Linux 简介

严格的来讲，Linux 不算是一个操作系统，只是一个 Linux 系统中的**内核**，即计算机软件与硬件通讯之间的平台；Linux 的全称是 GNU/Linux，这才算是一个真正意义上的 Linux 系统。GNU 是 Richard Stallman 组织的一个项目，世界各地的程序员可以变形 GNU 程序，同时遵循 GPL 协议，允许任何人任意改动。但是，修改后的程序必须遵循 GPL 协议。

Linux 是一个多用户多任务的操作系统，也是一款自由软件，完全兼容 POSIX 标准，拥有良好的用户界面，支持多种处理器架构，移植方便。

为程序分配系统资源，处理计算机内部细节的软件叫做操作系统或者内核。如果你希望详细了解操作系统的概念用户通过 Shell 与 Linux 内核交互。Shell 是一个命令行解释工具（是一个软件），它将用户输入的命令转换为内核能够理解的语言（命令）。

Linux 下，很多工作都是通过命令完成的，学好 Linux，首先要掌握常用命令。

Linux 版本

内核版本指的是在 Linus 领导下的开发小组开发出的系统内核的版本号。Linux 的每个内核版本使用形式为 x.y.zz-www 的一组数字来表示。其中：

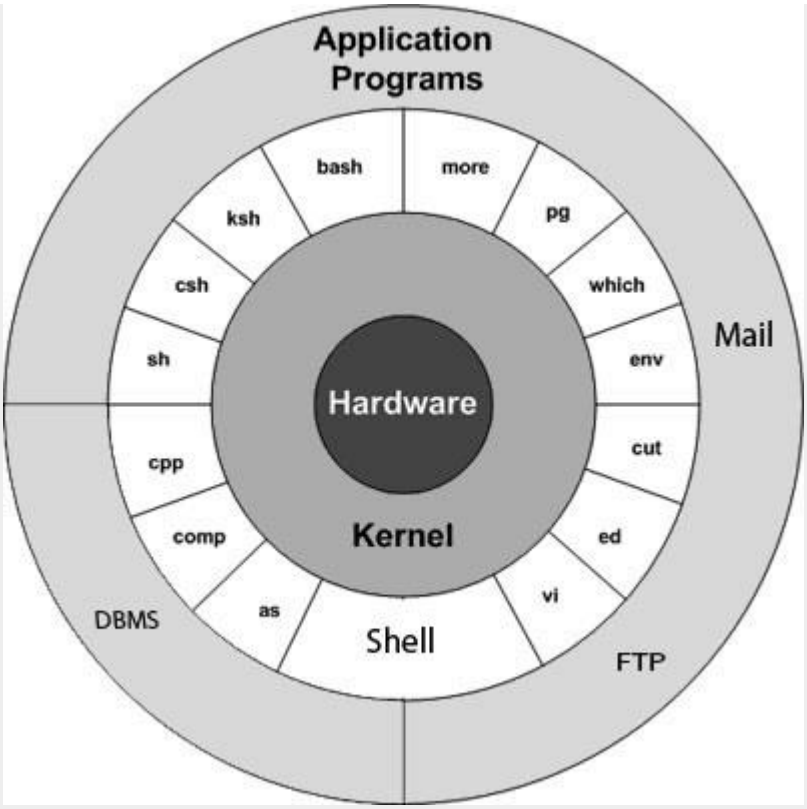
- x.y：为 linux 的主版本号。通常 y 若为奇数，表示此版本为测试版，系统会有较多 bug，主要用途是提供给用户测试。
- zz：为次版本号。
- www：代表发行号（注意，它与发行版本号无关）。

当内核功能有一个飞跃时，主版本号升级，如 Kernel2.2、2.4、2.6 等。如果内核增加了少量补丁时，常常会升级次版本号，如 Kernel2.6.15、2.6.20 等。

一些组织或厂家将 Linux 内核与 GNU 软件（系统软件和工具）整合起来，并提供一些安装界面和系统设定与管理工具，这样就构成了一个发型套件，例如 Ubuntu、Red Hat、Centos、Fedora、SUSE、Debian、FreeBSD 等。相对于内核版本，发行套件的版本号随着发布者的不同而不同，与系统内核的版本号是相对独立的。因此把 Red Hat 等直接说成是 Linux 是不确切的，它们是 Linux 的发行版本，更确切地说，应该叫做“以 linux 为核心的操作系统软件包”。

Linux 体系结构

下面是 Linux 体系结构的示意图：



在所有 Linux 版本中，都会涉及到以下几个重要概念：

- 内核：内核是操作系统的核心。内核直接与硬件交互，并处理大部分较低层的任务，如内存管理、进程调度、文件管理等。
- Shell：Shell 是一个处理用户请求的工具，它负责解释用户输入的命令，调用用户希望使用的程序。
- 命令和工具：日常工作中，你会用到很多系统命令和工具，如 cp、mv、cat 和 grep 等。在 Linux 系统中，有 250 多个命令，每个命令都有多个选项；第三方工具也有很多，他们也扮演着重要角色。
- 文件和目录：Linux 系统中所有的数据都被存储到文件中，这些文件被分配到各个目录，构成文件系统。Linux 的目录与 Windows 的文件夹是类似的概念。

系统启动（开机）

如果你有一台装有 Linux 的电脑，加电后系统会自动启动，然后提示你登录系统，只有登录后才能进行其他操作。

登录 Linux

第一次使用 Linux，会看到登录的提示，如下所示：

```
login:
```

登录步骤：

- 登录 Linux 必须有用户名（用户 ID）和密码，如果没有，可以向管理员所要。

- 在登录提示处输入用户名并回车；用户名是区分大小写的，输入时要注意。
- 然后会提示你输入密码，密码也是区分大小写的。
- 如果用户名和密码正确，那么会成功登录，并看到上次登录信息。

```
login : amrood
amrood's password:
Last login: Sun Jun 14 09:32:32 2009 from 62.61.164.73
$
```

登录后会出现命令提示符($)，你可以输入任何命令。下面通过 cal 命令来查看日历：

```
$ cal
     June 2009
Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6
7   8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30

$
```

修改密码

Linux 系统通过密码来保证数据和文件的安全，防止黑客破解和攻击。你可以通过以下方法来修改密码：

- 输入 password 命令。
- 输入你现在使用的密码。
- 输入新密码。注意密码不要过于简单，简单的密码往往会为入侵者大开便利之门。
- 确认密码，再输入一遍刚才的密码。

```
$ passwd
Changing password for amrood
(current) Linux password:******
New Linux password:*******
Retype new Linux password:*******
passwd: all authentication tokens updated  successfully

$
```

注意：输入的密码是看不到的，只会看到一个占位符(*)。

查看目录和文件

在 Linux 中，所有的数据都被保存在文件中，所有的文件又被分配到不同的目录；目录是一种类似树的结构，称为文件系统。

你可以使用 ls 命令来查看当前目录下的文件和目录。下面的例子，使用了 ls 命令的 -l 选项：

```
$ ls -l
total 19621
drwxrwxr-x  2 amrood amrood      4096 Dec 25 09:59 uml
-rw-rw-r--  1 amrood amrood      5341 Dec 25 08:38 uml.jpg
drwxr-xr-x  2 amrood amrood      4096 Feb 15  2006 univ
drwxr-xr-x  2 root   root        4096 Dec  9  2007 urlspedia
-rw-r--r--  1 root   root      276480 Dec  9  2007 urlspedia.tar
drwxr-xr-x  8 root   root        4096 Nov 25  2007 usr
-rwxr-xr-x  1 root   root        3192 Nov 25  2007 webthumb.php
-rw-rw-r--  1 amrood amrood     20480 Nov 25  2007 webthumb.tar
-rw-rw-r--  1 amrood amrood      5654 Aug  9  2007 yourfile.mid
-rw-rw-r--  1 amrood amrood    166255 Aug  9  2007 yourfile.swf

$putty 中文版
```

注意：以 d* 开头的为目录，如 uml、univ、urlspedia 等；其他的都是文件。

查看当前用户信息

登录系统后，如果你希望知道自己的用户名（用户 ID），可以使用 whoami 命令：

```
$ whoami
amrood

$
```

如果你希望了解更多关于当前用户的信息，可以使用 who am i 命令，读者可以自己尝试一下。

查看当前在线用户

如果你希望知道当前在线的用户（同时登录到系统的用户），可以使用 users、who 和 w 命令：

```
$ users
amrood bablu qadir
```

```
$ who
amrood ttyp0 Oct 8 14:10 (limbo)
bablu  ttyp2 Oct 4 09:08 (calliope)
qadir  ttyp4 Oct 8 12:09 (dent)


$
```

w 命令可以看到在线用户的更多信息，读者可以自己尝试。

退出登录

完成工作后，你需要退出系统，防止他人使用你的账户。

使用 logout 命令即可退出登录，系统会清理有关信息并断开连接。

关闭系统（关机）

关系 Linux 系统可以使用下列命令：

命令	说明
halt	直接关闭系统
init 0	使用预先定义的脚本关闭系统，关闭前可以清理和更新有关信息
init 6	重新启动系统
poweroff	通过断电来关闭系统
reboot	重新启动系统
shutdown	安全关闭系统

注意：一般情况下只有超级用户和 root 用户（Linux 系统中的最高特权用户）才有关闭系统的权限，但是给普通用户赋予相应权限也可以关闭系统。

Linux 文件管理

Linux 中的所有数据都被保存在文件中，所有的文件被分配到不同的目录。目录是一种类似于树的结构，称为文件系统。

当你使用 Linux 时，大部分时间都会和文件打交道，通过本节可以了解基本的文件操作，如创建文件、删除文件、复制文件、重命名文件以及为文件创建链接等。

在 Linux 中，有三种基本的文件类型：

1）普通文件

普通文件是以字节为单位的数据流，包括文本文件、源码文件、可执行文件等。文本和二进制对 Linux 来说并无区别，对普通文件的解释由处理该文件的应用程序进行。

2）目录

目录可以包含普通文件和特殊文件，目录相当于 Windows 和 Mac OS 中的文件夹。

3）设备文件

有些教程中称特殊文件，是一个含义。Linux 与外部设备（例如光驱，打印机，终端，modern 等）是通过一种被称为设备文件的文件来进行通信。Linux 输入输出到外部设备的方式和输入输出到一个文件的方式是相同的。Linux 和一个外部设备通讯之前，这个设备必须首先要有一个设备文件存在。

例如，每一个终端都有自己的设备文件来供 Linux 写数据（出现在终端屏幕上）和读取数据（用户通过键盘输入）。

设备文件和普通文件不一样，设备文件中并不包含任何数据。

设备文件有两种类型：字符设备文件和块设备文件。

- 字符设备文件以字母"c"开头。字符设备文件向设备传送数据时，一次传送一个字符。典型的通过字符传送数据的设备有终端、打印机、绘图仪、modern 等。字符设备文件有时也被称为"raw"设备文件。
- 块设备文件以字母"b"开头。块设备文件向设备传送数据时，先从内存中的 buffer 中读或写数据，而不是直接传送数据到物理磁盘。磁盘和 CD-ROMS 既可以使用字符设备文件也可以使用块设备文件。

查看文件

查看当前目录下的文件和目录可以使用 **ls** 命令，例如：

```
$ls


bin         hosts  lib      res.03
ch07        hw1    pub      test_results
ch07.bak    hw2    res.01   users
docs        hw3    res.02   work
```

通过 **ls** 命令的 **-l** 选项，你可以获取更多文件信息，例如：

```
$ls -l
total 1962188


drwxrwxr-x  2 amrood amrood      4096 Dec 25 09:59 uml
-rw-rw-r--  1 amrood amrood      5341 Dec 25 08:38 uml.jpg
drwxr-xr-x  2 amrood amrood      4096 Feb 15  2006 univ
drwxr-xr-x  2 root   root        4096 Dec  9  2007 urlspedia
```

```
-rw-r--r--  1 root   root      276480 Dec  9  2007 urlspedia.tar
drwxr-xr-x  8 root   root        4096 Nov 25  2007 usr
drwxr-xr-x  2    200    300      4096 Nov 25  2007 webthumb-1.01
-rwxr-xr-x  1 root   root        3192 Nov 25  2007 webthumb.php
-rw-rw-r--  1 amrood amrood     20480 Nov 25  2007 webthumb.tar
-rw-rw-r--  1 amrood amrood      5654 Aug  9  2007 yourfile.mid
-rw-rw-r--  1 amrood amrood    166255 Aug  9  2007 yourfile.swf
drwxr-xr-x 11 amrood amrood      4096 May 29  2007 zlib-1.2.3
$
```

每一列的含义如下：

- 第一列：文件类型。
- 第二列：表示文件个数。如果是文件，那么就是 1；如果是目录，那么就是该目录中文件的数目。
- 第三列：文件的所有者，即文件的创建者。
- 第四列：文件所有者所在的用户组。在 Linux 中，每个用户都隶属于一个用户组。
- 第五列：文件大小（以字节计）。
- 第六列：文件被创建或上次被修改的时间。
- 第七列：文件名或目录名。

注意：每一个目录都有一个指向它本身的子目录"." 和指向它上级目录的子目录"..", 所以对于一个空目录，第二列应该为 2。

通过 **ls -l** 列出的文件，每一行都是以 a、d、- 或 l 开头，这些字符表示文件类型：

前缀	描述
-	普通文件。如文本文件、二进制可执行文件、源代码等。
b	块设备文件。硬盘可以使用块设备文件。
c	字符设备文件。硬盘也可以使用字符设备文件。
d	目录文件。目录可以包含文件和其他目录。
l	符号链接（软链接）。可以链接任何普通文件，类似于 Windows 中的快捷方式。
p	具名管道。管道是进程间的一种通信机制。
s	用于进程间通信的套接字。

提示：通俗的讲软连接就是 windows 的快捷方式，原来文件删了，快捷方式虽然在但是不起作用了。

元字符

元字符是具有特殊含义的字符。* 和 ？ 都是元字符：

- * 可以匹配 0 个或多个任意字符；
- ？匹配一个字符。

例如

```
$ls ch*.doc
```

可以显示所有以 ch 开头，以 .doc 结尾的文件：

```
ch01-1.doc    ch010.doc   ch02.doc     ch03-2.doc
ch04-1.doc    ch040.doc   ch05.doc     ch06-2.doc
ch01-2.doc ch02-1.doc c
```

这里，* 匹配任意一个字符。如果你希望显示所有以 .doc 结尾的文件，可以使用

```
$ls *.doc。
```

隐藏文件

隐藏文件的第一个字符为英文句号或点号(.)，Linux 程序（包括 Shell）通常使用隐藏文件来保存配置信息。

下面是一些常见的隐藏文件：
.profile：Bourne shell (sh) 初始化脚本
.kshrc：Korn shell (ksh) 初始化脚本
.cshrc：C shell (csh) 初始化脚本
.rhosts：Remote shell (rsh) 配置文件

查看隐藏文件需要使用 **ls** 命令的 **-a** 选项：

```
$ ls -a


.         .profile       docs     lib     test_results
..        .rhosts        hosts    pub     users
.emacs    bin            hw1      res.01  work
.exrc     ch07           hw2      res.02
.kshrc    ch07.bak       hw3      res.03
$
```

一个点号(.)表示当前目录，两个点号(..)表示上级目录

注意：输入密码时，星号(*)作为占位符，代表你输入的字符个数。

创建文件

在 Linux 中，可以使用 vi 编辑器创建一个文本文件，例如：

```
$ vi filename
```

上面的命令会创建文件 filename 并打开，按下 i 键即可进入编辑模式，你可以向文件中写入内容。例如：

```
This is Linux file....I created it for the first time.....
I'm going to save this content in this file.
```

完成编辑后，可以按 esc 键退出编辑模式，也可以按组合键 Shift + ZZ 完全退出文件。这样，就完成了文件的创建。

```
$ vi filename
$
```

编辑文件

vi 编辑器可以用来编辑文件。由于篇幅限制，这里仅作简单介绍，将在后面章节进行详细讲解。

如下可以打开一个名为 filename 的文件：

```
$ vi filename
```

当文件被打开后，可以按 i 键进入编辑模式，按照自己的方式编辑文件。如果想移动光标，必须先按 esc 键退出编辑模式，然后使用下面的按键在文件内移动光标：

- l 键向右移动
- h 键向左移动
- k 键向上移动
- j 键向下移动

使用上面的按键，可以将光标快速定位到你想编辑的地方。定位好光标后，按 i 键再次进入编辑模式。编辑完成后按 esc 键退出编辑模式或者按组合键 Shift+ZZ 退出当前文件。

查看文件内容

可以使用 **cat** 命令来查看文件内容，下面是一个简单的例子：

```
$ cat filename
This is Linux file....I created it for the first time.....
I'm going to save this content in this file.
$
```

可以通过 **cat** 命令的 **-b** 选项来显示行号，例如：

```
$ cat -b filename
1   This is Linux file....I created it for the first time.....
2   I'm going to save this content in this file.
$
```

统计单词数目

可以使用 **wc** 命令来统计当前文件的行数、单词数和字符数，下面是一个简单的例子：

```
$ wc filename
2  19 103 filename
$
```

每一列的含义如下：

- 第一列：文件的总行数
- 第二列：单词数目
- 第三列：文件的字节数，即文件的大小
- 第四列：文件名

也可以一次查看多个文件的内容，例如：

```
$ wc filename1 filename2 filename3
```

复制文件

可以使用 **cp** 命令来复制文件。**cp** 命令的基本语法如下：

```
$ cp source_file destination_file
```

下面的例子将会复制 filename 文件：

```
$ cp filename copyfile
$
```

现在在当前目录中会多出一个和 filename 一模一样的 copyfile 文件。

重命名文件

重命名文件可以使用 **mv** 命令，语法为：

```
$ mv old_file new_file
```

下面的例子将会把 filename 文件重命名为 newfile：

```
$ mv filename newfile
```

```
$
```

现在在当前目录下，只有一个 newfile 文件。

mv 命令其实是一个移动文件的命令，不但可以更改文件的路径，也可以更改文件名。

删除文件

rm 命令可以删除文件，语法为：

```
$ rm filename
```

注意：删除文件是一种危险的行为，因为文件内可能包含有用信息，建议结合 **-i** 选项来使用 **rm** 命令。

下面的例子会彻底删除一个文件：

```
$ rm filename
$
```

你也可以一次删除多个文件：

```
$ rm filename1 filename2 filename3
$
```

标准的 Linux 流

一般情况下，每个 Linux 程序运行时都会创建三个文件流（三个文件）：

- 标准输入流(stdin)：stdin 的文件描述符为 0，Linux 程序默认从 stdin 读取数据。
- 标准输出流(stdout)：stdout 的文件描述符为 1，Linux 程序默认向 stdout 输出数据。
- 标准错误流(stderr)：stderr 的文件描述符为 2，Linux 程序会向 stderr 流中写入错误信息。

Linux 目录

目录也是一个文件，它的唯一功能是用来保存文件及其相关信息。所有的文件，包括普通文件、设备文件和目录文件，都会被保存到目录中。

主目录

登录后，你所在的位置就是你的主目录（或登录目录），接下来你主要是在这个目录下进行操作，如创建文件、删除文件等。

使用下面的命令可以随时进入主目录：

```
$cd ~
$
```

这里 ~ 就表示主目录。如果你希望进入其他用户的主目录，可以使用下面的命令：

```
$cd ~username
$
```

返回进入当前目录前所在的目录可以使用下面的命令：

```
$cd -
$
```

绝对路径和相对路径

Linux 的目录有清晰的层次结构，/ 代表根目录，所有的目录都位于 / 下面；文件在层次结构中的位置可以用路径来表示。

如果一个路径以 / 开头，就称为绝对路径；它表示当前文件与根目录的关系。举例如下：

```
/etc/passwd
/users/sjones/chem/notes
/dev/rdsk/0s3
```

不以 / 开头的路径称为相对路径，它表示文件与当前目录的关系。例如：

```
chem/notes
personal/res
```

获取当前所在的目录可以使用 pwd 命令：

```
$pwd
/user0/home/amrood


$
```

查看目录中的文件可以使用 **ls** 命令：

```
$ls dirname
```

下面的例子将遍历 /usr/local 目录下的文件：

```
$ls /usr/local


X11       bin          gimp       jikes       sbin
ace       doc          include    lib         share
atalk     etc          info       man         ami
```

创建目录

可以使用 **mkdir** 命令来创建目录，语法为：

```
$mkdir dirname
```

dirname 可以为绝对路径，也可以为相对路径。例如

```
$mkdir mydir
$
```

会在当前目录下创建 mydir 目录。又如

```
$mkdir /tmp/test-dir
$
```

会在 /tmp 目录下创建 test-dir 目录。**mkdir** 成功创建目录后不会输出任何信息。

也可以使用 **mkdir** 命令同时创建多个目录，例如

```
$mkdir docs pub
$
```

会在当前目录下创建 docs 和 pub 两个目录。

创建父目录

使用 **mkdir** 命令创建目录时，如果上级目录不存在，就会报错。下面的例子中，mkdir 会输出错误信息：

```
$mkdir /tmp/amrood/test
mkdir: Failed to make directory "/tmp/amrood/test";
No such file or directory
$
```

为 **mkdir** 命令增加 **-p** 选项，可以一级一级创建所需要的目录，即使上级目录不存在也不会报错。例如

```
$mkdir -p /tmp/amrood/test
$
```

会创建所有不存在的上级目录。

删除目录

可以使用 **rmdir** 命令来删除目录，例如：

```
$rmdir dirname
$
```

注意：删除目录时请确保目录为空，不会包含其他文件或目录。

也可以使用 **rmdir** 命令同时删除多个目录：

```
$rmdir dirname1 dirname2 dirname3
$
```

如果 dirname1、dirname2、dirname3 为空，就会被删除。**rmdir** 成功删除目录后不会输出任何信息。

改变所在目录

可以使用 **cd** 命令来改变当前所在目录，进入任何有权限的目录，语法为：

```
$cd dirname
```

dirname 为路径，可以为相对路径，也可以为绝对路径。例如

```
$cd /usr/local/bin
$
```

可以进入 /usr/local/bin 目录。可以使用相对路径从这个目录进入 /usr/home/amrood 目录：

```
$cd ../../home/amrood
$
```

重命名目录

mv (move) 命令也可以用来重命名目录，语法为：

```
$mv olddir newdir
```

下面的例子将会把 mydir 目录重命名为 yourdir 目录：

```
$mv mydir yourdir
$
```

点号(.)

一个点号(.)表示当前目录，两个点号(..)表示上级目录（父目录）。

ls 命令的 **-a** 选项可以查看所有文件，包括隐藏文件；**-l** 选项可以查看文件的所有信息，共有 7 列。例如：

```
$ls -la
drwxrwxr-x    4    teacher   class   2048  Jul 16 17.56 .
drwxr-xr-x    60   root              1536  Jul 13 14:18 ..
----------    1    teacher   class   4210  May 1 08:27 .profile
-rwxr-xr-x    1    teacher   class   1948  May 12 13:42 memo
$
```

Linux 文件权限和访问模式

为了更加安全的存储文件，Linux 为不同的文件赋予了不同的权限，每个文件都拥有下面三种权限：

- 所有者权限：文件所有者能够进行的操作
- 组权限：文件所属用户组能够进行的操作
- 外部权限（其他权限）：其他用户可以进行的操作。

查看文件权限

使用 **ls -l** 命令可以查看与文件权限相关的信息：

```
$ls -l /home/amrood
-rwxr-xr--  1 amrood   users 1024  Nov 2 00:10  myfile
drwxr-xr--- 1 amrood   users 1024  Nov 2 00:10  mydir
```

第一列就包含了文件或目录的权限。

第一列的字符可以分为三组，每一组有三个，每个字符都代表不同的权限，分别为读取(r)、写入(w)和执行(x)：

- 第一组字符(2-4)表示文件所有者的权限，-rwxr-xr-- 表示所有者拥有读取(r)、写入(w)和执行(x)的权限。
- 第二组字符(5-7)表示文件所属用户组的权限，-rwxr-xr-- 表示该组拥有读取(r)和执行(x)的权限，但没有写入权限。
- 第三组字符(8-10)表示所有其他用户的权限，rwxr-xr-- 表示其他用户只能读取(r)文件。

文件访问模式

文件权限是 Linux 系统的第一道安全防线，基本的权限有读取(r)、写入(w)和执行(x)：

- 读取：用户能够读取文件信息，查看文件内容。
- 写入：用户可以编辑文件，可以向文件写入内容，也可以删除文件内容。
- 执行：用户可以将文件作为程序来运行。

目录访问模式

目录的访问模式和文件类似，但是稍有不同：

- 读取：用户可以查看目录中的文件
- 写入：用户可以在当前目录中删除文件或创建文件
- 执行：执行权限赋予用户遍历目录的权利，例如执行 cd 和 ls 命令。

改变权限

可以使用 **chmod** (change mode) 命令来改变文件或目录的访问权限，权限可以使用符号或数字来表示。

使用符号表示权限

对于初学者来说最简单的就是使用符号来改变文件或目录的权限，你可以增加(+)和删除(-)权限，也可以指定特定权限。

g:表示所在组，u 表示文件的所有者，o 表示其他用户 a 表示所有用户

符号	说明
+	为文件或目录增加权限
-	删除文件或目录的权限
=	设置指定的权限

下面的例子将会修改 testfile 文件的权限：

```
$ls -l testfile
-rwxrwxr--  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod o+wx testfile
$ls -l testfile
-rwxrwxrwx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod u-x testfile
$ls -l testfile
-rw-rwxrwx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod g=rx testfile
$ls -l testfile
-rw-r-xrwx  1 amrood   users 1024  Nov 2 00:10  testfile
```

也可以同时使用多个符号：

```
$chmod o+wx,u-x,g=rx testfile
$ls -l testfile
-rw-r-xrwx  1 amrood   users 1024  Nov 2 00:10  testfile
```

使用数字表示权限

除了符号，也可以使用八进制数字来指定具体权限，如下表所示：

数字	说明	权限
0	没有任何权限	---
1	执行权限	--x
2	写入权限	-w-

3	执行权限和写入权限：1 (执行) + 2 (写入) = 3	-wx
4	读取权限	r--
5	读取和执行权限：4 (读取) + 1 (执行) = 5	r-x
6	读取和写入权限：4 (读取) + 2 (写入) = 6	rw-
7	所有权限: 4 (读取) + 2 (写入) + 1 (执行) = 7	rwx

下面的例子，首先使用 **ls -1** 命令查看 testfile 文件的权限，然后使用 **chmod** 命令更改权限：

```
$ls -l testfile
-rwxrwxr--  1 amrood   users 1024  Nov 2 00:10  testfile
$ chmod 755 testfile
$ls -l testfile
-rwxr-xr-x  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod 743 testfile
$ls -l testfile
-rwxr---wx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod 043 testfile
$ls -l testfile
----r---wx  1 amrood   users 1024  Nov 2 00:10  testfile
```

更改所有者和用户组

在 Linux 中，每添加一个新用户，就会为它分配一个用户 ID 和群组 ID，上面提到的文件权限也是基于用户和群组来分配的。

有两个命令可以改变文件的所有者或群组：

- **chown**：chown 命令是"change owner"的缩写，用来改变文件的所有者。
- **chgrp**：chgrp 命令是"change group"的缩写，用来改变文件所在的群组。

chown 命令用来更改文件所有者，其语法如下：

```
$ chown user filelist
```

user 可以是用户名或用户 ID，例如

```
$ chown amrood testfile
$
```

将 testfile 文件的所有者改为 amrood。

注意：超级用户 root 可以不受限制的更改文件的所有者和用户组，但是普通用户只能更改所有者是自己的文件或目录。

chgrp 命令用来改变文件所属群组，其语法为：

```
$ chgrp group filelist
```

group 可以是群组名或群组 ID，例如

```
$ chgrp special testfile
$
```

将文件 testfile 的群组改为 special。

用一个命令同时修改所有者和所有组
chown shiy:shiy b.txt

SUID 和 SGID 位

在 Linux 中，一些程序需要特殊权限才能完成用户指定的操作。

例如，用户的密码保存在 /etc/shadow 文件中，出于安全考虑，一般用户没有读取和写入的权限。但是当我们使用 **passwd** 命令来更改密码时，需要对 /etc/shadow 文件有写入权限。这就意味着，passwd 程序必须要给我们一些特殊权限，才可以向 /etc/shadow 文件写入内容。

Linux 通过给程序设置 SUID(Set User ID)和 SGID(Set Group ID)位来赋予普通用户特殊权限。当我们运行一个带有 SUID 位的程序时，就会继承该程序所有者的权限；如果程序不带 SUID 位，则会根据程序使用者的权限来运行。

SGID 也是一样。一般情况下程序会根据你的组权限来运行，但是给程序设置 SGID 后，就会根据程序所在组的组权限运行。

如果程序设置了 SUID 位，就会在表示文件所有者可执行权限的位置上出现's'字母；同样，如果设置了 SGID，就会在表示文件群组可执行权限的位置上出现's'字母。如下所示：

```
$ ls -l /usr/bin/passwd
-r-sr-xr-x  1   root   bin  19031 Feb 7 13:47  /usr/bin/passwd*
$
```

上面第一列第四个字符不是'x'或'-'，而是's'，说明 /usr/bin/passwd 文件设置了 SUID 位，这时普通用户会以 root 用户的权限来执行 passwd 程序。

注意：小写字母's'说明文件所有者有执行权限(x)，大写字母'S'说明程序所有者没有执行权限(x)。

如果在表示群组权限的位置上出现 SGID 位，那么也仅有三类用户可以删除该目录下的文件：目录所有者、文件所有者、超级用户 root。

为一个目录设置 SUID 和 SGID 位可以使用下面的命令：

```
$ chmod ug+s dirname
$ ls -l
drwsr-sr-x 2 root root  4096 Jun 19 06:45 dirname
$
```

Linux 环境变量

在 Linux 中，环境变量是一个很重要的概念。环境变量可以由系统、用户、Shell 以及其他程序来设定。

变量就是一个可以被赋值的字符串，赋值范围包括数字、文本、文件名、设备以及其他类型的数据。

下面的例子，我们将为变量 TEST 赋值，然后使用 **echo** 命令输出：

```
$TEST="Linux Programming"
$echo $TEST
Linux Programming
```

注意：变量赋值时前面不能加 $ 符号，变量输出时必须要加 $ 前缀。退出 Shell 时，变量将消失。

登录系统后，Shell 会有一个初始化的过程，用来设置环境变量。这个阶段，Shell 会读取 /etc/profile 和 .profile 两个文件，过程如下：

- Shell 首先检查 /etc/profile 文件是否存在，如果存在，就读取内容，否则就跳过，但是不会报错。
- 然后检查你的主目录（登录目录）中是否存在 .profile 文件，如果存在，就读取内容，否则就跳过，也不会报错。

```
PS1='[\u\u@aaaffffffaaa\h \t \w]\$'
export PS1
```

下表中的转义字符可以被用作 PS1 的参数，丰富命令提示符信息。

转义字符	描述
\t	当前时间，格式为 HH:MM:SS
\d	当前日期，格式为 Weekday Month Date
\n	换行
\W	当前所在目录
\w	当前所在目录的完整路径
\u	用户名
\h	主机名（IP 地址）

#	输入的命令的个数，每输入一个新的命令就会加 1
\$	如果是超级用户 root，提示符为#，否则为$。

source .profile
读取完上面两个文件，Shell 就会出现 $ 命令提示符：

```
$
```

出现这个提示符，就可以输入命令并调用相应的程序了。

注意：上面是 Bourne Shell 的初始化过程，bash 和 ksh 在初始化过程中还会检查其他文件。

.profile 文件

/etc/profile 文件包含了通用的 Shell 初始化信息，由 Linux 管理员维护，一般用户无权修改。

但是你可以修改主目录下的 .profile 文件，增加一些“私人定制”初始化信息，包括：

- 设置默认终端类型和外观样式；
- 设置 Shell 命令查找路径，即 PATH 变量；
- 设置命令提示符。

找到主目录下的 .profile 文件，使用 vi 编辑器打开并查看内容。

设置终端类型

一般情况下，我们使用的终端是由 login 或 getty 程序设置的，可能会不符合我们的习惯。

登陆信息修改: vi /etc/motd

对于没有使用过的终端，可能会比较生疏，不习惯命令的输出样式，交互起来略显吃力。所以，一般用户会将终端设置成下面的类型：

```
$TERM=vt100
$
```

vt100 是 virtual terminate 100 的缩写。虚拟终端是一种假的终端，真正有自己的显示器和键盘的终端，会通过特殊电缆 (如串口) 连到计算机主机。vt100 是被绝大多数 Linux 系统所支持的一种虚拟终端规范，常用的还有 ansi、xterm 等。

设置 PATH 变量

在命令提示符下输入一个命令时，Shell 会根据 PATH 变量来查找该命令对应的程序，PATH 变量指明了这些程序所在的路径。

一般情况下 PATH 变量的设置如下：

```
$PATH=/bin:/usr/bin
$
```

多个路径使用冒号(:)分隔。如果用户输入的命令在 PATH 设置的路径下没有找到，就会报错，例如：

```
$hello
hello: not found
$
```

常用环境变量

下表列出了部分重要的环境变量，这些变量可以通过上面提到的方式修改。

变量	描述
DISPLAY	用来设置将图形显示到何处。
HOME	当前用户的主目录。
IFS	内部域分隔符。
LANG	LANG 可以让系统支持多语言。例如，将 LANG 设为 pt_BR，则可以支持(巴西)葡萄牙语。
PATH	指定 Shell 命令的路径。
PWD	当前所在目录，即 cd 到的目录。
RANDOM	生成一个介于 0 和 32767 之间的随机数。
TERM	设置终端类型。
TZ	时区。可以是 AST(大西洋标准时间)或 GMT(格林尼治标准时间)等。
UID	以数字形式表示的当前用户 ID，shell 启动时会被初始化。

下面的例子中使用了部分环境变量：

```
$ echo $HOME
/root
]$ echo $DISPLAY


$ echo $TERM
xterm
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/home/amrood/bin:/usr/local/bin
```

```
$
```

Linux 管道和过滤器

有时候，我们可以把两个命令连起来使用，一个命令的输出作为另一个命令的输入，这就叫做**管道**。为了建立管道，需要在两个命令之间使用竖线(|)连接。

管道是 Linux 进程之间一种重要的通信机制；除了管道，还有共享内存、消息队列、信号、套接字(socket)等进程通信机制。

管道使用竖线(|)将两个命令隔开，竖线左边命令的输出就会作为竖线右边命令的输入。连续使用竖线表示第一个命令的输出会作为第二个命令的输入，第二个命令的输出又会作为第三个命令的输入，依此类推。

能够接受数据，过滤（处理或筛选）后再输出的工具，称为**过滤器**。

grep 命令

grep 是一个强大的文本搜索工具，可以使用正则表达式，并返回匹配的行，语法为：

```
$grep pattern file(s)
```

“grep”源于 ed(Linux 的一个行文本编辑器)的 g/re/p 命令，g/re/p 是“globally search for a regular expression and print all lines containing it”的缩写，意思是使用正则表达式进行全局检索，并把匹配的行打印出来。

正则表达式是一个包含了若干特殊字符的字符串，每个字符都有特殊含义，可以用来匹配文本，更多信息请查看正则表达式教程。

grep 可以看做是一个过滤器，如果没有为 grep 指定要检索的文件，那么它会从标准输入设备（一般是键盘）读取；其他过滤器也是如此。

grep 命令最简单的使用就是检索包含固定字符的文本。

例如，在管道中使用 grep 命令，只允许包含指定字符的行输出到显示器：

```
$ls -l | grep "Aug"
-rw-rw-rw-   1 john  doc     11008 Aug  6 14:10 ch02
-rw-rw-rw-   1 john  doc      8515 Aug  6 15:30 ch07
-rw-rw-r--   1 john  doc      2488 Aug 15 10:51 intro
-rw-rw-r--   1 carol doc      1605 Aug 23 07:35 macros
$
```

grep 命令有很多选项：

选项	说明

-v	反转查询，输出不匹配的行。例如，grep -v "test" demo.txt 将输出不包含"test"的行。
-n	输出匹配的行以及行号。
-l	输出匹配的行所在的文件名。
-c	输出匹配的总行数。
-i	不区分大小写进行匹配。

下面我们使用正则表达式来匹配这样的行：包含字符“carol”，然后包含任意数目（含零个）的其他字符，最后还要包含“Aug”。

使用 -i 选项进行不区分大小写的匹配：

```
$ls -l | grep -i "carol.*aug"
-rw-rw-r--   1 carol doc      1605 Aug 23 07:35 macros
$
```

sort 命令

sort 命令在 Linux 中非常有用，它将文件中的各行按字母或数进行排序。sort 命令既可以从特定的文件，也可以从 stdin 获取输入。

例如，对 foot 文件的各行进行排序：

```
$sort food
Afghani Cuisine
Bangkok Wok
Big Apple Deli
Isle of Java
Mandalay
Sushi and Sashimi
Sweet Tooth
Tio Pepe's Peppers
$
```

通过下面的选项可以控制排序规则：

选项	描述
-n	按照数字大小排序，例如，10 会排在 2 后面；-n 选项会忽略空格或 tab 缩进。
-r	降序排序。sort 默认是升序排序。
-f	不区分大小写。

+x	对第 x 列（从 0 开始）进行排序。
-k	按照第几列进行排序
-t	按照哪个字符来分隔

下面的例子通过管道将 ls、grep 和 sort 命令连起来使用，过滤包含“Aug”的行，并按照文件大小排序：

```
$ls -l | grep "Aug" | sort +4n
ll | sort -n -k 5 -t " "
-rw-rw-r--  1 carol doc      1605 Aug 23 07:35 macros
-rw-rw-r--  1 john  doc      2488 Aug 15 10:51 intro
-rw-rw-rw-  1 john  doc      8515 Aug  6 15:30 ch07
-rw-rw-rw-  1 john  doc     11008 Aug  6 14:10 ch02
$
```

上面的命令，对当前目录中八月份修改的文件按照大小排序；+4n 表示对第 5 列按照数字大小排序。

pg 和 more 命令

如果文件内容过多，全部显示会很乱，可以使用 **pg** 和 **more** 命令分页显示，每次只显示一屏。

例如，通过管道，使用 more 命令显示目录中的文件：

```
$ls -l | grep "Aug" | sort +4n | more
-rw-rw-r--  1 carol doc      1605 Aug 23 07:35 macros
-rw-rw-r--  1 john  doc      2488 Aug 15 10:51 intro
-rw-rw-rw-  1 john  doc      8515 Aug  6 15:30 ch07
-rw-rw-r--  1 john  doc     14827 Aug  9 12:40 ch03
.
.
.
-rw-rw-rw-  1 john  doc     16867 Aug  6 15:56 ch05
--More--(74%)
```

如上，一次只显示一屏文本，显示满后，停下来，并提示已显示全部内容的百分比，按空格键(space)可以查看下一屏，按 b 键可以查看上一屏。

Linux 进程管理

当我们运行程序时，Linux 会为程序创建一个特殊的环境，该环境包含程序运行需要的所有资源，以保证程序能够独立运行，不受其他程序的干扰。这个特殊的环境就称为进程。

每个 Linux 命令都与系统中的程序对应，输入命令，Linux 就会创建一个新的进程。例如使用 ls 命令遍历目录中的文件时，就创建了一个进程。

简而言之，进程就是程序的实例。

系统通过一个五位数字跟踪程序的运行状态，这个数字称为 pid 或进程 ID。每个进程都拥有唯一的 pid。

理论上，五位数字是有限的，当数字被用完时，下一个 pid 就会重新开始，所以 pid 最终会重复。但是，两个 pid 一样的进程不能同时存在，因为 Linux 会使用 pid 来跟踪程序的运行状态。

创建进程

有两种方式来创建进程：前台进程和后台进程。

前台进程

默认情况下，用户创建的进程都是前台进程；前台进程从键盘读取数据，并把处理结果输出到显示器。

我们可以看到前台进程的运行过程。例如，使用 ls 命令来遍历当前目录下的文件：

```
$ls ch*.doc
ch01-1.doc    ch010.doc   ch02.doc     ch03-2.doc
ch04-1.doc    ch040.doc   ch05.doc     ch06-2.doc
ch01-2.doc    ch02-1.doc
```

这个程序就运行在前台，它会直接把结果输出到显示器。如果 ls 命令需要数据（实际上不需要），那么它会等待用户从键盘输入。

当程序运行在前台时，由于命令提示符($)还未出现，用户不能输入其他命令；即使程序需要运行很长时间，也必须等待程序运行结束才能输入其他命令。

后台进程

后台进程与键盘没有必然的关系。当然，后台进程也可能会等待键盘输入。

后台进程的优点是不必等待程序运行结束就可以输入其他命令。

创建后台进程最简单的方式就是在命令的末尾加 &，例如：

```
$ls ch*.doc &
ch01-1.doc    ch010.doc   ch02.doc     ch03-2.doc
ch04-1.doc    ch040.doc   ch05.doc     ch06-2.doc
ch01-2.doc    ch02-1.doc
```

如果 ls 命令需要输入（实际上不需要），那么它会暂停，直到用户把它调到前台并从键盘输入数据才会继续运行。

查看正在运行的进程

可以使用 **ps** 命令查看进程的运行状态，包括后台进程，例如：

```
$ps
PID       TTY      TIME        CMD
18358     ttyp3    00:00:00    sh
18361     ttyp3    00:01:31    abiword
18789     ttyp3    00:00:00    ps
```

还可以结合 **-f** 选项查看更多信息，f 是 full 的缩写，例如：

```
$ps -f
UID      PID  PPID C STIME    TTY   TIME CMD
amrood   6738 3662 0 10:23:03 pts/6 0:00 first_one
amrood   6739 3662 0 10:22:54 pts/6 0:00 second_one
amrood   3662 3657 0 08:10:53 pts/6 0:00 -ksh
amrood   6892 3662 4 10:51:50 pts/6 0:00 ps -f
```

每列的含义如下：

列	描述
UID	进程所属用户的 ID，即哪个用户创建了该进程。
PID	进程 ID。
PPID	父进程 ID，创建该进程的进程称为父进程。
C	CPU 使用率。
STIME	进程被创建的时间。
TTY	与进程有关的终端类型。
TIME	进程所使用的 CPU 时间。
CMD	创建该进程的命令。

ps 命令还有其他一些选项：

选项	说明
-a	显示所有用户的所有进程。
-x	显示无终端的进程。

-u	显示更多信息，类似于 -f 选项。
-e	显示所有进程。

终止进程

当进程运行在前台时，可以通过 **kill** 命令或 Ctrl+C 组合键来结束进程。

如果进程运行在后台，那么首先要通过 **ps** 命令来获取进程 ID，然后使用 **kill** 命令“杀死”进程，例如：

```
$ps -f
UID      PID  PPID C STIME    TTY   TIME CMD
amrood   6738 3662 0 10:23:03 pts/6 0:00 first_one
amrood   6739 3662 0 10:22:54 pts/6 0:00 second_one
amrood   3662 3657 0 08:10:53 pts/6 0:00 -ksh
amrood   6892 3662 4 10:51:50 pts/6 0:00 ps -f
$kill 6738
Terminated
```

如上所示，kill 命令终结了 first_one 进程。

如果进程忽略 kill 命令，那么可以通过 kill -9 来结束：

```
$kill -9 6738
Terminated
```

父进程和子进程

每个 Linux 进程会包含两个进程 ID：当前进程 ID(pid)和父进程 ID(ppid)。可以暂时认为所有的进程都有父进程。

由用户运行的大部分命令都将 Shell 作为父进程，使用 **ps -f** 命令可以查看当前进程 ID 和父进程 ID。

僵尸进程和孤儿进程

正常情况下，子进程被终止时会通过 SIGCHLD 信号通知父进程，父进程可以做一些清理工作或者重新启动一个新的进程。但在某些情况下，父进程会在子进程之前被终止，那么这些子进程就没有了“父亲”，被称为**孤儿进程**。

init 进程会成为所有孤儿进程的父进程。init 的 pid 为 1，是 Linux 系统的第一个进程，也是所有进程的父进程。

如果一个进程被终止了，但是使用 ps 命令仍然可以查看该进程，并且状态为 Z，那么这就是一个**僵尸进**

程。僵尸进程虽然被终止了，但是仍然存在于进程列表中。一般僵尸进程很难杀掉，你可以先杀死他们的父进程，让他们变成孤儿进程，init 进程会自动清理僵尸进程。

常驻进程

常驻进程一般是系统级进程，以 root 权限运行在后台，可以处理其他进程的请求。

常驻进程没有终端，不能访问 /dev/tty 文件，如果使用 ps -ef 查看该进程，tty 这一列会显示问号(?)。

更确切地说，常驻进程通常运行在后台，等待指定事件发生，例如打印进程是一个常驻进程，它会等待用户输入打印相关的命令并进行处理。

top 命令

top 命令是一个很有用的工具，它可以动态显示正在运行的进程，还可以按照指定条件对进程进行排序，与 Windows 的任务管理器类似。

top 命令可以显示进程的很多信息，包括物理内存、虚拟内存、CPU 使用率、平均负载以及繁忙的进程等。例如：

```
$top
```

这里仅给出一个示意图，读者最好亲自运行一下：

```
top - 08:43:19 up 47 days,  6:55,  3 users,  load average: 1.10, 1.03, 1.01
Tasks: 290 total,   2 running, 288 sleeping,   0 stopped,   0 zombie
Cpu(s): 34.0%us,  0.2%sy,  0.0%ni, 65.0%id,  0.0%wa,  0.0%hi,  0.1%si,  0.7%st
Mem:   5951644k total,  5752244k used,   199400k free,   120304k buffers
Swap:  2096472k total,   471580k used,  1624892k free,  2726220k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
  435 oracle    20   0 11032 1220  764 R  0.3  0.0   0:00.07 top
 1095 root      20   0     0    0    0 S  0.3  0.0   2:31.63 flush-202:0
 1138 root      20   0     0    0    0 S  0.3  0.0   4:15.36 kjournald
 7087 oracle    20   0  557m  67m 3896 S  0.3  1.2  12:03.90 java
 7213 oracle    20   0 1219m  74m  69m S  0.3  1.3   1:26.37 oracle
28023 oracle    20   0  678m  57m 2624 S  0.3  1.0  60:42.75 java
    1 root      20   0 10364  528  496 S  0.0  0.0   0:59.85 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      RT   0     0    0    0 S  0.0  0.0   0:03.24 migration/0
    4 root      20   0     0    0    0 S  0.0  0.0   0:26.56 ksoftirqd/0
    5 root      RT   0     0    0    0 S  0.0  0.0   0:00.01 watchdog/0
    6 root      RT   0     0    0    0 S  0.0  0.0   0:03.25 migration/1
    7 root      20   0     0    0    0 S  0.0  0.0   0:22.39 ksoftirqd/1
    8 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 watchdog/1
    9 root      20   0     0    0    0 S  0.0  0.0   2:54.04 events/0
   10 root      20   0     0    0    0 S  0.0  0.0   8:39.94 events/1
   11 root      20   0     0    0    0 S  0.0  0.0   0:00.00 cpuset
   12 root      20   0     0    0    0 S  0.0  0.0   0:00.00 khelper
   13 root      20   0     0    0    0 S  0.0  0.0   0:00.00 netns
   14 root      20   0     0    0    0 S  0.0  0.0   0:00.00 async/mgr
   15 root      20   0     0    0    0 S  0.0  0.0   0:00.00 xenwatch
   16 root      20   0     0    0    0 S  0.0  0.0   0:00.00 xenbus
   17 root      20   0     0    0    0 S  0.0  0.0   0:06.92 sync_supers
   18 root      20   0     0    0    0 S  0.0  0.0   0:09.91 bdi-default
```

任务和进程

任务(task)是最抽象的，是一个一般性的术语，指由软件完成的一个活动。一个任务既可以是一个进程，也可以是多个进程。简而言之，它指的是一系列共同达到某一目的的操作。例如，读取数据并将数据放入内存中。这个任务可以由一个进程来实现，也可以由多个进程来实现。 每个任务都有一个数字表示的任务号。

进程(process)常常被定义为程序的执行。可以把一个进程看成是一个独立的程序，在内存中有其完备的数据空间和代码空间。一个进程所拥有的数据和变量只属于它自己。

jobs 命令可以用来查看系统中正在运行的任务，包括后台运行的任务。该命令可以显示任务号及其对应的进程 ID。一个任务可以对应于一个或者多个进程号。

jobs 命令的 -l 选项可以查看当前任务包含的进程 ID：

```
$jobs -l
```

```
[1] + 1903 running                ls ch*.doc &
$
```

其中，第一列表示任务号，第二列表示任务对应的进程 ID，第三列表示任务的运行状态，第四列表示启动任务的命令。

前台任务和后台任务的切换

fg 命令可以将后台任务调到前台，语法为：

```
$fg %jobnumber
```

jobnumber 是通过 jobs 命令获取的后台任务的的序号，注意不是 pid。如果后台只有一个任务，可以不指定 jobnumber。

bg 命令可以将后台暂停的任务，调到前台继续运行，语法为：

```
$bg %jobnumber
```

jobnumber 同样是通过 jobs 命令获取的后台任务的的序号，注意不是 pid。如果前台只有一个任务，可以不指定 jobnumber。

如果希望将当前任务转移到后台，可以先 Ctrl+z 暂停任务，再使用 bg 命令。任务转移到后台可以空出终端，继续输入其他命令。

Linux 网络通信工具

现在是一个互联网的时代，你不可避免的要和其他用户进行远程交流，连接到远程主机。

ping 命令

ping 命令会向网络上的主机发送应答请求，根据响应信息可以判断远程主机是否可用。

ping 命令的语法：

```
$ping hostname or ip-address
```

如果网络畅通，很快就可以看到响应信息。

例如，检测是否可以连接到谷歌的主机：

```
$ping google.com
PING google.com (74.125.67.100) 56(84) bytes of data.
64 bytes from 74.125.67.100: icmp_seq=1 ttl=54 time=39.4 ms
64 bytes from 74.125.67.100: icmp_seq=2 ttl=54 time=39.9 ms
```

```
64 bytes from 74.125.67.100: icmp_seq=3 ttl=54 time=39.3 ms
64 bytes from 74.125.67.100: icmp_seq=4 ttl=54 time=39.1 ms
64 bytes from 74.125.67.100: icmp_seq=5 ttl=54 time=38.8 ms
--- google.com ping statistics ---
22 packets transmitted, 22 received, 0% packet loss, time 21017ms
rtt min/avg/max/mdev = 38.867/39.334/39.900/0.396 ms
$
```

如果主机没有响应，可以看到类似下面的信息：

```
$ping giiiiiigle.com
ping: unknown host giiiiigle.com
$
```

ftp 工具

ftp 是 File Transfer Protocol 的缩写，称为文件传输协议。通过 ftp 工具，能够将文件上传到远程服务器，也可以从远程服务器下载文件。

ftp 工具有自己的命令（类似 Linux 命令），可以：

- 连接并登录远程主机；
- 查看目录，遍历目录下的文件；
- 上传或下载文件，包括文本文件、二进制文件等。

ftp 命令的用法如下：

```
$ftp hostname or ip-address
```

接下来会提示你输入用户名和密码，验证成功后会进入主目录，然后就可以使用 ftp 工具的命令进行操作了。

ftp 命令	说明
put filename	将本地文件上传到远程主机。
get filename	将远程文件下载到本地。
mput file list	将多个本地文件上传到远程主机。
mget file list	将多个远程文件下载到本地。
prompt off	关闭提示。默认情况下，使用 mput 或 mget 命令会不断提示你确认文件的上传或下载。
prompt on	打开提示。
dir	列出远程主机当前目录下的所有文件。

cd dirname	改变远程主机目录。
lcd dirname	改变本地目录。
quit	退出登录。

注意，所有的上传和下载都是针对本地主机和远程主机的当前目录，如果你希望上传指定目录下的文件，首先要 cd 到该目录，然后才能上传。

ftp 工具使用举例:

```
$ftp amrood.com
Connected to amrood.com.
220 amrood.com FTP server (Ver 4.9 Thu Sep 2 20:35:07 CDT 2009)
Name (amrood.com:amrood): amrood
331 Password required for amrood.
Password:
230 User amrood logged in.
ftp> dir
200 PORT command successful.
150 Opening data connection for /bin/ls.
total 1464
drwxr-sr-x   3 amrood   group       1024 Mar 11 20:04 Mail
drwxr-sr-x   2 amrood   group       1536 Mar  3 18:07 Misc
drwxr-sr-x   5 amrood   group        512 Dec  7 10:59 OldStuff
drwxr-sr-x   2 amrood   group       1024 Mar 11 15:24 bin
drwxr-sr-x   5 amrood   group       3072 Mar 13 16:10 mpl
-rw-r--r--   1 amrood   group     209671 Mar 15 10:57 myfile.out
drwxr-sr-x   3 amrood   group        512 Jan  5 13:32 public
drwxr-sr-x   3 amrood   group        512 Feb 10 10:17 pvm3
226 Transfer complete.
ftp> cd mpl
250 CWD command successful.
ftp> dir
200 PORT command successful.
150 Opening data connection for /bin/ls.
total 7320
-rw-r--r--   1 amrood   group       1630 Aug  8 1994  dboard.f
-rw-r-----   1 amrood   group       4340 Jul 17 1994  vttest.c
-rwxr-xr-x   1 amrood   group     525574 Feb 15 11:52 wave_shift
```

```
-rw-r--r--   1 amrood   group       1648 Aug  5 1994  wide.list
-rwxr-xr-x   1 amrood   group       4019 Feb 14 16:26 fix.c
226 Transfer complete.
ftp> get wave_shift
200 PORT command successful.
150 Opening data connection for wave_shift (525574 bytes).
226 Transfer complete.
528454 bytes received in 1.296 seconds (398.1 Kbytes/s)
ftp> quit
221 Goodbye.
$
```

telnet 工具

Telnet 工具可以让我们连接并登录到远程计算机。

一旦连接到了远程计算机，就可以在上面进行各种操作了，例如：

```
C:>telnet amrood.com
Trying...
Connected to amrood.com.
Escape character is '^]'.

login: amrood
amrood's Password:
**************************************************
*                                                *
*                                                *
*            WELCOME TO AMROOD.COM                *
*                                                *
*                                                *
**************************************************

Last unsuccessful login: Fri Mar  3 12:01:09 IST 2009
Last login: Wed Mar  8 18:33:27 IST 2009 on pts/10

   {  do your work }
```

```
$ logout
Connection closed.
C:>
```

finger 工具

finger 可以让我们查看本地主机或远程主机上的用户信息。有些系统为了安全会禁用 finger 命令。

例如，查看本机在线用户：

```
$ finger
Login     Name       Tty      Idle  Login Time   Office
amrood               pts/0          Jun 25 08:03 (62.61.164.115)
```

查看本机指定用户的信息：

```
$ finger amrood
Login: amrood                           Name: (null)
Directory: /home/amrood                 Shell: /bin/bash
On since Thu Jun 25 08:03 (MST) on pts/0 from 62.61.164.115
No mail.
No Plan.
```

查看远程主机上的在线用户：

```
$ finger @avtar.com
Login     Name       Tty      Idle  Login Time   Office
amrood               pts/0          Jun 25 08:03 (62.61.164.115)
```

查看远程主机上某个用户的信息：

```
$ finger amrood@avtar.com
Login: amrood                           Name: (null)
Directory: /home/amrood                 Shell: /bin/bash
On since Thu Jun 25 08:03 (MST) on pts/0 from 62.61.164.115
No mail.
No Plan.
```

vi 编辑器

Linux 下的文本编辑器有很多种，vi 是最常用的，也是各版本 Linux 的标配。注意，vi 仅仅是一个文本编辑器，可以给字符着色，可以自动补全，但是不像 Windows 下的 word 有排版功能。

vi 是十年磨一剑的产品，虽然命令繁多，并且大多数功能都是依靠键盘输入来完成，但是一旦你熟悉后，

会发现 vi 的功能和效率是其他图形界面编辑器无法比拟的。

Vim 是 **Vi** **im**proved 的缩写，是 vi 的改进版。在 Linux 中，vi 被认为是事实上的标准编辑器，因为：

- 所有版本的 Linux 都带有 vi 编辑器；
- 占用资源少；
- 与 ed、ex 等其他编辑器相比，vi 对用户更加友好。

你可以使用 vi 编辑器编辑现有的文件，也可以创建一个新文件，还能以只读模式打开文本文件。

进入 vi 编辑器

可以通过以下方式进入 vi 编辑器：

命令	描述
vi filename	如果 filename 存在，则打开；否则会创建一个新文件再打开。
vi -R filename	以只读模式（只能查看不能编辑）打开现有文件。
view filename	以只读模式打开现有文件。

例如，使用 vi 编辑器创建一个新文件并打开：

```
$vi testfile
|
~
~
~
~
~
~
~
~
~
~
~
~
~
"testfile" [New File]
```

竖线(|)代表光标的位置；波浪号(~)代表该行没有任何内容。如果没有 ~，也看不到任何内容，那说明这一行肯定是有空白字符（空格、tab 缩进、换行符等）或不可见字符。

工作模式

进一步了解 vi 之前先来了解一下 vi 的工作模式，vi 有三种工作模式：

1）普通模式

由 Shell 进入 vi 编辑器时，首先进入普通模式。在普通模式下，从键盘输入任何字符都被当作命令来解释。普通模式下没有任何提示符，输入命令后立即执行，不需要回车，而且输入的字符不会在屏幕上显示出来。

普通模式下可以执行命令、保存文件、移动光标、粘贴复制等。

2）编辑模式

编辑模式主要用于文本的编辑。该模式下用户输入的任何字符都被作为文件的内容保存起来，并在屏幕上显示出来。

3）命令模式

命令模式下，用户可以对文件进行一些高级处理。尽管普通模式下的命令可以完成很多功能，但要执行一些如字符串查找、替换、显示行号等操作还是必须要进入命令模式。

注意：有些教程中称有两种工作模式，是把命令模式合并到普通模式。

工作模式切换：

- 在普通模式下输入 i(插入)、c(修改)、o(另起一行) 命令时进入编辑模式；按 esc 键退回到普通模式。
- 在普通模式下输入冒号(:)可以进入命令模式。输入完命令按回车，命令执行完后会自动退回普通模式。

提示：如果不确定当前处于哪种模式，按两次 Esc 键将回到普通模式。

退出 vi 编辑器

一般在命令模式下退出 vi 编辑器。

退出命令	说明
q	如果文件未被修改，会直接退回到 Shell；否则提示保存文件。
q!	强行退出，不保存修改内容。
wq	w 命令保存文件，q 命令退出 vi，合起来就是保存并退出。
ZZ	保存并退出，相当于 wq，但是更加方便。

退出之前，你也可以在 w 命令后面指定一个文件名，将文件另存为新文件，例如：

```
w filename2
```

将当前文件另存为 filename2。

注意：vi 编辑文件时，用户的操作都是基于缓冲区中的副本进行的。如果退出时没有保存到磁盘，则缓冲区中的内容就会被丢失。

移动光标

为了不影响文件内容，必须在普通模式（按两次 Esc 键）下移动光标。使用下表中的命令每次可以移动一个字符：

命令	描述
k	向上移动光标（移动一行）
j	向下移动光标（移动一行）
h	向左移动光标（移动一个字符）
l	向右移动光标（移动一个字符）

两点提醒：

- vi 是区分大小写的，输入命令时注意不要锁定大写。
- 可以在命令前边添加一个数字作为前缀，例如，2j 将光标向下移动两行。

当然，还有很多其他命令来移动光标，不过记住，一定要在普通模式（按两次 Esc 键）下。

用来移动光标的命令

命令	说明
0 或 |	将光标定位在一行的开头。
$	将光标定位在一行的末尾。
w	定位到下一个单词。
b	定位到上一个单词。
(	定位到一句话的开头，句子是以！.？三种符号来界定的。
)	定位到一句话的结尾。
{	移动到段落开头。&&&&&&
}	移动到段落结束。&&&&&&&&&
[[	回到段落的开头处。&&&&&&&&&&
]]	向前移到下一个段落的开头处。&&&&&&&&&&
n|	移动到第 n 列（当前行）。

1G	移动到文件第一行。
G	移动到文件最后一行。
nG	移动到文件第 n 行。
:n	移动到文件第 n 行。
H	移动到屏幕顶部。
nH	移动到距离屏幕顶部第 n 行的位置。
M	移动到屏幕中间。
L	移动到屏幕底部。
nL	移动到距离屏幕底部第 n 行的位置。
:x	x 是一个数字，表示移动到行号为 x 的行。

控制命令

有一些控制命令可以与 Ctrl 键组合使用，如下：

命令	描述
CTRL+d	向前滚动半屏
CTRL+f	向前滚动全屏
CTRL+u	向后滚动半屏
CTRL+b	向后滚动整屏
CTRL+e	向上滚动一行
CTRL+y	向下滚动一行
CTRL+l	刷新屏幕

编辑文件

切换到编辑模式下才能编辑文件。有很多命令可以从普通模式切换到编辑模式，如下所示：

命令	描述
i	在当前光标位置之前插入文本
I	在当前行的开头插入文本
a	在当前光标位置之后插入文本
A	在当前行的末尾插入文本
o	在当前位置下面创建一行
O	在当前位置上面创建一行

删除字符

下面的命令，可以删除文件中的字符或行：

命令	说明
x	删除当前光标下的字符
X	删除光标前面的字符
dw	删除从当前光标到单词结尾的字符
d^	删除从当前光标到行首的字符
d$	删除从当前光标到行尾的字符
D	删除从当前光标到行尾的字符
dd	删除当前光标所在的行

可以在命令前面添加一个数字前缀，表示重复操作的次数，例如，2x 表示连续两次删除光标下的字符，2dd 表示连续两次删除光标所在的行。

建议各位读者多加练习上面的命令，再进一步深入学习。

修改文本

如果你希望对字符、单词或行进行修改，可以使用下面的命令：

命令	描述
cc	删除当前行，并进入编辑模式。
cw	删除当前字（单词），并进入编辑模式。
r	替换当前光标下的字符。
R	从当前光标开始替换字符，按 Esc 键退出。
s	用输入的字符替换当前字符，并进入编辑模式。
S	用输入的文本替换当前行，并进入编辑模式。

粘贴复制

vi 中的复制粘贴命令：

命令	描述
yy	复制当前行
nyy	复制 n 行

yw	复制一个字（单词）
nyw	复制 n 行
p	将复制的文本粘贴到光标后面
P	将复制的文本粘贴到光标前面

高级命令

下面的一些命令虽然看起来有些古怪，但是会让你的工作更有效率，如果你是 vi 重度用户，就了解一下吧。

命令	说明
J	将当前行和下一行连接为一行
<<	将当前行左移一个单位（一个缩进宽度）
>>	将当前行右移一个单位（一个缩进宽度）
~	改变当前字符的大小写
^G	Ctrl+G 组合键可以显示当前文件名和状态
U	撤销对当前行所做的修改
u	撤销上次操作，再次按 'u' 恢复该次操作
:f	以百分号(%)的形式显示当前光标在文件中的位置、文件名和文件的总行数
:f filename	将文件重命名为 filename
:w filename	保存修改到 filename
:e filename	打开另一个文件名为 filename 的文件
:cd dirname	改变当前工作目录到 dirname
:e #	在两个打开的文件之间进行切换
:n	如果用 vi 打开了多个文件，可以使用 :n 切换到下一个文件
:p	如果用 vi 打开了多个文件，可以使用 :n 切换到上一个文件
:N	如果用 vi 打开了多个文件，可以使用 :n 切换到上一个文件
:r file	读取文件并在当前行的后边插入
:nr file	读取文件并在第 n 行后边插入

文本查找

如果希望进行全文件搜索，可以在普通模式（按两次 Esc 键）下输入 / 命令，这时状态栏（最后一行）出现"/"并提示输入要查找的字符串，回车即可。

/ 命令是向下查找，如果希望向上查找，可以使用 ？ 命令。

这时，输入 n 命令可以按相同的方向继续查找，输入 N 命令可以按相反的方向继续查找。

搜索的字符串中可以包含一些有特殊含义的字符，如果希望搜索这些字符本身，需要在前面加反斜杠(\)。

部分特殊字符列表

字符	说明
^	匹配一行的开头
.	匹配一个字符
*	匹配 0 个或多个字符
$	匹配一行的结尾
[ ]	匹配一组字符

如果希望搜索某行中的单个字符，可以使用 f 或 F 命令，f 向上搜索，F 向下搜索，并且会把光标定位到匹配的字符。

也可以使用 t 或 T 命令：t 命令向上搜索，并把光标定位到匹配字符的前面；T 命令向下搜索，并把光标定位到匹配字符的后面。

set 命令

set 命令可以对 vi 编辑器进行一些设置。使用 set 命令需要进入命令模式。

:set 命令选项

命令	说明
:set ic	搜索时忽略大小写。
:set ai	设置自动缩进（自动对齐）。
:set noai	取消自动缩进（自动对齐）。
:set nu	显示行号。
:set sw	设置缩进的空格数，例如，将缩进空格数设置为 4：:set sw=4。
:set ws	循环搜索：如果直到文件末尾也没有查找到指定字符，那么会回到开头继续查找。
:set wm	设置自动换行，例如，设置距离边际 2 个字符时换行：:set wm=2 。
:set ro	将文件类型改为只读。
:set term	输出终端类型。

:set bf	忽略输入的控制字符，如 BEL(响铃)、BS(退格)、CR(回车)等。

运行命令

切换到命令模式，再输入 ！命令即可运行 Linux 命令。

例如，保存文件前，如果希望查看该文件是否存在，那么输入

```
:! ls
```

即可列出当前目录下的文件。

按任意键回到 vi 编辑器。

文本替换

切换到命令模式，再输入 s/ 命令即可对文本进行替换。语法为：

```
:s/search/replace/g
```

search 为检索的文本，replace 为要替换的文本，g 表示全局替换。

几点提示

vi 编辑器的使用讲解完毕，但是请记住下面几点：

- 输入冒号(:)进入命令模式，按两次 Esc 键进入普通模式。
- 命令大小写的含义是不一样的。
- 必须在编辑模式下才能输入内容。

Linux 文件系统

文件系统就是分区或磁盘上的所有文件的逻辑集合。

文件系统不仅包含着文件中的数据而且还有文件系统的结构，所有 Linux 用户和程序看到的文件、目录、软连接及文件保护信息等都存储在其中。

不同 Linux 发行版本之间的文件系统差别很少，主要表现在系统管理的特色工具以及软件包管理方式的不同，文件目录结构基本上都是一样的。

文件系统有多种类型，如：

- ext2 ： 早期 linux 中常用的文件系统；
- ext3 ： ext2 的升级版，带日志功能；
- RAMFS ： 内存文件系统，速度很快；

- iso9660：光盘或光盘镜像；
- NFS ： 网络文件系统，由 SUN 发明，主要用于远程文件共享；
- MS-DOS ： MS-DOS 文件系统；
- FAT ： Windows XP 操作系统采用的文件系统；
- NTFS ： Windows NT/XP 操作系统采用的文件系统。

分区与目录

文件系统位于磁盘分区中；一个硬盘可以有多个分区，也可以只有一个分区；一个分区只能包含一个文件系统。

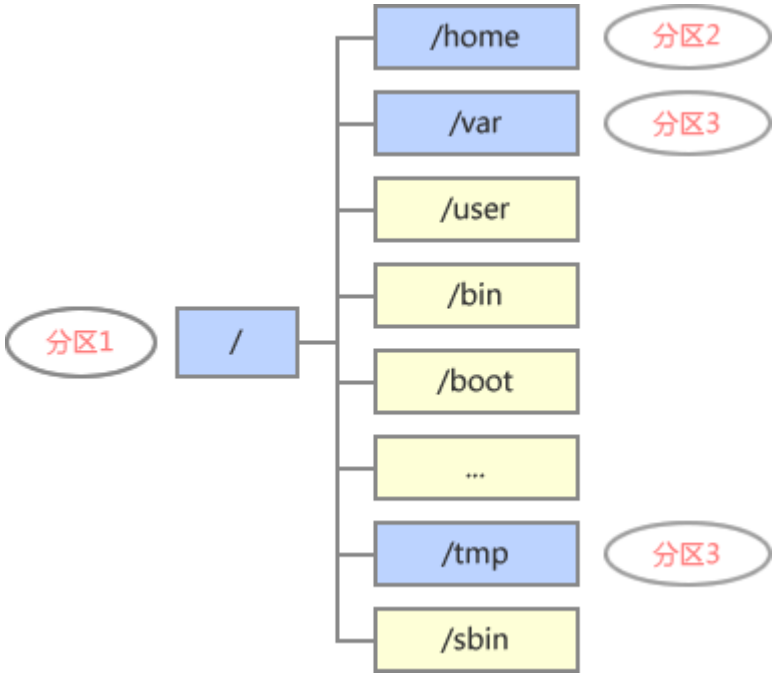
Linux 文件系统与 Windows 有较大的差别。Windows 的文件结构是多个并列的树状结构，最顶部的是不同的磁盘（分区），如 C、D、E、F 等。

Linux 的文件结构是单个的树状结构，根目录是“/”，其他目录都要位于根目录下。

每次安装系统的时候我们都会进行分区，Linux 下磁盘分区和目录的关系如下：

- 任何一个分区都必须对应到某个目录上，才能进行读写操作，称为“挂载”。
- 被挂载的目录可以是根目录，也可以是其他二级、三级目录，任何目录都可以是挂载点。
- 目录是逻辑上的区分。分区是物理上的区分。
- 根目录是所有 Linux 的文件和目录所在的地方，需要挂载上一个磁盘分区。

下图是常见的目录和分区的对应关系：



为什么要分区，如何分区？

- 可以把不同资料，分别放入不同分区中管理，降低风险。
- 大硬盘搜索范围大，效率低。
- /home、/var、/usr/local 经常是单独分区，因为经常会操作，容易产生碎片。

为了便于定位和查找，Linux 中的每个目录一般都存放特定类型的文件，下表列出了各种 Linux 发行版本的常见目录：

目录	说明
/	根目录，只能包含目录，不能包含具体文件。
/bin	存放可执行文件。很多命令就对应/bin 目录下的某个程序，例如 ls、cp、mkdir。/bin 目录对所有用户有效。
/dev	硬件驱动程序。例如声卡、磁盘驱动等，还有如 /dev/null、/dev/console、/dev/zero、/dev/full 等文件。
/etc	主要包含系统配置文件和用户、用户组配置文件。
/lib	主要包含共享库文件，类似于 Windows 下的 DLL；有时也会包含内核相关文件。
/boot	系统启动文件，例如 Linux 内核、引导程序等。
/home	用户工作目录（主目录），每个用户都会分配一个目录。
/mnt	临时挂载文件系统。这个目录一般是用于存放挂载储存设备的挂载目录的，例如挂载 CD-ROM 的 cdrom 目录。
/proc	操作系统运行时，进程（正在运行中的程序）信息及内核信息（比如 cpu、硬盘分区、内存信息等）存放在这里。/proc 目录伪装的文件系统 proc 的挂载目录，proc 并不是真正的文件系统。
/tmp	临时文件目录，系统重启后不会被保存。
/usr	/user 目下的文件比较混杂，包含了管理命令、共享文件、库文件等，可以被很多用户使用。
/var	主要包含一些可变长度的文件，会经常对数据进行读写，例如日志文件和打印队列里的文件。
/sbin	和 /bin 类似，主要包含可执行文件，不过一般是系统管理所需要的，不是所有用户都需要。

常用文件管理命令

你可以通过下面的命令来管理文件：

Command	Description
cat filename	查看文件内容。
cd dirname	改变所在目录。
cp file1 file2	复制文件或目录。
file filename	查看文件类型(binary, text, etc)。
find filename dir	搜索文件或目录。

head filename	显示文件的开头，与 tail 命令相对。
less filename	查看文件的全部内容，可以分页显示，比 more 命令要强大。
ls dirname	遍历目录下的文件或目录。
mkdir dirname	创建目录。
more filename	查看文件的全部内容，可以分页显示。
mv file1 file2	移动文件或重命名。
pwd	显示用户当前所在目录。
rm filename	删除文件。
rmdir dirname	删除目录。
tail filename	显示文件的结尾，与 head 命令相对。
touch filename	文件不存在时创建一个空文件，存在时修改文件时间戳。
whereis filename	查看文件所在位置。
which filename	如果文件在环境变量 PATH 中有定义，那么显示文件位置。

df 命令

管理磁盘分区时经常会使用 **df** (disk free) 命令，df -k 命令可以用来查看磁盘空间的使用情况（以千字节计），例如：

```
$df -k
Filesystem      1K-blocks      Used   Available Use% Mounted on
/dev/vzfs        10485760   7836644     2649116  75% /
/devices                0         0           0   0% /devices
$
```

每一列的含义如下：

列	说明
Filesystem	代表文件系统对应的设备文件的路径名（一般是硬盘上的分区）。
kbytes	分区包含的数据块（1024 字节）的数目。
used	已用空间。
avail	可用空间。
capacity	已用空间的百分比。
Mounted on	文件系统挂载点。

某些目录（例如 /devices）的 kbytes、used、avail 列为 0，use 列为 0%，这些都是特殊（或虚拟）文件系统，即使位于根目录下，也不占用硬盘空间。

你可以结合 -h (human readable) 选项将输出信息格式化，让人更易阅读。

du 命令

du (disk usage) 命令可以用来查看特定目录的空间使用情况。

du 命令会显示每个目录所占用数据块。根据系统的不同，一个数据块可能是 512 字节或 1024 字节。举例如下：

```
$du /etc
10      /etc/cron.d
126     /etc/default
6       /etc/dfs
...
$
```

结合 -h 选项可以让信息显示的更加清晰：

```
$du -h /etc
5k     /etc/cron.d
63k    /etc/default
3k     /etc/dfs
...
$
```

挂载文件系统

挂载是指将一个硬件设备（例如硬盘、U 盘、光盘等）对应到一个已存在的目录上。 若要访问设备中的文件，必须将文件挂载到一个已存在的目录上， 然后通过访问这个目录来访问存储设备。

这样就为用户提供了统一的接口，屏蔽了硬件设备的细节。Linux 将所有的硬件设备看做文件，对硬件设备的操作等同于对文件的操作。

注意：挂载目录可以不为空，但挂载后这个目录下以前的内容将不可用。

需要知道的是，光盘、软盘、其他操作系统使用的文件系统的格式与 linux 使用的文件系统格式是不一样的，挂载需要确认 Linux 是否支持所要挂载的文件系统格式。

查看当前系统所挂载的硬件设备可以使用 mount 命令：

```
$ mount
/dev/vzfs on / type reiserfs (rw,usrquota,grpquota)
proc on /proc type proc (rw,nodiratime)
```

```
devpts on /dev/pts type devpts (rw)
$
```

一般约定，/mnt 为临时挂载目录，例如挂载 CD-ROM、远程网络设备、软盘等。

也可以通过 mount 命令来挂载文件系统，语法为：

```
mount -t file_system_type device_to_mount directory_to_mount_to
```

例如：

```
$ mount -t iso9660 /dev/cdrom /mnt/cdrom
```

将 CD-ROM 挂载到 /mnt/cdrom 目录。

注意：file_system_type 用来指定文件系统类型，通常可以不指定，Linux 会自动正确选择文件系统类型。

挂载文件系统后，就可以通过 cd、cat 等命令来操作对应文件。

可以通过 umount 命令来卸载文件系统。例如，卸载 cdrom：

```
$ umount /dev/cdrom
```

不过，大部分现代的 Linux 系统都有自动挂载卸载功能，unmount 命令较少用到。

用户和群组配额

用户和群组配额可以让管理员为每个用户或群组分配固定的磁盘空间。

管理员有两种方式来分配磁盘空间：

- 软限制：如果用户超过指定的空间，会有一个宽限期，等待用户释放空间。
- 硬限制：没有宽限期，超出指定空间立即禁止操作。

下面的命令可以用来管理配额：

命令	说明
quota	显示磁盘使用情况以及每个用户组的配额。
edquota	编辑用户和群组的配额。
quotacheck	查看文件系统的磁盘使用情况，创建、检查并修复配额文件。
setquota	设置配额。
quotaon	开启用户或群组的配额功能。
quotaoff	关闭用户或群组的配额功能。
repquota	打印指定文件系统的配额。

Linux 文件存储结构

大部分的 Linux 文件系统（如 ext2、ext3）规定，一个文件由目录项、inode 和数据块组成：

- 目录项：包括文件名和 inode 节点号。
- Inode：又称文件索引节点，包含文件的基础信息以及数据块的指针。
- 数据块：包含文件的具体内容。

先说 inode

理解 inode，要从文件储存说起。文件储存在硬盘上，硬盘的最小存储单位叫做"扇区"（Sector），每个扇区储存 512 字节（相当于 0.5KB）。

操作系统读取硬盘的时候，不会一个扇区一个扇区地读取，这样效率太低，而是一次性连续读取多个扇区，即一次性读取一个"块"（block）。这种由多个扇区组成的"块"，是文件存取的最小单位。"块"的大小，最常见的是 4KB，即连续八个 sector 组成一个 block。

文件数据都储存在"块"中，那么很显然，我们还必须找到一个地方储存文件的元信息，比如文件的创建者、文件的创建日期、文件的大小等等。这种储存文件元信息的区域就叫做 inode，中文译名为"索引节点"。

inode 包含文件的元信息，具体来说有以下内容：

- 文件的字节数。
- 文件拥有者的 User ID。
- 文件的 Group ID。
- 文件的读、写、执行权限。
- 文件的时间戳，共有三个：ctime 指 inode 上一次变动的时间，mtime 指文件内容上一次变动的时间，atime 指文件上一次打开的时间。
- 链接数，即有多少文件名指向这个 inode。
- 文件数据 block 的位置。

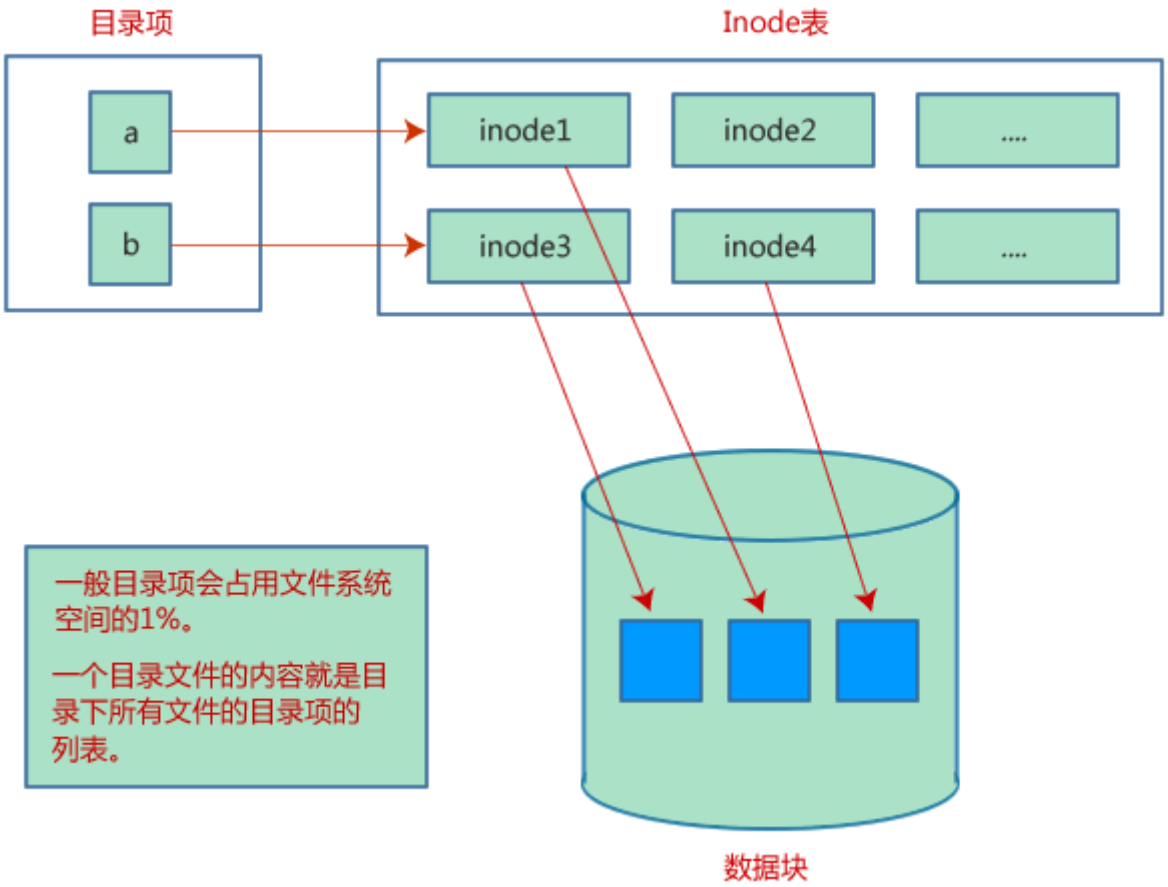
可以用 stat 命令，查看某个文件的 inode 信息：

```
stat demo.txt
```

总之，除了文件名以外的所有文件信息，都存在 inode 之中。至于为什么没有文件名，下文会有详细解释。

当查看某个文件时，会先从 inode 表中查出文件属性及数据存放点，再从数据块中读取数据。

请看文件存储结构示意图：



inode 的大小

inode 也会消耗硬盘空间，所以硬盘格式化的时候，操作系统自动将硬盘分成两个区域。一个是数据区，存放文件数据；另一个是 inode 区（inode table），存放 inode 所包含的信息。

每个 inode 节点的大小，一般是 128 字节或 256 字节。inode 节点的总数，在格式化时就给定，一般是每 1KB 或每 2KB 就设置一个 inode。假定在一块 1GB 的硬盘中，每个 inode 节点的大小为 128 字节，每 1KB 就设置一个 inode，那么 inode table 的大小就会达到 128MB，占整块硬盘的 12.8%。

查看每个硬盘分区的 inode 总数和已经使用的数量，可以使用 df -i 命令。

查看每个 inode 节点的大小，可以用如下命令：

```
sudo dumpe2fs -h /dev/hda | grep "Inode size"
```

由于每个文件都必须有一个 inode，因此有可能发生 inode 已经用光，但是硬盘还未存满的情况。这时，就无法在硬盘上创建新文件。

inode 号码

每个 inode 都有一个号码，操作系统用 inode 号码来识别不同的文件。

这里值得重复一遍，Linux 系统内部不使用文件名，而使用 inode 号码来识别文件。对于系统来说，文件名只是 inode 号码便于识别的别称或者绰号。表面上，用户通过文件名，打开文件。实际上，系统内部这

个过程分成三步：首先，系统找到这个文件名对应的 inode 号码；其次，通过 inode 号码，获取 inode 信息；最后，根据 inode 信息，找到文件数据所在的 block，读出数据。

使用 ls -i 命令，可以看到文件名对应的 inode 号码，例如：

```
ls -i demo.txt
```

目录项

Linux 系统中，目录（directory）也是一种文件。打开目录，实际上就是打开目录文件。

目录文件的结构非常简单，就是一系列目录项（dirent）的列表。每个目录项，由两部分组成：所包含文件的文件名，以及该文件名对应的 inode 号码。

ls 命令只列出目录文件中的所有文件名：

```
ls /etc
```

ls -i 命令列出整个目录文件，即文件名和 inode 号码：

```
ls -i /etc
```

如果要查看文件的详细信息，就必须根据 inode 号码，访问 inode 节点，读取信息。ls -l 命令列出文件的详细信息。

```
ls -l /etc
```

硬链接和软链接

硬链接

一般情况下，文件名和 inode 号码是"一一对应"关系，每个 inode 号码对应一个文件名。但是，Linux 系统允许，多个文件名指向同一个 inode 号码。这意味着，可以用不同的文件名访问同样的内容；对文件内容进行修改，会影响到所有文件名；但是，删除一个文件名，不影响另一个文件名的访问。这种情况就被称为"硬链接"（hard link）。

ln 命令可以创建硬链接，语法为：

```
ln source_file target_file
```

运行上面这条命令以后，源文件与目标文件的 inode 号码相同，都指向同一个 inode。inode 信息中有一项叫做"链接数"，记录指向该 inode 的文件名总数，这时就会增加 1。反过来，删除一个文件名，就会使得 inode 节点中的"链接数"减 1。当这个值减到 0，表明没有文件名指向这个 inode，系统就会回收这个 inode 号码，以及其所对应 block 区域。

这里顺便说一下目录文件的"链接数"。创建目录时，默认会生成两个目录项："."和".."。前者的 inode 号码就是当前目录的 inode 号码，等同于当前目录的"硬链接"；后者的 inode 号码就是当前目录的父目录的

inode 号码，等同于父目录的"硬链接"。所以，任何一个目录的"硬链接"总数，总是等于 2 加上它的子目录总数（含隐藏目录）,这里的 2 是父目录对其的“硬链接”和当前目录下的".硬链接 “。

软链接

除了硬链接以外，还有一种特殊情况。文件 A 和文件 B 的 inode 号码虽然不一样，但是文件 A 的内容是文件 B 的路径。读取文件 A 时，系统会自动将访问者导向文件 B。因此，无论打开哪一个文件，最终读取的都是文件 B。这时，文件 A 就称为文件 B 的"软链接"（soft link）或者"符号链接（symbolic link）。

这意味着，文件 A 依赖于文件 B 而存在，如果删除了文件 B，打开文件 A 就会报错："No such file or directory"。这是软链接与硬链接最大的不同：文件 A 指向文件 B 的文件名，而不是文件 B 的 inode 号码，文件 B 的 inode"链接数"不会因此发生变化。

ln -s 命令可以创建软链接，语法为：

```
ln -s source_file target_file


file 察看文件类型
```

Linux 用户管理　git　rm -rf

在 Linux 中，有三种用户：

- Root 用户：也称为超级用户，对系统拥有完全的控制权限。超级用户可以不受限制的运行任何命令。Root 用户可以看做是系统管理员。
- 系统用户：系统用户是 Linux 运行某些程序所必须的用户，例如 mail 用户、sshd 用户等。系统用户通常为系统功能所必须的，不建议修改这些用户。
- 普通用户：一般用户都是普通用户，这些用户对系统文件的访问受限，不能执行全部 Linux 命令。

Linux 支持用户组，用户组就是具有相同特征的用户的集合。一个组可以包含多个用户，每个用户也可以属于不同的组。用户组在 Linux 中扮演着重要的角色，方便管理员对用户进行集中管理。

与用户和组有关的系统文件

与用户和组有关的系统文件：

系统文件	说明
/etc/passwd	保存用户名和密码等信息，Linux 系统中的每个用户都在/etc/passwd 文件中有一个对应的记录行。这个文件对所有用户都是可读的。
/etc/shadow	/etc/shadow 中的记录行和/etc/passwd 中的相对应，他由 pwconv 命令根据/etc/passwd 中的数据自动产生，它的格式和/etc/passwd 类似，只是对密码进行了加密。并不是所有的系统都支持这个文件。
/etc/group	以记录行的形式保存了用户组的所有信息。

来看一下/etc/passwd 文件的结构：

```
$cat /etc/passwd
root:x:0:0:Superuser:/:
daemon:x:1:1:System daemons:/etc:
bin:x:2:2:Owner of system commands:/bin:
sys:x:3:3:Owner of system files:/usr/sys:
adm:x:4:4:System accounting:/usr/adm:
uucp:x:5:5:UUCP administrator:/usr/lib/uucp:
auth:x:7:21:Authentication administrator:/tcb/files/auth:
cron:x:9:16:Cron daemon:/usr/spool/cron:
listen:x:37:4:Network daemon:/usr/net/nls:
lp:x:71:18:printer administrator:/usr/spool/lp:
sam:x:200:50:Sam san:/usr/sam:/bin/sh
```

可以看到，/etc/passwd 文件中一行记录对应着一个用户，每行记录又被冒号分隔为 7 个字段，其格式和具体含义如下图所示：

对每个字段的说明：

字段	说明
用户名	用户名是惟一的，长度根据不同的 linux 系统而定，一般是 8 位。
密码	由于系统中还有一个/etc/shadow 文件用于存放加密后的口令，所以在这里这一项是“x”来表示，如果用户没有设置口令，则该项为空。如果 passwd 字段中的第一个字符是“*”的话，那么，就表示该账号被查封了，系统不允许持有该账号的用户登录。
用户 ID	系统内部根据用户 ID 而不是用户名来识别不同的用户，用户 ID 有以下几种： 0 代表系统管理员，如果你想建立一个系统管理员的话，可以建立一个普通帐户，然后将该账户的用户 ID 改为 0 即可。 1~500 系统预留的 ID。 500 以上是普通用户使用。
组 ID	其实这个和用户 ID 差不多，用来管理群组，与/etc/group 文件相关。
描述信息	这个字段几乎没有什么用，只是用来解释这个账号的意义。在不同的 Linux 系统中，这个字段的 格式并没有统一。在许多 Linux 系统中，这个字段存放的是一段任意的注释性描述文字，用做 finger 命令的输出。
用户主目录	用户登录系统的起始目录。用户登录系统后将首先进入该目录。root 用户默认是/，普通用

	户是/home/username。
用户 Shell	用户登录系统时使用的 Shell。

管理用户和组

下面是一些常用的管理用户和组的命令：

命令	说明
useradd	添加用户。
usermod	修改用户信息。
userdel	删除用户。
groupadd	添加用户组。
groupmod	修改用户组信息。
groupdel	删除用户组。

创建用户组

添加用户时，可以将用户添加到现有的用户组，或者创建一个新的用户组。可以在 /etc/groups 文件中看到所有的用户组信息。

默认的用户组通常用来管理系统用户，不建议将普通用户添加到这些用户组。使用 groupadd 命令创建用户组的语法为：

```
groupadd [-g gid [-o]] [-r] [-f] groupname
```

每个选项的含义如下：

选项	说明
-g GID	以数字表示的用户组 ID。
-o	可以使用重复的组 ID。
-r	建立系统组，用来管理系统用户。
-f	强制创建。
groupname	用户组的名称。

如果不指定选项，系统将使用默认值。例如创建一个 developers 用户组：

```
$ groupadd developers
```

修改用户组

groupmod 命令可以用来修改用户组，语法为：

```
$ groupmod -n new_modified_group_name old_group_name
```

例如，将用户组 developers_2 重命名为 developer：

```
$ groupmod -n developer developer_2
```

将 developer 用户组的 ID 改为 545：

```
$ groupmod -g 545 developer
```

删除用户组

通过 groupdel 命令可以删除用户组。例如，删除 developer 组：

```
$ groupdel developer
```

groupdel 仅仅删除用户组，并不删除与之相关的文件，这些文件仍然可以被所有者访问。

添加用户

添加用户可以使用 useradd 命令，语法为：

```
useradd -d homedir -g groupname -m -s shell -u userid accountname
```

每个选项的含义如下：

选项	描述
-d homedir	指定用户主目录。
-g groupname	指定用户组。
-m	如果主目录不存在，就创建。
-s shell	为用户指定默认 Shell。
-u userid	指定用户 ID。
accountname	用户名。

如果不指定任何选项，系统将使用默认值。useradd 命令将会修改 /etc/passwd、/etc/shadow、and /etc/group 三个文件，并创建用户主目录。

下面的例子将会添加用户 mcmohd，并设置主目录为 /home/mcmohd，用户组为 developers，默认 Shell 为 Korn Shell：

```
$ useradd -d /home/mcmohd -g developers -s /bin/ksh mcmohd
```

注意：添加用户前请确认 developers 用户组存在。

用户被创建后，可以使用 passwd 命令来设置密码，例如：

```
$ passwd mcmohd20
Changing password for user mcmohd20.
New Linux password:******
Retype new UNIX password:******
```

```
passwd: all authentication tokens updated successfully.
```

注意：如果你是管理员，输入 $ passwd username 可以修改你所管理的用户的密码；否则只能修改你自己的密码（不需要提供 username）。

修改用户

usermod 命令可以修改现有用户的信息。usermod 命令的选项和 useradd 相同，不过可以增加 -l 选项来更改用户名。

下面的例子将用户 mcmohd 的用户名修改为 mcmohd20，主目录修改为 /home/mcmohd20：

```
$ usermod -d /home/mcmohd20 -m -l mcmohd mcmohd20
```

删除用户

userdel 命令可以用来删除现有用户。userdel 是一个危险的命令，请谨慎使用。

userdel 命令仅有一个选项 -r，用来删除用户主目录和本地邮件。例如，删除用户 mcmohd20：

```
$ userdel -r mcmohd20
```

为了便于恢复被误删的用户，可以忽略 -r 选项，保留用户主目录，之后确认无误可以随时删除主目录。

Linux 系统性能分析

这篇教程的目的是向大家介绍一些免费的系统性能分析工具（命令），使用这些工具可以监控系统资源使用情况，便于发现性能瓶颈。

系统的整体性能取决于各种资源的平衡，类似木桶理论，某种资源的耗尽会严重阻碍系统的性能。

Linux 中需要监控的资源主要有 CPU、主存（内存）、硬盘空间、I/O 时间、网络时间、应用程序等。

影响系统性能的主要因素有：

因素	说明
用户态 CPU	CPU 在用户态运行用户程序所花费的时间，包括库调用，但是不包括内核花费的时间。
内核态 CPU	CPU 在内核态运行系统服务所花费的时间。所有的 I/O 操作都需要调用系统服务，程序员可以通过阻塞 I/O 传输来影响这部分的时间。
I/O 时间和网络时间	响应 I/O 请求、处理网络连接所花费的时间。
内存	切换上下文和交换数据（虚拟内存页导入和导出）花费的时间。
应用程序	程序等待运行的时间——CPU 正在运行其他程序，等待切换到当前程序。

说明：一般认为用户态 CPU 和内核态 CPU 花费的时间小于 70%时是良好状态。

下面的命令可以用来监控系统性能并作出相应调整：

命令	说明
nice	启动程序时指定进程优先级。
renice	调整现有进程的优先级。
netstat	显示各种网络相关信息，包括网络连接情况、路由表、接口状态(Interface Statistics)、masquerade 连接、多播成员 (Multicast Memberships)等。实际上，netstat 用于显示与 IP、TCP、UDP 和 ICMP 协议相关的统计数据，一般用于检验本机各端口的网络连接情况。
time	检测一个命令运行时间以及资源（CPU、内存、I/O 等）使用情况。
uptime	查看系统负载情况。
ps	查看系统中进程的资源使用情况（瞬时状态，不是动态监控）。
vmstat	报告虚拟内存使用情况。
gprof	精确分析程序的性能，能给出函数调用时间、调用次数、调用关系等。
top	实时监控系统中各个进程资源的资源使用情况。

一开始执行程序就指定 nice 值：nice
nice -n -5 /usr/local/mysql/bin/mysqld_safe &

linux nice 命令详解

功能说明：设置优先权。

语　　法：nice [-n <优先等级>][--help][--version][执行指令]

补充说明：nice 指令可以改变程序执行的优先权等级。

参　　数：-n<优先等级>或-<优先等级>或–adjustment=<优先等级>　　设置欲执行的指令的优先权等级。等级的范围从-20-19，其中-20 最高，19 最低，只有系统管理者可以设置负数的等级。
–help　　在线帮助。
–version　　显示版本信息。
————————————————————————————————

2.1、调整已存在进程的 nice：renice

renice -5 -p 5200
#PID 为 5200 的进程 nice 设为-5
linux renice 命令详解

功能说明：调整优先权。

语　　法：renice [优先等级][-g <程序群组名称>...][-p <程序识别码>...][-u <用户名称>...]

补充说明：renice 指令可重新调整程序执行的优先权等级。预设是以程序识别码指定程序调整其优先权，您亦可以指定程序群组或用户名称调整优先权等级，并修改所有隶属于该程序群组或用户的程序的优先权。等级范围从-20–19，只有系统管理者可以改变其他用户程序的优先权，也仅有系统管理者可以设置负数等级。
参　　数：
-g <程序群组名称>　　使用程序群组名称，修改所有隶属于该程序群组的程序的优先权。
-p <程序识别码>　　改变该程序的优先权等级，此参数为预设值。
-u <用户名称>　　指定用户名称，修改所有隶属于该用户的程序的优先权。

常用命令组合：

- vmstat、sar、mpstat 检测是否存在 CPU 瓶颈；
- vmstat、free 检测是否存在内存瓶颈；
- iostat 检测是否存在磁盘 I/O 瓶颈；
- netstat 检测是否存在网络 I/O 瓶颈。

Linux 系统日志及日志分析

Linux 系统拥有非常灵活和强大的日志功能，可以保存几乎所有的操作记录，并可以从中检索出我们需要的信息。

大部分 Linux 发行版默认的日志守护进程为 syslog，位于 /etc/syslog 或 /etc/syslogd，默认配置文件为 /etc/syslog.conf，任何希望生成日志的程序都可以向 syslog 发送信息。

Linux 系统内核和许多程序会产生各种错误信息、警告信息和其他的提示信息，这些信息对管理员了解系统的运行状态是非常有用的，所以应该把它们写到日志文件中去。完成这个过程的程序就是 syslog。syslog 可以根据日志的类别和优先级将日志保存到不同的文件中。例如，为了方便查阅，可以把内核信息与其他信息分开，单独保存到一个独立的日志文件中。默认配置下，日志文件通常都保存在“/var/log”目录下。

日志类型

下面是常见的日志类型，但并不是所有的 Linux 发行版都包含这些类型：

类型	说明
auth	用户认证时产生的日志，如 login 命令、su 命令。
authpriv	与 auth 类似，但是只能被特定用户查看。
console	针对系统控制台的消息。
cron	系统定期执行计划任务时产生的日志。
daemon	某些守护进程产生的日志。
ftp	FTP 服务。
kern	系统内核消息。
local0.local7	由自定义程序使用。
lpr	与打印机活动有关。
mail	邮件日志。
mark	产生时间戳。系统每隔一段时间向日志文件中输出当前时间，每行的格式类似于 May 26 11:17:09 rs2 -- MARK --，可以由此推断系统发生故障的大概时间。
news	网络新闻传输协议(nntp)产生的消息。
ntp	网络时间协议(ntp)产生的消息。
user	用户进程。
uucp	UUCP 子系统。

日志优先级

常见的日志优先级请见下标：

优先级	说明
emerg	紧急情况，系统不可用（例如系统崩溃），一般会通知所有用户。
alert	需要立即修复，例如系统数据库损坏。
crit	危险情况，例如硬盘错误，可能会阻碍程序的部分功能。
err	一般错误消息。
warning	警告。
notice	不是错误，但是可能需要处理。
info	通用性消息，一般用来提供有用信息。
debug	调试程序产生的信息。

none	没有优先级，不记录任何日志消息。

常见日志文件

所有的系统应用都会在 /var/log 目录下创建日志文件，或创建子目录再创建日志文件。例如：

文件/目录	说明
/var/log/boot.log	开启或重启日志。
/var/log/cron	计划任务日志
/var/log/maillog	邮件日志。
/var/log/messages	该日志文件是许多进程日志文件的汇总，从该文件可以看出任何入侵企图或成功的入侵。
/var/log/httpd 目录	Apache HTTP 服务日志。
/var/log/samba 目录	samba 软件日志

/etc/syslog.conf 文件

/etc/syslog.conf 是 syslog 的配置文件，会根据日志类型和优先级来决定将日志保存到何处。典型的 syslog.conf 文件格式如下所示：

```
*.err;kern.debug;auth.notice /dev/console
daemon,auth.notice           /var/log/messages
lpr.info                     /var/log/lpr.log
mail.*                       /var/log/mail.log
ftp.*                        /var/log/ftp.log
auth.*                       @see.xidian.edu.cn
auth.*                       root,amrood
netinfo.err                  /var/log/netinfo.log
install.*                    /var/log/install.log
*.emerg                      *
*.alert                      |program_name
mark.*                       /dev/console
```

第一列为日志类型和日志优先级的组合，每个类型和优先级的组合称为一个选择器；后面一列为保存日志的文件、服务器，或输出日志的终端。syslog 进程根据选择器决定如何操作日志。

对配置文件的几点说明：

- 日志类型和优先级由点号(.)分开，例如 kern.debug 表示由内核产生的调试信息。

- kern.debug 的优先级大于 debug。
- 星号(*)表示所有，例如 *.debug 表示所有类型的调试信息，kern.* 表示由内核产生的所有消息。
- 可以使用逗号(,)分隔多个日志类型，使用分号(;)分隔多个选择器。

对日志的操作包括：

- 将日志输出到文件，例如 /var/log/maillog 或 /dev/console。
- 将消息发送给用户，多个用户用逗号(,)分隔，例如 root, amrood。
- 通过管道将消息发送给用户程序，注意程序要放在管道符(|)后面。
- 将消息发送给其他主机上的 syslog 进程，这时 /etc/syslog.conf 文件后面一列为以@开头的主机名，例如@see.xidian.edu.cn。

logger 命令

logger 是 Shell 命令，可以通过该命令使用 syslog 的系统日志模块，还可以从命令行直接向系统日志文件写入一行信息。

logger 命令的语法为：

```
logger [-i] [-f filename] [-p priority] [-t tag] [message...]
```

每个选项的含义如下：

选项	说明
-f filename	将 filename 文件的内容作为日志。
-i	每行都记录 logger 进程的 ID。
-p priority	指定优先级；优先级必须是形如 facility.priority 的完整的选择器，默认优先级为 user.notice。
-t tag	使用指定的标签标记每一个记录行。
message	要写入的日志内容，多条日志以空格为分隔；如果没有指定日志内容，并且 -f filename 选项为空，那么会把标准输入作为日志内容。

例如，将 ping 命令的结果写入日志：

```
$ ping 192.168.0.1 | logger -it logger_test -p local3.notice&
$ tail -f /var/log/userlog
Oct 6 12:48:43 kevein logger_test[22484]: PING 192.168.0.1 (192.168.0.1) 56(84) bytes of data.
Oct 6 12:48:43 kevein logger_test[22484]: 64 bytes from 192.168.0.1: icmp_seq=1 ttl=253 time=49.7 ms
Oct 6 12:48:44 kevein logger_test[22484]: 64 bytes from 192.168.0.1: icmp_seq=2 ttl=253 time=68.4 ms
```

```
Oct 6 12:48:45 kevein logger_test[22484]: 64 bytes from 192.168.0.1: icmp_seq=3 ttl=253 time=315
ms
Oct 6 12:48:46 kevein logger_test[22484]: 64 bytes from 192.168.0.1: icmp_seq=4 ttl=253 time=279
ms
Oct 6 12:48:47 kevein logger_test[22484]: 64 bytes from 192.168.0.1: icmp_seq=5 ttl=253 time=347
ms
Oct 6 12:48:49 kevein logger_test[22484]: 64 bytes from 192.168.0.1: icmp_seq=6 ttl=253 time=701
ms
Oct 6 12:48:50 kevein logger_test[22484]: 64 bytes from 192.168.0.1: icmp_seq=7 ttl=253 time=591
ms
Oct 6 12:48:51 kevein logger_test[22484]: 64 bytes from 192.168.0.1: icmp_seq=8 ttl=253 time=592
ms
Oct 6 12:48:52 kevein logger_test[22484]: 64 bytes from 192.168.0.1: icmp_seq=9 ttl=253 time=611
ms
Oct 6 12:48:53 kevein logger_test[22484]: 64 bytes from 192.168.0.1: icmp_seq=10 ttl=253 time=931
ms
```

ping 命令的结果成功输出到 /var/log/userlog 文件。

命令 logger -it logger_test -p local3.notice 各选项的含义：

- -i：在每行都记录进程 ID；
- -t logger_test：每行记录都加上“logger_test”这个标签；
- -p local3.notice：设置日志类型和优先级。

日志转储

日志转储也叫日志回卷或日志轮转。Linux 中的日志通常增长很快，会占用大量硬盘空间，需要在日志文件达到指定大小时分开存储。

syslog 只负责接收日志并保存到相应的文件，但不会对日志文件进行管理，因此经常会造成日志文件过大，尤其是 WEB 服务器，轻易就能超过 1G，给检索带来困难。

大多数 Linux 发行版使用 logrotate 或 newsyslog 对日志进行管理。logrotate 程序不但可以压缩日志文件，减少存储空间，还可以将日志发送到指定 E-mail，方便管理员及时查看日志。

例如，规定邮件日志 /var/log/maillog 超过 1G 时转储，每周一次，那么每隔一周 logrotate 进程就会检查 /var/log/maillog 文件的大小：

- 如果没有超过 1G，不进行任何操作。
- 如果在 1G~2G 之间，就会创建新文件 /var/log/maillog.1，并将多出的 1G 日志转移到该文件，以给 /var/log/maillog 文件瘦身。

- 如果在 2G~3G 之间，会继续创建新文件 /var/log/maillog.2，并将 /var/log/maillog.1 的内容转移到该文件，将 /var/log/maillog 的内容转移到 /var/log/maillog.1，以保持 /var/log/maillog 文件不超过 1G。

可以看到，每次转存都会创建一个新文件（如果不存在），命名格式为日志文件名加一个数字（从 1 开始自动增长），以保持当前日志文件和转存后的日志文件不超过指定大小。

logrotate 的主要配置文件是 /etc/logrotate.conf，/etc/logrotate.d 目录是对 /etc/logrotate.conf 的补充，或者说为了不使 /etc/logrotate.conf 过大而设置。

可以通过 cat 命令查看它的内容：

```
$cat /etc/logrotate.conf
# see "man logrotate" for details  //可以查看帮助文档
# rotate log files weekly
weekly                              //设置每周转储一次
# keep 4 weeks worth of backlogs
rotate 4                            //最多转储 4 次
# create new (empty) log files after rotating old ones
create                              //当转储后文件不存储时创建它
# uncomment this if you want your log files compressed
#compress                           //以压缩方式转储
# RPM packages drop log rotation information into this directory
include /etc/logrotate.d            //其他日志文件的转储方式，包含在该目录下
# no packages own wtmp -- we'll rotate them here
/var/log/wtmp {                     //设置/var/log/wtmp 日志文件的转储参数
    monthly                         //每月转储
    create 0664 root utmp          //转储后文件不存在时创建它，文件所有者为 root，所属组为 utmp，对应的权限为 0664
    rotate 1                        //最多转储一次
}
```

注意：include 允许管理员把多个分散的文件集中到一个，类似于 C 语言的 #include，将其他文件的内容包含进当前文件。

include 非常有用，一些程序会把转储日志的配置文件放在 /etc/logrotate.d 目录，这些配置文件会覆盖或增加 /etc/logrotate.conf 的配置项，如果没有指定相关配置，那么采用 /etc/logrotate.conf 的默认配置。

所以，建议将 /etc/logrotate.conf 作为默认配置文件，第三方程序在 /etc/logrotate.d 目录下自定义配

置文件。

logrotate 也可以作为命令直接运行来修改配置文件。

Linux 信号机制与信号处理

信号(signal)是 Linux 进程间通信的一种机制，全称为软中断信号，也被称为软中断。信号本质上是在软件层次上对硬件中断机制的一种模拟。

与其他进程间通信方式（例如管道、共享内存等）相比，信号所能传递的信息比较粗糙，只是一个整数。但正是由于传递的信息量少，信号也便于管理和使用，可以用于系统管理相关的任务，例如通知进程终结、中止或者恢复等。

每种信号用一个整型常量宏表示，以 SIG 开头，比如 SIGCHLD、SIGINT 等，它们在系统头文件<signal.h>中定义。

信号由内核(kernel)管理，产生方式多种多样：

- 可以由内核自身产生，比如出现硬件错误、内存读取错误，分母为 0 的除法等，内核需要通知相应进程。
- 也可以由其他进程产生并发送给内核，再由内核传递给目标进程。

信号传递的过程：

- 内核中针对每一个进程都有一个表来保存信号。
- 当内核需要将信号传递给某个进程时，就在该进程对应的表中写入信号，这样就生成了信号。
- 当该进程由用户态陷入内核态，再次切换到用户态之前，会查看表中的信号。如果有信号，进程就会首先执行信号对应的操作，此时叫做执行信号。
- 从生成信号到将信号传递给对应进程这段时间，信号处于等待状态。
- 我们可以编写代码，让进程阻塞(block)某些信号，也就是让这些信号始终处于等待的状态，直到进程取消阻塞(unblock)或者忽略信号。

信号种类

下表列出了一些常见信号：

信号名称	数字表示	说明
SIGHUP	1	终端挂起或控制进程终止。当用户退出 Shell 时，由该进程启动的所有进程都会收到这个信号，默认动作为终止进程。
SIGINT	2	键盘中断。当用户按下<Ctrl+C>组合键时，用户终端向正在运行中的由该终端启动的程序发出此信号。默认动作为终止进程。
SIGQUIT	3	键盘退出键被按下。当用户按下<Ctrl+D>或<Ctrl+\>组合键时，用户终端向正在运行中的由该终端启动的程序发出此信号。默认动作为退出程序。

SIGFPE	8	发生致命的运算错误时发出。不仅包括浮点运算错误，还包括溢出及除数为 0 等所有的算法错误。默认动作为终止进程并产生 core 文件。
SIGKILL	9	无条件终止进程。进程接收到该信号会立即终止，不进行清理和暂存工作。该信号不能被忽略、处理和阻塞，它向系统管理员提供了可以杀死任何进程的方法。
SIGALRM	14	定时器超时，默认动作为终止进程。
SIGTERM	15	程序结束信号，可以由 kill 命令产生。与 SIGKILL 不同的是，SIGTERM 信号可以被阻塞和终止，以便程序在退出前可以保存工作或清理临时文件等。

通过 kill -l 命令可以查看系统支持的所有信号：

```
$ kill -l
1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

上面仅是一个演示，不同的 Linux 发行版支持的信号可能不同。

每种信号都会有一个默认动作。默认动作就是脚本或程序接收到该信号所做出的默认操作。常见的默认动作有终止进程、退出程序、忽略信号、重启暂停的进程等，上表中也对部分默认动作进行了说明。

发送信号

有多种方式可以向程序或脚本发送信号，例如按下<Ctrl+C>组合键会发送 SIGINT 信号，终止当前进程。

还可以通过 kill 命令发送信号，语法为：

```
$ kill -signal pid
```

signal 为要发送的信号，可以是信号名称或数字；pid 为接收信号的进程 ID。例如：

```
$ kill -1 1001
```

将 SIGHUP 信号发送给进程 ID 为 1001 的程序，程序会终止执行。

又如，强制杀死 ID 为 1001 的进程：

```
$ kill -9 1001
```

捕获信号

通常情况下，直接终止进程并不是我们所希望的。例如，按下<Ctrl+C>，进程被立即终止，不会清理创建的临时文件，带来系统垃圾，也不会保存正在进行的工作，导致需要重做。

可以通过编程来捕获这些信号，当终止信号出现时，可以先进行清场和保存处理，再退出程序。

用户程序可以通过 C/C++等代码捕获信号，这将在 Linux C 编程中进行讲解，这里仅介绍如果通过 Linux 命令捕获信号。

通过 trap 命令就可以捕获信号，语法为：

```
$ trap commands signals
```

commands 为 Linux 系统命令或用户自定义命令；signals 为要捕获的信号，可以为信号名称或数字。

捕获到信号后，可以有三种处理：

- 执行一段脚本来做一些处理工作，例如清理临时文件；
- 接受（恢复）信号的默认操作；
- 忽略当前信号。

1）清理临时文件

脚本捕获到终止信号后一个常见的动作就是清理临时文件。例如：

```
$ trap "rm -f $WORKDIR/work1$$ $WORKDIR/dataout$$; exit" 2
```

当用户按下<Ctrl+C>后，脚本先清理临时文件 work1$$ 和 dataout$$ 再退出。

注意：exit 命令是必须的，否则脚本捕获到信号后会继续执行而不是退出。

修改上面的脚本，使接收到 SIGHUP 时进行同样的操作：

```
$ trap "rm $WORKDIR/work1$$ $WORKDIR/dataout$$; exit" 1 2
```

几点注意：

- 如果执行多个命令，需要将命令用引号包围；
- 只有脚本执行到 trap 命令时才会捕获信号；
- 再次接收到信号时还会执行同样的操作。

上面的脚本，执行到 trap 命令时就会替换 WORKDIR 和 $$ 的值。如果希望接收到 SIGHUP 或 SIGINT 信号时再替换其值，那么可以将命令放在单引号内，例如：

```
$ trap 'rm $WORKDIR/work1$$ $WORKDIR/dataout$$; exit' 1 2
```

2）忽略信号

如果 trap 命令的 commands 为空，将会忽略接收到的信号，即不做任何处理，也不执行默认动作。例如：

```
$ trap '' 2
```

也可以同时忽略多个信号：

```
$ trap '' 1 2 3 15
```

注意：必须被引号包围，不能写成下面的形式：

```
$ trap  2
```

3）恢复默认动作

如果希望改变信号的默认动作后再次恢复默认动作，那么省略 trap 命令的 commands 即可，例如：

```
$ trap 1 2
```

将恢复 SIGHUP 和 SIGINT 信号的默认动作。

Shell 简介：什么是 Shell，Shell 命令的两种执行方式

Shell本身是一个用C语言编写的程序,它是用户使用Unix/Linux的桥梁,用户的大部分工作都是通过Shell完成的。Shell 既是一种命令语言，又是一种程序设计语言。作为命令语言，它交互式地解释和执行用户输入的命令；作为程序设计语言，它定义了各种变量和参数，并提供了许多在高级语言中才具有的控制结构，包括循环和分支。

它虽然不是 Unix/Linux 系统内核的一部分，但它调用了系统核心的大部分功能来执行程序、建立文件并以并行的方式协调各个程序的运行。因此，对于用户来说，shell 是最重要的实用程序，深入了解和熟练掌握 shell 的特性极其使用方法，是用好 Unix/Linux 系统的关键。

可以说，shell 使用的熟练程度反映了用户对 Unix/Linux 使用的熟练程度。

注意：单独地学习 Shell 是没有意义的，请先参考 Unix/Linux 入门教程，了解 Unix/Linux 基础。

Shell 有两种执行命令的方式：

- 交互式（Interactive）：解释执行用户的命令，用户输入一条命令，Shell 就解释执行一条。

- 批处理（Batch）：用户事先写一个 Shell 脚本(Script)，其中有很多条命令，让 Shell 一次把这些命令执行完，而不必一条一条地敲命令。

Shell 脚本和编程语言很相似，也有变量和流程控制语句，但 Shell 脚本是解释执行的，不需要编译，Shell 程序从脚本中一行一行读取并执行这些命令，相当于一个用户把脚本中的命令一行一行敲到 Shell 提示符下执行。

Shell 初学者请注意，在平常应用中，建议不要用 root 帐号运行 Shell 。作为普通用户，不管您有意还是无意，都无法破坏系统；但如果是 root，那就不同了，只要敲几个字母，就可能导致灾难性后果。

上面提到过，Shell 是一种脚本语言，那么，就必须有解释器来执行这些脚本。

Unix/Linux 上常见的 Shell 脚本解释器有 bash、sh、csh、ksh 等，习惯上把它们称作一种 Shell。我们常说有多少种 Shell，其实说的是 Shell 脚本解释器。

bash

bash 是 Linux 标准默认的 shell，本教程也基于 bash 讲解。bash 由 Brian Fox 和 Chet Ramey 共同完成，是 BourneAgain Shell 的缩写，内部命令一共有 40 个。

Linux 使用它作为默认的 shell 是因为它有诸如以下的特色：

- 可以使用类似 DOS 下面的 doskey 的功能，用方向键查阅和快速输入并修改命令。
- 自动通过查找匹配的方式给出以某字符串开头的命令。
- 包含了自身的帮助功能，你只要在提示符下面键入 help 就可以得到相关的帮助。

sh

sh 由 Steve Bourne 开发，是 Bourne Shell 的缩写，sh 是 Unix 标准默认的 shell。

ash

ash shell 是由 Kenneth Almquist 编写的，Linux 中占用系统资源最少的一个小 shell，它只包含 24 个内部命令，因而使用起来很不方便。

csh

csh 是 Linux 比较大的内核，它由以 William Joy 为代表的共计 47 位作者编成，共有 52 个内部命令。该 shell 其实是指向/bin/tcsh 这样的一个 shell，也就是说，csh 其实就是 tcsh。

ksh

ksh 是 Korn shell 的缩写，由 Eric Gisin 编写，共有 42 条内部命令。该 shell 最大的优点是几乎和商业发行版的 ksh 完全兼容，这样就可以在不用花钱购买商业版本的情况下尝试商业版本的性能了。

注意：bash 是 Bourne Again Shell 的缩写，是 linux 标准的默认 shell ，它基于 Bourne shell，吸收了 C shell 和 Korn shell 的一些特性。bash 完全兼容 sh，也就是说，用 sh 写的脚本可以不加修改的在 bash 中执行。

Shell 脚本语言与编译型语言的差异

大体上，可以将程序设计语言可以分为两类：编译型语言和解释型语言。

编译型语言

很多传统的程序设计语言，例如 Fortran、Ada、Pascal、C、C++和 Java，都是编译型语言。这类语言需要预先将我们写好的源代码(source code)转换成目标代码(object code)，这个过程被称作“编译”。

运行程序时，直接读取目标代码(object code)。由于编译后的目标代码(object code)非常接近计算机底层，因此执行效率很高，这是编译型语言的优点。

但是，由于编译型语言多半运作于底层，所处理的是字节、整数、浮点数或是其他机器层级的对象，往往实现一个简单的功能需要大量复杂的代码。例如，在 C++里，就很难进行“将一个目录里所有的文件复制到另一个目录中”之类的简单操作。

解释型语言

解释型语言也被称作“脚本语言”。执行这类程序时，解释器(interpreter)需要读取我们编写的源代码(source code)，并将其转换成目标代码(object code)，再由计算机运行。因为每次执行程序都多了编译的过程，因此效率有所下降。

使用脚本编程语言的好处是，它们多半运行在比编译型语言还高的层级，能够轻易处理文件与目录之类的对象；缺点是它们的效率通常不如编译型语言。不过权衡之下，通常使用脚本编程还是值得的：花一个小时写成的简单脚本，同样的功能用 C 或 C++来编写实现，可能需要两天，而且一般来说，脚本执行的速度已经够快了，快到足以让人忽略它性能上的问题。脚本编程语言的例子有 awk、Perl、Python、Ruby 与 Shell。

什么时候使用 Shell

因为 Shell 似乎是各 UNIX 系统之间通用的功能，并且经过了 POSIX 的标准化。因此，Shell 脚本只要“用心写”一次，即可应用到很多系统上。因此，之所以要使用 Shell 脚本是基于：

- 简单性：Shell 是一个高级语言；通过它，你可以简洁地表达复杂的操作。
- 可移植性：使用 POSIX 所定义的功能，可以做到脚本无须修改就可在不同的系统上执行。
- 开发容易：可以在短时间内完成一个功能强大又好用的脚本。

但是，考虑到 Shell 脚本的命令限制和效率问题，下列情况一般不使用 Shell：

1. 资源密集型的任务，尤其在需要考虑效率时（比如，排序，hash 等等）。
2. 需要处理大任务的数学操作，尤其是浮点运算，精确运算，或者复杂的算术运算（这种情况一般使用 C++ 或 FORTRAN 来处理）。
3. 有跨平台（操作系统）移植需求（一般使用 C 或 Java）。
4. 复杂的应用，在必须使用结构化编程的时候（需要变量的类型检查，函数原型，等等）。
5. 对于影响系统全局性的关键任务应用。
6. 对于安全有很高要求的任务，比如你需要一个健壮的系统来防止入侵、破解、恶意破坏等等。
7. 项目由连串的依赖的各个部分组成。
8. 需要大规模的文件操作。
9. 需要多维数组的支持。
10. 需要数据结构的支持，比如链表或数等数据结构。
11. 需要产生或操作图形化界面 GUI。
12. 需要直接操作系统硬件。
13. 需要 I/O 或 socket 接口。
14. 需要使用库或者遗留下来的老代码的接口。
15. 私人的、闭源的应用（shell 脚本把代码就放在文本文件中，全世界都能看到）。

第一个 Shell 脚本

打开文本编辑器，新建一个文件，扩展名为 sh（sh 代表 shell），扩展名并不影响脚本执行，见名知意就好，如果你用 php 写 shell 脚本，扩展名就用 php 好了。

输入一些代码：

```
#!/bin/bash
echo "Hello World !"
```

“#!” 是一个约定的标记，它告诉系统这个脚本需要什么解释器来执行，即使用哪一种 Shell。echo 命令用于向窗口输出文本。

运行 Shell 脚本有两种方法。

作为可执行程序

将上面的代码保存为 test.sh，并 cd 到相应目录：

```
chmod +x ./test.sh  #使脚本具有执行权限
./test.sh  #执行脚本
```

注意，一定要写成./test.sh，而不是 test.sh。运行其它二进制的程序也一样，直接写 test.sh，linux 系统会去 PATH 里寻找有没有叫 test.sh 的，而只有/bin, /sbin, /usr/bin，/usr/sbin 等在 PATH 里，你的当前目录通常不在 PATH 里，所以写成 test.sh 是会找不到命令的，要用./test.sh 告诉系统说，就在当前目

录找。

通过这种方式运行 bash 脚本，第一行一定要写对，好让系统查找到正确的解释器。

这里的"系统"，其实就是 shell 这个应用程序（想象一下 Windows Explorer），但我故意写成系统，是方便理解，既然这个系统就是指 shell，那么一个使用/bin/sh 作为解释器的脚本是不是可以省去第一行呢？是的。

作为解释器参数

这种运行方式是，直接运行解释器，其参数就是 shell 脚本的文件名，如：

```
/bin/sh test.sh
/bin/php test.php
```

这种方式运行的脚本，不需要在第一行指定解释器信息，写了也没用。

再看一个例子。下面的脚本使用 **read** 命令从 stdin 获取输入并赋值给 PERSON 变量,最后在 stdout 上输出：

```
#!/bin/bash

# Author : mozhiyan
# Copyright (c) http://see.xidian.edu.cn/cpp/linux/
# Script follows here:

echo "What is your name?"
read PERSON
echo "Hello, $PERSON"
```

运行脚本：

```
chmod +x ./test.sh
$./test.sh
What is your name?
mozhiyan
Hello, mozhiyan
$
```

Shell 变量：Shell 变量的定义、删除变量、只读变量、变量类型

Shell 支持自定义变量。

定义变量

定义变量时，变量名不加美元符号（$），如：

```
variableName="value"
```

注意，变量名和等号之间不能有空格，这可能和你熟悉的所有编程语言都不一样。同时，变量名的命名须遵循如下规则：

- 首个字符必须为字母（a-z，A-Z）。
- 中间不能有空格，可以使用下划线（_）。
- 不能使用标点符号。
- 不能使用 bash 里的关键字（可用 help 命令查看保留关键字）。

变量定义举例：

```
myUrl="http://see.xidian.edu.cn/cpp/linux/"
myNum=100
```

使用变量

使用一个定义过的变量，只要在变量名前面加美元符号（$）即可，如：

```
your_name="mozhiyan"
echo $your_name
echo ${your_name}
```

变量名外面的花括号是可选的，加不加都行，加花括号是为了帮助解释器识别变量的边界，比如下面这种情况：

```
for skill in Ada Coffe Action Java
do
    echo "I am good at ${skill}Script"
done
```

如果不给 skill 变量加花括号，写成 echo "I am good at $skillScript"，解释器就会把$skillScript 当成一个变量（其值为空），代码执行结果就不是我们期望的样子了。

推荐给所有变量加上花括号，这是个好的编程习惯。

重新定义变量

已定义的变量，可以被重新定义，如：

```
myUrl="http://see.xidian.edu.cn/cpp/linux/"
echo ${myUrl}
```

```
3.
4. myUrl="http://see.xidian.edu.cn/cpp/shell/"
5. echo ${myUrl}
```

这样写是合法的，但注意，第二次赋值的时候不能写 $myUrl="http://see.xidian.edu.cn/cpp/shell/"，使用变量的时候才加美元符（$）。

只读变量

使用 **readonly** 命令可以将变量定义为只读变量，只读变量的值不能被改变。

下面的例子尝试更改只读变量，结果报错：

```
1. #!/bin/bash
2.
3. myUrl="http://see.xidian.edu.cn/cpp/shell/"
4. readonly myUrl
5. myUrl="http://see.xidian.edu.cn/cpp/danpianji/"
```

运行脚本，结果如下：

```
/bin/sh: NAME: This variable is read only.
```

删除变量

使用 **unset** 命令可以删除变量。语法：

```
1. unset variable_name
```

变量被删除后不能再次使用；unset 命令不能删除只读变量。

举个例子：

```
1. #!/bin/sh
2.
3. myUrl="http://see.xidian.edu.cn/cpp/u/xitong/"
4. unset myUrl
5. echo $myUrl
```

上面的脚本没有任何输出。

变量类型

运行 shell 时，会同时存在三种变量：

1）局部变量

局部变量在脚本或命令中定义，仅在当前 shell 实例中有效，其他 shell 启动的程序不能访问局部变量。

2）环境变量

所有的程序，包括 shell 启动的程序，都能访问环境变量，有些程序需要环境变量来保证其正常运行。必要的时候 shell 脚本也可以定义环境变量。

3) shell 变量

shell 变量是由 shell 程序设置的特殊变量。shell 变量中有一部分是环境变量，有一部分是局部变量，这些变量保证了 shell 的正常运行

Shell 特殊变量：Shell $0, $#, $*, $@, $?, $$和命令行参数

前面已经讲到，变量名只能包含数字、字母和下划线，因为某些包含其他字符的变量有特殊含义，这样的变量被称为特殊变量。

例如，$ 表示当前 Shell 进程的 ID，即 pid，看下面的代码：

```
$echo $$
```

运行结果

```
29949
```

特殊变量列表

变量	含义
$0	当前脚本的文件名
$n	传递给脚本或函数的参数。n 是一个数字，表示第几个参数。例如，第一个参数是$1，第二个参数是$2。
$#	传递给脚本或函数的参数个数。
$*	传递给脚本或函数的所有参数。
$@	传递给脚本或函数的所有参数。被双引号(" ")包含时，与 $* 稍有不同，下面将会讲到。
$?	上个命令的退出状态，或函数的返回值。
$$	当前 Shell 进程 ID。对于 Shell 脚本，就是这些脚本所在的进程 ID。

命令行参数

运行脚本时传递给脚本的参数称为命令行参数。命令行参数用 $n 表示，例如，$1 表示第一个参数，$2 表示第二个参数，依次类推。

请看下面的脚本：

```
#!/bin/bash

echo "File Name: $0"
echo "First Parameter : $1"
echo "First Parameter : $2"
echo "Quoted Values: $@"
echo "Quoted Values: $*"
echo "Total Number of Parameters : $#"
```

运行结果：

```
$./test.sh Zara Ali
File Name : ./test.sh
First Parameter : Zara
Second Parameter : Ali
Quoted Values: Zara Ali
Quoted Values: Zara Ali
Total Number of Parameters : 2
```

$* 和 $@ 的区别

$* 和 $@ 都表示传递给函数或脚本的所有参数，不被双引号(" ")包含时，都以"$1" "$2" … "$n" 的形式输出所有参数。

但是当它们被双引号(" ")包含时，"$*" 会将所有的参数作为一个整体，以"$1 $2 … $n"的形式输出所有参数；"$@" 会将各个参数分开，以"$1" "$2" … "$n" 的形式输出所有参数。

下面的例子可以清楚的看到 $* 和 $@ 的区别：

```
#!/bin/bash
echo "\$*=" $*
echo "\"\$*\"=" "$*"

echo "\$@=" $@
echo "\"\$@\"=" "$@"

echo "print each param from \$*"
for var in $*
do
    echo "$var"
```

```
done

echo "print each param from \$@"
for var in $@
do
    echo "$var"
done

echo "print each param from \"\$*\""
for var in "$*"
do
    echo "$var"
done

echo "print each param from \"\$@\""
for var in "$@"
do
    echo "$var"
done
```

执行 ./test.sh "a" "b" "c" "d"，看到下面的结果：

```
$*= a b c d
"$*"= a b c d
$@= a b c d
"$@"= a b c d
print each param from $*
a
b
c
d
print each param from $@
a
b
c
d
print each param from "$*"
a b c d
print each param from "$@"
a
b
```

```
c
d
```

退出状态

$? 可以获取上一个命令的退出状态。所谓退出状态，就是上一个命令执行后的返回结果。

退出状态是一个数字，一般情况下，大部分命令执行成功会返回 0，失败返回 1。

不过，也有一些命令返回其他值，表示不同类型的错误。

下面例子中，命令成功执行：

```
$./test.sh Zara Ali
File Name : ./test.sh
First Parameter : Zara
Second Parameter : Ali
Quoted Values: Zara Ali
Quoted Values: Zara Ali
Total Number of Parameters : 2
$echo $?
0
$
```

$? 也可以表示函数的返回值，后续将会讲解。

Shell 替换：Shell 变量替换，命令替换，转义字符

如果表达式中包含特殊字符，Shell 将会进行替换。例如，在双引号中使用变量就是一种替换，转义字符也是一种替换。

举个例子：

```
#!/bin/bash

a=10
echo -e "Value of a is $a \n"
```

运行结果：

```
Value of a is 10
```

这里 -e 表示对转义字符进行替换。如果不使用 -e 选项，将会原样输出：

```
Value of a is 10\n
```

下面的转义字符都可以用在 echo 中：

转义字符	含义
\\	反斜杠
\a	警报，响铃
\b	退格（删除键）
\f	换页(FF)，将当前位置移到下页开头
\n	换行
\r	回车
\t	水平制表符（tab 键）
\v	垂直制表符

可以使用 echo 命令的 -E 选项禁止转义，默认也是不转义的；使用 -n 选项可以禁止插入换行符。

命令替换

命令替换是指 Shell 可以先执行命令，将输出结果暂时保存，在适当的地方输出。

命令替换的语法：

```
1. `command`
```

注意是反引号，不是单引号，这个键位于 Esc 键下方。

下面的例子中，将命令执行结果保存在变量中：

```
1. #!/bin/bash
2.
3. DATE=`date`
4. echo "Date is $DATE"
5.
6. USERS=`who | wc -l`
7. echo "Logged in user are $USERS"
8.
9. UP=`date ; uptime`
10. echo "Uptime is $UP"
```

运行结果：

```
Date is Thu Jul  2 03:59:57 MST 2009
```

```
Logged in user are 1
Uptime is Thu Jul  2 03:59:57 MST 2009
03:59:57 up 20 days, 14:03,  1 user,  load avg: 0.13, 0.07, 0.15
```

变量替换

变量替换可以根据变量的状态（是否为空、是否定义等）来改变它的值

可以使用的变量替换形式：

形式	说明
${var}	变量本来的值
${var:-word}	如果变量 var 为空或已被删除(unset)，那么返回 word，但不改变 var 的值。
${var:=word}	如果变量 var 为空或已被删除(unset),那么返回 word,并将 var 的值设置为 word。
${var:?message}	如果变量 var 为空或已被删除(unset)，那么将消息 message 送到标准错误输出，可以用来检测变量 var 是否可以被正常赋值。 若此替换出现在 Shell 脚本中，那么脚本将停止运行。
${var:+word}	如果变量 var 被定义，那么返回 word，但不改变 var 的值。

请看下面的例子：

```
#!/bin/bash

echo ${var:-"Variable is not set"}
echo "1 - Value of var is ${var}"

echo ${var:="Variable is not set"}
echo "2 - Value of var is ${var}"

unset var
echo ${var:+"This is default value"}
echo "3 - Value of var is $var"

var="Prefix"
echo ${var:+"This is default value"}
echo "4 - Value of var is $var"

echo ${var:?"Print this message"}
```

```
echo "5 - Value of var is ${var}"
```

运行结果：

纯文本复制

```
Variable is not set
1 - Value of var is
Variable is not set
2 - Value of var is Variable is not set

3 - Value of var is
This is default value
4 - Value of var is Prefix
Prefix
5 - Value of var is Prefix
```

Shell 运算符：Shell 算数运算符、关系运算符、布尔运算符、字符串运算符等

Bash 支持很多运算符，包括算数运算符、关系运算符、布尔运算符、字符串运算符和文件测试运算符。

原生 bash 不支持简单的数学运算，但是可以通过其他命令来实现，例如 awk 和 expr，expr 最常用。

expr 是一款表达式计算工具，使用它能完成表达式的求值操作。

例如，两个数相加：

```
#!/bin/bash

val=`expr 2 + 2`
echo "Total value : $val"
```

运行脚本输出：

```
Total value : 4
```

两点注意：

- 表达式和运算符之间要有空格，例如 2+2 是不对的，必须写成 2 + 2，这与我们熟悉的大多数编程语言不一样。
- 完整的表达式要被 `` 包含，注意这个字符不是常用的单引号，在 Esc 键下边。

算术运算符

先来看一个使用算术运算符的例子：

```
#!/bin/sh

a=10
b=20
val=`expr $a + $b`
echo "a + b : $val"

val=`expr $a - $b`
echo "a - b : $val"

val=`expr $a \* $b`
echo "a * b : $val"

val=`expr $b / $a`
echo "b / a : $val"

val=`expr $b % $a`
echo "b % a : $val"

if [ $a == $b ]
then
   echo "a is equal to b"
fi

if [ $a != $b ]
then
   echo "a is not equal to b"
fi
```

运行结果：

```
a + b : 30
a - b : -10
a * b : 200
b / a : 2
b % a : 0
a is not equal to b
```

注意：

- 乘号(*)前边必须加反斜杠(\)才能实现乘法运算；
- if...then...fi 是条件语句，后续将会讲解。

算术运算符列表		
运算符	**说明**	**举例**
+	加法	`expr $a + $b` 结果为 30。
-	减法	`expr $a - $b` 结果为 10。
*	乘法	`expr $a \* $b` 结果为 200。
/	除法	`expr $b / $a` 结果为 2。
%	取余	`expr $b % $a` 结果为 0。
=	赋值	a=$b 将把变量 b 的值赋给 a。
==	相等。用于比较两个数字，相同则返回 true。	[$a == $b] 返回 false。
!=	不相等。用于比较两个数字，不相同则返回 true。	[$a != $b] 返回 true。

注意：条件表达式要放在方括号之间，并且要有空格，例如 [$a==$b] 是错误的，必须写成 [$a == $b]。

关系运算符

关系运算符只支持数字，不支持字符串，除非字符串的值是数字。

先来看一个关系运算符的例子：

```
#!/bin/sh

a=10
b=20
if [ $a -eq $b ]
then
   echo "$a -eq $b : a is equal to b"
else
   echo "$a -eq $b: a is not equal to b"
fi

if [ $a -ne $b ]
then
   echo "$a -ne $b: a is not equal to b"
else
   echo "$a -ne $b : a is equal to b"
fi

```

```
if [ $a -gt $b ]
then
   echo "$a -gt $b: a is greater than b"
else
   echo "$a -gt $b: a is not greater than b"
fi

if [ $a -lt $b ]
then
   echo "$a -lt $b: a is less than b"
else
   echo "$a -lt $b: a is not less than b"
fi

if [ $a -ge $b ]
then
   echo "$a -ge $b: a is greater or  equal to b"
else
   echo "$a -ge $b: a is not greater or equal to b"
fi

if [ $a -le $b ]
then
   echo "$a -le $b: a is less or  equal to b"
else
   echo "$a -le $b: a is not less or equal to b"
fi
```

运行结果：

```
10 -eq 20: a is not equal to b
10 -ne 20: a is not equal to b
10 -gt 20: a is not greater than b
10 -lt 20: a is less than b
10 -ge 20: a is not greater or equal to b
10 -le 20: a is less or  equal to b
```

关系运算符列表

运算符	说明	举例
-eq	检测两个数是否相等，相等返回 true。	[$a -eq $b] 返回 true。
-ne	检测两个数是否相等，不相等返回 true。	[$a -ne $b] 返回 true。

-gt	检测左边的数是否大于右边的，如果是，则返回 true。	[$a -gt $b] 返回 false。
-lt	检测左边的数是否小于右边的，如果是，则返回 true。	[$a -lt $b] 返回 true。
-ge	检测左边的数是否大等于右边的，如果是，则返回 true。	[$a -ge $b] 返回 false。
-le	检测左边的数是否小于等于右边的，如果是，则返回 true。	[$a -le $b] 返回 true。

布尔运算符

先来看一个布尔运算符的例子：

```
#!/bin/sh

a=10
b=20

if [ $a != $b ]
then
   echo "$a != $b : a is not equal to b"
else
   echo "$a != $b: a is equal to b"
fi

if [ $a -lt 100 -a $b -gt 15 ]
then
   echo "$a -lt 100 -a $b -gt 15 : returns true"
else
   echo "$a -lt 100 -a $b -gt 15 : returns false"
fi

if [ $a -lt 100 -o $b -gt 100 ]
then
   echo "$a -lt 100 -o $b -gt 100 : returns true"
else
   echo "$a -lt 100 -o $b -gt 100 : returns false"
fi

if [ $a -lt 5 -o $b -gt 100 ]
then
   echo "$a -lt 100 -o $b -gt 100 : returns true"
else
   echo "$a -lt 100 -o $b -gt 100 : returns false"
fi
```

运行结果：

```
10 != 20 : a is not equal to b
10 -lt 100 -a 20 -gt 15 : returns true
10 -lt 100 -o 20 -gt 100 : returns true
10 -lt 5 -o 20 -gt 100 : returns false
```

布尔运算符列表

运算符	说明	举例
!	非运算，表达式为 true 则返回 false，否则返回 true。	[! false] 返回 true。
-o	或运算，有一个表达式为 true 则返回 true。	[$a -lt 20 -o $b -gt 100] 返回 true。
-a	与运算，两个表达式都为 true 才返回 true。	[$a -lt 20 -a $b -gt 100] 返回 false。

字符串运算符

先来看一个例子：

```
#!/bin/sh

a="abc"
b="efg"

if [ $a = $b ]
then
   echo "$a = $b : a is equal to b"
else
   echo "$a = $b: a is not equal to b"
fi

if [ $a != $b ]
then
   echo "$a != $b : a is not equal to b"
else
   echo "$a != $b: a is equal to b"
fi

if [ -z $a ]
then
   echo "-z $a : string length is zero"
else
```

```
    echo "-z $a : string length is not zero"
fi

if [ -n $a ]
then
    echo "-n $a : string length is not zero"
else
    echo "-n $a : string length is zero"
fi

if [ $a ]
then
    echo "$a : string is not empty"
else
    echo "$a : string is empty"
fi
```

运行结果：

```
abc = efg: a is not equal to b
abc != efg : a is not equal to b
-z abc : string length is not zero
-n abc : string length is not zero
abc : string is not empty
```

字符串运算符列表

运算符	说明	举例
=	检测两个字符串是否相等，相等返回 true。	[$a = $b] 返回 false。
!=	检测两个字符串是否相等，不相等返回 true。	[$a != $b] 返回 true。
-z	检测字符串长度是否为 0，为 0 返回 true。	[-z $a] 返回 false。
-n	检测字符串长度是否为 0，不为 0 返回 true。	[-z $a] 返回 true。
str	检测字符串是否为空，不为空返回 true。	[$a] 返回 true。

文件测试运算符

文件测试运算符用于检测 Unix 文件的各种属性。

例如，变量 file 表示文件“/var/www/tutorialspoint/unix/test.sh”，它的大小为 100 字节，具有 rwx 权限。下面的代码，将检测该文件的各种属性：

```
#!/bin/sh

file="/var/www/tutorialspoint/unix/test.sh"

if [ -r $file ]
then
   `rm $file`
else
   echo "File does not have read access"
fi

if [ -w $file ]
then
   echo "File has write permission"
else
   echo "File does not have write permission"
fi

if [ -x $file ]
then
   echo "File has execute permission"
else
   echo "File does not have execute permission"
fi

if [ -f $file ]
then
   echo "File is an ordinary file"
else
   echo "This is sepcial file"
fi

if [ -d $file ]
then
   echo "File is a directory"
else
   echo "This is not a directory"
fi

if [ -s $file ]
then
   echo "File size is zero"
else
   echo "File size is not zero"
```

```
fi

if [ -e $file ]
then
    echo "File exists"
else
    echo "File does not exist"
fi
```

运行结果：

```
File has read access
File has write permission
File has execute permission
File is an ordinary file
This is not a directory
File size is zero
File exists
```

文件测试运算符列表

操作符	说明	举例
-b file	检测文件是否是块设备文件，如果是，则返回 true。	[-b $file] 返回 false。
-c file	检测文件是否是字符设备文件，如果是，则返回 true。	[-b $file] 返回 false。
-d file	检测文件是否是目录，如果是，则返回 true。	[-d $file] 返回 false。
-f file	检测文件是否是普通文件（既不是目录，也不是设备文件），如果是，则返回 true。	[-f $file] 返回 true。
-g file	检测文件是否设置了 SGID 位，如果是，则返回 true。	[-g $file] 返回 false。
-k file	检测文件是否设置了粘着位(Sticky Bit)，如果是，则返回 true。	[-k $file] 返回 false。
-p file	检测文件是否是具名管道，如果是，则返回 true。	[-p $file] 返回 false。
-u file	检测文件是否设置了 SUID 位，如果是，则返回 true。	[-u $file] 返回 false。

-r file	检测文件是否可读，如果是，则返回 true。	[-r $file] 返回 true。
-w file	检测文件是否可写，如果是，则返回 true。	[-w $file] 返回 true。
-x file	检测文件是否可执行，如果是，则返回 true。	[-x $file] 返回 true。
-s file	检测文件是否为空（文件大小是否大于 0），不为空返回 true。	[-s $file] 返回 true。
-e file	检测文件（包括目录）是否存在，如果是，则返回 true。	[-e $file] 返回 true。

Shell 注释

以“#”开头的行就是注释，会被解释器忽略。

sh 里没有多行注释，只能每一行加一个#号。只能像这样：

```
#--------------------------------------------
# 这是一个自动打 ipa 的脚本，基于 webfrogs 的 ipa-build 书写：
# https://github.com/webfrogs/xcode_shell/blob/master/ipa-build

# 功能：自动为 etao ios app 打包，产出物为 14 个渠道的 ipa 包
# 特色：全自动打包，不需要输入任何参数
#--------------------------------------------

##### 用户配置区 开始 #####
#
#
# 项目根目录，推荐将此脚本放在项目的根目录，这里就不用改了
# 应用名，确保和 Xcode 里 Product 下的 target_name.app 名字一致
#
##### 用户配置区 结束  #####
```

如果在开发过程中，遇到大段的代码需要临时注释起来，过一会儿又取消注释，怎么办呢？每一行加个#符号太费力了，可以把这一段要注释的代码用一对花括号括起来，定义成一个函数，没有地方调用这个函数，这块代码就不会执行，达到了和注释一样的效果。

Shell 字符串

字符串是 shell 编程中最常用最有用的数据类型（除了数字和字符串，也没啥其它类型好用了），字符串可以用单引号，也可以用双引号，也可以不用引号。单双引号的区别跟 PHP 类似。

单引号

```
str='this is a string'
```

单引号字符串的限制：

- 单引号里的任何字符都会原样输出，单引号字符串中的变量是无效的；
- 单引号字串中不能出现单引号（对单引号使用转义符后也不行）。

双引号

```
your_name='qinjx'
str="Hello, I know your are \"$your_name\"! \n"
```

双引号的优点：

- 双引号里可以有变量
- 双引号里可以出现转义字符

拼接字符串

```
your_name="qinjx"
greeting="hello, "$your_name" !"
greeting_1="hello, ${your_name} !"

echo $greeting $greeting_1
```

获取字符串长度

```
string="abcd"
echo ${#string} #输出 4
```

提取子字符串

```
string="alibaba is a great company"
echo ${string:1:4} #输出 liba
```

查找子字符串

纯文本复制

```
string="alibaba is a great company"
echo `expr index "$string" is`
```

Shell 数组：shell 数组的定义、数组长度

Shell 在编程方面比 Windows 批处理强大很多，无论是在循环、运算。

bash 支持一维数组（不支持多维数组），并且没有限定数组的大小。类似与 C 语言，数组元素的下标由 0 开始编号。获取数组中的元素要利用下标，下标可以是整数或算术表达式，其值应大于或等于 0。

定义数组

在 Shell 中，用括号来表示数组，数组元素用“空格”符号分割开。定义数组的一般形式为：

array_name=(value1 ... valuen)

例如：

```
array_name=(value0 value1 value2 value3)
```

或者

```
array_name=(
value0
value1
value2
value3
)
```

还可以单独定义数组的各个分量：

```
array_name[0]=value0
array_name[1]=value1
array_name[2]=value2
```

可以不使用连续的下标，而且下标的范围没有限制。

读取数组

读取数组元素值的一般格式是：

${array_name[index]}

例如：

```
valuen=${array_name[2]}
```

举个例子：

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

```
2.
3. NAME[0]="Zara"
4. NAME[1]="Qadir"
5. NAME[2]="Mahnaz"
6. NAME[3]="Ayan"
7. NAME[4]="Daisy"
8. echo "First Index: ${NAME[0]}"
9. echo "Second Index: ${NAME[1]}"
```

运行脚本，输出：

```
$./test.sh
First Index: Zara
Second Index: Qadir
```

使用@ 或 * 可以获取数组中的所有元素，例如：

```
1. ${array_name[*]}
2. ${array_name[@]}
```

举个例子：

```
1. #!/bin/sh
2.
3. NAME[0]="Zara"
4. NAME[1]="Qadir"
5. NAME[2]="Mahnaz"
6. NAME[3]="Ayan"
7. NAME[4]="Daisy"
8. echo "First Method: ${NAME[*]}"
9. echo "Second Method: ${NAME[@]}"
```

运行脚本，输出：

```
$./test.sh
First Method: Zara Qadir Mahnaz Ayan Daisy
Second Method: Zara Qadir Mahnaz Ayan Daisy
```

获取数组的长度

获取数组长度的方法与获取字符串长度的方法相同，例如：

纯文本复制

```
1. # 取得数组元素的个数
2. length=${#array_name[@]}
```

```
# 或者
length=${#array_name[*]}
# 取得数组单个元素的长度
lengthn=${#array_name[n]}
```

Shell echo 命令

echo 是 Shell 的一个内部指令，用于在屏幕上打印出指定的字符串。命令格式：

```
echo arg
```

您可以使用 echo 实现更复杂的输出格式控制。

显示转义字符

```
echo "\"It is a test\""
```

结果将是：
"It is a test"

双引号也可以省略。

显示变量

```
name="OK"
echo "$name It is a test"
```

结果将是：
OK It is a test

同样双引号也可以省略。

如果变量与其它字符相连的话，需要使用大括号（{}）：

```
mouth=8
echo "${mouth}-1-2009"
```

结果将是：
8-1-2009

显示换行

```
echo "OK!\n"
```

```
2. echo "It is a test"
```

输出:
OK!
It is a test

显示不换行

```
1. echo "OK!\c"
2. echo "It is a test"
```

输出:
OK!It si a test

显示结果重定向至文件

```
1. echo "It is a test" > myfile
```

原样输出字符串

若需要原样输出字符串（不进行转义），请使用单引号。例如:

```
1. echo '$name\"'
```

显示命令执行结果

```
1. echo `date`
```

结果将显示当前日期

shell printf 命令：格式化输出语句

printf 命令用于格式化输出， 是 echo 命令的增强版。它是 C 语言 printf()库函数的一个有限的变形，并且在语法上有些不同。

注意：printf 由 POSIX 标准所定义，移植性要比 echo 好。

如同 echo 命令，printf 命令也可以输出简单的字符串:

```
1. $printf "Hello, Shell\n"
2. Hello, Shell
3. $
```

printf 不像 echo 那样会自动换行，必须显式添加换行符(\n)。

printf 命令的语法：

```
printf  format-string  [arguments...]
```

format-string 为格式控制字符串，arguments 为参数列表。

printf()在 C 语言入门教程中已经讲到，功能和用法与 printf 命令类似，请查看：C 语言格式输出函数 printf()详解

这里仅说明与 C 语言 printf()函数的不同：

- printf 命令不用加括号
- format-string 可以没有引号，但最好加上，单引号双引号均可。
- 参数多于格式控制符(%)时，format-string 可以重用，可以将所有参数都转换。
- arguments 使用空格分隔，不用逗号。

请看下面的例子：

```
# format-string 为双引号
$ printf "%d %s\n" 1 "abc"
1 abc
# 单引号与双引号效果一样
$ printf '%d %s\n' 1 "abc"
1 abc
# 没有引号也可以输出
$ printf %s abcdef
abcdef
# 格式只指定了一个参数，但多出的参数仍然会按照该格式输出，format-string 被重用
$ printf %s abc def
abcdef
$ printf "%s\n" abc def
abc
def
$ printf "%s %s %s\n" a b c d e f g h i j
a b c
d e f
g h i
j
# 如果没有 arguments，那么 %s 用 NULL 代替，%d 用 0 代替
$ printf "%s and %d \n"
and 0
```

```
# 如果以 %d 的格式来显示字符串，那么会有警告，提示无效的数字，此时默认置为 0
$ printf "The first program always prints'%s,%d\n'" Hello Shell
-bash: printf: Shell: invalid number
The first program always prints 'Hello,0'
$
```

注意，根据 POSIX 标准，浮点格式%e、%E、%f、%g 与%G 是“不需要被支持”。这是因为 awk 支持浮点预算，且有它自己的 printf 语句。这样 Shell 程序中需要将浮点数值进行格式化的打印时，可使用小型的 awk 程序实现。然而，内建于 bash、ksh93 和 zsh 中的 printf 命令都支持浮点格式。

Shell if else 语句

if 语句通过关系运算符判断表达式的真假来决定执行哪个分支。Shell 有三种 if ... else 语句：

- if ... fi 语句；
- if ... else ... fi 语句；
- if ... elif ... else ... fi 语句。

1) if ... else 语句

if ... else 语句的语法：

```
if [ expression ]
then
   Statement(s) to be executed if expression is true
fi
```

如果 expression 返回 true，then 后边的语句将会被执行；如果返回 false，不会执行任何语句。

最后必须以 fi 来结尾闭合 if，fi 就是 if 倒过来拼写，后面也会遇见。

注意：expression 和方括号([ ])之间必须有空格，否则会有语法错误。

举个例子：

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
```

```
    echo "a is equal to b"
fi

if [ $a != $b ]
then
    echo "a is not equal to b"
fi
```

运行结果：

```
a is not equal to b
```

2) if ... else ... fi 语句

if ... else ... fi 语句的语法：

```
if [ expression ]
then
   Statement(s) to be executed if expression is true
else
   Statement(s) to be executed if expression is not true
fi
```

如果 expression 返回 true，那么 then 后边的语句将会被执行；否则，执行 else 后边的语句。

举个例子：

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
    echo "a is equal to b"
else
    echo "a is not equal to b"
fi
```

执行结果：

```
a is not equal to b
```

3) if ... elif ... fi 语句

if ... elif ... fi 语句可以对多个条件进行判断，语法为：

```
if [ expression 1 ]
then
   Statement(s) to be executed if expression 1 is true
elif [ expression 2 ]
then
   Statement(s) to be executed if expression 2 is true
elif [ expression 3 ]
then
   Statement(s) to be executed if expression 3 is true
else
   Statement(s) to be executed if no expression is true
fi
```

哪一个 expression 的值为 true，就执行哪个 expression 后面的语句；如果都为 false，那么不执行任何语句。

举个例子：

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
   echo "a is equal to b"
elif [ $a -gt $b ]
then
   echo "a is greater than b"
elif [ $a -lt $b ]
then
   echo "a is less than b"
else
   echo "None of the condition met"
fi
```

运行结果：

```
a is less than b
```

if ... else 语句也可以写成一行，以命令的方式来运行，像这样：

```
if test $[2*3] -eq $[1+5]; then echo 'The two numbers are equal!'; fi;
```

if ... else 语句也经常与 test 命令结合使用，如下所示：

```
num1=$[2*3]
num2=$[1+5]
if test $[num1] -eq $[num2]
then
    echo 'The two numbers are equal!'
else
    echo 'The two numbers are not equal!'
fi
```

输出：

```
The two numbers are equal!
```

test 命令用于检查某个条件是否成立，与方括号([ ])类似。

Shell case esac 语句

case ... esac 与其他语言中的 switch ... case 语句类似，是一种多分枝选择结构。

case 语句匹配一个值或一个模式，如果匹配成功，执行相匹配的命令。case 语句格式如下：

```
case 值 in
模式 1)
    command1
    command2
    command3
    ;;
模式 2）
    command1
    command2
    command3
    ;;
*)
    command1
    command2
    command3
    ;;
esac
```

case 工作方式如上所示。取值后面必须为关键字 in，每一模式必须以右括号结束。取值可以为变量或常数。匹配发现取值符合某一模式后，其间所有命令开始执行直至 ;;。;; 与其他语言中的 break 类似，意思是跳到整个 case 语句的最后。

取值将检测匹配的每一个模式。一旦模式匹配，则执行完匹配模式相应命令后不再继续其他模式。如果无一匹配模式，使用星号 * 捕获该值，再执行后面的命令。

下面的脚本提示输入 1 到 4，与每一种模式进行匹配：

```
echo 'Input a number between 1 to 4'
echo 'Your number is:\c'
read aNum
case $aNum in
    1)  echo 'You select 1'
    ;;
    2)  echo 'You select 2'
    ;;
    3)  echo 'You select 3'
    ;;
    4)  echo 'You select 4'
    ;;
    *)  echo 'You do not select a number between 1 to 4'
    ;;
esac
```

输入不同的内容，会有不同的结果，例如：

```
Input a number between 1 to 4
Your number is:3
You select 3
```

再举一个例子：

```
#!/bin/bash

option="${1}"
case ${option} in
   -f) FILE="${2}"
      echo "File name is $FILE"
      ;;
   -d) DIR="${2}"
      echo "Dir name is $DIR"
      ;;
   *)
```

```
        echo "`basename ${0}`:usage: [-f file] | [-d directory]"
        exit 1 # Command to come out of the program with status 1
        ;;
esac
```

运行结果：

```
$./test.sh
test.sh: usage: [ -f filename ] | [ -d directory ]
$ ./test.sh -f index.htm
$ vi test.sh
$ ./test.sh -f index.htm
File name is index.htm
$ ./test.sh -d unix
Dir name is unix
$
```

Shell for 循环

与其他编程语言类似，Shell 支持 for 循环。

for 循环一般格式为：

```
for 变量 in 列表
do
    command1
    command2
    ...
    commandN
done
```

列表是一组值（数字、字符串等）组成的序列，每个值通过空格分隔。每循环一次，就将列表中的下一个值赋给变量。

in 列表是可选的，如果不用它，for 循环使用命令行的位置参数。

例如，顺序输出当前列表中的数字：

```
for x in one two three four
do

    echo $x
done
```

```
for x in /var/log/*
do
        #echo "$x is a file living in /var/log"
        echo $(basename $x) is a file living in /var/log
done



echo "for: Traditional form: for var in ..."
for j in $(seq 1 5)
do
        echo $j
done

echo "for: C language form: for (( exp1; exp2; exp3 ))"

for (( i=1; i<=5; i++ ))
do
        echo "i=$i"
done
```

显示主目录下以 .bash 开头的文件:

```
#!/bin/bash

for FILE in $HOME/.bash*
do
   echo $FILE
done
```

运行结果:

```
/root/.bash_history
/root/.bash_logout
/root/.bash_profile
/root/.bashrc
```

Shell while 循环

while 循环用于不断执行一系列命令，也用于从输入文件中读取数据；命令通常为测试条件。其格式为:

```
while command
do
     Statement(s) to be executed if command is true
done
```

命令执行完毕，控制返回循环顶部，从头开始直至测试条件为假。

以下是一个基本的 while 循环，测试条件是：如果 COUNTER 小于 5，那么返回 true。COUNTER 从 0 开始，每次循环处理时，COUNTER 加 1。运行上述脚本，返回数字 1 到 5，然后终止。

```
myvar=1
while [ $myvar -le 10 ]
do
        echo $myvar
        myvar=$(( $myvar + 1 ))
done
```

```
i=0;
while [ $i -lt 4 ];
do
      echo $i;
      i=`expr $i + 1`;
      # let i+=1;
      # ((i++));
      # i=$[$i+1];
      # i=$(( $i + 1 ))
done
```

Shell until 循环

until 循环执行一系列命令直至条件为 true 时停止。until 循环与 while 循环在处理方式上刚好相反。一般 while 循环优于 until 循环，但在某些时候，也只是极少数情况下，until 循环更加有用。

until 循环格式为：

```
until command
do
   Statement(s) to be executed until command is true
done
```

command 一般为条件表达式，如果返回值为 false，则继续执行循环体内的语句，否则跳出循环。

例如，使用 until 命令输出 0 ~ 9 的数字：

```
1. #!/bin/bash
2.
3. myvar=1
```

```
until [ $myvar -gt 10 ]
do
        echo $myvar
        myvar=$(( $myvar + 1 ))
done
```

运行结果：

```
0
1
2
3
4
5
6
7
8
9
```

Shell break 和 continue 命令

在循环过程中，有时候需要在未达到循环结束条件时强制跳出循环，像大多数编程语言一样，Shell 也使用 break 和 continue 来跳出循环。

break 命令

break 命令允许跳出所有循环（终止执行后面的所有循环）。

下面的例子中，脚本进入死循环直至用户输入数字大于 5。要跳出这个循环，返回到 shell 提示符下，就要使用 break 命令。

```
#!/bin/bash
while :
do
    echo -n "Input a number between 1 to 5: "
    read aNum
    case $aNum in
        1|2|3|4|5) echo "Your number is $aNum!"
        ;;
        *) echo "You do not select a number between 1 to 5, game is over!"
            break
        ;;
    esac
done
```

在嵌套循环中，break 命令后面还可以跟一个整数，表示跳出第几层循环。例如：

```
break n
```

表示跳出第 n 层循环。

下面是一个嵌套循环的例子，如果 var1 等于 2，并且 var2 等于 0，就跳出循环：

```
#!/bin/bash

for var1 in 1 2 3
do
   for var2 in 0 5
   do
      if [ $var1 -eq 2 -a $var2 -eq 0 ]
      then
         break 2
      else
         echo "$var1 $var2"
      fi
   done
done
```

如上，break 2 表示直接跳出外层循环。运行结果：

```
1 0
1 5
```

continue 命令

continue 命令与 break 命令类似，只有一点差别，它不会跳出所有循环，仅仅跳出当前循环。

对上面的例子进行修改：

```
#!/bin/bash
while :
do
    echo -n "Input a number between 1 to 5: "
    read aNum
    case $aNum in
        1|2|3|4|5) echo "Your number is $aNum!"
        ;;
        *) echo "You do not select a number between 1 to 5!"
            continue
            echo "Game is over!"
```

```
        ;;
    esac
done
```

运行代码发现，当输入大于 5 的数字时，该例中的循环不会结束，语句

```
echo "Game is over!"
```

永远不会被执行。

同样，continue 后面也可以跟一个数字，表示跳出第几层循环。

再看一个 continue 的例子：

```
#!/bin/bash

NUMS="1 2 3 4 5 6 7"

for NUM in $NUMS
do
    Q=`expr $NUM % 2`
    if [ $Q -eq 0 ]
    then
        echo "Number is an even number!!"
        continue
    fi
    echo "Found odd number"
done
```

运行结果：

```
Found odd number
Number is an even number!!
Found odd number
Number is an even number!!
Found odd number
Number is an even number!!
Found odd number
```

Shell 函数：Shell 函数返回值、删除函数、在终端调用函数

函数可以让我们将一个复杂功能划分成若干模块，让程序结构更加清晰，代码重复利用率更高。像其他编程语言一样，Shell 也支持函数。Shell 函数必须先定义后使用。

Shell 函数的定义格式如下：

```
function_name () {
    list of commands
    [ return value ]
}
```

如果你愿意，也可以在函数名前加上关键字 function：

```
function function_name () {
    list of commands
    [ return value ]
}
```

函数返回值，可以显式增加 return 语句；如果不加，会将最后一条命令运行结果作为返回值。

Shell 函数返回值只能是整数，一般用来表示函数执行成功与否，0 表示成功，其他值表示失败。如果 return 其他数据，比如一个字符串，往往会得到错误提示：“numeric argument required”。

如果一定要让函数返回字符串，那么可以先定义一个变量，用来接收函数的计算结果，脚本在需要的时候访问这个变量来获得函数返回值。

先来看一个例子：

```
#!/bin/bash

# Define your function here
Hello () {
    echo "Url is http://see.xidian.edu.cn/cpp/shell/"
}

# Invoke your function
Hello
```

运行结果：

```
$./test.sh
Hello World
$
```

调用函数只需要给出函数名，不需要加括号。

再来看一个带有 return 语句的函数：

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

```
funWithReturn(){
    echo "The function is to get the sum of two numbers..."
    echo -n "Input first number: "
    read aNum
    echo -n "Input another number: "
    read anotherNum
    echo "The two numbers are $aNum and $anotherNum !"
    return $(($aNum+$anotherNum))
}
funWithReturn
# Capture value returnd by last command
ret=$?
echo "The sum of two numbers is $ret !"
```

运行结果：

```
The function is to get the sum of two numbers...
Input first number: 25
Input another number: 50
The two numbers are 25 and 50 !
The sum of two numbers is 75 !
```

函数返回值在调用该函数后通过 $? 来获得。

再来看一个函数嵌套的例子：

```
#!/bin/bash

# Calling one function from another
number_one () {
    echo "Url_1 is http://see.xidian.edu.cn/cpp/shell/"
    number_two
}

number_two () {
    echo "Url_2 is http://see.xidian.edu.cn/cpp/u/xitong/"
}

number_one
```

运行结果：

```
Url_1 is http://see.xidian.edu.cn/cpp/shell/
Url_2 is http://see.xidian.edu.cn/cpp/u/xitong/
```

像删除变量一样，删除函数也可以使用 unset 命令，不过要加上 -f 选项，如下所示：

纯文本复制

```
$unset -f function_name
```

如果你希望直接从终端调用函数，可以将函数定义在主目录下的 .profile 文件，这样每次登录后，在命令提示符后面输入函数名字就可以立即调用。

Shell 函数参数

在 Shell 中，调用函数时可以向其传递参数。在函数体内部，通过 $n 的形式来获取参数的值，例如，$1 表示第一个参数，$2 表示第二个参数...

带参数的函数示例：

```
#!/bin/bash
funWithParam(){
    echo "The value of the first parameter is $1 !"
    echo "The value of the second parameter is $2 !"
    echo "The value of the tenth parameter is $10 !"
    echo "The value of the tenth parameter is ${10} !"
    echo "The value of the eleventh parameter is ${11} !"
    echo "The amount of the parameters is $# !"  # 参数个数
    echo "The string of the parameters is $* !"  # 传递给函数的所有参数
}
funWithParam 1 2 3 4 5 6 7 8 9 34 73
```

运行脚本：

```
The value of the first parameter is 1 !
The value of the second parameter is 2 !
The value of the tenth parameter is 10 !
The value of the tenth parameter is 34 !
The value of the eleventh parameter is 73 !
The amount of the parameters is 12 !
The string of the parameters is 1 2 3 4 5 6 7 8 9 34 73 !
```

注意，$10 不能获取第十个参数，获取第十个参数需要${10}。当 n>=10 时，需要使用${n}来获取参数。

另外，还有几个特殊变量用来处理参数，前面已经提到：

特殊变量	说明
$#	传递给函数的参数个数。
$*	显示所有传递给函数的参数。

$@	与$*相同，但是略有区别，请查看 Shell 特殊变量。
$?	函数的返回值。

Shell 输入输出重定向：Shell Here Document，/dev/null 文件

Unix 命令默认从标准输入设备(stdin)获取输入，将结果输出到标准输出设备(stdout)显示。一般情况下，标准输入设备就是键盘，标准输出设备就是终端，即显示器。

输出重定向

命令的输出不仅可以是显示器，还可以很容易的转移向到文件，这被称为输出重定向。

命令输出重定向的语法为：

```
$ command > file
```

这样，输出到显示器的内容就可以被重定向到文件。

例如，下面的命令在显示器上不会看到任何输出：

```
$ who > users
```

打开 users 文件，可以看到下面的内容：

```
$ cat users
oko         tty01   Sep 12 07:30
ai          tty15   Sep 12 13:32
ruth        tty21   Sep 12 10:10
pat         tty24   Sep 12 13:07
steve       tty25   Sep 12 13:03
$
```

输出重定向会覆盖文件内容，请看下面的例子：

```
$ echo line 1 > users
$ cat users
line 1
$
```

如果不希望文件内容被覆盖，可以使用 >> 追加到文件末尾，例如：

```
$ echo line 2 >> users
$ cat users
line 1
```

```
line 2
$
```

输入重定向

和输出重定向一样，Unix 命令也可以从文件获取输入，语法为：

```
1. command < file
```

这样，本来需要从键盘获取输入的命令会转移到文件读取内容。

注意：输出重定向是大于号(>)，输入重定向是小于号(<)。

例如，计算 users 文件中的行数，可以使用下面的命令：

```
$ wc -l users
2 users
$
```

也可以将输入重定向到 users 文件：

```
$ wc -l < users
2
$
```

注意：上面两个例子的结果不同：第一个例子，会输出文件名；第二个不会，因为它仅仅知道从标准输入读取内容。

重定向深入讲解

一般情况下，每个 Unix/Linux 命令运行时都会打开三个文件：

- 标准输入文件(stdin)：stdin 的文件描述符为 0，Unix 程序默认从 stdin 读取数据。
- 标准输出文件(stdout)：stdout 的文件描述符为 1，Unix 程序默认向 stdout 输出数据。
- 标准错误文件(stderr)：stderr 的文件描述符为 2，Unix 程序会向 stderr 流中写入错误信息。

默认情况下，command > file 将 stdout 重定向到 file，command < file 将 stdin 重定向到 file。

如果希望 stderr 重定向到 file，可以这样写：

```
1. $command 2 > file
```

如果希望 stderr 追加到 file 文件末尾，可以这样写：

```
$command 2 >> file
```

2 表示标准错误文件(stderr)。

如果希望将 stdout 和 stderr 合并后重定向到 file，可以这样写：

```
$command > file 2>&1
```

或

```
$command >> file 2>&1
```

如果希望对 stdin 和 stdout 都重定向，可以这样写：

```
$command < file1 >file2
```

command 命令将 stdin 重定向到 file1，将 stdout 重定向到 file2。

全部可用的重定向命令列表

命令	说明
command > file	将输出重定向到 file。
command < file	将输入重定向到 file。
command >> file	将输出以追加的方式重定向到 file。
n > file	将文件描述符为 n 的文件重定向到 file。
n >> file	将文件描述符为 n 的文件以追加的方式重定向到 file。
n >& m	将输出文件 m 和 n 合并。
n <& m	将输入文件 m 和 n 合并。
<< tag	将开始标记 tag 和结束标记 tag 之间的内容作为输入。

Here Document

Here Document 目前没有统一的翻译，这里暂译为”嵌入文档“。Here Document 是 Shell 中的一种特殊的重定向方式，它的基本的形式如下：

```
command << delimiter
    document
delimiter
```

它的作用是将两个 delimiter 之间的内容(document) 作为输入传递给 command。

注意：

- 结尾的 delimiter 一定要顶格写，前面不能有任何字符，后面也不能有任何字符，包括空格和 tab 缩进。
- 开始的 delimiter 前后的空格会被忽略掉。

下面的例子，通过 wc -l 命令计算 document 的行数：

```
$wc -l << EOF
    This is a simple lookup program
    for good (and bad) restaurants
    in Cape Town.
EOF
3
$
```

也可以 将 Here Document 用在脚本中，例如：

```
#!/bin/bash

cat << EOF
This is a simple lookup program
for good (and bad) restaurants
in Cape Town.
EOF
```

运行结果：

```
This is a simple lookup program
for good (and bad) restaurants
in Cape Town.
```

下面的脚本通过 vi 编辑器将 document 保存到 test.txt 文件：

```
#!/bin/sh

filename=test.txt
vi $filename <<EndOfCommands
i
This file was created automatically from
a shell script
^[
```

```
9. ZZ
10. EndOfCommands
```

运行脚本：

```
$ sh test.sh
Vim: Warning: Input is not from a terminal
$
```

打开 test.txt，可以看到下面的内容：

```
$ cat test.txt
This file was created automatically from
a shell script
$
```

/dev/null 文件

如果希望执行某个命令，但又不希望在屏幕上显示输出结果，那么可以将输出重定向到 /dev/null：

```
1. $ command > /dev/null
```

/dev/null 是一个特殊的文件，写入到它的内容都会被丢弃；如果尝试从该文件读取内容，那么什么也读不到。但是 /dev/null 文件非常有用，将命令的输出重定向到它，会起到”禁止输出“的效果。

如果希望屏蔽 stdout 和 stderr，可以这样写：

纯文本复制

```
1. $ command > /dev/null 2>&1
```

Shell 文件包含

像其他语言一样，Shell 也可以包含外部脚本，将外部脚本的内容合并到当前脚本。

Shell 中包含脚本可以使用：

```
1. . filename
```

或

```
1. source filename
```

两种方式的效果相同，简单起见，一般使用点号(.)，但是注意点号(.)和文件名中间有一空格。

例如，创建两个脚本，一个是被调用脚本 subscript.sh，内容如下：

```
1. url="http://see.xidian.edu.cn/cpp/view/2738.html"
```

一个是主文件 main.sh，内容如下：

```
1. #!/bin/bash
2. . ./subscript.sh
3. echo $url
```

执行脚本：

```
$chomd +x main.sh
./main.sh
http://see.xidian.edu.cn/cpp/view/2738.html
$
```

注意：被包含脚本不需要有执行权限。

作业： 写一个 JAVA 文件,在 linux 环境下编译执行,并且写一个 shell,使 java 的编译执行过程批处理化

A.java

```
public class A{

    public static void main(String args[]){

        System.out.println("helllo world");

    }

}
```

```
#!/bin/bash
echo "program start----------"
javac A.java
java A
echo "program end----------"
```