about linux command line

一、基础命令

此部分主要与文件相关

1.1 查看文件/目录

1.1.1 查看目录

ls

> 显示文件(ls xxx/xxx/)
>
> - **-a** 显示全部文件
> - **-l** 显示详细信息

1.1.2 查看文件

cat

查看文件内容

tail

> 直接查看文件末尾 (tail dir_file)
>
> - **-n** 指定行数　tail -n num dir_file
> - **-f** 实时更新

head

查看文件开头若干行(与tail类似)

...可以用上述功能合并文件内容为一个新文件

下面的 * 为通配符，后面会介绍

```
$  cat may*.txt > combined.txt
```

less/more

翻页查看文件

1.2 创建文件/目录

1.2.1 创建目录

mkdir

> 创建目录
>
> - **-p** (parent)递归创建　　(mkdir /xxx/xxx/xxx)
> - **-m** 添加权限　(mkdir -m 755 xxx)
> - **‘ ’** 转义　　(mkdir xxx/‘folder 1’) (文件夹名称中有空格)

1.2.2 创建文件

touch

创建文件

ln

> 创建链接文件 (ln /xxx/xxx /xxx/xxx_link_file)
>
> - **-s** 软链接文件 链接文件与原文件inode编号不同，大小不同
> - **无参** 硬链接文件 链接文件与原文件指向同一位置，大小相同，inode号相同，需要在统一存储设备中才能创建，不能跨文件系统

cp

> 备份文件
>
> - cp xxx1 xxx1_bak (复制xxx1并命名为xxx1_bak)

1.2.3 编辑文件

vim / vi

编辑/创建文件

1.3 移动文件/目录

mv

> 移动/重命名文件 (mv /src /dir)
>
> - 重命名后时间戳 and inode号不变
> - mv xxx1 xxx2 (将xxx1在当前文件夹下重命名为xxx2)
> - mv xxx1 xxx2 xxxx* dir1 (将xxx1,xxx2,xxxx开头的文件都移动至dir1下)

1.4 删除文件/目录

rm

> 删除文件
>
> - 注意！！！ rm的删除彻底且不可撤销
> - **-r** 递归删除，包括目录内的文件和文件夹 （rm -r folder1）
> - **-i** 交互选项，删除文件需要确认 y删除，n保留

rmdir

> 删除文件夹
>
> - **-p** 同时删除父目录 (rmdir -p folder2)

1.5 其他

chmod

> 修改文件权限

权限项	文件类型	读	写	执行	读	写	执行	读	写	执行
字符表示	（d|l|c|s|p）	(r)	(w)	(x)	(r)	(w)	(x)	(r)	(w)	(x)
数字表示		4	2	1	4	2	1	4	2	1
权限分配		文件所有者			文件所属组用户			其他用户		


```
$ chmod 777 xxx
$ chmod u+x xxx
```

whoami

当前用户名

echo

输出参数/内容

*****

通配符，表示零个或多个字符，(cat may*) (输出所有文件名开头为"may"的文件中内容)

?

表示文件名中的单个字符，(cat may?.txt)

>

重定向　(cat xxx.txt > xxx1.txt) (注意！重定向会覆盖文件原内容，追加写应当使用 >>)

>>

追加

sort

> 按首字母（或其他）对文件内行进行排序
>
> - -b 忽略每行前面开始出的空格字符。
> - -c 检查文件是否已经按照顺序排序。
> - -d 排序时，处理英文字母、数字及空格字符外，忽略其他的字符。
> - -f 排序时，将小写字母视为大写字母。
> - -i 排序时，除了040至176之间的ASCII字符外，忽略其他的字符。
> - -m 将几个排序好的文件进行合并。
> - -M 将前面3个字母依照月份的缩写进行排序。
> - -n 依照数值的大小排序。
> - -u 意味着是唯一的(unique)，输出的结果是去完重了的。
> - -o<输出文件> 将排序后的结果存入指定的文件。
> - -r 以相反的顺序来排序。
> - -t<分隔字符> 指定排序时所用的栏位分隔字符。
> - +<起始栏位>-<结束栏位> 以指定的栏位来排序，范围由起始栏位到结束栏位的前一栏位。
> - --help 显示帮助。
> - --version 显示版本信息。
> - [-k field1[,field2]] 按指定的列进行排序。

wc

> (word count) 文件字数统计
>
> - wc test.txt
> - 13 16 128 (text.txt 有13行，16个单词，128字节)
> - -l 仅查看行数

| 管道符

uniq

> 删除文件内重复行

> - -c或--count 在每列旁边显示该行重复出现的次数。
> - -d或--repeated 仅显示重复出现的行列。
> - -f<栏位>或--skip-fields=<栏位> 忽略比较指定的栏位。
> - -s<字符位置>或--skip-chars=<字符位置> 忽略比较指定的字符。
> - -u或--unique 仅显示出一次的行列。
> - -w<字符位置>或--check-chars=<字符位置> 指定要比较的字符。
> - --help 显示帮助。
> - --version 显示版本信息。
> - [输入文件] 指定已排序好的文本文件。如果不指定此项，则从标准读取数据；
> - [输出文件] 指定输出的文件。如果不指定此选项，则将内容显示到标准输出设备（显示终端）。

二、监测进程

ps

显示当前运行的进程(某一时刻)

可选项众多，需要掌握一组常用的，能够输出进程信息的选项

top

实时输出当前的进程信息

强大的工具，可排序

> f, q, d...

进程的信号是预定义的消息，进程通过编程实现识别这些信号，决定作出反应或者是忽略。如何处理通过编程决定)

kill

通过pid向进程发送信号　　(kill 3940)

pkill

使用程序名代替pid来终止进程(允许使用通配符)

三、挂载存储设备

mount

输出当前系统已挂载的设备列表

> - -t 限制某些输出?

umount

卸载可移动设备(移除可移动设备时，不能直接拔下，需要先卸载)

有程序正在使用设备上文件的话，不允许卸载

df

> -h 查看磁盘空间

du

> 显示特定目录的磁盘使用情况
>
> - **-c** 显示所有已列出文件的总大小
> - **-h** 按人类方便阅读的格式输出大小
> - **-s** 输出每个参数的汇总信息

四、处理数据文件

sort

> 用来对输出的数据进行排序的命令
>
> - **-n** 将数字按值排序(默认情况下会按ascll码，当作字符来排序)
> - **-M** 将含有时间戳日期的文件 中的数字 按月排序
> - **-k** 对按字段分隔的数据进行排序 (sort -t ':' -k 3 -n /etc/passwd)
> - 按第三个字段(用户的id)排序
> - **-n** 用于排序数值型输出

grep

> 数据搜索
>
> - grep hello file.txt　/　grep hello dir/file/
> - **-i** 忽略大小写
> - **-v** 反向查找，打印不匹配的行
> - **-n** 显示行号
> - **-r** 递归查找子目录中文件
> - **-l** 只打印匹配的文件名
> - **-c** 只打印匹配的行数

gzip

压缩文件

tar

> 用于将文件写入磁带设备以作归档，也可以将输出写入文件
>
> - tar function [option] object1 object2

> **关于function:**
>
> - **-A** 将一个tar归档文件追加到另一个tar归档文件末尾
> - **-c** 创建新的tar归档文件
> - **-d** 检查归档文件和文件系统的不同之处 or 从tar归档文件中删除文件
> - **-r** 将文件追加到tar归档文件末尾
> - **-t** 列出tar归档文件的内容
> - **-u** 将比tar归档文件中已有的同名文更新的文件追加到该归档文件
> - **-x** 从tar归档文件中提取文件

> **关于option**
>
> - **-C** 切换到指定目录
> - **-f** 将结果输出到文件(设备)
> - **-j** 将输出传给bzip2命令进行压缩
> - **-J** 将输出传给xz命令进行压缩
> - **-p** 保留文件的所有权限
> - **-v** 在处理文件时显示文件名
> - **-z** 将输出传给gzip命令进行压缩
> - **-Z** 将输出传给compress命令进行压缩

这些选项常常组合使用

五、一些小东西...?

5.1 关于 shell

5.1.1 查看 shell

shell的类型

bash rbash dash ...

5.1.2 显示 shell 的类型

```
$ which bash
```

输出当前shell的bash类型

多数的/bin/bash其实是链接，指向/usr/bin/bash

```
$ cat /ect/shells
```

输出此系统中包含的shell类型

```
$ echo $0
```

显示当前shell　　(仅限在当前shell中使用，在脚本中将输出脚本名称)

5.2 操作 shell

5.2.1 启动

bash

在CLI中输出 bash 将启动一个子shell， 子shell同样等待命令输入。可以通过 ps -f 查看

在 ps -f 中，可以看到子shell的**PPID**是父shell的**PID**

还有 **ps --forest** 可以查看(如果创建了多个子shell)

5.2.2 退出

exit

使用exit退出子shell

在一行中使用多个； 来分隔命令

```
$ pwd ; cd ; pwd ; cd wish/shell ; pwd ; ls *.txt
```

所有命令依次执行

若使用 () 的话，将启动一个子进程来实现命令　(pwd ; cd ; pwd ; cd wish/shell ; pwd ; ls *.txt)

使用 echo $BASH_SUBSHELL 来查看生成了多少个子进程

需要注意，生成子shell需要占用更多的内存，如内存和处理能力，明显拖慢任务进度

可以将子shell丢入后台模式..?好像也好不到哪去(后台的好处是，可以让父子shell不再占用一个CLI)

5.2.3 睡眠

sleep

```
sleep 10 使当前进程睡眠10秒
sleep 10 & 使当前进程在后台睡眠10秒
```

5.2.4 后台 shell 与查看后台

jobs

查看当前所有的后台进程　[进程号] - / + 的正负号表示子进程的创建顺序，最近启动的为+

后台作业完成后会显示出结束状态

使用后台模式来**tar创建备份文件**是一个实用的例子

coproc

协程　　做了两件事：在后台生成一个子shell，在子shell中执行命令（coproc sleep 10)(创建一个后台进程并睡眠10秒)

取名：

```
$ coproc My_job { sleep 10; }
```

需要注意格式，{}的开始与结束都需要空格，命令行语句以分号结尾

内建命令与外部命令：外部命令更耗空间，which只能识别外部命令

外部命令执行时生成了一个子进程，而内部不需要

5.3 其他

history

查看历史命令

历史命令保存在主用户文件夹下的.**bash_history**中

alias

命令别名

命令别名仅在其被定义的shell中才能实用(父进程中创建，子进程无法使用)

关于数组

mynums=(zero one two three dmdasha) 数组

```
$ echo $mynums 仅会输出zero 数组第一个值
$ echo ${mynums[2]}       指定数组输出值需要引用，且加花括号
$ echo ${mynums[*]}  要显示整个数组变量，可以使用通配符 *
```

mynums[2]=dmnice　修改数组单个值

unset mynums[1]　　删除单个数组值。删除后，输出整个数组变量时会自动补齐，但单独访问mynums[1]时会发现此处为空

数组下标从0开始

数组在编程中并不常用

六、环境变量

环境变量分为局部变量和全局变量

为了区分，大部分**全局变量**都为**大写**，**局部变量**都为**小写**

全局变量对所有shell会话和所有生成的子shell都可见，局部变量仅对创建的shell可见

6.1 查看环境变量

printenv / env

查看全局变量

也可以使用 **echo** 来显示变量名

set

查看所有的变量

6.2 操作环境变量

My_variable=Hello

自定义局部变量(变量名，等号，值之间没有空格)

echo My_variable

My_variable="hello world" 使用空格时需要加引号

export

设置全局变量(在局部变量已存在的基础上)

```
$ export My_variable    (此时My_variable已创建)
$ export My_var=hello   (在创建时声明为全局变量)
```

修改子shell中的值不会影响父shell中的值，是否声明为全局都一样

unset

删除环境变量 (unset My_var)(不需要$)

tips:如果**需要使用环境变量(作参数，引用...)**，则**加上$**, 若需要**操作环境变量(创建，修改值，删除...)**，则**不需要$**

在子shell中删除变量后，依然无法对父shell中的变量产生影响

bash shell中有许多的默认环境变量

6.3 设置PATH环境变量

echo $PATH将输出一些目录，shell会在其中查找命令和程序

有些脚本编写人员会在脚本中使用 **#!/usr/bin/env bash** 这样的优点在于env会在$PATH中搜索bash，拥有更好的移植性

将新的搜索目录添加到现有的$PATH中，无须从头定义

```
$ PATH=$PATH:/home/Sobloom/wish/shell    (将脚本所在目录加入PATH中)
```

若希望子shell也可以使用，需要将修改后的PATH导出

登录linux系统时，bash shell会作为登录shell启动，通常会从5个不同的启动文件中读取命令，/etc/profile是主启动文件，另外四个文件是针对用户的，位于用户的主目录中，**$HOME/.bash_profile**, **$HOME/.bash_login**, **$HOME/.profile**, **$HOME/.bashrc**

若不是登录时启动的bash shell，则此时shell称为交互式shell (也就是 ctrl +alt + t 打开的那个东西吧)

系统执行脚本时，用的是非交互式shell，不同之处在于它没有命令行提示符

大多数发行版中，保存用户个人永久性bash shell变量的地点是**$HOME/.bashrc**文件。alias命令设置无法持久生效，可以将个人的alias放入$HOME/.bashrc启动文件中，效果永久化

七、linux文件权限

linux安全系统的核心是用户账户。用户对系统中各种对象的访问权限取决于他们登录系统时所用的账户

用户权限是通过创建用户时分配的用户ID来跟踪的。UID唯一

7.1 系统内的用户信息

/etc/passwd

文件来匹配登录名与对应的UID值，root账户为0，ubuntu系统普通账户id从1000开始

文件中包含了：登录用户名 用户密码 用户账户的UID 用户账户的组ID 用户账户的文本描述(备注) 用户$HOME目录的位置 用户默认shell

/etc/shadow

密码系统，更好的控制用户密码。内部包含了上次修改密码的时间，多久后可修改，多久后必须修改...

7.2 操作用户

useradd

添加新用户

> -D 显示默认值

表 7-1 useradd 命令行选项

选　项	描　述
-c *comment*	给新用户添加备注
-d *home_dir*	为主目录指定一个名字（如果不想用登录名作为主目录名的话）
-e *expire_date*	用 YYYY-MM-DD 格式指定账户过期日期
-f *inactive_days*	指定账户密码过期多少天后禁用该账户；0 表示密码一过期就立即禁用，-1 表示不使用这个功能
-g *initial_group*	指定用户登录组的 GID 或组名
-G *group ...*	指定除登录组之外用户所属的一个或多个附加组
-k	必须和-m 一起使用，将/etc/skel 目录的内容复制到用户的$HOME 目录
-m	创建用户的$HOME 目录
-M	不创建用户的$HOME 目录，即便默认设置里要求创建
-n	创建一个与用户登录名同名的新组
-r	创建系统账户
-p *passwd*	为用户账户指定默认密码
-s *shell*	指定默认的登录 shell
-u *uid*	为账户指定一个唯一的 UID

表 7-2 useradd 修改系统默认值

选　项	描　述
-b *default_home*	修改用户$HOME 目录默认创建的位置
-e *expiration_date*	修改新账户的默认过期日期
-f *inactive*	修改从密码过期到账户被禁用的默认天数
-g *group*	修改默认的组名称或 GID
-s *shell*	修改默认的登录 shell

```
$ useradd -D -s /bin/tcsh    修改默认值
```

userdel

删除用户

默认状态下，只删除 /etc/passwd 和 /etc/shadow 文件中的用户信息，属于账户的文件会保留

-r 会删除用户的 $HOME 目录以及邮件目录。but，系统中仍可能存有已删除用户的其他文件。在大量用户环境中，一个用户的$HOME目录下可能存放了其他用户或程序的重要文件，删除前需要检查清楚

usermod

> 修改用户账户
>
> - **-c** 修改备注字段
> - **-e** 修改过期日期
> - **-g** 修改默认登陆组
> - **-l** 修改登录名
> - **-L** 锁定账户，禁止登录
> - **-p** 修改密码
> - **-U** 解除锁定，允许登录

passwd

> 修改用户密码
>
> - 普通用户仅能修改自己的，root才能修改别人的

chsh

修改登录shell

chfn

将Unix的finger命令的信息存入备注字段

chage

修改账户有效期

> - **-E** 设置密码过期日期
> - **-I** 设置密码过期多少天后锁定账户
> - **-m** 修改密码间隔最小天数
> - **-M** 修改密码间隔最大天数
> - **-W** 密码过期多久前提醒用户

7.3 修改用户组

/etc/group

文件保存了linux组的信息

GID：系统的小于500，用户的从500开始

groupadd

添加新组

usermod 向组中添加成员

groupmod

修改已有组的GID(-g)或组名(-n)

```
sobloom@sobloomdm:~$ ls -l
total 40
drwxr-xr-x 2 sobloom sobloom 4096 10月  23 18:23 Desktop
drwxr-xr-x 7 sobloom sobloom 4096 10月  23 18:56 Documents
drwxr-xr-x 2 sobloom sobloom 4096 10月  23 19:55 Downloads
drwxr-xr-x 2 sobloom sobloom 4096 10月  23 18:23 Music
drwxr-xr-x 2 sobloom sobloom 4096 10月  23 18:23 Pictures
drwxr-xr-x 2 sobloom sobloom 4096 10月  23 18:23 Public
drwx------ 7 sobloom sobloom 4096 12月   5 16:35 snap
drwxr-xr-x 2 sobloom sobloom 4096 10月  23 18:23 Templates
drwxr-xr-x 2 sobloom sobloom 4096 10月  23 18:23 Videos
drwxrwxr-x 4 sobloom sobloom 4096  1月  25 20:13 wish
```

修改组名时，仅组名发生变化，GID 和 组成员不变。所有的安全权限均基于GID

> 列出的文件首个字符代表

> - **-** 代表文件
> - **d** 代表目录
> - **l** 代表链接
> - **c** 字符设备
> - **d** 块设备
> - **p** 具名管道
> - **s** 网络套接字

属主，属组，其他用户

7.4 修改权限

umask

配置默认文件权限

chmod

修改文件和目录的安全设置

作用对象 作用操作 作用权限

u 用户 g 组 o 其他用户 a 以上所有

+添加权限 - 移除权限 = 设置权限

r, w, x

chown/chgrp

修改所属关系 (属主，属组)

```
$ chmod options owner/group file
$ chown Sobloom withyou.sh  修改withyou.sh文件属主为SoBloom
$ chown Sobloom.dm withyou.sh   修改属主和属组(dm)
```

只有root用户能修改文件的属主，任何用户都可以修改属组，但用户必须为原属组或新属组成员

7.5 共享文件

linux每个文件有三个额外信息位：SUID, SGID, 粘滞位

首先，mkdir创建希望共享的目录，chgrp修改目录的默认属组，使其包含所有需要共享的用户。最后，设置目录的SGID位，保证目录中的新建文件都以shared作为默认属组

访问控制列表：ACL

setfacl

getfacl

八、管理文件系统

8.1 发展历史

最初为**文件扩展系统(ext)**，后经版本迭代至**ext4**

从**ext2**开始，出现了日志文件系统。先将文件变更写入临时文件，数据被成功写到存储设备和i节点表之后再删除日志目录，避免了在数据写入前断电或崩溃导致的数据写入失败。(若发生以上情况，日志文件系统可以在日志中读取，处理尚未提交的数据)

数据模式：i节点和文件数据都会写入日志。数据丢失风险低，性能差

有序模式：只有i节点数据会被写入日志，直到文件数据被成功写入后才会将其删除。性能和安全性间折中

回写模式：只有i节点数据会被写入日志，但不控制文件数据何时写入。数据丢失风险高，一般不用于日志

日志技术的替代选择：写时复制(COW, copy-on-write)，通过快照兼顾安全性和性能

ZFS：稳定的文件系统，拥有数据完整性和自动修复功能

8.2 磁盘管理

创建磁盘分区：fdisk, gdisk, GNU parted

SATA驱动器和SCSI驱动器：设备命名 /dev/sd**x**， 第一个检测到的驱动器为a, 第二个为b... 用以替代x

SSD NVMe 驱动器：设备命名为/dev/nvme**N**n**#**，N根据驱动器的检测顺序决定(从0起始)。#是分配给该驱动器的名称空间变化(从1起始)

IDE驱动器：设备命名/dev/hd**x**，第一个检测到的驱动器为a, 第二个为b... 用以替代x

fdisk

在任何存储设备上创建和管理分区

最大处理2TB的硬盘

fdisk是一个交互式程序，输入命令逐步完成硬盘分区操作，需要root权限

若存储设备首次分区，fdisk会警告该设备无分区表

不允许调整现有分区大小，只能删除现有分区后重新创建

gdisk

存储设备要采用GUID分区表(GPT)进行分区

gdisk 会识别存储设备采用的分区类型，若未使用GPT方法，gdisk会提供相应选项，然后转换为GPT

提供命令行提示符来输入命令进行分区操作

GNU parted

允许调整现有分区大小，容易扩大或收缩磁盘分区

逻辑卷管理器 (logical volume manager， LVM)

LVM将多个分区组合在一起，作为单个分区(逻辑卷)进行格式化，在linux虚拟目录结构上挂载、存储数据

物理卷：PV,使用pvcreate命令。指定一个未使用的磁盘分区由LVM使用，在此过程中，LVM结构、卷标和元数据都会添加到该分区

卷组：VG，使用vgcreate命令创建，将PV加入存储池。使用vgcreate将一个或多个PV加入VG时，同时会添加卷组的元数据。被指定为PV的分区只能属于单个VG，被指定为PV的其他分区可以属于其他VG

逻辑卷：LV，使用lvcreate命令创建。LV由VG的存储空间块组成

创建好LV，可以将其视作普通分区，不同之处在于可以根据需要扩大或收缩此分区。但在此之前，必须将LV挂载到虚拟目录结构中。

九、软件安装

rpm 和 apt

十、文本编辑器Vim

Vim编辑器有三种模式：命令模式，Ex模式，插入模式

h,j,k,l：　左下上右

ctrl + F　下翻一屏

ctrl + B　上翻一屏

G　移到缓冲区最后一行

num G　移到缓冲区第num行

gg　移到缓冲区第一行

q　未对文件内容作更改，退出

q!　取消对文件内容的修改，强制退出

w filename　将文件另存为其他名称

wq　将缓冲区数据保存到文件中并退出

x　删除光标当前所在位置的字符

dd　删除光标当前所在行

dw　删除光标当前所在位置的单词

d$　删除光标当前所在位置至行尾的内容

J　删除光标当前所在行结尾的换行符(合并行)

u　撤销

a　在光标当前位置后追加数据

A　在光标当前所在行尾追加数据

r char　用char替代光标当前所在位置的单个字符

R text　用text覆盖光标当前所在位置的内容，直到按下ESC键

有些编辑命令允许使用数字修饰符来指定重复该命令多少次。命令2x会从光标当前位置开始删除两个字符，命令5dd会删除从光标当前所在行开始的5行

p　粘贴

按下v 移动光标(用上下左右移动)　选中复制区域

y　复制

:s/old/new/g 替换当前行内出现的所有old

:n,ms/old/new/g　替换第n行和第m行之间出现的所有old

:%s/old/new/g　替换整个文件中出现的所有old

:%s/old/new/gc　替换整个文件中出现的所有old，每次替换时提示

十一、shell基础脚本构建

11.1 shell 基础

11.1.1 shell 的运行

第一行:

```
#!/bin/bash
or
#!/usr/bin/env bash
```

第一行的 # 用来告诉shell用哪个shell运行脚本

其他行的 # 用来作注释

创建了脚本后，需要给脚本添加可执行权限

```
$ chmod a+x ./script.sh
```

11.1.2 输出内容

echo "words"　echo输出字符串可以不加引号，但是字符串中包含''(单引号)时需要添加

想要 **echo** 输出的内容和命令输出的内容可以使用 -n 选项

```
$ echo -n "hey, who are you, my owner??   "; whoami
hey, who are you, my owner??   Sobloom
```

echo 输出 $ 符号时，需要加\来进行转义

```
$ echo "the cost is \\$15"
```

11.1.3 shell 中的变量

shell脚本中依然可以引用shell中的环境变量

用户自定义变量

Var_name=content　　不带空格，大小写敏感

shell脚本会以字符串的形式存储所有的变量值，脚本的生命周期中会一直保存，结束后删除

用户变量可以通过 **$** 引用

```
$ echo "Var_name is $Var_name"
```

命令替换

将shell命令的输出赋给变量

```
date1=$(date)

date2=$(date +%y%m%d)    (250304)
```

11.1.4 重定向输入和输出

输出重定向

command > *outputfile*

重定向会将命令内容重定向至文件(若没有文件则创建文件)，重定向会覆盖文件的原内容

追加写

command >> *outputfile*

输入重定向

command < *inputfile*　　(wc < test6.txt)

将文件内容定向至command命令作为输入

内联重定向

command << *marker*

举例:

```
$ wc << EOF
< data1 write something here
< data2 emmm
< data3 dm is a pretty shab
< EOF
3   12  65
```

次提示符 **<** 会持续显示，获取更多的输入数据，直到输入了作为文本标记的那个字符串

管道

将左侧命令的输出作为右侧命令的输入

两侧命令同时执行

多用在大量输出传送给 **more** 命令

```
$ ls -l | more
```

11.2 数学运算

expr

这是个运算符

var=$(expr 5 + 2)

表 11-1　expr 命令运算符

运　算　符	描　　述
ARG1 | ARG2	如果 ARG1 既不为 null 也不为 0，就返回 ARG1；否则，返回 ARG2
ARG1 & ARG2	如果 ARG1 和 ARG2 都不为 null 或 0，就返回 ARG1；否则，返回 0
ARG1 < ARG2	如果 ARG1 小于 ARG2，就返回 1；否则，返回 0
ARG1 <= ARG2	如果 ARG1 小于或等于 ARG2，就返回 1；否则，返回 0
ARG1 = ARG2	如果 ARG1 等于 ARG2，就返回 1；否则，返回 0
ARG1 != ARG2	如果 ARG1 不等于 ARG2，就返回 1；否则，返回 0
ARG1 >= ARG2	如果 ARG1 大于或等于 ARG2，就返回 1；否则，返回 0
ARG1 > ARG2	如果 ARG1 大于 ARG2，就返回 1；否则，返回 0
ARG1 + ARG2	返回 ARG1 和 ARG2 之和
ARG1 - ARG2	返回 ARG1 和 ARG2 之差
ARG1 * ARG2	返回 ARG1 和 ARG2 之积
ARG1 / ARG2	返回 ARG1 和 ARG2 之商
ARG1 % ARG2	返回 ARG1 和 ARG2 之余数
STRING : REGEXP	如果 REGEXP 模式匹配 STRING，就返回该模式匹配的内容
match STRING REGEXP	如果 REGEXP 模式匹配 STRING，就返回该模式匹配的内容
substr STRING POS LENGTH	返回起始位置为 POS（从 1 开始计数）、长度为 LENGTH 的子串
index STRING CHARS	返回 CHARS 在字符串 STRING 中所处的位置；否则，返回 0
length STRING	返回字符串 STRING 的长度
+ TOKEN	将 TOKEN 解释成字符串，即使 TOKEN 属于关键字
(EXPRESSION)	返回 EXPRESSION 的值

上述运算符部分需要加上 \ 进行转义

bash shell的数学运算符仅支持整数运算

[] 方括号

用法和expr差不多，但是不需要加转义符

var=$[5 * 2]

浮点数计算

bc

scale 表示小数位数，默认scale为0

scale=4表示四位小数

variable=$(echo "options; expression" | bc)

```
$ cat script
#!/usr/bin/env bash
var1=$(echo " scale=4; 6 / 5" | bc)
echo "var1 is $var1"
exit
```

bc可以接受输入重定向，在进行大量计算时可以选择

variable=$(bc << EOF
options
statement
expression
EOF
)

```
$ cat script_var
#!/usr/bin/env bash
var1=11.11
var2=12.12
var3=13.13
var4=14.14
var5=$(bc << EOF
scale=4
a1 = ($var1 * $var2)
a2 = ($var3 * $var4)
a1 + a2
EOF
)
echo $var5
exit
```

退出状态码

退出状态码时一个0~255的整数值，命令结束运行时传给shell，可以获取该值并在脚本中使用

变量： $?

> 存储着最后一个命令执行后的退出状态码
>
> 0　成功结束
>
> 1　一般未知性错误(可能是命令的参数不对)
>
> 2　不适合的shell命令
>
> 126　命令无法执行(无权限)
>
> 127　没找到命令

> 128 无效的退出参数
>
> 128+x 与linux信号x相关的严重错误
>
> 130 通过 ctrl + c 退出的命令
>
> 255 正常范围外的退出状态码

可以使用 **exit** 指定一个退出状态码，也可以使用变量作为exit的参数

退出状态码最大为255, 若大于255会对256进行取余后输出

十二、结构化命令

12.1 if 语句

12.1.1 if-then 语句

```
if command ; then
    commands
fi
```

bash shell 会运行**if语句之后的命令**，若命令**退出状态码为 0** (即为命令正常完成)， 位于then部分的**commands命令就会执行**，若为**其他值**，则**不会执行**

```
if pwd; then
    echo "worked"
fi
```

12.1.2 else的加入

```
if command1; then
    echo "command1 worked"
else
    echo "command1 not worked"
fi
```

12.1.3 嵌套的if-then

```
if command1; then
    echo "command1 worked"
    echo "oh yeah"
elif command2; then
    echo "command2 worked"
    echo "oh no"
fi
```

12.1.4 test 在if-then语句中测试不同的条件

若test命令中列出的条件成立，则test命令退出并返回状态码0

test condition　　condition为要测试的一系列参数和值

```
if test condition; then
    commands
fi
```

在bash shell中，可以使用

```
if [ condition ]; then
    commands
fi
```

方括号(注意空格)来替代test

> test 命令可以判断三类条件：数值比较，字符串比较，文件比较

表 12-1　test 命令的数值比较功能

比　较	描　述
n1 -eq *n2*	检查 *n1* 是否等于 *n2*
n1 -ge *n2*	检查 *n1* 是否大于或等于 *n2*
n1 -gt *n2*	检查 *n1* 是否大于 *n2*
n1 -le *n2*	检查 *n1* 是否小于或等于 *n2*
n1 -lt *n2*	检查 *n1* 是否小于 *n2*
n1 -ne *n2*	检查 *n1* 是否不等于 *n2*

表 12-2　test 命令的字符串比较功能

比　较	描　述
str1 = *str2*	检查 *str1* 是否和 *str2* 相同
str1 != *str2*	检查 *str1* 是否和 *str2* 不同
str1 < *str2*	检查 *str1* 是否小于 *str2*
str1 > *str2*	检查 *str1* 是否大于 *str2*
-n *str1*	检查 *str1* 的长度是否不为 0
-z *str1*	检查 *str1* 的长度是否为 0

大于小于号需要转义 \ ，否则会变成重定向

大于小于顺序和sort命令采用的不同

表 12-3　test 命令的文件比较功能

比　较	描　述
-d *file*	检查 *file* 是否存在且为目录
-e *file*	检查 *file* 是否存在
-f *file*	检查 *file* 是否存在且为文件
-r *file*	检查 *file* 是否存在且可读
-s *file*	检查 *file* 是否存在且非空
-w *file*	检查 *file* 是否存在且可写
-x *file*	检查 *file* 是否存在且可执行
-O *file*	检查 *file* 是否存在且属当前用户所有
-G *file*	检查 *file* 是否存在且默认组与当前用户相同
file1 -nt *file2*	检查 *file1* 是否比 *file2* 新
file1 -ot *file2*	检查 *file1* 是否比 *file2* 旧

-d　检测指定目录是否存在于系统中。若打算将文件写入目录或切换到目录，先测试一下较好

-e　可以使用-e 选项多次判断(目录和文件)

```
#!/usr/bin/env bash
location=$HOME
file_name="dm.txt"
if [ -e $location/$file_name ]; then
    echo "file exist"
fi
```

-f　是否存在且为文件

...

12.2 if 语句的其他表达

12.2.1 复合条件测试

[condition1] && [condition2]

[condition1] || [condition2]

12.2.2 高级特性

在子shell执行命令：使用单括号 **()**

高级数学表达式：双括号 **(())**

表 12-4　双括号命令符号

符　　号	描　　述
val++	后增
val--	后减
++val	先增
--val	先减
!	逻辑求反
~	位求反
**	幂运算
<<	左位移
>>	右位移
&	位布尔 AND
|	位布尔 OR
&&	逻辑 AND
||	逻辑 OR

模式匹配：双方括号 **[[]]**

在bash shell中运行良好，其他shell不一定支持

可以通过通配符和正则表达式来匹配字符串

```
if [[ %BASH_VERSION == 5.* ]]; then
    echo "BASH SHELL version > 5.0"
fi
```

12.3 case 命令

```
case $USER in
damo | best)
    echo ...
    echo ...;;
mud | may)
    echo ...;;
huh)
    echo ...;;
*)
    echo "default situations"
esac
```

注意每个case的结尾语句处有两个分号

十三、更多的结构化命令(for, while)

13.1 for 命令

```
for var in list; do
    commands
done
```

需要提供一系列的值作为迭代的list参数，指定这些值的方法不止一种

每次迭代，var 会包含列表中的当前值，用完为止

```
for var in aaa bbb ccc ddd eee fff; do
    echo "var is $var"
done
```

最后一次迭代结束后，$var 的值在 shell 脚本的剩余部分依然有用，保持最后一次迭代的值

若是迭代的值(也就是字符串)中有 '' 或者 **空格**、**tab**、**换行**，需要特殊处理(否则一个变量会被识别为两个变量)

'单引号　可以使用 \转义 或者 ” “ 将内容框起来

空格　　使用 "" 将内容框起来　tips: 用双引号将值框起来的时候，双引号不会作为值的一部分

注意引号是加在 for 循环的 $list[@] 两边的

```
list=("aaa ccc" "ddd" "fff" 233 ddd trtt")
#list="$list append"
#上句为原笔记中的写法，实际操作后发现 append 追加在了 aaa ccc 变量之后，变成了 aaa ccc append
#下句为其他写法，仅供参考
list+=("append")
#list+=("huh" "bab" "aoe")
for var in "$list[@]"; do
    commands
done
```

上述代码中，list=$list" append" 一句为向list列表中追加一个值append

```
file="AprilComes.txt"
for var in $(cat $file); do
    commands
done
```

注意，AprilComes.txt 文件中的每个值各占一行，而不是以空格分隔。这并没有解决数据中还有空格的情况。若是一行的内容中有空格，依然会当作多个值输出　　(举例：txt中第一行为"aaa vvv ddd"，脚本中的for循环会进行三次，依次获取aaa vvv ddd)

更改字段分隔符

特殊的环境变量 **IFS** 定义了bash shell用作字段分隔符的一系列字符。默认情况下，bash shell会将 **空格** **制表符** **换行符** 当作字段分隔符

```
IFS=$'\n'
```

使其只能识别换行符，忽略空格和制表符

若要遍历文件中以冒号分隔的值(如/etc/passwd)，将IFS值修改为冒号即可 IFS=:

```
 IFS=$'\n:;"'
```

指定多个值

使用通配符读取目录

```
for file in /home/Sobloom/wish/*; do
    if [ -d "$file" ]; then
        echo "$file is a directory"
    elif [ -f "$file" ]; then
        echo "$file is a file"
    fi
done
```

在linux中，目录名和文件名中可能会包含空格，所以要加上引号避免输出时发生错误(bash shell中，test会将额外的单词视为参数，引发错误)

可以在for命令中列出多个目录通配符

```
for file in /home/Sobloom/wish/* /home/Sobloom/code/*; do
```

C语言风格的for命令

```
for (( variable assignment ;  condition ;  iteration process ))
```

与标准的for命令不同的是，这里的变量赋值可以有空格，迭代条件的变量不以美元符号开头，迭代算式不适用expr命令格式

```
for (( i=1 ; i <= 10 ; i++ )); do
    echo "The next number is $i"
done
```

同样可以使用多个变量 for ((a=1, b=2 ; a <= 10 ; a++, b++))

13.2 while命令

```
while test command; do
    commands
done
while [ $var -gt 0 ]; do
    echo $var
    var=$[ $var - 1 ]
done
```

while 命令允许在while语句内定义多个测试命令，只有最后一个测试命令的退出状态码会被用于决定是否结束循环

```
while echo "this is a test"
[ $var -gt 0 ]; do
    echo $var
    var=$[ $var - 1 ]
done
```

注意！！！**每个测试命令都要单独放在一行！！！**

until命令

判断条件与while相反，命令退出状态码不为0就一直执行

可以进行循环嵌套，举例

```
IFS.OLD=$IFS
IFS=$'\n'
for entry in $(cat /etc/passwd); do
    echo "Value in $entry -"
    IFS=:
    for value in $entry; do
        echo "  $value"
    done
done
```

第一个IFS 值解析出/etc/passwd文件中的各行，内层for循环接着将IFS值修改为冒号，解析出/etc/passwd文件中各行的字段

IFS.OLD 存在的意义为记录IFS的旧值，以便恢复

break 退出循环

> break n 可以跳出多层循环，默认n为1，若n为2，会停止下一级的外层循环

continue 终止某次循环

> continue n　指定继续循环哪一级

可以将循环的输出定向至文件中

```
for ...; do
    echo ..
done > some.txt
echo "finished"
```

类似的，可以将结果传输到其他命令中　done | sort

十四、处理用户输入

14.1 传递参数

```
./script 10 20
```

bash shell 会将所有的命令行参数都指派给称作**位置参数**的特殊变量，包括脚本名称

$0 对应脚本名，**$1** 对应第一个命令行参数，**$2** 对应第二个命令行参数，以此类推

命令行参数间需要用空格分开，shell会将其分配给对应的位置变量(包含空格的参数，需要在两边加上引号)

注意：如果脚本需要的命令行参数不止9个，在第九个变量之后需要在变量名两侧加上花括号。$8, $9, ${10},${11}

使用位置变量 **$0** 获取脚本名

如果使用另一个命令来运行脚本的话，命令名和脚本名会重叠在一起，若运行脚本使用的绝对路径的话，$0 会包含整个路径

basename 可以返回不包含路径的脚本名

在使用位置变量前一定要检查是否为空　　if (-n "$1")

14.2 特殊参数变量

$# 含有脚本运行时包含的命令行参数个数 if [$# -ne ...] ; then ...

**${!#}** 表示最后一个位置变量　　**注意** 命令行中无参数时，$#的值为0，${!#}会返回命令行中的脚本名

$*** 会将所有参数视为单个参数，**$@ 会单独处理每个参数，可以用于遍历

shift 会将所有位置的变量值都向左移动一个位置　　**$3** 会移至 **$2**，**$2** 会移至 **$1**。**$0** 的值不会改变(不知道多少个参数时，可以一直操作第一个参数)

tips:　　当 **$** 出现在双引号内时，会被扩展成有多个命令行参数组成的单个单词，以 **IFS** 变量的第一个字符分隔("$*" 会被分隔为 "$1c$2c$3c...", c为IFS变量的首个字符)　　$@出现在双引号内时，各个命令行参数会被扩展成独立的单词。

碰到了同时使用选项和参数的情况，linux中处理该问题的方法是使用双连字符 **--** ，使用该字符表示选项部分结束，剩下的均为参数部分

```
while [ -n "$1" ]
do
    case "$1" in
        -a) echo "Found the -a option" ;;
        -b) echo "Found the -b option" ;;
        -c) echo "Found the -c option" ;;
        --) shift
            break;;
        *) echo "$1 is not an option" ;;
    esac
    shift
done
...commands
```

若是选项带参:

```
while [ -n "$1" ]
do
    case "$1" in
        -a) echo "Found the -a option" ;;
        -b) echo "Found the -b option" ;;
            param=$2
            echo "Found the -b option with parameter value $param"
            shift ;;
        -c) echo "Found the -c option" ;;
        --) shift
            break;;
        *) echo "$1 is not an option" ;;
    esac
    shift
done
```

...commands

14.3 获取参数的命令

getopt

```
getopt optstring parameters
```

getopt 可以接受一系列任意形式的命令行选项和参数，并自动转换为适当的格式

optstring 是关键所在，它定义了有效的命令行选项字母，定义了哪些选项字母需要参数值

举例： **getopt ab:cd -a -b Value -cd test1 test2**

-a -b BValue -c -d -- test1 test2

定义了四个有效选项a、b、c、d，b 后接了 : 表示b选项需要一个参数值，自动将 -cd 识别为两个单独的命令

在脚本中使用:

```
set -- $(getopt -q ab:cd "$0")
```

位置变量原先的值会被getopt命令的输出替换掉

getopt 并不擅长处理带空格和引号的参数值，会把空格当作参数分隔符

getopts

bash shell 的内建命令

```
getopts optstring variable
```

optstring 如上所述　　getopts 在处理完所有的参数后会退出并返回一个非零的退出状态码

getopts 需要用到两个环境变量，OPTARG 用在选项需要加带参数值的情况；OPTIND 保存着列表参数中getopts正在处理的参数位置

不想显示错误信息，可以在optstring 前加上 :

```
while getopts :ab\:c opt
do
    case "$opt" in
        a) echo... ;;
        b) echo... ;;
        c) commands... ;;
        *) ... ;;
    esac
done
```

注意到，getopts 命令会移除起始的连字符

在处理每个选项时，getopts 会将 OPTIND 环境变量值增1，处理完选项后，可以使用 shift 和 OPTIND 来移动参数

```
shift $[ $OPTIND - 1]
count=1
for param in "$0"; do
    echo "Parameter $count : $param"
    count=$[ $count + 1 ]
done
```

表 14-1　常用的 Linux 命令行选项

选　项	描　述
-a	显示所有对象
-c	生成计数
-d	指定目录

选　项	描　述
-e	扩展对象
-f	指定读入数据的文件
-h	显示命令的帮助信息
-i	忽略文本大小写
-l	产生长格式输出
-n	使用非交互模式（批处理）
-o	将所有输出重定向至指定的文件
-q	以静默模式运行
-r	递归处理目录和文件
-s	以静默模式运行
-v	生成详细输出
-x	排除某个对象
-y	对所有问题回答 yes

14.4 获取用户输入

read

从标准输入接受输入

> -p 直接指定提示符

```
read -p "Please enter a word" word
echo "The word is $word"
```

read 命令会将提示符后输入的数据分配给单个变量。若指定多个变量，输入的每个数据值都会分给变量列表中的下一个变量，若数量不够，则全部分配给最后一个变量

若不指定任何变量，则将接收的所有数据都放入特殊环境变量 REPLY

> -t 使用此选项指定read等待输入的秒数，若超时read命令会返回非0退出状态码
>
> -n 指定字数　read -n 1 ..... [Y/N]
>
> -s 无显示读取

十五、呈现数据

如何将脚本的输出重定向到linux系统的不同位置

15.1 输入和输出

输出：输出到显示器上，输出重定向至文件中

15.1.1 标准文件描述符

linux 会将每个对象当作文件来处理，包括输入和输出。linux用文件描述符来标识每个文件对象。

文件描述符	缩写	描述
0	STDIN	标准输入
1	STDOUT	标准输出
2	STDERR	标准错误

STDIN

代表了shell的标准输入。对终端界面来说，标准输入就是键盘。shell会从STDIN文件描述符对应的键盘获得输入并进行处理。

使用输入重定向符 < 时，linux会用重定向指定的文件替换标准输入文件描述符。命令便会从文件中读取数据，好像从键盘读入一样

STDOUT

STDOUT代表shell的标准输出。在终端上，标准输出就是终端显示器。shell所有输出会被送往终端

类似的，可以使用重定向符 > 来将输出重定向到指定的文件中，也可以使用 >> 追加

但是，当命令产生错误消息时，shell 不会将错误信息重定向到指定文件。文件会被创建，但没有内容

STDERR

shell 或 运行在shell中的脚本和程序报错时，生成的错误消息都会被送到这里。一般，STDERR 和 STDOUT 指向同一个位置，但是STDOUT改变时，STDERR不会改变。

分别重定向错误消息和正常消息：

```
$ ls -al aaa bbb cc ddd errtest 2> errlog 1> log
```

以上命令中，通过**在重定向符 > 前加上STDERR的文件描述符2**将错误消息重定向至errlog，**在重定向符 > 前加上STDOUT的文件描述符1**将正常输出定向至log

&> 会将正常和错误都定向至一个文件中

临时重定向

脚本中，对错误信息做出反应时，需要在文件描述符索引值前加一个&

```
$ echo "simple out"
$ echo "error out" >&2
```

正常运行脚本时，上面两句话都会输出

当脚本重定向STDERR 时，脚本中被送往ERR的文本都会被重定向(也就是只输出 simple out)

永久重定向

exec

exec 1>testout　输出到脚本中

exec 2>errlog

exec 会启动一个新的shell并将STDOUT文件描述符重定向到指定文件。脚本中送往STDOUT 的所有输出都会被重定向。

类似的， exec 0< filename

15.2 创建自己的重定向

15.2.1 创建输出文件描述符

```
$ exec 3>testlog
$ echo "..."
$ echo "...to3" >&3
```

使用exec命令将自己创建的文件描述符3对应的输出定向至目标文件中，特定信息定向至特定文件。当然也可以追加

```
$ exec 3>>testlog
```

tips: 如果文件描述符大于9，可能会和系统的文件描述符冲撞

```
$ exec 3>&1 3定向至1，1指向屏幕
$ exec 1>test1out　1指向文件test1out，3指向屏幕(不因1的改变而改变)
$ exec 1>&3 1指向3，3指向屏幕
```

上例可用于在脚本中临时修改定向

15.2.2 创建输入文件描述符

```
$ exec 6<&0 用6来保存原位置
$ exec 0< testfile　修改输入为文件
$ exec 0<&6 修改至原位置
```

15.2.3 读写文件描述符

```
$ exec 3<> filename
```

写入文件的数据会覆盖文件的原数据

15.2.4 关闭文件描述符

```
$ exec 3>&-
```

若是关闭后再打开，并且打开的还是同一个输出文件，之后创建的输出文件会覆盖原文件

```
$ exec 3>logfile
$ exec 3>&-
$ exec 3>logfile     (在此处新建了文件并覆盖原文件)
```

15.2.5 列出打开的文件描述符

lsof

15.3 命令的其他输入输出

15.3.1 抑制命令输出

若是不想错误信息有任何输出，可以定向到/dev/null

15.3.2 使用临时文件

/tmp 目录下保存了那些不需要永久保留的文件，系统启动时会自动清除

mktemp

使用 mktemp 命令时，需要在文件名末尾加上.XXXXXX，命令会任意的将6个X替换为同等数量的字符，保证文件在目录中唯一

```
$ mktemp test.XXXXXX
$ rm -f test.*
```

> -t 会让 mktemp 命令在系统的临时目录中创建文件，使用此特性时，mktemp命令会返回创建文件的完整路径
>
> -d 创建一个临时目录

```
$ mktemp -d dir.XXXXXX
```

15.3.3 记录消息

tee

将输出同时送往显示器和文件

tee *filename*

会将STDIN的数据同时送往STDIN和文件中

> -a 追加送到指定文件

十六、脚本控制

16.1 处理信号

> - 1 SIGHUP 挂起进程
> - 2 SIGINT 中断
> - 3 SIGQUIT 停止
> - 9 SIGKILL 无条件终止

> - 15 SIGTERM 尽可能终止
> - 18 SIGCONT 继续运行停止的进程
> - 19 SIGSTOP 无条件停止，但不终止进程
> - 20 SIGTSTP 停止或暂停(pause)，但不终止进程

默认情况下， bash shell 会忽略收到的任何 SIGQUIT(3) 和 SIGTERM(15) (因此交互式shell 才不会被意外终止)。但是， bash shell 会处理所有收到的 SIGHUP(1) 和 SIGINT(2)

若收到了 SIGHUP 信号，bash shell 就会退出。退出前，它会将 SIGHUP 信号传给所有由该shell启动的进程，包括正在运行的脚本

随着收到 SIGINT 信号，shell会被中断。linux 内核将不再为 shell 分配 CPU 处理时间。出现这种情况时，shell 会将 SIGINT 信号传给由其启动的所有进程，以此告知情况

ctrl + c 产生中断信号

ctrl + z 暂停信号

16.1.1 捕获信号

```
trap commands signals
trap "echo ' ...somewords '" SIGINT
```

在脚本运行时，检测到由键盘的 ctrl+c 产生的 SIGINT 信号时，就不会中断脚本，而是执行 commands(echo '...somewords')了

tips: trap "" SIGINT 这样可以让脚本完全忽略SIGINT信号，继续执行重要的任务

捕获脚本退出

```
trap commands EXIT
```

脚本退出时，会执行 commands 语句

检测到 SIGINT 提前退出，退出前已经触发了 EXIT ， shell 依然会执行 commands 命令

修改或移除信号捕获

在不同的位置修改信号捕获，产生的效果不一样

```
trap command1 SIGINT
...
trap command2 SIGINT
..
trap -- SIGINT
```

移除信号捕获(移除后会按信号的默认行为来执行)

16.2 后台模式运行脚本

脚本名后加 &

```
$ ./script &
```

在后台运行脚本时，依然会使用终端显示器来显示 STDOUT 和 STDERR 消息

最好将 STDOUT 和 STDERR 进行重定向，否则显示页面会很混乱

若是启动了多个后台进程，在shell终端退出时，这些后台进程也会结束

16.2.1 在非控制台下运行脚本

nohup *command*

关闭终端会话，脚本依然运行

和普通后台进程一样，shell 会给 *command* 分配一个作业号，linux 系统会给其分配一个 PID。nohup 可以阻断发给特定进程的 SIGHUP 信号。

nohup 命令会解除终端和进程之间的关联，进程不再同 STDOUT 和 STDERR 绑定在一起。为保存命令产生的输出，nohup 会自动将 STDOUT 和 STDERR 产生的消息重定向到一个名为 nohup.out 的文件中(一般在当前工作目录创建，否则在 $HOME 目录创建)

16.3 作业控制

16.3.1 查看作业

jobs

$$ 变量用来查看linux 系统分配给该脚本的 PID

查看时，会发现一个 + 一个 - 如果作业控制命令没有指定作业号，默认作业就是 + 所指向的作业。默认作业完成后，下一个默认作业就是 - 所指向的作业。任何时候，shell 中只有一个带 + 的作业和一个带 - 的作业

> - -l 列出进程的 PID 以及作业号
> - -n 只列出上次 shell 发出通知后状态发生改变的作业
> - -p 只列出作业的PID
> - -r 只列出运行中的作业
> - -s 只列出停止的作业

需要删除已停止的作业，使用 kill 命令向其PID发送 SIGKILL(9) 信号即可

```
$ kill -9 PIDNUM
```

16.3.2 重启已停止的作业

bg/fg

bg 在后台启动已被停止的作业(可以指定作业号，若未指定启动默认作业)

fg 在前台启动

16.3.3 调整谦让度

由 shell 启动的脚本优先级默认都是相同的(为0)

-20 ~ 20，越低优先级越高

```
$ nice -n 10 ./script &
```

普通用户可以在**脚本启动时**降低优先级，但不能提高优先级。root或特权用户才行

```
$ renice -n 10 -p 16642
```

在运行时修改优先级

16.4 定时运行作业

16.4.1 at 命令

at命令运行指定Linux 系统何时运行脚本。该命令会将作业提交到队列中，指定shell何时运行该作业

at 的守护进程 atd 在后台运行，在作业队列中检查待运行的作业

at [-f *filename*] time

默认情况下， at 会将 STDIN 的输入放入队列。可以使用 -f 指定用于从中读取命令(脚本文件)的文件名

time 指定了希望运行的时间。若是时间已经过去，at命令会在第二天的同一时刻运行指定的作业(时间形式多种多样)

10:15 PM　　10:15,23:15　now,noon,midnight,teatime　　MMDDYY　　Jun 4　　Now + 25 minutes

10:15 PM tomorrow　10:15 + 7 days

作业队列有52个(a~z, A~Z)，字母排序越高，优先级越低。默认放入a，-q 选项可以加入其他队列

Linux 系统中运行 at 时，系统会将用户的 email 作为 STDOUT 和 STDERR。通过邮件发送给用户。

建议在使用 at 时重定向

-M 可以直接禁止输出

16.4.2 atq 列出等待的作业

```
$ atq
```

16.4.3 atrm 删除等待中的作业

```
$ atrm workNum
```

只能删除自己提交的作业，不能删除别人的

16.4.4 调度需要定期运行的脚本

cron 时间表

> minutepasthour hourofday dayofmonth month dayofweek command

若是每天的10:15运行一个命令，可以在时间表中(未实践)

```
15 10 * * * command
```

每周一的10:15

```
15 10 * * 1 command
```

每个月的最后一天中午12:00

```
00 12 28-31 * * if [ "$(date +%d -d tomorrow)" = 01] ; then
    command ;
fi
```

脚本需要是绝对路径

默认情况下，用户的 cron 时间表并不存在。可以使用 -e 选项向 cron 时间表添加字段

添加字段时，crontab 命令会启动一个文本编辑器

> crontab -l 列出已有的时间表
>
> crontab -e 编辑一个时间表

浏览 cron 目录

```
$ ls /etc/cron.*ly
```

若是对脚本执行时间的精确性要求不高，可以使用预配置的 cron 脚本目录

```
$ ls /etc/cron.*ly
```

将脚本复制到对应的目录即可

anacron

系统关闭的时候不会运行脚本，anacron 会在linux 系统重新启动的时候尽快运行未运行的脚本

anacron 只处理位于 cron 目录的程序，通过时间戳来判断作业是否在计划的间隔内运行了

anacron 使用自己的程序表

```
period delay identifier command
```

period 定义了作业的运行频率　delay 指定了在系统启动后，anacron 需要等待多少分钟再运行错过的脚本

每次启动一个新的 shell 时都会运行一个启动文件

source

另一种运行脚本的方式，与使用 bash 运行差不多，但是不会创建子shell

除了空命令("")之外，source不弄很好的处理 trap 列出的任何命令。使用空命令会导致souce脚本直接忽略trap中的任何信号。

十七、创建函数

17.1 函数基础

17.1.1 创建函数

```
function name {
    commands
}
name() {
    commands
}
```

两种语法规则都可以

17.1.2 使用函数

在脚本中使用函数，只需要像其他shell命令一样写出函数名即可

```
function func1 {
    echo " ..."
}
func1
```

函数需要放在被调用的位置之前

函数名是唯一的，若函数同名则新函数覆盖原函数，不会报错

17.2 函数返回值

bash shell 把函数视为一个小型脚本，运行结束时返回一个退出状态码，使用标准变量 **$?** 来确定函数的退出状态码

17.2.1 默认的退出状态码

```
func1() {
    echo "..."
}
echo "test start"
func1
echo "the exit status is $?"
```

17.2.2 使用return命令

```
funcRe() {
    read -p "Enter a value:  " value
    echo "doubleing the value"
    return $[ $value * 2 ]
}
funcRe
echo "New value is $?"
```

输出时使用 **$?** 来显示 value 的计算结果

> - 函数执行结束立即读取值
> - 退出状态码介于 0~255 间
> - $? 变量保存的是最后执行命令的退出状态码

17.2.3 使用函数输出

```
result = $(funcRe)
```

将函数的输出赋给 $result 变量

```
funcRe() {
    read -p "Enter a value:  " value
    echo $[ $value * 2 ]
}
result=$(funcRe)
```

在终端上不会出现 echo $[$value * 2] 命令的输出

17.3 函数中使用变量

17.3.1 向函数传递参数

和向普通脚本传递参数一样，参数和函数名放在同一行

函数中使用的 $1 和 $2 和脚本主体的 $1 和 $2 变量不是一回事。要在函数中使用脚本的命令行参数，需要在调用函数时手动传入

```
func() {
    if [ $# -eq 0 ]; then
        echo "No input in func"
    elif [ $# -eq 2 ]; then
        echo $[ $1 * $2 ]
    fi
}
```

错误示例：

```
$ ./script 10 20
#脚本内
result=$(func)
```

正确示例：

```
$ ./script 10 20
#脚本内
result=$(func $1 $2)
```

17.3.2 函数中处理变量

对脚本的其他部分而已，函数中定义的变量是无效的

全局变量和局部变量用法与 C 相同

局部变量不同的是，需要在变量前加上 **local** 关键字，即使与全局变量同名，也可以互不干扰

17.4 数组变量和函数

17.4.1向函数传递数组

将数组变量当作单个参数传递给函数的话没有意义。需要在脚本内保存所有的数组元素在将其作为参数传递给函数，函数根据参数重新构建数组变量

```
func() {
    local newarray
    newarray=(\`echo "$@"\`)
    echo "the newarray is ${newarray[*]}"
}
array=(1 2 4 5 6)
echo "original array is ${array[*]}"
func ${array[*]}
```

17.4.2 从函数返回数组

函数的最后使用 echo ${newarray[*]}

```
arr=(1 2 4 5 6)
arg=$(echo ${arr[*]})
result=($(func $arg))
```

脚本通过 arg 将数组元素变为参数传递给 func ，最后 echo 输出每个数组元素的值，脚本用函数的输出重组新的数组变量

17.5 函数递归

```
func() {
    if [ $1 -eq 1 ]; then
        echo 1
    else
        local num=$[ $1 - 1]
        local result=\`func $num1`
        echo $[ $result * $1 ]
    fi
}
```

17.6 创建库

库文件内只需要写函数即可，和脚本操作相同

```
func () {
    ...
}
```

其他文件要调用库文件时，需要使用 **source** 命令，或其别名 **.** (点操作符)。

. ./myfunc

. 后是库文件的路径

17.7 在命令行中使用函数

只是为了方便

在shell中定义了函数之后，可以在整个系统的任意目录使用它，无须担心该函数是否位于PATH变量中

17.7.1 命令行中创建函数

在命令行中定义函数时，每个命令后都需要加上分号

或者：

```
$ funname() {
\> echo $[ $1 * $2 ]
\> ...commands
\> }
$
$ funname 10 20
```

17.1.2 在.bashrc文件中定义函数

.bashrc 文件在用户主目录下，在文件末尾定义函数即可

可以通过 source 或 . 来在 .bashrc 中引入库文件

十八、图形化编程

没学...

十九、sed 和 gawk

这两款工具可以极大地简化数据处理任务

19.1 文本处理

19.1.1 sed编辑器

sed 编辑器根据命令来处理数据流中的数据，这些命令要么从命令行中输入，要么保存在命令文本文件中

sed 编辑器可以

> - 从输入中读取一行数据
> - 根据所提供的编辑器命令匹配数据
> - 按照命令修改数据流中的数据
> - 将新的数据输出到 STDOUT

在流编辑器匹配并针对一行数据执行所有命令之后，会读取下一行数据并重复这个过程。处理完所有数据后结束运行

由于命令是按顺序逐行执行的，因此 sed 编辑器只需对数据流处理一遍即可完成编辑

sed *options script file*

options:

> - -e commands　　在处理输入时，加入额外的 sed 命令
> - -f file　　在处理输入时，将 file 中指定的命令添加到已有的命令中
> - -n　　不产生命令输出，使用 p (print) 命令完成输出

script 参数指定了应用于流数据中的单个命令。若需要多个命令，要么使用 -e 在命令行中指定，要么 -f 在单独的文件中指定。

1. 在命令行中定义编辑器命令

默认情况下，sed 编辑器会将指定的命令应用于 STDIN 输入流中。可以将数据通过管道传入 sed 编辑器进行处理

```
$ echo "..." | sed 's/old/new/'
```

替换命令

sed 编辑器**不会**修改文本文件的数据，只是将修改后的发送到 STDOUT

2. 在命令行中使用多个编辑器命令

```
$ sed -e 's/old1/new1; s/old2/new2/' test.txt
```

命令间使用分号分隔，命令末尾和分号之间不能出现空格

或者

```
$ sed -e '
\> s/old1/new1/
\> s/old2/new2/
\> s/old3/new3/' test.txt
...
```

输入第一个单引号之后，bash 会提示继续输入命令，检测到第二个单引号后，标示结束

sed 命令会将指定的命令应用于文本文件中的每一行

3. 从文件中读取编辑器命令

```
$ sed -f script.sed data1.txt
```

在 script.sed 中，不需要每个命令后都加上分号，每一行都是一个单独的命令

19.1.2 gawk 编辑器

更加贴近编程

> - 定义变量来保存数据
> - 使用算数和字符串运算符来处理数据
> - 使用结构化编程概念(if-then语句和循环)为数据添加处理逻辑
> - 提取文件中的数据并将其重新排列组合，最后生成格式化报告

gawk 的报告生成能力多用于从大文本文件中提取数据并将格式化成可读性报告。gawk 能够从日志文件中过滤出所需的数据，将其格式化，更容易阅读

1. gawk 命令格式

gawk *options program file*

options:

> - -F fs 指定行中划分数据字段的字段分隔符
> - -f file 从指定文件中读取 gawk 脚本代码
> - -v var=value 定义 gawk 脚本中的变量及其默认值
> - -L [ketword] 指定 gawk 的兼容模式或警告级别

gawk 的强大之处在于脚本。你可以编写脚本来读取文本行中的数据，然后对其进行处理并显示，形成各种输出报告

2. 从命令行读取 gawk 脚本

gawk 脚本用一对花括号来定义。必须将脚本命令放到一对花括号 ({ }) 之间

gawk 命令行假定脚本时单个文本字符串，必须将脚本放到单引号中。

```
$ gawk '{print "hello world"}'
```

接下来会进入 STDIN 的文本输入模式。不管输入什么，都会打印 hello world 。需要结束可以 ctrl + D

3. 使用数据字段变量

gawk 会自动为每一行的各个数据元素分配一个变量

$0 代表整个文本行 **$1** 代表文本行的第一个数据字段 **$2** 代表文本行第二个数据字段 **$n** 代表第n个

读取文本时，gawk 会用预先定义的字段分隔符划分数据字段

-F 指定字段分隔符

```
$ gawk -F: '{print $1}' /etc/passwd
```

4. 在脚本中使用多条命令

gawk 运行将多条命令合成一个常规的脚本。执行多条命令只需要在命令间加入 : 分号即可

```
$ echo "My world is xxxx" | gawk '{$4="pretty"; print $0}'
```

第一条命令为字段4赋值，第二条打印整行

5. 从文件中读取脚本

```
gawk -F: script.gawk /etc/passwd
script:
{
text = " ' s home dirctory is "
print $1 text $6
}
#处理结果是打印每个数据的第一个字段和第六个字段
abal 's home dirctory is /sss/sss/sss
```

6. 在处理数据前运行脚本

若是需要在处理数据前运行，比如创建一个标题。可以使用 BEGIN 。在读取数据前先执行命令。但显示后脚本就会直接结束，不显示任何数据。所以需要用另一个区域来定义脚本 (相当于写两个)

gawk 'BEGIN {print "the title is :"}; {print $0}' data.txt

注意：两段脚本被视为 gawk 命令行中的一个文本字符串，需要加上单引号

7. 在处理数据后运行脚本

END 关键字，处理完后运行脚本。可以添加页脚

```
script.gawk:
BEGIN {
commands
print "the title is : "
print "-------------------------------"
FS=":"
}

{
commands
}

END {
commands
}
```

在 BEGIN 中，使用 FS=":" 来定义字段分隔符

19.2 sed 编辑器基础命令

19.2.1 更多的替换选项

1. 替换标志

s/pattern/replacement/flags

4种可用的替换标志

> - 数字，指明新文本将替换每行中第几处匹配
> - g, 指明新文本将替换行中所有匹配
> - p, 指明打印出替换后的行
> - w file, 将替换的结果写入文件

替换标志 p 和 sed 的 -n 选项配合，可以做到只输出被替换命令修改的行

2. 替代字符

```
$ sed 's!/bin/bash!/bin/csh!'
```

此处将 ! 作为替换命令的分隔符 (原分隔符为 /)，这样替换路径时便不显得繁琐难懂

19.2.2 使用地址

行寻址

> - 以数字形式表示的行区间
> - 匹配行内文本的模式

[*address*]

{
commands

}

1. 数字行寻址

```
sed '2s/dog/cat/'
```

第二行的 dog 替换为 cat

```
sed '2,3s/dog/cat/'
```

第2~3行的 dog 替换为 cat

```
sed '2,$s/dog/cat/'
```

第2行到结尾的 dog 替换为 cat

2. 使用文本模式过滤

/pattern/command

将指定的**模式(pattern)**放入正斜线内。sed 编辑器会将该命令应用于包含匹配模式的行

sed '**/rich**/s/old/new/' data.txt

行中包含 rich 的行进行替换命令

3. 命令组

单行中执行多条命令，用 { } 括起来

sed '2{s/sad/happy/; s/March/April/}' data.txt

19.2.3 删除行

```
sed 'd' data.txt
sed '3d' data.txt
sed '2,3d' data.txt
sed '/poor/d' data.txt  (包含poor的行进行删除)
sed '/1/,/3/d' data.txt (删除从包含 1 的行到 包含 3 的行)
```

19.2.4 插入和附加文本

> - 插入 (insert) (i) 命令会在指定行前增加一行
> - 附加 (append) (a) 命令会在指定行后增加一行

```
$ echo "..." | sed 'i\the new line*'
```

the new line 会出现在第一行

同理，若是 sed 'a*the new line*' 则 the new line 会出现在最后一行

可以像删除一样**指定行**

$ 代表最后一行

若要插入多行，可以用 sed 'i*the new line*.*the other line*.' data.txt

在新文本末尾使用反斜线 \

19.2.5 修改行

修改 (c) 命令

和插入，附加命令相同，指定单独一行

可以使用模式来寻址

```
$ sed '/hhuhuh/c\the new line' data.txt
```

若使用区间，则会将区间内的文本内容替换为一句新内容 (2~3行变成1行)

19.2.6 转换命令

处理单个字符

[address]y/inchars/outchars/

inchars 和 outchars 的长度需要相同，对应位进行替换，全部替换完成后命令结束

是一个全局命令，对文本行中匹配到的所有指定字符进行转换，不考虑位置

sed 'y/123/456/' data.txt

1 换为 4， 2 换为 5， 3 换为 6

文本中的每个指定字符都会被替换

19.2.7 打印

使用标志 p 和替换命令显示 sed 编辑器修改过的行

> - 打印 (p) 命令用于打印文本行
> - 等会 (=) 命令用于打印行号
> - 列出 (l) 命令用于列出行

1. 打印行

-n 抑制其他行的输出，只打印匹配的行

```
$ sed -n '/3/{p; s/old/new/p}' data.txt
```

首先打印包含数字3的行，然后打印将 old 改为 new 之后的指定行

输出同时显示了原始文本行和新的文本行，在做修改前可以用作检查

2. 打印行号

```
$ sed -n '/searchWord/{=; p; }' data.txt
```

先打印包含 searchWord 的行号，然后打印包含 searchWord 的行

3. 列出行

列出用于打印数据流中的文本和不可打印字符(例如制表符\t，换行符)

19.2.8 使用 sed 处理文件

1. 写入文件

写入 (w)

[address]w *filename*

```
$ sed '1,2w test.txt' data.txt
```

将 test.txt 中的前两行写入 data.txt

2. 从文件读取数据

读取 (r)

[address]r *filename*

将一条独立文件中的数据插入数据流

filename 为数据文件的绝对路径或相对路径

读取命令无法使用地址区间，只能指定单行或文本模式地址

sed 会将文件内容插入指定地址后

```
$ sed '3r data.txt' dirdata.txt
```

将 data.txt 中的内容插入到 dirdata.txt的 LIST 行后

```
$ cat dir.txt
Here is the list of name

LIST

end here
$ sed '/LIST/{r data.txt; d}' dir.txt
```

将 data.txt 的内容插入dir.txt 中并删除 LIST

```
Here is the list of name

aaa sdfds

bbb fwerqe

cc asqwer

end here
```

3. sed 的 F 选项

用来打印当前正在处理的文件名。且不受 -n 选项的影响

若只需要显示一次，在 F 命令前加上数字 1 (否则，所处理的每个文件的每一行都会显示文件名)

二十、正则表达式

在 shell 脚本中成功运用 sed 和 gawk 的关键在于熟练掌握正则表达式。从大量数据中过滤特定数据可能会很复杂

20.1 正则表达式基础

20.1.1 定义

正则表达式是一种可供 linux 工具过滤文本的自定义模板。linux 工具(例如 sed 或 gawk)会在读取数据时使用正则表达式对数据进行模式匹配。若匹配模式，则会进行处理；不匹配则弃用

正则表达式使用元字符(一些特殊的字符)来描述数据流中的一个或多个字符。例如 ls 加上通配符 * 列出满足特定要求的文件和目录

正则表达式的工作方式和通配符类似。包含文本和特殊字符，定义了 sed 和 gawk 匹配数据时用的模板。

20.1.2 正则表达式的类型

正则表达式是由**正则表达式引擎**实现的。这是一种底层软件，负责解释正则表达式并用这些模式进行文本匹配

linux 中流行的正则表达式引擎有两种

> - POSIX 基础正则表达式(basic regular expression, BRE) 引擎
> - POSIX 扩展正则表达式(extended regular expression, ERE) 引擎

sed 仅符合 BRE 的引擎规范的一个子集，gawk 使用 ERE 引擎来处理正则表达式

20.2 定义 BRE 模式

最基本的 BRE 模式时匹配数据流中的文本字符。

20.2.1 普通文本

```
sobloom@sobloomdm:~$ echo "This is a test" | sed -n '/test/p'
This is a test
sobloom@sobloomdm:~$ echo "This is a test" | sed -n '/Nop/p'
sobloom@sobloomdm:~$ echo "This is a test" | gawk '/test/{print $0}'
This is a test
sobloom@sobloomdm:~$ echo "This is a test" | gawk '/Nop/{print $0}'
sobloom@sobloomdm:~$
```

正则表达式并不关心模式在数据流中出现的位置，也不在意模式出现了多少次。

正则表达式对匹配的模式非常挑剔

第一条，区分大小写。

在正则表达式中，无须写出整个单词，定义的文本出现在数据流中，正则表达式就可以匹配

notes 中出现了 note ，也可以匹配

```
sobloom@sobloomdm:~$ echo "notes here" | sed -n '/note/p'
notes here
```

在正则表达式中，空格和普通的字符没什么区别

```
$ echo "note here" | sed -n '/e h/p'
note here
```

20.2.2 特殊字符

正则表达式能识别的特殊字符如下所示

.*[]^${}\+?|()

不能在匹配普通文本的模式中单独使用这些字符

要将某个字符视作普通字符，必须转义　　使用 \ 反斜线

正斜线虽然不是特殊字符，但使用时也需要转义

20.2.3 锚点字符

1. 锚定行首

脱字符 (^) 可以指定位于数据流中文本行行首的模式

若正则表达式只能匹配出现在行首的模式 (下例中，bloom 必须位于行首)

若使用脱字节，必须将其置于正则表达式之前

```
$ echo "Flowers bloom here" | sed -n '/^bloom/p'
$
$ echo "blooming flower" | sed -n '/^bloom/p'
blooming flower
$
```

若是将脱字符放到正则表达式开头之外的位置，便和普通字符一样了 (若只匹配脱字符，可以不用转义，但若是脱字符和其他文本连接在一起，则需要转义) (this is a ^ ，不需要转义 '/^/p'　 this is a ^word，需要转义

'/\^word/p')

2. 锚定行尾

美元符号 ($) 定义了行尾锚点

```
$ echo "Flowers bloom here" | sed -n '/here$/p'
Flowers bloom here
$ echo "Flowers bloom here" | sed -n '/her$/p'
```

这个字符放在正则表达式之后表示数据行必须以该模式结尾

且必须清楚要查找什么。here 改成 her 便不能匹配了

必须是行的最后一部分

3. 组合锚点

```
$ cat data
this is a test of using both anchors
I think I love flowers
I love flowers
I am sure I love flowers
$ sed -n '/^I love flowers$/p' data
I love flowers
$
```

sed 忽略了哪些不单单包含指定模式的行

将这两个锚点组合在一起，不加任何文本，可以过滤数据流中的空行

```
$ cat data
this is a flower

this is a book
$ sed '/^$/d' data
this is a flower
this is a book
$
```

指定的正则表达式会查找行首和行尾之前什么都没有的行，然后使用删除命令 d 来删除匹配的行，也就是删除了所有的空行。这是从文档中删除空行的一种方法

20.2.4 点号字符

点号字符可以匹配除换行符之外的任意单个字符。点号字符必须匹配一个字符，若点号所在位置没有可以匹配的字符则模式不成立

```
$ cat data
this is a apple
this is a oier
this is a bottle of water
this is a er
er haha huh
$ sed -n '/.er/p' data
this is a oier
this is a bottle of water
this is a er
$
```

第一行没有 er，不匹配。第二三四行都有 oier, water，er (空格也算字符)，第五行 er 在开头，前面无字符

20.2.5 字符组

在正则表达式中定义用来匹配某个位置的一组字符。若字符组中的某个字符出现在了数据流中，则匹配该模式

方括号用于定义字符组

```
$ sed -n '/[ch]at/p' data
The cat is sleeping
That is a nice hat.
```

过滤了只包含 at 的行，匹配这个模式的单词有 cat 和 hat

字符组中必须有个字符来匹配相应的位置

不确定字符的大小写时可以使用字符组

```
$ echo "Yes" | sed -n '/[Yy]es'/p
Yes
```

单个表达式中可以使用多个字符组

```
$ echo "Yes" | sed -n '/[Yy][Ee][Ss]'/p
Yes
```

字符组中并非只能由字母，也可以包含数字

正则表达式可以匹配数据流中任何位置的文本。如果确定只匹配 3 位数，就必须和其他字符分开，要么用空格，要么明确匹配字符的起始位置

20.2.6 排除型字符组

可以在字符组的开头加上脱字符，表示匹配字符组中没有的字符

```
$ sed -n '/[^abc]at/p' data
This is a data
```

即使是排除型，字符组所在的位置仍必须匹配一个字符

20.2.7 区间

可以用单连字符在字符组中表示字符区间，只需指定区间的第一个和最后一个字符即可

```
$ sed -n '/[0-9][a-d][h-z]/p' data
```

可以在单个字符组内指定多个不连续的区间

```
$ sed -n '/[0-36-9]ahuha/p' data
```

20.2.8 特殊的字符组

BRE 提供了一些特殊的字符组，用来匹配特定类型的字符

表 20-1　BRE 特殊字符组

字　符　组	描　　述
[[:alpha:]]	匹配任意字母字符，无论是大写还是小写
[[:alnum:]]	匹配任意字母数字字符，0~9、A~Z 或 a~z
[[:blank:]]	匹配空格或制表符
[[:digit:]]	匹配 0~9 中的数字
[[:lower:]]	匹配小写字母字符 a~z
[[:print:]]	匹配任意可打印字符
[[:punct:]]	匹配标点符号
[[:space:]]	匹配任意空白字符：空格、制表符、换行符、分页符（formfeed）、垂直制表符和回车符
[[:upper:]]	匹配任意大写字母字符 A~Z

```
$ echo "996" | sed -n '/[[:alpha:]]/p'
$ echo "abc" | sed -n '/[[:alpha:]]/p'
abc
```

20.2.9 星号

在字符后放星号 * 表示该字符必须在匹配模式的文本中出现 0 次或多次 (不确定出现或不出现时使用)

```
$ echo "flowr" | sed -n '/we*r/p'
flowr
$ echo "flower" | sed -n '/we*r/p'
flower
$ echo "floweeer" | sed -n '/we*r/p'
floweeer
$ echo "floweeeeeeeeeeeeeeeeeeer" | sed -n '/we*r/p'
floweeeeeeeeeeeeeeeeeeer
```

广泛用于处理有拼写错误或不同语言中有拼写变化的单词。

星号也可以用于字符组

20.3 扩展正则表达式

这里用的 gawk 命令

在 gawk 命令中使用正则表达式区间时，需要加入 --re-interval 命令

20.3.1 问号

表明前面的字符出现 0 次或 1 次，不会匹配多次出现的字符

同样，可以用于字符组

```
$ echo "eolute" | gawk '/e[u-w]?o/{print $0}'
eolute
$ echo "evolute" | gawk '/e[u-w]?o/{print $0}'
evolute
$ echo "eaolute" | gawk '/e[u-w]?o/{print $0}'
$
$ echo "euvolute" | gawk '/e[u-w]?o/{print $0}'
$
$ echo "evvvolute" | gawk '/e[u-w]?o/{print $0}'
$
```

只有均不出现，或者其中一个出现一次才会匹配

其他字符，多个字符出现均不匹配

20.3.2 加号

前面的字符可以出现一次或多次，至少一次

和问号类似

不同的是，上面的 euvolute, evvvolute，加号也能匹配(允许出现多个字符)

20.3.3 花括号

ERE 中的花括号允许为正则表达式指定具体的可重复次数，被称为**区间**

- m : 正则表达式恰好出现 *m* 次
- m, n : 正则表达式至少出现 *m* 次，至多出现 *n* 次

同样用于字符组，出现次数为字符组内字符出现的累计次数

```
$ echo "aoaaabalabala" | gawk '/oa{3}b/{print $0}'
aoaaabalabala
$ echo "aoaoebalabala" | gawk '/o[aoe]{3}b/{print $0}'
aoaoebalabala
```

20.3.4 竖线符号

以逻辑 OR 方式指定正则表达式引擎要使用的两个或多个模式。若其中一个匹配了数据流文本，就视为匹配。

expr1|*expr2*...

```
$ echo "April is comming" | gawk '/March|April/{print $0}'
April is comming
```

竖线符号两侧的子表达式可以采用正则表达式可用的任何模式符号

20.3.5 表达式分组

用圆括号对正则表达式进行分组。分组后，每一组会被视为一个整体 (下例中的 ? 表示出现 1 次或 0 次)

```
$ echo "Sat" | gawk '/Sat(urday)?/{print $0}'
Sat
$ echo "Saturday" | gawk '/Sat(urday)?/{print $0}'
Saturday
```

匹配 Saturday 或其缩写 Sat

可以将分组和竖线结合起来

```
$ echo "kfc" | gawk '/(a|k)f(c|z)/{print $0}'
kfc
```

afc, afz, kfc, kfz 均可匹配

二十一、sed 进阶

21.1 多行命令

若是在数据流中寻找短语，而短语被分成了两行，对于单行命令的普通 sed , 就无法找到了

用于处理多行文本的特殊命令

- N: 加入数据流中的下一行，创建一个多行组进行处理

- D: 删除多行组中的一行
- P: 打印多行组中的一行

21.1.1 next 命令

多行 next(N) 命令

1. 单行 next 命令

单行 next (n) 命令会告诉 sed 编辑器移动到数据流中的下一行，不用返回到命令列表最开始的位置

```
$ cat data
First

Second

Third
$ sed '/First/{n ; d}' data
First
Second

Third
```

使用 next 命令，查找到包含 First 的行，然后 n 命令移动到下一行，进行删除。这时，sed 编辑器会继续执行命令列表，即使用删除局命令删除空行。sed 编辑器在执行完命令脚本后会读取数据流中下一行文本，并从头开始执行脚本。sed 编辑器找不到包含 First 的行了，所以不会有其他行被删除

2. 合并文本行

单行的 next 是将数据流中的下一行移入 sed 编辑器的工作空间。多行版本的 next (N)命令是将下一行添加到模式空间已有文本之后

sed 编辑器会将两行文本**当成一行**来处理

```
$ cat data
First
Second
Third
Fourth
$ sed '/Second/{ N ; s/\n/ / }' data
First
Second Third
Fourth
```

用N与下一行进行合并，之后用空格替代 \n ，两行合为一行

可以用此解决短语分散在两行的问题

```
$ cat data
April is
comming soon
Wish we have bright future
$ sed 'N ; s/is.comming soon/will come again/' data
April will come agin
Wish we have bright future
```

用 N 命令将第一个单词所在行与下一行合并，即使短语内有换行仍然可以查找

在 is 和 comming 之间用了 . (点号模式) 来匹配空格和换行符这两种情况。但若匹配的是换行符，则会导致两行变成一行，不是原结果

可以在 sed 中使用两个替换命令来解决

```
$ sed 'N
> s/is\ncomming soon/will\ncome again/
> s/is comming soon/will come again/
> ' data
April will
come again
Wish we have bright future
```

第一个替换命令用于专门查找两个单词间的换行符，这样如果是两行的话可以还原原文本格式

但是，若是匹配的文本在最后一行，脚本在执行 sed 编辑器前总是先将下一行文本读入，最后一行无法读入下一行，则 N 命令会叫停 sed 编辑器，无法找到最后一行的文本

可以将单行编辑命令放在 N 命令之前。

```
$ cat data
Wish we
have a apple
Wish we have a good mood
$ sed '
> s/Wish we have/There will be/
> N
> s/Wish we\nhave/There will\nbe
> ' data
There will
be a apple
There will be a good mood
```

21.1.2 多行删除命令

多行删除 (D)

只会删除模式空间中的第一行

删除该行的换行符及其之前的字符(其实就是一整行)

第二行虽然被 N 加入模式空间，但仍然完好

如果需要删除目标的前一行，很好用

```
$ cat data

First
Second

Third
$ sed '/^$/{N ; /First/D}' data
First
Second

Third
```

sed 编辑器会查找空行然后用 N 将下一行加入模式空间

若模式空间中有单词 First，则 D 命令会删除模式空间的第一行

此操作需要结合 N 和 D

21.1.3 多行打印命令

P

只打印模式空间中的第一行

出现多行匹配时，P 只打印模式空间的第一行

D 命令的独特在于其删除模式空间中的第一行后，会强制 sed 编辑器返回到脚本的起始处，对当前模式空间中的内容重新执行此命令(D 命令不会从数据流中读取新行)。在脚本中加入 N 命令，就能单步扫过整个模式空间，对多行进行匹配

P 打印第一行，D 删除第一行并绕回脚本起始处， N 会读取下一行并重新开始此过程

21.2 保留空间

sed 编辑器有两块空间：模式空间(一块活跃的缓冲区) 和保留空间 (也是缓冲区)

在处理模式空间时，可以用保留空间临时保存部分行

- h　将模式空间复制到保留空间
- H　将模式空间附加到保留空间
- g　将保留空间复制到模式空间
- G　将保留空间附加到模式空间
- x　交换模式空间和保留空间间的内容

通常，使用 h 命令或 H 命令将字符串移入保留空间后，最终还是要用 g，G，x 命令将保存的字符串移回模式空间

```
$ cat data
First
Second
Third
Fourth
$ sed -n '/Second/ {
> h ; p ;
> n ; p ;
> g ; p }
> ' data
Second
Third
Second
```

1. sed 脚本使用正则表达式作为地址，过滤出含有单词 Second 的行
2. 出现含有 Second 的行时，{ } 中的第一个命令 h 会将该行复制到保留空间。这时，保留空间和模式空间中的内容一致
3. p 命令会打印出模式空间的内容 （ Second ），也就是被复制进保留空间的内容
4. n 命令会提取数据流中的下一行 (Third)，将其放入模式空间。现在，模式空间和保留空间中的内容**不一样**了
5. p 命令会打印出模式空间中的内容 (Third) ，
6. g 命令会将保留空间中的内容 (Second) 放回模式空间，替换模式空间中的当前文本。模式空间和保留空间中的内容又相同了
7. p 命令会打印出模式空间的当前内容 (Second)

通过保留空间来移动文本行，可以强制 Second 输出在 Third 之后。去掉第一个 p 命令，可以将这两行以相反的顺序输出

```
$ sed -n '/Second/ {
> h ;
> n ; p
> g ; p }
> ' data
Third
Second
```

可以借此创建一个脚本反转整个文件的各行文本

21.3 排除命令

可以指示命令**不应用于**数据流中的特定地址或地址区间

感叹号 (!) 用于排除 (negate) 命令，也就是让原本会起作用的命令失效

```
$ sed -n '/First/!p' data
Second
Third
Fourth
```

本来应当只打印 First 行，加了感叹号之后，除了 First 行全部进行了打印。情况反了过来

```
$ sed 'N
> s/is\ncomming soon/will\ncome again/
> s/is comming soon/will come again/
> ' data
...
$ sed '$!N
> s/is\ncomming soon/will\ncome again/
> s/is comming soon/will come again/
> ' data
...
```

区别在于，上半部分的命令为对所有行 (除了最后一行) 读取下一行，然后进行判断。下半部分为，除了对最后一行不读行，其他行都读行进行判断($ 表示最后，! 表示反转)

这样，上半的命令无法对最后一行进行处理，但是下半的命令可以

如果要反转文本行

1. 读取第一行 -> 保留第一行
2. 读取第二行 -> 保留的第一行附加到模式的第二行 -> 保留第一二行
3. 读取第三行 -> 保留的第一二行附加到模式的第三行 -> 保留第一二三行
4. ...重复操作

若不想将保留空间的文本附加到要处理的第一行文本之后，可以使用感叹号 (!) 命令

1!G

接下来将新的模式空间 (包含已反转的文本行) 放入保留空间

将模式空间中的所有文本都反转后，只需打印结果。到达数据流的最后一行时，就得到了模式空间的所有内容

```
$ sed -n '{1!G ; h ; $p}' data
```

21.4 改变执行流程

通常，sed 编辑器会从脚本的顶部开始，一直执行到脚本尾部 (D 命令除外，强制 sed 编辑器在不读取新行的情况下返回到脚本的顶部)

21.4.1 分支

这种方法可以基于地址、地址模式或地址区间排除一整段命令。允许只对数据流中的特定行执行部分命令

分支 (b) 命令的格式如下：

[address]b [label]

address 参数决定了哪些行会触发分支命令。label 参数定义了要跳转到的位置。若没有 label 参数，则跳过触发分支命令的行，继续处理余下的文本行

```
$ cat data
First line
Second line
Third line
Fourth line
$ sed -n '{2,3b ;
> s/line/bala/}
> ' data
First bala
Second line
Third line
Fourth bala
```

分支命令在数据流中的第二行和第三行跳过了两次替换命令

若不想跳到脚本末尾，可以定义 label 参数，指定分支命令要跳转的位置。标签以冒号开始，最多有 7 个字符

:label2

要指定哪些 label ，把它放在分支命令之后即可。有了标签，就可以使用其他命令处理匹配 *address* 的那些行。对于其他行，仍可以用脚本中原先的命令处理。

```
$ sed '{/Second/b skip1 ;
> s/line/bala/
> :skip1
> s/line/skip bala/}
> ' data
First bala
Second skip bala
Third bala
Fourth bala
```

分支命令指定，若**文本行**出现了 Second ，则程序**跳**至**标签**为 skip1 的**脚本行**。若文本行**不匹配**分支 *address*，则 sed 编辑器会**继续执行脚本中的命令**，**包括分支标签 skip1 之后的命令**。(因此，两个替换命令都会被应用于不匹配分支 *address* 的行)

若某行匹配 *address* 分支，则 sed 编辑器就会跳转到带有分支标签 skip1 的那一行，故只有最后一个替换命令会被执行

此例演示了跳转到 sed 脚本下方的标签。下例演示了跳转到靠前的标签，以实现循环

```
$ echo "There, is, a, pretty, flower" |
> sed -n '{
> :start
> s/,//1p
> b start
> }'
There is, a, pretty, flower.
There is a, pretty, flower.
There is a pretty, flower.
There is a pretty flower.
^C
```

脚本的每次迭代都会删除文本中的第一个逗号并打印字符串。有一个问题，就是它会永远不停的查找逗号，形成死循环，除非 ctrl + C

为了避免，可以为分支命令指定一个地址模式，若不匹配则不会跳转

```
$ echo "There, is, a, pretty, flower." |
> sed -n '{
> :start
> s/,//1p
> /,/b start
> }'
There is, a, pretty, flower.
There is a, pretty, flower.
There is a pretty, flower.
There is a pretty flower.
```

现在分支命令只会在行中有逗号的情况下跳转。在最后一个逗号被删除后，分支命令不再执行，脚本正常结束。

21.4.2 测试

测试 (t) 命令也可以改变 sed 编辑器脚本的执行流程。测试命令会根据先前的替换命令结果跳转至某个 label 处，而不是根据 address 进行跳转

若替换命令成功匹配并完成了替换，测试命令就会跳转到指定的标签。若未完成替换，则不会跳转

[address]t [label]

和分支命令一样，若没有 label ，测试成功时，sed 会跳转到脚本结尾

测试命令提供了一种低成本的方法来对数据流中的文本执行基本的 if-then 语句。若需要进行二选一的替换操作，测试命令好用(无须指定 label)

```
$ sed '{s/Second/yes/ ; t
> s/line/bala/}
> ' data
First bala
yes line
Third bala
Fourth bala
```

第一个替换命令会查找模式文本 Second。若匹配，就替换文本，且执行后面的替换命令。若第一个替换命令不匹配，则执行后面的替换命令

有了分支，可以避免之前的死循环

```
$ echo "There, is, a, pretty, flower." |
> sed -n '{
> :start
> s/,//1p
> t start
> }'
There is, a, pretty, flower.
There is a, pretty, flower.
There is a pretty, flower.
There is a pretty flower.
```

没有逗号可以替换时，测试命令不再跳转，继续执行剩下的脚本

21.5 模式替换

.at 可以匹配 hat,cat,bat...，但是 s/.at/“.at”/ 中的 ".at" 却不能直接表示 hat, cat, bat 这样的单词，只是 ".at" 而已

21.5.1 & 符号

& 符号可以代表替换命令中的匹配模式。不管匹配到什么样的文本，可以用 & 符号代表这部分内容。这样就可以处理匹配到的任何单词了

```
$ echo "I like king" |
> sed -n '/s/.ing/&/p'
I like "reading"
(原文此处的flag为g，不太懂什么意思)
(原笔记中 king 为 reading，若不使用 & 符号的话，匹配的部分应该为 ding，而不是 reading，算是犯的一个小错误)
```

匹配到 reading，被替换为 "reading"

21.5.2 替换单独的单词

想要替换字符串中的单词

使用圆括号来定义匹配模式中的子模式

随后使用特殊字符组合来引用每个子模式匹配到的文本。反向引用有反斜线和数字组成。数字表明子模式的序号，第一个子模式为 \1，第二个子模式为 \2

在替换命令中使用圆括号，必须使用转义字符。以表明这不是普通的圆括号

```
$ echo "The flower is haokan" |
> sed '
> s/\(The\) flower/\1 book/'
The book is haokan
```

\1 即为 The，第一个圆括号中的子模式

若需要使用一个单词来替换一个短语，这个单词又恰好是短语的子串，用子模式会方便很多

```
$ echo "nice king" |
> sed 's/nice \(.ing\)/\1/'
nice
(若此处为 reading 的话，会出现无法匹配的情况)
(此例主要用于表现简化吧)
```

在这种情况下，不能用 & 符号，因为它代表了整个模式匹配的文本。反向引用允许将某个子模式匹配到的文本作为替换内容

需要在两个或多个子模式间插入文本时，这个特性尤其有用。下面的脚本使用子模式在大数之间插入逗号。

```
$ echo "123456789" | sed '{
> :start
> s/\(.*[0-9]\)\([0-9]\{3\}\)/\1,\2/
> t start}'
123,456,789
(此处的匹配为，每次匹配连续的数字字符串，取后三位为一个子模式，前面的全部为一个子模式，在两个子模式中间加逗号)
(仅供参考)
```

分成了两个子模式

.*[0-9]

[0-9]{3}

自后向前添加逗号

21.6 在脚本中使用 sed

21.6.1 使用包装器

shell 脚本包装器 ChangeScriptShell.sh

在shell脚本中，可以将普通的 shell 变量及命令行参数和 sed 编辑器脚本一起使用。

```
$ cat turn.sh
#!/usr/bin/env bash
sed -n '{1!G; h; $p}' $1
exit
$ ./turn.sh data
Fourth
Third
Second
First
```

脚本通过位置变量 $1 来获取第一个命令行参数

21.6.2 重定向 sed 的输出

sed 编辑器会默认输出到 STDOUT。可以重定向

用 $ ( ) 将 sed 编辑器命令的输出重定向到一个变量中

可以使用 sed 将阶乘的结果加上逗号

```
commands...

result=$(echo $number |
sed '{
:start
s/\(.*[0-9]\)\([0-9]\{3\}\)/\1,\2/
t start
}')

commands...
echo ...
exit
```

21.7 创建 sed 实用工具

21.7.1 加倍行间距

```
$ sed 'G' data
First

Second

Third

Fourth
```

21.7.2 对可能含有空行的文件加倍行间距

```
$ sed '/^$/d ; $!G' data
```

删除空行，除了最后一行不加倍其他加倍

21.7.3 给文件中的行编号

```
$ sed '=' data | sed 'N; s/\n/ /'
( why )
```

21.7.4 打印末尾行

```
$ sed '{
> :start
> $q ; N ; 11,$D
> b start
> }' data
```

若当前行为数据流的最后一行，则退出命令会停止循环，N 命令会将下一行附加到模式空间中的当前行之后。若当前行在第十行之后，则 11, $D 命令会删除模式空间中的第1行。在模式空间中创造了滑动窗口的效果。只显示文件的最后10行

21.7.5 删除行

```
$ sed '/./,/^$/!d' data
(删除多余的空行)
```

```
$ sed '/./,$!d' data
(删除开头的空行)
```

```
sed '{
:start
/^\n*$/{$d; N; b start }
}'
(删除结尾后的空行)
```

21.7.6 删除 html 标签

```
$ sed 's/<[^>]*>//g' ; /^$/d' data
```

二十二、gawk 进阶

22.1 使用变量

gawk 语言支持两种变量

- 内建变量
- 自定义变量

gawk 的内建变量包含用于处理数据文件中的数据字段和记录的信息。可以在 gawk 脚本中创建自己的变量

22.1.1 内建变量

1. 字段和记录分隔符变量

gawk 的一种内建变量：**数据字段变量**，允许使用美元符号 ($) 和字段在记录中的位置值来引用对应的字段。因此，要引用记录中的第一个数据字段，就用 $1；第二个就用 $2 ，以此类推

字段分隔符一般是空白字符(空格和制表符) 可以在命令行使用 -F 或者在脚本中使用特殊内建变量 FS 修改字段分隔符

有一组内建变量可以控制 gawk 对输入数据和输出数据中字段和记录的处理方式，FS 是其中之一

- FIELWDIDTHS　　由空格分隔的一列数字，定义了每个数据字段的确切宽度
- FS　输入字段分隔符
- RS　输入记录分隔符
- OFS 输出字段分隔符
- ORS 输出记录分隔符

一般 OFS 是空格，置于输出的每个字段间

```
$ cat data
kfc,mdl,wls,abk
$ gawk 'BEGIN{FS=','} {print $1,$2,$3}' data
kfc mdl wls
$ gawk 'BEGIN{FS=','; OFS='-'} {print $1,$2,$3}' data
kfc-mdl-wls
```

FIELDWIDTHS 变量可以不通过字段分隔符来读取记录。有些应用程序没有使用字段分隔符，而是将数据放在特定的列

设置了 FIELDWIDTHS，gawk 会忽略 FS 变量，根据提供的字段宽度来计算字段

```
$ cat data
138121382333777
3.0101-013.0963
$ gawk 'BEGIN{FIELDWIDTHS="3 5 4 3"}{print $1,$2,$3,$4}' data
138 12138 2333 777
3.0 101-0 13.0 777
```

FIELDWIDTHS 的值，一但设置了就不能改动。不适用于变长的数据字段

RS 和 ORS 一般为换行符

若是数据流中遇到占据多行的数据，使用默认的 FS 和 RS 来读取数据， gawk 就会把每一行当作一条单独的数据来读取

要解决，可以把 FS 设置为换行符，RS 设置为空字符串，在每个数据段之间留下一个空行

```
$ cat data
robin
19191919
queen

jack
10101010
king

bala
88888888
local
$ gawk 'BEGIN{FS="\n"; RS=""} {print $1,$2}' data
robin 19191919
jack 10101010
bala 88888888
```

2. 数据变量

变　量	描　述
ARGC	命令行参数的数量
ARGIND	当前处理的文件在 ARGV 中的索引
ARGV	包含命令行参数的数组
CONVFMT	数字的转换格式（参见 printf 语句），默认值为%.6g
ENVIRON	当前 shell 环境变量及其值组成的关联数组
ERRNO	当读取或关闭输入文件发生错误时的系统错误号
FILENAME	用作 gawk 输入的数据文件的名称
FNR	当前数据文件中的记录数
IGNORECASE	设成非 0 值时，忽略 gawk 命令中出现的字符串的大小写
NF	数据文件中的字段总数
NR	已处理的输入记录数
OFMT	数字的输出显示格式。默认值为%.6g.，以浮点数或科学计数法显示，以较短者最多使用 6 位小数
RLENGTH	由 match 函数所匹配的子串的长度
RSTART	由 match 函数所匹配的子串的起始位置

ARGV数组从索引 0 开始，代表命令。第一个数组值是 gawk 命令后的第一个命令行参数

ENVIRON 使用关联数组来提取 **shell** 环境变量的值

```
$ gawk '
> BEGIN{
> print ENVIRON["HOME"]
> print ENVIRON["PATH"]
> }'
...
```

NF 变量可以在不知道具体位置的情况下引用记录中的最后一个数据字段。其含有数据文件中最后一个字段的编号。可以在 NF 变量之前加上美元符号，将其用作字段变量

FNR 和 NR 变量类似，但略有不同。FNR 变量包含当前数据文件中已处理的记录数，NR 变量包含已处理过的记录总数。

处理多个文件时，每处理一个文件，FNR 的值会重置，而 NR 会继续计数

22.1.2 自定义变量

和在 shell 中一样

但是，允许在正常代码外赋值

```
$ cat script
BEGIN{FS=','}
{print $n}
$ gawk -f script n=2 data
```

可以在不修改脚本代码的情况下改变脚本行为

-v 可以在 BEGIN 之前设定变量

```
$ gawk -v n=3 -f script data
```

22.2 处理数组

关联数组的索引可以是任意文本字符串。类似于哈希表

22.2.1 定义数组变量

var[index] = element

cat["lihua"] = "verybeautiful"

cat["jumao"] = "veryfat"

22.2.2 遍历数组变量

for (var in *array*)

{

    statements

}

在循环时将关联数组的下一个索引赋值给 var ， 然后执行一遍 statements 。**是索引不是数组元素值**

索引值**没有特定的返回顺序**，但他们都能够指向对应的数组元素值。

22.2.3 删除数组变量

delete *array[index]*

22.3 使用模式

关键字 BEGIN 和 END 可以在读取数据流之前或之后执行命令。接下来用匹配模式来限制将脚本作用于哪些记录

22.3.1 正则表达式

用基础 (BRE) 或 扩展 (ERE) 正则表达式来筛选行

正则表达式出现在对应脚本的左花括号之前

```
$ cat data
a pretty flower
flower1
$ gawk '/flower/{print $1}' data
a
flower1
```

/flower/ 字段匹配了数组字段中含有字符串 flower 的记录

22.3.2 匹配操作符

匹配操作符 (~) 能将正则表达式限制在记录的特定数据字段。

$1 ~ /^data/

$1 变量代表记录中的第一个数据字段。表达式会过滤第一个数据字段以文本 data 开头的所有记录。

可以用其搜索特定的元素

```
$ gawk -F: '$1 ~ /rich/{print $1,$NF}' /etc/passwd
rich /bin/bash
```

在第一个数据字段中查找文本 rich ,若匹配则打印第一个数据字段和最后一个数据字段

可以用 ! 符号来排除正则表达式的匹配

22.3.3 数学表达式

想显示所有属于 root 用户组 (组 ID 为 0) 的用户

```
$ gawk -F: '$4 == 0{print $1}' /etc/passwd
root
```

== <= >= < > 都可以

可以对文本数据使用表达式。和正则表达式不同，表达式必须完全匹配 (相等)

22.4 结构化命令

22.4.1 if 语句

if (*condition*)

    statement

可以简写在一行

if (*condition*) *statement*

多条语句就加花括号

和 C 编程一样，但是换行时不需要在命令后加分号 (和shell一样)

22.4.2 while 语句

while (condition)

{

    statement

}

```
$ cat data
100 200 300
400 500 600
700 800 900
$ gawk '{
> total = 0
> i = 1
> while (i < 4)
> {
> total += $i
> i++
> }
> avg = total / 3
> print "average is ", avg
```

```
> }' data
200
500
800
(这里的平均数可以是小数)
```

在 while 中可以使用 break 和 continue

22.4.3 do-while 语句

do

{

    statement

} while (*condition*)

22.4.4 for 语句

```
$ gawk '{
> total = 0
> for (i = 1; i < 4; i++)
> {
>   total += $i
> }
> avg = total / 3
> print "average:",avg
> }' data
```

22.5 格式化打印

printf "*format string*", *var1*, *var2*

format string 会用文本元素和**格式说明符**

控制字母	描　　述
c	将数字作为 ASCII 字符显示
d	显示整数值
i	显示整数值（和 d 一样）
e	用科学计数法显示数字
f	显示浮点值
g	用科学计数法或浮点数显示（较短的格式优先）
o	显示八进制值
s	显示字符串
x	显示十六进制值
X	显示十六进制值，但用大写字母 A~F

和 C 一样

- width 指定输出字段的最小宽度。若短于此值，进行右对齐，空格填充。若长则按实际宽度来
- prec 指定浮点数中小数点右侧的位数或字符串中显示的最大字符数
- - (减号) 指明格式化空间中的数据采用左对齐而非右对齐

需要在 printf 的末尾手动添加 \n 换行符

{printf "%-16s %s\n", $1, $4}

22.6 内建函数

22.6.1 数学函数

函　数	描　述
atan2(*x*, *y*)	*x*/*y* 的反正切，*x* 和 *y* 以弧度为单位
cos(*x*)	*x* 的余弦，*x* 以弧度为单位
exp(*x*)	*x* 的指数
int(*x*)	*x* 的整数部分，取靠近 0 一侧的值
log(*x*)	*x* 的自然对数
rand( )	比 0 大且比 1 小的随机浮点值
sin(*x*)	*x* 的正弦，*x* 以弧度为单位
sqrt(*x*)	*x* 的平方根
srand(*x*)	为计算随机数指定一个种子值

gawk 对于其能处理的数值有一个限定区间

- and(*v1*, *v2*)：对 *v1* 和 *v2* 执行按位 AND 运算。
- compl(*val*)：对 *val* 执行补运算。
- lshift(*val*, *count*)：将 *val* 左移 count 位。
- or(*v1*, *v2*)：对 *v1* 和 *v2* 执行按位 OR 运算。
- rshift(*val*, *count*)：将 *val* 右移 count 位。
- xor(*v1*, *v2*)：对 *v1* 和 *v2* 执行按位 XOR 运算。

位操作函数在处理数据中的二进制值时特别有用。

22.6.2 字符串函数

函　数	描　述
asort(*s* [,*d*])	将数组 *s* 按照数组元素值排序。索引会被替换成表示新顺序的连续数字。另外，如果指定了 *d*，则排序后的数组会被保存在数组 *d* 中
asorti(*s* [,*d*])	将数组 *s* 按索引排序。生成的数组会将索引作为数组元素值，用连续数字索引表明排序顺序。另外，如果指定了 *d*，则排序后的数组会被保存在数组 *d* 中
gensub(*r*, *s*, *h* [,*t*])	针对变量$0 或目标字符串 *t*（如果提供了的话）来匹配正则表达式 *r*。如果 *h* 是一个以 *g* 或 *G* 开头的字符串，就用 *s* 替换匹配的文本。如果 *h* 是一个数字，则表示要替换 *r* 的第 *h* 处匹配
gsub(*r*, *s* [,*t*])	针对变量$0 或目标字符串 *t*（如果提供了的话）来匹配正则表达式 *r*。如果找到了，就将所有的匹配之处全部替换成字符串 *s*
index(*s*, *t*)	返回字符串 *t* 在字符串 *s* 中的索引位置；如果没找到，则返回 0
length([*s*])	返回字符串 *s* 的长度；如果没有指定，则返回$0 的长度

函　数	描　述
match(*s*, *r* [,*a*])	返回正则表达式 *r* 在字符串 *s* 中匹配位置的索引。如果指定了数组 *a*，则将 *s* 的匹配部分保存在该数组中
split(*s*, *a* [,*r*])	将 *s* 以 FS（字段分隔符）或正则表达式 *r*（如果指定了的话）分割并放入数组 *a* 中。返回分割后的字段总数
sprintf(*format*, *variables*)	用提供的 *format* 和 *variables* 返回一个类似于 printf 输出的字符串
sub(*r*, *s* [,*t*])	在变量$0 或目标字符串 *t* 中查找匹配正则表达式 *r* 的部分。如果找到了，就用字符串 *s* 替换第一处匹配
substr(*s*, *i* [,*n*])	返回 *s* 中从索引 *i* 开始、长度为 *n* 的子串。如果未提供 *n*，则返回 *s* 中剩下的部分
tolower(*s*)	将 *s* 中的所有字符都转换成小写
toupper(*s*)	将 *s* 中的所有字符都转换成大写

22.6.3 时间函数

函　数	描　述
mktime(*datespec*)	将一个按 YYYY MM DD HH MM SS [DST]格式指定的日期转换成时间戳[①]
strftime(*format* [, *timestamp*])	将当前时间的时间戳或 timestamp（如果提供了的话）转化为格式化日期（采用 shell 命令 date 的格式）
systime()	返回当前时间的时间戳

22.7 自定义函数

22.7.1 定义函数

function *name([variables])*

{

　　statements

}

函数名唯一

```
function getrand(limit)
{
    return int(limit * rand())
}
```

22.7.2 使用自定义函数

和 C 一样，若带参，参数在圆括号内

22.7.3 创建函数库

```
$ cat funclib
function name1()
{
...
}
function name2()
{
...
}
function name3()
{
...
}
```

加上 -f 命令行选项就可以使用该文件了

-f 选项不能和内联 gawk 脚本一起使用，但是可以在一个命令行中指定多个 -f 选项

二十三、实用其他 shell 编写脚本

二十四、编写简单的脚本实用工具

24.1 备份

也叫做归档

24.1.1 日常备份

可以创建一个 shell 脚本自动备份特定目录。避免从**主归档文件**执行耗时的恢复过程

1. 功能序求

使用 tar 命令将整个目录归档到单个文件中

```
$ tar -cf archive.tar /home/Sobloom/Project/*.*
```

tar 文件会占用大量的磁盘空间，最好清理一下。可以加上 -z 选项，使用 gzip 压缩 tar，由此生成的文件叫做 tarball，需要使用恰当的文件扩展名来表示这种文件是一个 tarball，.tar.gz 或者 .tgz 都行

```
$ tar -zcf archive.tgz Project/*.* 2>/dev/null
```

可以借助配置文件，包含我们希望纳入归档的所有目录的绝对路径

```
$ cat Files_To_Backup.txt
/home/code/...
/home/shell/...
...
```

为了让脚本读取配置文件，将每个目录名都加入配置列表中，使用 read 命令。不使用循环，使用 exec 命令来重定向 STDIN

exec 0 < $config_file

read file_name

配置文件中的每一条记录都会被读入。只要 read 命令在配置文件中发现还有记录可读，就会在？变量中返回一个表示成功的退出状态码0

这可以作为 while 循环的测试条件来读取配置文件中的所有记录

while [$? -eq 0]

do

[...]

read filename

done

一旦 read 命令读到配置文件的末尾，就会在？变量中返回一个非零状态码。脚本退出 while 循环

while 循环中，首先必须将目录名纳入归档列表，其次需要检查目录是否存在

if [-f $filename -o -d $filename]; then

    file_list="$file_list $file_name"

else

    echo ...

fi

file_no=$[$file_no + 1]

归档配置文件中的记录可以是文件名也可以是目录，if 语句会用 -f 和 -d 选项测试两者是否存在。 -o 选项使得其中一个测试为真即可

file_no 告诉我们归档配置文件中的哪一行含有不正确或缺失的文件，目录

2. 创建按日归档文件的存放位置

少量文件,在用户主目录下即可.若多人,最好在根目录下创建一个目录进行归档

创建好归档目录后,需要授予某些用户归档权限

用 sudo 命令创建一个用户组,来为归档目录中创建文件的用户授权

```
$ sudo groupadd Archivers
$ sudo chgrp Archivers /archive
$ ls -d /archive
$ sudo usermod -aG Archivers Sobloom
$ sudo chmod 775 /archive
$ ls -ld /archive
```

将用户添加到 archive 后,需要用户先登出后登入才能使组员关系生效

3. 创建按日归档的脚本

```
$ cat Daily_Archive.sh
#!/usr/bin/env bash
today=$(date +%y%m%d)
backupFile=archive$today.tgz
config_file=/archive/Files_To_Backup.txt
```

```
destination=/archive/$backupfile
if [ -f $config_file ]; then
    echo
else
    echo
    echo "config_file does not exist"
    echo "Backup not completed due to missing Configuration File"
    echo
    exit
fi

file_no=1
exec 0< $config_file
read file_name
while [ $? -eq 0 ]; do
    if [ -f $filename -o -d $file_name ]; then
        file_list="$file_list $file_name"
    else
        echo
        echo "$file_name does not exist"
        echo "It is listed on line $file_no of the config file"
        echo
    fi

    file_no=$[$file_no + 1]
    read file_name
done

echo "Starting archive"
echo

tar -czf $destination $file_list 2> /dev/null

echo "Archive completed"
echo "Resulting archive file is: $destination"
echo

exit
```

4. 运行按日归档的脚本

chmod u+x Daily_Archive.sh

24.1.2 创建按小时归档的脚本