路径
- v 详细显示、版本

## 第 21 章 创建屏幕输出

### 1. tput



1) 使用 tput 命令可以增强应用外观及脚本的控制，tput 使用文件/etc/terminfo 或/etc/termcap。

2) tput Usage: tput [-V] [-S] [-Ttermtype] capname

① 选项-V: 显示程序所用的 ncurses 版本;

选项-S: 从标准输入获取 capname 参数。(eg: tput -S << MAYDAY)

② 选项-T termtype : 省略此参数时，tput 从环境变量 TERM 中获取 termtype 值(T 与 termtype 间可无空格)

③ tput [-T termtype] init // 在使用 tput 前，需要在脚本或命令行中使用此命令初始化终端

④ tput [-T termtype] reset // 将环境变量 TERM 复位(reset)为 termtype 给定的值

⑤ tput [-T termtype] longname // 打印终端类型 termtype 的正式长格式名称

3) tput 产生三种不同的输出：字符型、数字型和布尔型(真/假)。即: capname 参数

① 大部分常用字符串：

```
bel          警铃
blink       闪烁模式
bold        粗体
civis       隐藏光标
clear       清屏
cnorm       不隐藏光标
cup x y     移动光标到屏幕位置(x,y)
el          清除到行尾
ell         清除到行首
smso        启动突出模式
rmso        停止突出模式
smul        开始下划线模式
rmul        结束下划线模式
sc          保存当前光标位置
rc          恢复光标到最后保存位置
sgr0        正常屏幕
rev         逆转视图
```

② 大部分常用数字输出:

```
cols       列数目
it         tab 设置宽度
lines      屏幕行数
```

③ 两种布尔操作符:

```
chts       光标不可见
hs         具有状态行
```

4) 举例: tput clear // 清屏

tput cup 23 4 // 定位光标到行 23 列 4

tput bel // 响铃

2.使用转义序列和产生控制码

1) 所有控制字符均以转义序列\033开始,然后后紧跟字符[, 最后是表示控制字符的实际序列, 用于打开或关闭某终端属性

2) 解析: `echo -e "\033[?25l"` // 发送一转义序列以关闭光标使之不可见。 \003 为转义键取值  
 // 表示接下来是一个控制字符的序列; [ 为分隔符  
 // ?25l 为控制字符的实际序列

3) 控制序列一般嵌在 `echo` 语句中, 可用的语法有:

① Linux/BSD : `echo -e "\033[<控制字符实际序列>"`

② System V : `echo "\033[<控制字符实际序列>"`

③ Generic : `echo "<Ctrl+v><Esc>[<控制字符实际序列>"` // 用<Ctrl+v><Esc>代替\033, 显示为 ^[

4) 常用的 <控制字符实际序列>

关闭光标	?25l(是字母 l)
打开光标	?25h
清屏	2J

### 3.使用颜色控制码

1) 屏幕颜色控制代码: <背景色代码;前景色代码 m>

eg: `echo -e "\033[40;32m"` // 产生一个黑色背景加绿色前景色

2) 常用前景色代码:

- 30 黑色
- 34 蓝色
- 31 红色
- 35 紫色
- 32 绿色
- 36 青色
- 33 黄(或棕)色
- 37 白(或灰)色

3) 常用背景色代码:

- 40 黑色
- 44 青色
- 41 红色
- 45 蓝色
- 42 绿色
- 46 青色
- 43 黄(或棕)色
- 47 白(或灰)色

### 4.创建精致菜单(按单键选择菜单), shell 编程使用的主要结构

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

```

# just read a single key please
get_char()
{
# get_char
# save current stty settings
SAVEDSTTY=`stty -g`
    stty cbreak
    dd if=/dev/tty bs=1 count=1 2> /dev/null
    stty -cbreak
# restore stty
stty $SAVEDSTTY
}
echo -e -n "\tYour Choice [1,2,3,4,5] :."
read CHOICE
CHOICE=`get_char`
case $CHOICE in
1) doing submenu1's things
    ;;
2) doing submenu2's things
    ;;
3) doing submenu3's things
    ;;
4) doing submenu4's things
    ;;
5) doing submenu5's things
    ;;
*) default
    ;;

```

## 第 22 章 创建屏幕输入

1. 屏幕输入或数据输入是接受输入（这里指键盘）并验证其有效的能力。如果有效，接受它，如果无效，放弃该输入。
2. 本章主要给出了几个 shell 脚本例子，没有提到特别的知识点。

## 第 23 章 调试 shell 脚本

1.当 shell 打印出一个脚本错误后，不要只看那些疑问行。而是要观察整个相关代码段。shell 不会对错误进行精确定位，而是在试图结束一个语句时进行错误统计。

## 2.shell 脚本的一般错误

- 1) 循环错误: for、while、until 和 case 语句中的错误是指实际语句段不正确。也许漏写了固定结构中的一个保留字。
- 2) 典型的漏写引号: 解决这类错误的唯一方案是在脚本中确保所有引号成对出现。使用 vi 的 set nu 选项调试错误,可以定位文本行号。
- 3) 测试语句错误: 使用 -eq 语句时忘记在测试条件一边使用数字取值;在变量和方括号间忘记加空格;在方括号里漏写操作符。
- 4) 字符大小写: 经验上讲大多数错误是由于使用变量时大小写保持一致。
- 5) for 循环: 使用 for 循环时，有时会忘了在循环的列表部分用\$符号，特别是在读取字符串时。

## 3.调试脚本工具

### 1) echo

最有用的调试脚本工具是 echo 命令。一般在可能出现问题的脚本重要部分加入 echo 命令，例如在变量读取或修改操作其前后加入 echo 命令。使用最后状态命令判断命令是否成功，这里需要注意的是，不要使用 echo 命令后直接加最后状态命令，因为此命令永远为真。

### 2) set

#### ① set 命令常用的调试选项:

- set -n 读命令但并不执行。
- set -v 显示读取的所有行。
- set -x 显示所有命令及其参数。
- 将 set 选项关闭，只需用+替代-。

② 可以在脚本开始时将 set 选项打开，然后在结束时关闭它。或在认为有问题的特殊语句段前后打开及关闭它。

4.跟踪错误的最好方式是亲自查阅脚本，并使用 set 命令并加大量的 echo 语句。

## 第 24 章 shell 嵌入命令

### 1.shell 嵌入命令

这些命令是在实际的 Bourne shell 里创建而不是存在于/bin 或/usr/bin 目录里。嵌入命令比系统里的相同命令要快。(类似于 dos 的内部命令和外部命令之分)

### 2.bash shell 嵌入命令完整列表

: 空，永远返回为 true

. 从当前 shell 中执行操作

break 退出 for、while、until 或 case 语句

cd 改变到当前目录

continue 执行循环的下一步

echo 反馈信息到标准输出

eval 读取参数，执行结果命令

exec 执行命令，但不在当前 shell

exit 退出当前 shell

export 导出变量，使当前 shell 可利用它

pwd 显示当前目录

read 从标准输入读取一行文本

readonly 使变量只读

return 退出函数并带有返回值

set 控制各种参数到标准输出的显示

shift 命令行参数向左偏移一个

test 评估条件表达式

times 显示 shell 运行过程的用户和系统时间

trap 当捕获信号时运行指定命令

ulimit 显示或设置 shell 资源

umask 显示或设置缺省文件创建模式

unset 从 shell 内存中删除变量或函数

wait 等待直到子进程运行完毕，报告终止

### 3. 一些 shell 嵌入命令

1) pwd : 显示当前目录。

2) set : 在查看调试脚本、打开或关闭 shell 选项时，曾用到 set 命令。

set 也可用于在脚本内部给出其运行参数，在脚本中使用 set param1 param2 ... 且在命令行下不向该脚本传递参数时，系统会将 param1、param2..当作\$1、\$2..来看待。当测试一段脚本且脚本包含参数时，这样使用 set 命令有很多用处。其一就是不必在每次运行脚本时重复输入参数。

3) times : times 命令给出用户脚本或任何系统命令的运行时间。第一行给出 shell 消耗时间，第二行给出运行命令消耗的时间。

4) type : 使用 type 查询命令是否仍驻留系统及命令类型。type 打印命令名是否有效及其路径位置。

5) ulimit : ulimit 设置运行在 shell 上的显示限制。通常此命令出现在文件/etc/profile 中。也可以从当前 shell 或用户的.profile 文件中将之移入用户需要的位置。

① ulimit 一般格式: ulimit options

② 一些常用的选项:

-a 显示当前限制

-c 限制内核垃圾大小

-f        限制运行进程创建的输出文件的大小

6) wait : wait 命令等待直到一个用户子进程完成，可以在 wait 命令中指定进程 ID 号。如果并未指定，则等待直到所有子进程完成。

## 第五部分 高级 shell 编程技巧

### 第 25 章 深入讨论<<

#### 1. << 的用法

- 1) 该命令的一般形式为: `command <<word`      // 当 shell 看到"<<"时, 它就会知道下一个词是一个分界符  
                                  `text...`                      // 在该分界符以后的内容都被当作输入  
                                  `word`                      // 直到 shell 又看到该分界符(位于单独的一行)

2) 分界符 word 可以是你所定义的任何字符串。如果在 text 中使用 tab 键, 可以在"<<"之后加一个横杠"-"

3) 可以使用"<<"来创建文件、显示文件列表、排序文件列表以及创建屏幕输入等。

#### 2. << 应用举例

- 1) 快速创建一个文件: `cat >> myfile <<ENDFILE`

现在可以输入一些文本, 结束时只要在新的一行键入 ENDFILE 即可, 这样就创建了一个名为 myfile 的文件, 该文件中包含了一些文本。

- 2) 快速创建打印文档: `lpr <<QUICKDOC`

现在可以输入一些文本, 结束时只要在新的一行键入 ENDFILE 即可, 这样输入的文本就可以被 lpr 打印了。

3) "<<"的用途很广, 特别是在连接某些应用程序如使用 ftp 时。你可以灵活地使用"<<"来自动运行以前编写的脚本, 从而完成各种不同的任务。

- 4) 一个自动 ftp 传输脚本示例: 当要定期登陆 ftp 下载文件时, 编写脚本很有用。

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

```
ftp -i -n 172.25.151.123 <<FTPEND // 使用 ftp -i -n 选项, 表示不要自动登陆, 且关闭交互模式
```

```
user anonymous zqf@site.com // 匿名登陆
```

```
binary // 设置 ftp 的传输模式为 binary
```

```
cd /pub/data // 在 ftp 服务器上切换目录
```

```
get data.doc ~zqf/data.ftp.doc // 下载文件
```

```
quit // 退出 ftp 程序
```

```
FTPEND
```

### 第 26 章 shell 工具

#### 1. 创建保存信息的文件

- 1) 任何脚本都应该能够创建临时文件或日志文件。在运行脚本做备份时, 最好是保存一个日志文件。

- 2) 在开发脚本的时候，可能总要创建一些临时的文件。在正常运行脚本的时候，也要使用临时文件保存信息，以便作为另外一个进程的输入，在脚本的末尾一般用 rm 命令删除临时文件。
- 3) 在创建日志文件或临时文件时，最好能够使它的文件名具有唯一性。

## 2.使日志文件的文件名具有唯一性: 文件名中加入日期和时间信息

### 1) date 命令用法:

① Usage: date [option] +<format>

② 常用<format>:

```

+%d%m%y          030105
+%d-%m-%Y        03-01-2005
+%R              16:34
+%T              16:34:50
+%a              Mon
+%A              Monday
+%b              Jan
+%B              January
+%D              01/03/05
+%r              04:41:28 PM
+%x              01/03/2005

```

③ 举例: date +%d/%m/%Y" "%R // 打印 19/02/2005 21:12

- 2) 在文件名中含有日期的一个简单办法就是使用置换。把含有日期格式的变量附加在相应的日志文件名后面即可。

eg: 脚本代码示例:

```

MYDATE=`date +%d%m%y`
LOGFILE=/tmp/backup_log.$MYDATE
# create the file
>$LOGFILE

```

## 3.使临时文件的文件名具有唯一性: 使用环境变量\$\$

环境变量\$\$中保存有所运行的当前进程的进程号。因为脚本在运行时的进程号是唯一的，如果运行的脚本中需要创建一个临时文件，我们只要创建一个文件并在后面附加上\$\$即可保证临时文件的文件名具有唯一性(类上)。

## 4.信号

- 1) 信号:信号就是系统向脚本或命令发出的消息，告知它们某个事件的发生。信号可以用数字代表。
- 2) 最常用的信号及它们的含义:

signum	sigspec	meanings
1	SIGHUP	挂起或父进程被杀死
2	SIGINT	来自键盘的中断信号，通常是<CTRL-C>



3	SIGQUIT	从键盘退出，通常是<CTRL-D>
9	SIGKILL	无条件终止
11	SIGSEGV	段(内存)冲突
15	SIGTERM	软件终止(缺省杀进程信号)

3) 列出系统所有的信号: `kill -l` 或 `trap -l`

4) 杀死一个进程: `kill [-s sigspec | -n signum | -sigspec] <pid | job>`

用 `kill` 发送信号 1 将使一个进程重新读入配置文件，`kill` 缺省发送信号 15。

eg: `kill -1 httpd` // 使 `httpd` 进程从新读取配置文件

## 5.信号捕获

1) 有些信号可以被应用程序或脚本捕获，并依据该信号采取相应的行动。另外一些信号不能被捕获。

2) 在编写 shell 脚本时，一般只需关心信号 1、2、3 和 15。

3) 当脚本捕捉到一个信号后，它可能会采取下面三种操作之一：

① 不采取任何行动，由系统来进行处理。

② 捕获该信号，但忽略它。

③ 捕获该信号，并采取相应的行动。

第一种处理方式是默认方式，如果想要采取另外两种处理方法，必须使用 `trap` 命令。

## 6.trap 在脚本中捕捉信号

1) trap Usage: `trap [arg] [signal_spec ...]`

① `arg` 是捕捉到信号以后所采取的一系列操作，可以是一个函数或命令行，`arg` 要用双引号引起来,也可以用单引号。

② `signal_spec ...` 是信号列表(空格分割)，可以是 `signum` 或 `sigspec`

2) 一些最常见的 `trap` 命令用法:

`trap "" 2 3` // 忽略信号 2 和信号 3，用户不能终止该脚本

`trap "commands" 2 3` // 如果捕捉到信号 2 或 3，就执行相应的 `commands` 命令

`trap 2 3` // 复位信号 2 和 3，用户可以终止该脚本

3) 举例 1: `trap "echo the terminal stays locked" 2 3 15`

`trap` 命令捕捉信号 2、3 和 15。如果一个用户试图中断该脚本的运行，将会得到一个不成功的提示。

举例 2: `trap "" 1 2 3 15` 和 `trap 1 2 3 15`

在编写脚本时，如果希望在脚本运行的某些关键时刻脚本运行不被中断，通过设置 `trap` 来屏蔽某些信号就可以解决这个问题。在这些关键性的处理过程结束后，再重新打开信号。

4) 当脚本捕捉到信号时，通过使用 `trap` 命令，可以更好地控制脚本的运行。捕获信号并进行处理是一个脚本健壮性的标志。

## 7.eval

1) eval 命令将会首先扫描命令行进行所有的变量置换，然后再执行该命令。该命令适用于那些一次扫描无法实现其功能的变量。该命令对变量进行两次扫描。

2) 举例: `cmdline="cat /etc/password"; eval $cmdline` // 显示结果与 `cat /etc/password` 等效。

## 8.logger

1) 当希望在系统全局的日志文件 `messages(/var/log 或 /var/adm 目录下)` 中记录信息的时候，`logger` 命令是一个非常好的工具。

2) 用户可能会出于下列的原因向该全局日志文件 `messages` 中发送消息:

- ① 在某一个特定的时间段出现的访问或登录。
- ② 你的某些执行关键任务的脚本运行失败。
- ③ 监控脚本的报告。

3) `logger` Usage: `logger [-p pri] [-i] message`

- ① `-p pri` 指定消息 `message` 的优先级，缺省值为提示用户注意的优先级。
- ② `-i` 在每个消息中记录发送消息的进程号。
- ③ `message` 欲发送的消息字符串，使用双引号，其中可以使用变量替换。

eg: `logger -p notice "This is a test message"` // 此语句将附加到文件 `/var/log/messages` 尾

4) 向日志文件中发送信息的一个更为合理的用途就是用于脚本非正常退出时。如果希望向日志文件中发送消息，只要在捕获信号的退出函数中包含 `logger` 命令即可。

## 第 27 章 几个脚本例子

1.在对系统进行某些更新时，你可能不希望用户登录，这时可以使用 `/etc/nologin` 文件，大多数系统都提供这个文件。一旦在 `/etc` 目录中使用 `touch` 命令创建了一个名为 `nologin` 的文件，除 `root` 以外的任何用户都将无法登录。记住，该文件要对所有用户可读。当决定恢复用户登录时，只要删除该文件即可。

## 第 28 章 运行级别脚本

### 1.运行级别脚本

1) 如果希望在系统启动时自动运行某些应用程序、服务或脚本，或者在系统关机时能够正确地关闭这些程序，那么需要创建运行级别脚本。外，绝大多数 LINUX 发行版都含有这种基于 `System V` 的运行级别配置目录。

2) 如果需要在某一个特定的运行级别启动或停止程序，就得创建运行级别脚本(它们通常被称为 `rc` 脚本)

3) 任何用关键字 start 或 stop 调用的、能够启动或停止程序运行的脚本都可以看作是一个 rc 脚本。注意，应当由用户来保证他或她所提交的脚本是一个有效的脚本，能够正确地启动或停止某一服务。

4) 运行级别配置目录的机制使得 rc 脚本只在系统切换运行级别时有效。它不负责检查某一运行级别中所有的特定服务是否都已经被启动或停止。这是 shell 编程者的事。

## 2.运行级别目录

1) Linux 的运行级别目录: /etc/rc.d 和 /etc/rc.d/rcX.d (X=0-6)

2) 确定 Linux 当前的运行级别: runlevel

3) 运行级别目录中含有一系列启动服务的脚本。这里的“服务”可以是守护进程、应用程序、服务器、子系统或脚本进程。

## 3.inittab 脚本

1) 在系统启动的过程中，将会启动一个名为 init 的进程(它是系统中所有进程的祖先)。它所完成的一部分工作就是看看需要启动哪些服务，应当缺省地进入哪一个运行级别。它通过查看配置文件/etc/inittab 来获得上述信息的。init 进程还按照该文件中的设置加载特定的进程。

2) inittab 文件所包含的域具有严格的格式。该文件中每个条目的格式为:

id:rstart:action:process

① id 域是相应进程的唯一标识。rstart 域所包含的数字表示运行该进程的级别。

② action 域告诉 init 进程如何对待 process 所对应的进程。这里可以有很多种动作，但是最常见的是 wait 和 respawn。wait 意味着当进程启动后等待它结束。respawn 则意味着如果该进程不存在，则启动相应的进程，如果它存在，那么只要它一掉下来就立即重新启动它。

③ process 域包含了实际要运行的命令

3) inittab 脚本的行内容举例:

l3:3:wait:/etc/rc.d/rc 3

1:2345:respawn:/sbin/mingetty tty1

x:5:respawn:/etc/X11/prefdm -nodaemon

## 4.运行级别

1) init 进程在系统完全就绪之前所做的最后几项工作之一就是执行缺省运行级别所包含的所有脚本。该进程是通过/etc/rc.d/rc(或/etc/rc.init)来启动这些脚本的。它的作用是首先杀死该运行级别所包含的进程再启动这些进程。

2) rc 脚本将会使用 for 循环来依次查看相应运行级别目录中的文件，给每一个链接名以 K 开头的相应脚本赋予参数 stop；给每一个链接名以 S 开头的相应脚本赋予参数 start。在运行级别切换时，rc 脚本也会完成同样的工作，只不过根据相应的运行级别来启动或停止对应的脚本。

3) /etc/rc.d/rcX.d 目录中的脚本只是一些链接，真正的脚本保存在目录/etc/rc.d/init.d 中。在这个目录中含有一些能够启动或停止某一服务的脚本(可称为 rc 脚本)。

4) 一般来说，rc 脚本都应当能够接受参数 start 和 stop。可选的参数包括 restart 和 status。

5) 各种运行级别(run level)

运行级别 0 : 启动和停止整个系统

运行级别 1 : 单用户或管理模式

运行级别 2 : 多用户模式；部分网络服务被启动。有些系统将其作为正常运行模式，而不是级别 3

运行级别 3 : 正常操作运行模式，启动所有的网络服务

运行级别 4 : 用户定义的模式，可以使用该级别来定制所需要运行的服务

运行级别 5 : 有些 UNIX 系统变体将其作为缺省 X-windows 模式，还有些系统把它作为系统维护模式

运行级别 6 : 重新启动

6) 运行级别脚本名(链接)的格式

脚本名的格式: SnnScript\_name 或 KnnScript\_name

① 链接脚本名实际是在 rc 脚本的名称前加上 Snn 或 Knn。

② S 代表启动相应的进程,K 代表杀死相应的进程。

③ nn: 是 00 到 99 的两位数字，同一个 rc 脚本在不同的 rcX.d 目录中的链接名应采用相同的数字。

④ 当 init 进程调用相应的运行级别脚本时，杀进程按照从高到低的 K 序号进行。而启动进程按照从低到高的序号进行。如果使用的是 LINUX 系统，K 序号将按照从高到低的顺序执行。

5.安装自己的运行级别脚本的步骤

1) 编写该 rc 脚本，确保它符合调用标准(脚本要接受 start 和 stop 参数)。

2) 确信它能够启动或终止相应的服务。

3) 将该脚本放置于(取决于操作系统)/etc/init.d 或/usr/sbin/init.d 或/etc/rc.d 中。

4) 在相应的 rcX.d 目录中按照合理的命名方式创建链接。

6.启动和停止服务的其他方法(如果系统不支持运行级别的话)

1) 在 inittab 文件尾加入相应的条目。如果希望用户的 rc 脚本在系统刚刚就绪之后运行，使用 inittab 是很好的方式。

2) 大多数系统都含有文件/etc/rc.d/rc.local(或在其他目录下)，该脚本文件将在 inittab 和运行级别脚本之后运行。可以在该文件中加入任何命令，或从中调用最习惯用的启动脚本。

3) 有些系统还在/bin 目录下(更多的是在/usr/sbin 目录下)含有一个名为 shutdown 的脚本文件。可以使用它来关闭某些服务。

## 第 29 章 cgi 脚本

1.本章讲授使用 shell 脚本编写 cgi 程序，此技术基本没有什么价值，省略。