H


wuzhangjin@gmailcom
编程范例
泰晓实验室
December
2013
Shell
Computer
TinyLaborg
Wu
Comments
Zhangjin
Linux
Bash
Read
Language
Tags
falcon
TinyLab
Posted
Comment
Blog
printfriendly
Embedded
Archive
200711
20080
none
Shell编程学习笔记
Shell编程范例之文件操作
Shell编程范例之开篇
Shell编程范例之数值运算
Shell编程范例之文件系统操作
Shell编程范例之网络操作
Shell编程范例之索引篇
Shell编程范例之布尔运算
vnStatSVG
marginright
Recent
Android
bottom
Wiki
Tools
Archives
verticalalign
Kernel
Loongson
border
auto
float
12px
System
display
Time
Management
RSS
22
20

She11编程范例序列

TinyLab.org | 泰晓实验室[1]

2014-01-01

[1]这是[Wu Zhangjin/Falcon](wuzhangjin@gmail.com)于2007至2008年编写的一个Blog序列，因为老Blog系统因故无法访问，整个序列的更新一度停顿，作者现已新开个人网站：http://tinylab.org，并且已经用Markdown对整个序列做了初步整理，目前基于http://github.com/larrycai/kaiyuanbook在TinyLab的项目仓库以自由书籍的方式长期维护:https://gitorious.org/tinylab/pleac-shell，并以PDF的方式发布第一版初稿： v0.1，更多详细内容请访问项目首页：http://www.tinylab.org/project/pleac-shell/

简介

背景

早在2007年11月，当我在Linux操作系统下面学习Shell编程的时候，为了系统化地学习和总结Shell编程，专门制定了一个Shell编程范例的总结计划，当时的计划是：

> 这个系列将以面向“对象”（即我们操作的对象）来展开，并引入大量的实例，这样有助于让我们真正去学以致用，并在用的过程中提高兴趣。所以这个系列将不会专门介绍shell的语法， 而是假设读者对shell编程有了一定的基础。
>
> 另外，该系列到最后可能会涵盖：数值、逻辑值、字符串、文件、进程、文件系统等所有我们可以操作的“对象”，这个操作对象也将从低级到高级，进而上升到网络层面，整个通过各种 方式连接起来的计算机的集合。实际上这也未尝不是在摸索unix的哲学，那“K.I.S.S”（Keep It Simple, Stupid）蕴藏的巨大能量。
>
> 摘自《兰大开源社区 >> 脚本编程 >> Shell编程范例序列》

在2008年4月底，整个序列大部分内容和框架基本完成，后来因为实习和工作原因，整个序列并没有得以持续完善。不过相关的范例章节却得到了比较大的反响，很多热心的网友有大量评论和转载，例如，在百度文库转载的一份《Shell编程范例之字符串操作》的访问量都已经达到将近3000的阅读量。说明，整个序列还是有比较大的阅读群体。

现状

考虑到整个Linux世界的蓬勃发展，Shell的使用环境越来越多，相关的使用群体会不断增加，所以最近计划把整个序列重新整理和完善，以自由书籍的方式不断更新，以便惠及更多的读者。

目前已经把早期的内容重新整理到本站，整个序列用Markdown重写，可以直接通过TinyLab.org每个页面右上角的Print/PDF插件直接下载所有章节的PDF版本。

整个初稿的索引篇是：《Shell编程范例之索引篇》，其内容结构如下：

* Shell编程范例之开篇（更新时间：2007-07-21）
* Shell编程范例之数值运算（更新时间：2007-11-9）
* Shell编程范例之布尔运算（更新时间：2007-10-30）
* Shell编程范例之字符串操作（更新时间：2007-11-21）
* Shell编程范例之文件操作（更新时间：2007-12-5）
* Shell编程范例之文件系统操作（更新时间：2007-12-29）
* Shell编程范例之进程操作（更新时间：2008-02-22）

* Shell编程范例之网络操作（更新时间：2008-04-19）
* Shell编程范例之总结篇（更新时间：2008-07-21）

基于一个Markdown的开源书籍模版，最近刚重新整理了该书籍，并发布在gitorious.org上TinyLab的项目仓库中。项目相关信息如下：

* 项目首页：http://www.tinylab.org/project/pleac-shell/
* 代码仓库：https://gitorious.org/tinylab/pleac-shell

计划

后续除了继续在TinyLab.org以Blog的形式持续更新以外，打算重新规划、增补整个序列的内容，并以开源项目的方式在TinyLab.org持续维护，并通过这个平台接受读者的反馈，直到可以出版，再找出版商正式发行出版。

欢迎大家指出本书初稿中的不足，甚至参与到相关章节的写作、校订和完善当中来。

如果有时间和兴趣，欢迎参与，可以通过Contact TinyLab发送邮件给我们，也可以直接在TinyLab.org的相关页面进行评论回复。

目录

第 1 章

准备工作

1.1 前言

到最后一节来写“开篇”，确实有点古怪。不过，在第一篇（数值操作）的开头实际上也算是一个小的开篇，那里提到整个序列的前提是需要有一定的shell编程基础，因此，为了能够让没有shell编程基础的读者也可以阅读这个序列，我到最后来重写这个开篇。开篇主要介绍什么是shell，shell运行环境，shell基本语法和调试技巧。

1.2 什么是Shell

首先让我们从下图看看shell在整个操作系统中所处的位置吧，该图的外圆描述了整个操作系统（比如debian/ubuntu/slackware等），内圆描述了操作系统的核心（比如linux kernel），而SHELL和GUI一样作为用户和操作系统之间的接口。

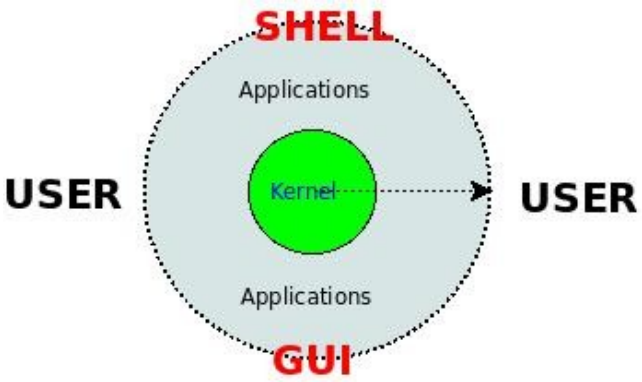

图 1.1: Shell和GUI用户接口

GUI提供了一种图形化的用户接口，使用起来非常简便易学；而SHELL则为用户提供了一种命令行的接口，接收用户的键盘输入，并分析和执行输入字符串中的命令，然后给用户返回执行结果，使用起来可能会复杂一些，但是由于占用的资源少，而且在操作熟练以后可能会提高工作效率，而且具有批处理的功能，因此在某些应用场合还非常流行。

SHELL作为一种用户接口，它实际上是一个能够解释和分析用户键盘输入，执行输入中的命令，然后返回结果的一个解释程序(interpreter，例如在linux下比较常用的bash)，我们可以通过下面的命令查看当前的SHELL：

```
$ echo $SHELL
/bin/bash
$ ls -l /bin/bash
-rwxr-xr-x 1 root root 702160 2008-05-13 02:33 /bin/bash
```

该解释程序不仅能够解释简单的命令，而且可以解释一个具有特定语法结构的文件，这种文件被称作脚本(script)。它具体是如何解释这些命令和脚本文件的，这里不深入分析，请看《Linux命令行上程序执行的那一刹那》。

既然该程序(bash)可以解释具有一定语法结构的文件，那么我们就可以遵循某一语法来编写它，它有什么样的语法，如何运行，如何调试呢？下面我们以bash为例来讨论这几个方面。

1.3 搭建运行环境

为了方便后面的练习，我们先搭建一个基本运行环境：在一个linux操作系统中，有一个运行有bash的命令行在等待我们键入命令，这个命令行可以是图形界面下的terminal，也可以是字符界面的console，如果你发现当前SHELL不是bash，请用下面的方法替换它：

```
$ chsh $USER -s /bin/bash
$ su $USER
```

或者是简单地键入bash

```
$ bash
$ echo $SHELL  # 确认一下
/bin/bash
```

如果没有安装linux操作系统，也可以考虑使用一些公共社区提供的Linux虚拟实验服务，一般都有提供远程SHELL，你可以通过telnet或者是ssh的客户端登录上去进行练习。

有了基本的运行环境，那么如何来运行用户键入的命令或者是用户编写好的脚本文件呢?

假设我们编写好了一个shell脚本，叫test.sh。

第一种方法是确保我们执行的命令具有可执行权限，然后直接键入该命令执行它。

```
$ chmod +x /path/to/test.sh
$ /path/to/test.sh
```

第二种方法是直接把sc.sh作为bash解释器的参数传入。

```
$ bash /path/to/test.sh
```

或

```
$ source /path/to/test.sh
```

或

```
$ . /path/to/test.sh
```

1.4 基本语法介绍

先来一个Hello，World程序。

下面来介绍一个shell程序的基本结构，以Hello，World为例：

```
#!/bin/bash -v
# test.sh
echo "Hello, World"
```

把上面的代码保存为test.sh，然后通过上面两种不同的方式运行一下，可以看到如下的效果：

方法一：

```
$ chmod +x test.sh
$ ./test.sh
 ./test.sh
 #!/bin/bash -v

 echo "Hello, World"
 Hello, World
```

方法二：

```
$ bash test.sh
Hello, World

$ source test.sh
Hello, World

$ . test.sh
Hello, World
```

我们发现两者运行结果有区别，为什么呢？这里我们需要关注一下test.sh文件的内容，它仅仅有两行，第二行打印了Hello，World，两种方法都达到了目的，但是第一种方法却多打印了脚本文件本身的内容，为什么呢？

原因在该文件的第一行，当我们直接运行该脚本文件(test.sh)时，该行告诉操作系统使用用#!符号之后面的解释器以及相应的参数来解释该脚本文件，通过分析第一行，我们发现对应的解释器以及参数是**/bin/bash -v**，而-v刚好就是要打印程序的源代码；但是我们在用第二种方法时没有给bash传递任何额外的参数，因此，它仅仅解释了脚本文件本身。

其他语法细节请直接看《Shell编程学习笔记》。

1.5 Shell程序设计过程

SHELL语言作为解释型语言，它的程序设计过程跟编译型语言有些区别，其基本过程如下：

* 设计算法
* 用SHELL编写脚本程序实现算法
* 直接运行脚本程序

可见它没有编译型语言的“麻烦的”编译和链接过程，不过正是因为这样，它出错时调试起来不是很方便，因为语法错误和逻辑错误都在运行时出现。下面我们简单介绍一下调试方法。

1.6 调试方法介绍

可以直接参考资料：shell脚本调试技术。

1.7 小结

SHELL语言作为一门解释型语言，可以使用大量的现有工具，包括数值计算、符号处理、文件操作、网络操作等，因此，编写过程可能更加高效，但是因为它是解释型的，需要在执行过程中从磁盘上不断调用外部的程序并进行进程之间的切换，在运行效率方面可能有劣势，所以我们应该根据应用场合选择使用SHELL或是用其他的语言来编程。

1.8 参考资料

* 《Linux命令行上程序执行的那一刹那》
* 《linux shell编程学习笔记》
* 《shell脚本调试技术》

第 2 章

数值运算

2.1 前言

从本文开始，打算结合自己平时的积累和进一步的实践，通过一些范例来介绍Shell编程。因为范例往往能够给人以学有所用的感觉，而且给人以动手实践的机会，从而激发人的学习热情。

考虑到易读性，这里的范例将非常简单，但是实用，希望它们能够成为你解决常规问题的参照物或者是“茶余饭后”的小点心，当然这些“点心”肯定还有值得探讨、优化的地方。

更复杂有趣的例子请参考Advanced Bash-Scripting Guide(一本深入学习shell脚本艺术的书籍)。

该序列概要:

* 目的：享受用shell解决问题的乐趣；和朋友们一起交流和探讨

* 计划：先零散地写些东西，之后再不断补充，最后整理成册。

* 适合读者：已经熟悉linux基本知识，比如文件系统结构、常用命令行工具、shell编程基础等。

* 建议：大家在看这些范例的时候，参考网络中流传的《shell基础十二篇》和《shell十三问》，见ChinaUnix Shell讨论区。

* 环境：如果没有特别说明，以后《shell编程范例》系列使用的shell将特指bash，版本在3.1.17以上。

* 说明：本序列的组织不是依据Shell语法，而是面向某些潜在的操作对象和操作本身，它们反应了现实应用。当然，在这个过程当中肯定会涉及到Shell的语法。另外，欢迎您对帖子里头存在的问题进行批评指正，也欢迎您对一些范例进行改进。

这一篇打算讨论一下Shell编程中的基本数值运算，这类运算包括:

* 数值（包括整数和浮点数）间的加、减、乘、除、求幂、求模等

* 产生指定范围的随机数

* 产生指定范围的数列

貌似Shell本身（注：Shell本身是一个解释程序，你可以在命令行打印SHELL变量找到当前的shell程序）只可以完成整数运算，一些复杂的运算可以通过外部命令实现，比如expr,bc,awk等。至于随机数，shell可以通过RANDOM环境变量产生一个从0到32767的随机数，一些外部工具，比如awk可以通过rand()函数产生随机数。而seq命令可以用来产生一个数列。下面对它们分别进行介绍。

2.2 整数运算

2.2.1 范例：对某个数加1

```
$ i=0;
$ ((i++))
$ echo $i
1

$ let i++
$ echo $i
2

$ expr $i + 1
3
$ echo $i
2

$ echo $i 1 | awk '{printf $1+$2}'
3
```

说明：expr之后的$i,+,1之间有空格分开。如果进行乘法运算，需要对运算符进行转义，否则shell会把乘号解释为通配符，导致语法错误；awk后面的$1和2分别指i和1,即从左往右的第1个和第2个数。

用shell的内置命令查看各个命令的类型如下：

```
$ type type
type is a shell builtin
$ type let
let is a shell builtin
$ type expr
expr is hashed (/usr/bin/expr)
$ type bc
bc is hashed (/usr/bin/bc)
$ type awk
awk is /usr/bin/awk
```

从上面的演示可以看出：let是shell内置命令，其他几个是外部命令，都在/usr/bin目录下。而expr和bc因为刚用过，已经加载在内存的hash表中。这个结果将有助于我们理解下面范例的结果。

如果要查看不同命令的帮助，对于let和type等shell内置命令，可以通过shell的一个内置命令help来查看相关帮助，而一些外部命令可以通过shell的一个外部命令man来查看帮助，用法诸如help let，man expr等。

2.2.2 范例：从1加到某个数

```
#!/bin/bash
# calc.sh

i=0;
while [ $i -lt 10000 ]
do
    ((i++))
done
echo $i
```

说明：这里通过**while [条件表达式]; do .... done循环**来实现。-lt是小于号(<)，具体见test命令的用法：**man test**。

如何执行该脚本?

第一种办法直接把脚本文件当成子Shell(bash)的一个参数传入。

```
$ bash calc.sh
$ type bash
bash is hashed (/bin/bash)
```

第二种办法是通过bash的内置命令.或source执行。

```
$ . ./calc.sh
```

或

```
$ source ./calc.sh
$ type .
. is a shell builtin
$ type source
source is a shell builtin
```

第三种办法是修改文件为可执行，直接在当前shell下执行。

```
$ chmod ./calc.sh
$ ./calc.sh
```

下面，逐一演示用其他方法计算变量加一，即把((i++))行替换成下面的某一个：

```
let i++;

i=$(expr $i + 1)

i=$(echo $i+1|bc)

i=$(echo "$i 1" | awk '{printf $1+$2;}')
```

比较计算时间如下：

```
$ time calc.sh
10000

real    0m1.319s
user    0m1.056s
sys     0m0.036s
$ time calc_let.sh
10000

real    0m1.426s
user    0m1.176s
sys     0m0.032s
$  time calc_expr.sh
1000

real    0m27.425s
user    0m5.060s
sys     0m14.177s
$ time calc_bc.sh
1000

real    0m56.576s
user    0m9.353s
sys     0m24.618s
$ time ./calc_awk.sh
100

real    0m11.672s
user    0m2.604s
sys     0m2.660s
```

说明：time命令可以用来统计命令执行时间，这部分时间包括总的运行时间，用户空间执行时间，内核空间执行时间，它通过ptrace系统调用实现。

通过上面的比较，我们发现(())的运算效率最高。而let作为shell内置命令，效率也很高，但是expr,bc,awk的计算效率就比较低。所以，在shell本身能够完成相关工作的情况下，建议优先使用shell本身提供的功能。但是shell本身好像无法完成浮点运算，所以就需要外部命令的帮助。

let，expr，bc都可以用来求模，运算符都是%，而let和bc可以用来求幂，运算符不一样，前者是**，后者是^。例如：

2.2.3 范例：求模

```
$ expr 5 % 2
1

$ let i=5%2
$ echo $i
1

$ echo 5 % 2 | bc
1

$ ((i=5%2))
$ echo $i
1
```

2.2.4 范例：求幂

```
$ let i=5**2
$ echo $i
25

$ ((i=5**2))
$ echo $i

25
$ echo "5^2" | bc
25
```

2.2.5 范例：进制转换

进制转换也是比较常用的操作，可以用Bash的内置支持也可以用bc来完成，例如把8进制的11转换为10进制，则可以：

```
$ echo "obase=10;ibase=8;11" | bc -l
9
```

```
$ echo $((8#11))
9
```

上面都是把某个进制的数转换为10进制的，如果要进行任意进制之间的转换还是bc比较灵活，因为它可以直接指定进制源和进制转换目标。

2.2.6 范例：ascii字符编码

如果要把某些字符串以特定的进制表示，可以用od命令，例如默认的分隔符IFS包括空格、TAB以及换行，可以用man ascii佐证。

```
$ echo -n "$IFS" | od -c
0000000      t  n
0000003
$ echo -n "$IFS" | od -b
0000000 040 011 012
0000003
```

2.3 浮点运算

let和expr都无法进行浮点运算，但是bc和awk可以。

2.3.1 范例：求1除以13，保留3位有效数字

```
$ echo "scale=3; 1/13"  | bc
.076

$ echo "1 13" | awk '{printf("%0.3fn",$1/$2)}'
0.077
```

说明：bc在进行浮点运算的时候需要指定小数点位数，否则默认为0，即进行浮点运算的时候，默认求出的结果只保留整数。而awk在控制小数位数的时候非常灵活，仅仅通过printf的格式控制就可以实现。

补充：在用bc进行运算的时候，如果不指定scale，而在bc后加上-l选项，也可以进行浮点运算，只不过这时的浮点运算的小数点默认是20位。例如：

```
$ echo 1/13100 | bc -l
.00007633587786259541
```

2.3.2 范例：余弦值转角度

用bc -l计算，可以获得高精度：

```
$ export cos=0.996293; echo "scale=100; a(sqrt(1-$cos^2)/$cos)*180/(a(1)*4)" |bc -l
4.93495475541138363271983403693184060515970639865524387537276491773 2
549550415976601152707828600407213 1
```

当然也可以用awk来计算：

```
$ echo 0.996293 | awk '{ printf("%sn", atan2(sqrt(1-$1^2),$1)*180/3.1415926535);}'
4.93495
```

2.3.3 范例：有一组数据，存有某村所有家庭的人数和月总收入，找出人均月收入最高家庭

在这里随机产生了一组测试数据，文件名为income.txt。

```
1 3 4490
2 5 3896
3 4 3112
4 4 4716
5 4 4578
6 6 5399
7 3 5089
8 6 3029
9 4 6195
10 5 5145
```

说明：上面的三列数据分别是家庭编号、家庭人数、家庭月总收入。
分析：为了求出月均收入最高的家庭，我们需要对后面两列数进行除法运算，即求出每个家庭的月均收入，然后按照月均收入排序，找出收入最高的家庭。
实现：

```
#!/bin/bash
# gettopfamily.sh

[ $# -lt 1 ] && echo "please input the income file" && exit -1
[ ! -f $1 ] && echo "$1 is not a file" && exit -1

income=$1
awk '{
    printf("%d %0.2fn", $1, $3/$2);
}' $income | sort -k 2 -n -r
```

说明：

* [$# -lt 1]：要求用户至少收入一个参数，S#是shell中传入参数的个数

* `[ ! -f $1 ]`：要求用户传入的参数是一个文件，-f的用法见test命令，`man test`
* `income=$1`：把用户传入的参数赋值给income变量，并在后面作为awk的参数，即需要处理的文件

* `awk`：用文件中的第三列除以第二列，求出月均收入，考虑到精确性，保留了两位有效数字。

* `sort -k 2 -n -r`：这里对结果的awk结果的第二列(-k 2)，即月均收入进行排序，按照数字排序(-n)，并按照递减的顺序排序（-r)。

演示：

```
$ ./gettopfamily.sh income.txt
7 1696.33
9 1548.75
1 1496.67
4 1179.00
5 1144.50
10 1029.00
6 899.83
2 779.20
3 778.00
8 504.83
```

补充：之前的income.txt数据是随机产生的。在做一些实验时，往往需要随机产生一些数据，在下一小节，我们将详细介绍它。这里是产生income.txt数据的脚本：

```
#!/bin/bash
# genrandomdata.sh

for i in $(seq 1 10)
do
    echo $i $(($RANDOM/8192+3)) $((RANDOM/10+3000))
done
```

说明：上述脚本中还用到seq命令产生从1到10的一列数，这个命令的详细用法在该篇最后一节也会进一步介绍。

2.4 随机数

环境变量RANDOM产生0到32767的随机数，而awk的rand函数可以产生0到1之间的随机数。

2.4.1 范例：获取一个随机数

```
$ echo $RANDOM
81

$ echo "" | awk '{srand(); printf("%f", rand());}'
0.237788
```

说明：srand在无参数时，采用当前时间作为rand随机数产生器的一个seed。

2.4.2 范例：随机产生一个从0到255之间的数字

可以通过RANDOM变量的缩放和awk中rand的放大来实现。

```
$ expr $RANDOM / 128

$ echo "" | awk '{srand(); printf("%dn", rand()*255);}'
```

思考：如果要随机产生某个IP段的IP地址，该如何做呢？看例子：友善地获取一个可用的IP地址。

```
#!/bin/bash
# getip.sh -- get an usable ipaddress automatically
# author: falcon &lt;zhangjinw@gmail.com>
# update: Tue Oct 30 23:46:17 CST 2007

# set your own network, default gateway, and the time out of "ping" command
net="192.168.1"
default_gateway="192.168.1.1"
over_time=2

# check the current ipaddress
ping -c 1 $default_gateway -W $over_time
[ $? -eq 0 ] && echo "the current ipaddress is okey!" && exit -1;

while :; do
    # clear the current configuration
    ifconfig eth0 down
    # configure the ip address of the eth0
    ifconfig eth0 \
        $net.$(($RANDOM /130 +2)) \
        up
    # configure the default gateway
    route add default gw $default_gateway
    # check the new configuration
    ping -c 1 $default_gw -W $over_time
    # if work, finish
```

```
    [ $? -eq 0 ] && break
done
```

说明：如果网关地址不是1，那么用ifconfig配置地址时不能配置为网关地址，否则你的IP地址将和网关一样，导致整个网络出现问题。

2.5 其他运算

其实通过一个循环就可以产生一序列数，但是有相关的小工具为什么不用呢！seq就是这么一个小工具，它可以产生一序列数，你可以指定数的递增间隔，也可以指定相邻两个数之间的分割符。

2.5.1 范例：获取一序列数

```
$ seq 5
1
2
3
4
5
$ seq 1 5
1
2
3
4
5
$ seq 1 2 5
1
3
5
$ seq -s: 1 2 5
1:3:5
$ seq 1 2 14
1
3
5
7
9
11
13
$ seq -w 1 2 14
01
03
05
07
```

```
09
11
13
$ seq -s: -w 1 2 14
01:03:05:07:09:11:13
$ seq -f "0x%g" 1 5
0x1
0x2
0x3
0x4
0x5
```

一个比较典型的使用seq的例子，构造一些特定格式的链接，然后用wget下载这些内容：

```
$ for i in `seq -f"http://thns.tsinghua.edu.cn/thnsebooks/ebook73/%02g.pdf" 1 21`;do wget -c $i; dor
```

或者

```
$ for i in `seq -w 1 21`;do wget -c "http://thns.tsinghua.edu.cn/thnsebooks/ebook73/$i"; done
```

补充：在bash版本3中，在for循环的in后面，可以直接通过{1..5}更简洁地产生自1到5的数字(注意，1和5之间只有两个点)，例如：

```
$ for i in 1..5; do echo -n "$i "; done
1 2 3 4 5
```

2.5.2 范例：统计字符串中各单词出现次数

首先，我们统计某个文件中所有单词的个数。

这里的单词我定义为：由字母组成的单个或者多个字符序列。所以，可以这样实现。

说明：为了方便演示，这里采用。

统计每个单词出现的次数：

```
$ wget -c http://tinylab.org
$ cat index.html | sed -e "s/[^a-zA-Z]/\n/g" | grep -v ^$ | sort | uniq -c
```

统计出现频率最高的前10个单词：

```
$ wget -c http://tinylab.org
$ cat index.html | sed -e "s/[^a-zA-Z]/\n/g" | grep -v ^$ | sort | uniq -c | sort -n -k 1 -r | head
    524 a
    238 tag
    205 href
```

```
    201 class
    193 http
    189 org
    175 tinylab
    174 www
    146 div
    128 title
```

说明：

* cat index.html：输出index.html文件里的内容
* sed -e "s/[^a-zA-Z]/\n/g"：把非字母的字符全部替换成空格，这样整个文本只剩下字母字符
* grep -v ^$:去掉空行
* sort：排序
* uniq -c：统计相同行的个数，即每个单词的个数
* sort -n -k 1 -r：按照第一列(-k 1)的数字(-n)逆序(-r)排序
* head -10：取出前十行

2.5.3 范例：统计指定单词出现次数

可以考虑采取两种办法：

* 只统计那些需要统计的单词
* 用上面的算法把所有单词的个数都统计出来，然后再返回那些需要统计的单词给用户

不过，这两种办法都可以通过下面的结构来实现。

```
#!/bin/bash
# statistic_words.sh

if [ $# -lt 1 ]; then
    echo "ERROR: you should input 2 words at least";
    echo "Usage: basename $0 FILE WORDS ...."
    exit -1
fi

FILE=$1
((WORDS_NUM=$#-1))
```

```
for n in $(seq $WORDS_NUM)
do
    shift
    cat $FILE | sed -e "s/[^a-zA-Z]/\n/g" \
        | grep -v ^$ | sort | grep ^$1$ | uniq -c
done
```

说明：

* if 条件部分：要求用户输入至少两个参数，第一个是需要统计单词的文件名，第二之后的所有参数是需要统计的单词。

* FILE=$1：获取文件名，即脚本之后的第一个字符串。

* ((WORDS_NUM=$#-1))：获取单词个数，即总的参数个数($#)减去那个文件名参数(1个)

* for循环部分：首先通过seq产生需要统计的单词个数序列，shift是shell内置变量（请通过help shift获取帮助），它把用户从命令行中传入的参数依次往后移动位置，并把当前参数作为第一个参数即$1，这样通过$1就可以遍历用户所有输入的单词(仔细一想，这里貌似有数组下标的味道)。你可以考虑把shift之后的那句替换成echo $1测试shift的用法。

演示：

```
$ chmod +x statistic_words.sh
$ ./statistic_words.sh index.html tinylab linux python
    175 tinylab
     43 linux
      3 python
```

采用第二种办法，我们只需要修改shift之后的那句即可。

```
#!/bin/bash
# statistic_words.sh

if [ $# -lt 1 ]; then
    echo "ERROR: you should input 2 words at least";
    echo "Usage: basename $0 FILE WORDS ...."
    exit -1
fi

FILE=$1
((WORDS_NUM=$#-1))

for n in $(seq $WORDS_NUM)
do
```

```
    shift
    cat $FILE | sed -e "s/[^a-zA-Z]/\n/g" \
        | grep -v ^$ | sort | uniq -c | grep " $1$"
done
```

演示：

```
$ ./statistic_words.sh index.html tinylab linux python
    175 tinylab
     43 linux
      3 python
```

说明：很明显，采用第一种办法效率要高很多，因为第一种办法提前找出了需要统计的单词，然后再统计，而后者则不然。实际上，如果使用grep的-E选项，我们无须引入循环，而用一条命令就可以搞定：

```
$ cat index.html | sed -e "s/[^a-zA-Z]/\n/g" | grep -v ^$ | sort | grep -E "^tinylab$|^linux$" | uni
     43 linux
    175 tinylab
```

或者

```
$ cat index.html | sed -e "s/[^a-zA-Z]/\n/g" | grep -v ^$ | sort | egrep  "^tinylab$|^linux$" | unic
     43 linux
    175 tinylab
```

说明：需要注意到sed命令可以直接处理文件，而无需通过cat命令输出以后再通过管道传递，这样可以减少一个不必要的管道操作，所以上述命令可以简化为：

```
$ sed -e "s/[^a-zA-Z]/\n/g" index.html | grep -v ^$ | sort | egrep  "^tinylab$|^linux$" | uniq -c
     43 linux
    175 tinylab
```

所以，可见这些命令sed,grep,uniq,sort是多么有用，它们本身虽然只完成简单的功能，但是通过一定的组合，就可以实现你想要实现的功能啦。对了，统计单词还有个非常有用的命令wc -w，需要用到的时候也可以用它。

补充：在Advanced Bash-Scripting Guide一书中还提到jot命令和factor命令，由于机器上没有，所以没有测试，factor命令可以产生某个数的所有素数。如：

```
$ factor 100
100: 2 2 5 5
```

2.6 小结

到这里，shell编程范例之数值计算就结束啦。该篇主要介绍了：

* shell编程中的整数运算、浮点运算、随机数的产生、数列的产生
* shell的内置命令、外部命令的区别，以及如何查看他们的类型和帮助，关于内置命令和外部命令的比较
* shell脚本的几种执行办法
* 几个常用的shell外部命令：sed,awk,grep,uniq,sort等
* 范例：数字递增；求月均收入；自动获取IP地址；统计单词个数
* 其他：相关的用法，比如命令列表，条件测试等，在上述范例中都已经涉及，请认真阅读之

如果您有时间，请温习之。

2.7 资料

* Advanced Bash-Scripting Guide
* shell十三问
* shell基础十二篇
* SED手册
* AWK使用手册
* 几个shell讨论区
* LinuxSir.org
* ChinaUnix.net

2.8 后记

大概花了3个多小时才写完，目前是23:33，该回宿舍睡觉啦，明天起来修改错别字和补充一些内容，朋友们晚安！

10月31号，修改部分措辞，增加一篇统计家庭月均收入的范例，添加总结和参考资料，并用附录所有代码。

SHELL编程是一件非常有趣的事情，如果您想一想：上面计算家庭月均收入的例子，然后和用MS Excel来做这个工作比较，你会发现前者是那么简单和省事，而且给您以运用自如的感觉。

第 3 章

布尔运算

3.1 前言

上个礼拜介绍了Shell编程范例之数值运算，对Shell下基本数值运算方法做了简单的介绍，这周将一起探讨布尔运算，即如何操作“真假值”。

在Bash里有这样的常量(实际上是两个内置命令，在这里我们姑且这么认为，后面将介绍)，即true和false，一个表示真，一个表示假。对它们可以进行与、或、非运算等常规的逻辑运算，在这一节，我们除了讨论这些基本逻辑运算外，还将讨论SHELL编程中的条件测试和命令列表，并介绍它们和布尔运算的关系。

3.2 常规的布尔运算

这里主要介绍Bash里头常规的逻辑运算，与、或、非。

3.2.1 在shell下如何进行逻辑运算

范例：true or false

单独测试true和false，可以看出true是真值，false为假

```
$ if true;then echo "YES"; else echo "NO"; fi
YES
$ if false;then echo "YES"; else echo "NO"; fi
NO
```

范例：与运算

```
$ if true && true;then echo "YES"; else echo "NO"; fi
YES
$ if true && false;then echo "YES"; else echo "NO"; fi
NO
$ if false && false;then echo "YES"; else echo "NO"; fi
NO
```

```
$ if false && true;then echo "YES"; else echo "NO"; fi
NO
```

范例：或运算

```
$ if true || true;then echo "YES"; else echo "NO"; fi
YES
$ if true || false;then echo "YES"; else echo "NO"; fi
YES
$ if false || true;then echo "YES"; else echo "NO"; fi
YES
$ if false || false;then echo "YES"; else echo "NO"; fi
NO
```

范例：非运算，即取反

```
$ if ! false;then echo "YES"; else echo "NO"; fi
YES
$ if ! true;then echo "YES"; else echo "NO"; fi
NO
```

可以看出true和false按照我们对逻辑运算的理解进行着，但是为了能够更好的理解shell对逻辑运算的实现，我们还得弄清楚，true和false是怎么工作的?

3.2.2 Bash里头的true和false是我们通常认为的1和0么？

回答是：否。

范例：返回值 v.s. 逻辑值

true和false它们本身并非逻辑值，它们是shell内置命令，返回了“逻辑值”

```
$ true
$ echo $?
0
$ false
$ echo $?
1
```

范例：查看true和false帮助和类型

```
$ help true false
true: true
    Return a successful result.
```

```
false: false
     Return an unsuccessful result.
$ type true false
true is a shell builtin
false is a shell builtin
```

说明：**$?**是一个特殊的变量，存放有上一个程序的结束状态(退出状态码)。

从上面的操作不难联想到在C语言程序设计中为什么会强调在main函数前面加上int，并在末尾加上**return 0**。因为在shell里头，将把**0**作为程序是否成功结束的标志，这就是shell里头**true**和**false**的实质，它们用以反应某个程序是否正确结束，而并非传统的真假值（**1**和**0**），相反的，它们返回的是**0**和**1**。不过庆幸的是，我们在做逻辑运算时，无须关心这些。

3.3 条件测试

从上一节中，我们已经清楚的了解了shell下的“逻辑值”是什么：是程序结束后的返回值，如果成功返回，则为真，如果不成功返回，则为假。

而条件测试正好使用了test这么一个指令，它用来进行数值测试（各种数值属性测试）、字符串测试（各种字符串属性测试）、文件测试（各种文件属性测试），我们通过判断对应的测试是否成功，从而完成各种常规工作，在加上各种测试的逻辑组合后，将可以完成更复杂的工作。

3.3.1 条件测试基本使用

范例：数值测试

```
$ if test 5 -eq 5;then echo "YES"; else echo "NO"; fi
YES
$ if test 5 -ne 5;then echo "YES"; else echo "NO"; fi
NO
```

范例：字符串测试

```
$ if test -n "not empty";then echo "YES"; else echo "NO"; fi
YES
$ if test -z "not empty";then echo "YES"; else echo "NO"; fi
NO
$ if test -z "";then echo "YES"; else echo "NO"; fi
YES
$ if test -n "";then echo "YES"; else echo "NO"; fi
NO
```

范例：文件测试

```
$ if test -f /boot/System.map; then echo "YES"; else echo "NO"; fi
YES
$ if test -d /boot/System.map; then echo "YES"; else echo "NO"; fi
NO
```

3.3.2 各种逻辑测试的组合

范例：如果a,b,c都等于下面对应的值，那么打印YES，这里通过-a进行“与”测试

```
$ a=5;b=4;c=6;
$ if test $a -eq 5 -a $b -eq 4 -a $c -eq 6; then echo "YES"; else echo "NO"; fi
YES
```

范例：测试某个“东西”是文件或者目录，这里通过-o进行“或”运算

```
$ if test -f /etc/profile -o -d /etc/profile;then echo "YES"; else echo "NO"; fi
YES
```

范例：测试非运算

```
$ if test ! -f /etc/profile; then echo "YES"; else echo "NO"; fi
NO
```

上面仅仅演示了test命令一些非常简单的测试，你可以通过**help test**获取test的更多使用方法。在这里需要注意的是，test命令内部的逻辑运算和shell的逻辑运算符有一些区别，对应的为-a和&&，-o与||，这两者不能混淆使用。而非运算都是!，下面对它们进行比较。

3.3.3 比较-a与&&, -o与||，！test与test！

范例：要求某个文件有可执行权限并且有内容，用-a和&&分别实现

```
$ cat > test.sh
#!/bin/bash

echo "test"
$ chmod +x test.sh
$ if test -s test.sh -a -x test.sh; then echo "YES"; else echo "NO"; fi
YES
$ if test -s test.sh && test -x test.sh; then echo "YES"; else echo "NO"; fi
YES
```

范例：要求某个字符串要么为空，要么和某个字符串相等

```
$ str1="test"
$ str2="test"
$ if test -z "$str2" -o "$str2" == "$str1"; then echo "YES"; else echo "NO"; fi
YES
$ if test -z "$str2" || test "$str2" == "$str1"; then echo "YES"; else echo "NO"; fi
YES
```

范例：测试某个数字不满足指定的所有条件

```
$ i=5
$ if test ! $i -lt 5 -a $i -ne 6; then echo "YES"; else echo "NO"; fi
YES
$ if ! test $i -lt 5 -a $i -eq 6; then echo "YES"; else echo "NO"; fi
YES
```

很容易找出它们的区别，-a和-o使用在测试命令的内部，作为测试命令的参数，而&&和||是用来运算测试的返回值，!为两者通用。需要关注的是：

* 有时候我们可以不用!运算符，比如-eq和-ne刚好是相反的，用来测试两个数值是否相等；-z与-n也是对应的，用来测试某个字符串是否为空。
* 在bash里，test命令可以用[ ]运算符取代，但是需要注意，[之后与]之前需要加上额外的空格。
* 在测试字符串的时候，所有变量建议用双引号包含起来，以防止变量内容为空的时候出现仅有测试参数，没有测试内容的情况。

下面我们用实例来演示上面三个注意事项：
-ne和-eq对应的，我们有时候可以免去!运算

```
$ i=5
$ if test $i -eq 5; then echo "YES"; else echo "NO"; fi
YES
$ if test $i -ne 5; then echo "YES"; else echo "NO"; fi
NO
$ if test ! $i -eq 5; then echo "YES"; else echo "NO"; fi
NO
```

用[ ]可以取代test，这样看上去会“美观”很多

```
$ if [ $i -eq 5 ]; then echo "YES"; else echo "NO"; fi
YES
$ if [ $i -gt 4 ] && [ $i -lt 6 ]; then echo "YES"; else echo "NO"; fi
YES
```

记得给一些字符串变量加上""，记得[之后与]之前多加一个空格

```
$ str=""
$ if [ "$str" = "test"]; then echo "YES"; else echo "NO"; fi
-bash: [: missing `]'
NO
$ if [ $str = "test" ]; then echo "YES"; else echo "NO"; fi
-bash: [: =: unary operator expected
NO
$ if [ "$str" = "test" ]; then echo "YES"; else echo "NO"; fi
NO
```

到这里，条件测试就介绍完了，下面我们将介绍“命令列表”，实际上在上面我们似乎已经使用过了，即多个test命令的组合，通过&&，||和!组合起来的命令序列。这个命令序列可以有效替换if/then的条件分支结构。这不难想到我们在C语言程序设计中经常做的如下的选择题(很无聊的例子，但是有意义)：下面是否会打印j，如果打印，将打印什么?

```
#include <stdio.h>
int main()
{
    int i, j;

    i=5;j=1;
    if ((i==5) && (j=5)) printf("%d\n", j);

    return 0;
}
```

很容易知道将打印数字5，因为i==5这个条件成立，而且随后是&&，要判断整个条件是否成立，我们得进行后面的判断，可是这个判断并非常规的判断，而是先把j修改为5，再转换为真值，所以条件为真，打印出5。因此，这句可以解释为：如果i等于5，那么把j赋值为5，如果j大于1（因为之前已经为真），那么打印出j的值。这样用&&连结起来的判断语句替代了两个if条件分支语句。

正是基于逻辑运算特有的性质，我们可以通过&&，||来取代if/then等条件分支结构，这样就产生了命令列表。

3.4 命令列表

3.4.1 命令列表的执行规律

命令列表的执行规律符合逻辑运算的运算规律，用&&连接起来的命令，如果前者成功返回，将执行后面的命令，反之不然；用||连接起来的命令，如果前者成功返回，将不执行后续命令，反之不然。

范例：如果ping通www.lzu.edu.cn，那么打印连通信息

```
$ ping -c 1 www.lzu.edu.cn -W 1 && echo "=======connected======="
```

非常有趣的问题出来了，即我们上面已经提到的：为什么要让C程序在main函数的最后返回0？如果不这样，把这种程序放入命令列表会有什么样的结果？你自己写个简单的C程序看看，然后放入命令列表看看。

3.4.2 命令列表的作用

在有些时候取代if/then等条件分支结构，这样可以省略一些代码，而且使得程序比较美观、易读，例如：

范例：在脚本里判断程序的参数个数，和参数类型

```
#!/bin/bash

echo $#
echo $1
if [ $# -eq 1 ] && echo $1 | grep ^[0-9]*$ >/dev/null;then
    echo "YES"
fi
```

上例要求参数个数为1并且类型为数字。

再加上exit 1，我们将省掉if/then结构

```
#!/bin/bash

echo $#
echo $1
! ([ $# -eq 1 ] && echo $1 | grep ^[0-9]*$ >/dev/null) && exit 1

echo "YES"
```

这样处理后，对程序参数的判断仅仅需要简单的一行代码，而且变得更美观。

3.5 小结

这一节介绍了shell编程中的逻辑运算，条件测试和命令列表。

第 4 章

字符串操作

4.1 前言

忙活了一个礼拜，终于等到周末，可以空下来写点东西。

之前已经完成《She11编程范例之数值运算》和《She11编程范例之布尔运算》，这次介绍 字符串操作 了，这里先得明白两个东西，什么是字符串，对字符串有哪些操作?

下面是“在线新华字典”的解释：

> 字符串： 简称“串”。有限字符的序列。数据元素为字符的线性表，是一种数据的逻辑结构。在计算机中可有不同的存储结构。在串上可进行求子串、插入字符、删除字符、置换字符等运算。

而字符呢?

> 字符： 计算机程序设计及操作时使用的符号。包括字母、数字、空格符、提示符及各种专用字符等。

照这样说，之前介绍的数值运算中的数字，逻辑运算中的真假值，都是以字符的形式呈现出来的，是一种特别的字符，对它们的运算只不过是字符操作的特例罢了。而这里将研究一般字符的运算，它具有非常重要的意义，因为对我们来说，一般的工作都是处理字符而已。这些运算实际上将围绕上述两个定义来做。

* 找出字符或者字符串的类型，是数字、字母还是其他特定字符，是可打印字符，还是不可打印字符（一些控制字符）。
* 找出组成字符串的字符个数和字符串的存储结构（比如数组）。
* 对串的常规操作：求子串、插入字符、删除字符、置换字符、字符串的比较等。
* 对串的一些比较复杂而有趣的操作，这里将在最后介绍一些有趣的范例。

4.2 字符串的属性

4.2.1 字符串的类型

字符有可能是数字、字母、空格、其他特殊字符，而字符串有可能是它们任何一种或者多种的组合，在组合之后还可能形成一个具有特定意义的字符串，诸如邮件地址，URL地址等。

判断字符的类型

范例：数字或者数字组合 能够返回结果，即程序退出状态是0，说明属于这种类型，反之不然

```
$ i=5;j=9423483247234;
$ echo $i | grep [0-9]*
5
$ echo $j | grep [0-9]*
9423483247234
$ echo $j | grep [0-9]* >/dev/null
$ echo $?
0
```

范例：字符组合（小写字母、大写字母、两者的组合）

```
$ c="A"; d="fwefewjuew"; e="fewfEFWefwefe"
$ echo $c | grep [A-Z]
A
$ echo $d | grep "[a-z]*"
fwefewjuew
$ echo $e | grep "[a-zA-Z]*"
fewfEFWefwefe
```

范例：字母和数字的组合

```
$ ic="432fwfwefeFWEwefwef"
$ echo $ic | grep "[0-9a-zA-Z]*"
432fwfwefeFWEwefwef
```

范例：空格或者Tab键等

```
$ echo " " | grep " "

$ echo -e "\t" | grep "[[:space:]]" #[[:space:]]会同时匹配空格和TAB键

$ echo -e " \t" | grep "[[:space:]]"

$ echo -e "\t" | grep "" #为在键盘上按下TAB键，而不是字符
```

范例：匹配邮件地址

```
$ echo "test2007@lzu.cn" | grep "[0-9a-zA-Z\.]*@[0-9a-zA-Z\.]"
test2007@lzu.cn
```

范例：匹配URL地址(以http链接为例)

```
$ echo "http://news.lzu.edu.cn/article.jsp?newsid=10135" | grep "http://[0-9a-zA-Z\./=?]*"
http://news.lzu.edu.cn/article.jsp?newsid=10135
```

说明：

* /dev/null和/dev/zero是非常有趣的两个设备，它们都犹如一个黑洞，什么东西掉进去都会消失殆尽；后者则是一个能源箱，你总能从那里取到0，直到你退出。两者的部分用法见：关于zero及NULL设备的一些问题
* [[:space:]]是grep用于匹配空格或者TAB键类型字符串的一种标记，其他类似的标记请查看grep的帮助：man grep。
* 上面都是用grep来进行模式匹配，实际上sed，awk都可以用来做模式匹配，关于匹配中用到的正则匹配模式知识，大家可以参考正则匹配模式，更多相关资料请看参考资料。
* 如果仅仅想判断字符串是否为空，即判断字符串的长度是否为零，那么可以简单的通过test命令的-z选项来判断，具体用法见test命令，man test.

判断字符是否可打印

范例：用grep判断某个字符是否为可打印字符

```
$ echo "\\t\\n" | grep "[[:print:]]"
\\t\\n
$ echo $?
0
$ echo -e "\\t\\n" | grep "[[:print:]]"
$ echo $?
1
```

控制字符在终端的显示

范例：用echo的-e选项在屏幕控制字符显示位置、颜色、背景等

```
$ echo -e "\33[31;40m" #设置前景色为黑色，背景色为红色
$ echo -e "\33[11;29H Hello, World\!" #在屏幕的第11行，29列开始打印字符串Hello,World!
```

范例：在屏幕的某个位置动态显示当前系统时间

```
$ while :; do echo -e "\33[11;29H "$(date "+%Y-%m-%d %H:%M:%S"); done
```

范例：过滤掉某些控制字符串 用col命令过滤掉某些控制字符，在处理诸如script,screen等截屏命令的输出结果时，很有用

```
$ screen -L
$ cat /bin/cat
$ exit
$ cat screenlog.0 | col -b   # 把一些控制字符过滤后，就可以保留可读的操作日志
```

更多关于字符在终端的显示控制方法，请参考资料[20]和字符显示实例[21]：用shell实现的一个动态时钟。

4.2.2 字符串的长度

除了组成字符串的字符类型外，字符串还有哪些属性呢？组成字符串的字符个数。

下面我们来计算字符串的长度，即所有字符的个数，并简单介绍几种求字符串中指定字符个数的方法。

范例：计算某个字符串的长度

即所有字符的个数[这计算方法是五花八门，择其优着而用之]

```
$ var="get the length of me"
$ echo ${var}     # 这里等同于$var
get the length of me
$ echo ${#var}
20
$ expr length "$var"
20
$ echo $var | awk '{printf("%d\n", length($0));}'
20
$ echo -n $var |  wc -c
20
```

范例：计算某些指定一个字符或者多个字符的个数

```
$ echo $var | tr -cd g | wc -c
2
$ echo -n $var | sed -e 's/[^g]//g' | wc -c
2
$ echo -n $var | sed -e 's/[^gt]//g' | wc -c
5
```

范例：统计单词个数

更多相关信息见《shell编程之数值计算》之 单词统计 范例。

```
$ echo $var | wc -w
5
```

```
$ echo "$var" | tr " " "\n" | grep get | uniq -c
1
$ echo "$var" | tr " " "\n" | grep get | wc -l
1
```

说明：

操作符在*Bash*里头一个“大牛”，能胜任相当多的工作，具体就看看网中人的《*shell*十三问》之《*Shell*十三))　()　有{ }差在哪？”　吧。

4.3 字符串的存储

在我们看来，字符串是一连串的字符而已，但是为了操作方便，我们往往可以让字符串呈现出一定的结构。在这里，我们不关心字符串在内存中的实际存储结构，仅仅关系它呈现出来的逻辑结构。比如，这样一个字符串：“get the length of me”，我们可以从不同的方面来呈现它。

* 通过字符在串中的位置来呈现它

这样我们就可以通过指定位置来找到某个子串。这在c语言里头通常可以利用指针来做。而在shell编程中，有很多可用的工具，诸如expr，awk都提供了类似的方法来实现子串的查询动作。两者都几乎支持模式匹配(match)和完全匹配(index)。这在后面的字符串操作中将详细介绍。

* 根据某个分割符来取得字符串的各个部分

这里最常见的就是行分割符、空格或者TAB分割符了，前者用来当行号，我们似乎已经司空见惯了，因为我们的编辑器就这样“莫名”地处理着行分割符（在unix下为\n，在其他系统下有一些不同，比如windows下为\r\n）。而空格或者TAB键经常用来分割数据库的各个字段，这似乎也是司空见惯的事情。

正是因为这样，所以产生了大量优秀的行编辑工具，诸如grep,awk,sed等。在“行内”（姑且这么说吧，就是处理单行，即字符串里头不再包含行分割符）的字符串分割方面，cut和awk提供了非常优越的“行内”（处理单行）处理能力。

* 更方便地处理用分割符分割好的各个部分

同样是用到分割符，但为了更方便的操作分割以后的字符串的各个部分，我们抽象了“数组”这么一个数据结构，从而让我们更加方便地通过下标来获取某个指定的部分。bash提供了这么一种数据结构，而优秀的awk也同样提供了它，我们这里将简单介绍它们的用法。

4.3.1 范例：利用数组存放“get the length of me”的用空格分开的各个部分

* bash提供的数组数据结构，它是以数字为下标的，和C语言从0开始的下标一样

```
$ var="get the length of me"
$ var_arr=($var)    #这里把字符串var存放到字符串数组var_arr中了，默认以空格作为分割符
$ echo ${var_arr[0]} ${var_arr[1]} ${var_arr[2]} ${var_arr[3]} ${var_arr[4]}
```

```
get the length of me
$ echo ${var_arr[@]}    #这个就是整个字符串所有部分啦，这里可以用*代替@，下同
get the length of me
$ echo ${#var_arr[@]}    #记得上面求某个字符串的长度么，#操作符，如果想求某个数组元素的字符串长度，那
5
```

你也可以直接给某个数组元素赋值

```
$ var_arr[5]="new_element"
$ echo ${var_arr[5]}
6
$ echo ${var_arr[5]}
new_element
```

bash里头实际上还提供了一种类似于“数组”的功能，即“for i in 用指定分割符分开的字符串” 的用法 即，你可以很方便的获取某个字符串的某个部分

```
$  for i in $var; do echo -n $i" "; done;
get the length of me
```

* awk里头的数组，注意比较它和bash提供的数组的异同

split把一行按照空格分割，存放到数组var_arr中，并返回数组的长度。注意：这里的第一个元素下标不是0，而是1

```
$ echo $var | awk '{printf("%d %s\n", split($0, var_arr, " "), var_arr[1]);}'
5 get
```

实际上，上面的操作很类似awk自身的行处理功能：awk默认把一行按照空格分割为多个域，并可以通过$1,$2,$3…来获取,$0表示整行 这里的NF是该行的域的总数，类似于上面数组的长度，它同样提供了一种通过“下标”访问某个字符串的功能

```
$ echo $var | awk '{printf("%d | %s %s %s %s %s | %s\n", NF, $1, $2, $3, $4, $5, $0);}'
5 | get the length of me | get the length of me
```

awk的“数组”功能何止于此呢，看看它的for引用吧，注意，这个和bash里头的for不太一样，i不是元素本身，而是下标

```
$ echo $var | awk '{split($0, var_arr, " "); for(i in var_arr) printf("%s ",var_arr);}'
get the length of me
$ echo $var | awk '{split($0, var_arr, " "); for(i in var_arr) printf("%s ",i);}'
1 2 3 4 5
```

awk还有更“厉害”的处理能力，它的下标可以不是数字，而可以是字符串，从而变成了“关联”数组，这种“关联”的作用在某些方便将让我们非常方便 比如，我们这里就实现一个非凡的应用，把某个文件中的某个系统调用名替换成地址，如果你真正用起它，你会感慨它的“鬼斧神工”的。 这就是我在一个场合最好才发现的随好的实现方案：有兴趣看看awk手册帖子中我在3楼回复的实例吧。

```
$ cat symbol
sys_exit
sys_read
sys_close
$ ls /boot/System.map*
$ awk '{if(FILENAME ~ "System.map") map[$3]=$1; else {printf("%s\n", map[$1])}}' /boot/System.map-2.
c0129a80
c0177310
c0175d80
```

另外，awk还支持删除某个数组元素，如果你不用了就可以用delete函数给删除掉。如果某些场合有需要的话，别忘了awk还支持二维数组。

okay，就介绍到这里啦。为什么要介绍这些内容？再接着看下面的内容，你就会发现，那些有些的工具是怎么产生和发展起来的了，如果累了，看看最后一篇参考资料吧，它介绍了一些linux命令名字的由来，说不定可以帮助你理解本节下面的部分呢。

4.4 字符串常规操作

字符串操作包括取子串、查询子串、插入子串、删除子串、子串替换、子串比较、子串排序、子串进制转换、子串编码转换等。

4.4.1 取子串

取子串的方法主要有：直接到指定位置求子串，字符匹配求子串。

范例：按照位置取子串

比如从什么位置开始，取多少个字符

```
$ var="get the length of me"
$ echo ${var:0:3}
get
$ echo ${var:(-2)}   # 方向相反呢
me

$ echo `expr substr "$var" 5 3` #记得把$var引起来，否则expr会因为空格而解析错误
the

$ echo $var | awk '{printf("%s\n", substr($0, 9, 6))}'
length
```

awk把$var按照空格分开为多个变量，依次为$1,$2,$3,$4,$5

```
$ echo $var | awk '{printf("%s\n", $1);}'
get
$ echo $var | awk '{printf("%s\n", $5);}'
me
```

差点把cut这个小东西忘记啦，用起来和awk类似，-d指定分割符，如同awk用-F指定分割符一样，-f指定“域”，如同awk的$数字。

```
$ echo $var | cut -d" " -f 5
```

范例：匹配字符求子串

用Bash内置支持求字串

```
$ echo ${var%% *} #从右边开始计算，删除最左边的空格右边的所有字符
get
$ echo ${var% *} #从右边开始计算，删除第一个空格右边的所有字符
get the length of
$ echo ${var##* }  #从左边开始计算，删除最右边的空格左边的所有字符
me
$ echo ${var#* }  #从左边开始计算，删除第一个空格左边的所有字符
the length of me
```

删除所有 空格＋字母串 的字符串，所以get后面的全部被删除了

```
$ echo $var | sed 's/ [a-z]*//g'
get
$ echo $var | sed 's/[a-z]* //g'
me
```

sed有按地址（行）打印(p)的功能，记得先用tr把空格换成行号

```
$ echo $var | tr " " "\n" | sed -n 1p
get
$ echo $var | tr " " "\n" | sed -n 5p
me
```

tr也可以用来取子串哦，它也可以类似#和%来“拿掉”一些字符串来实现取子串

```
$ echo $var | tr -d " "
getthelengthofme
$ echo $var | tr -cd "[a-z]" #把所有的空格都拿掉了，仅仅保留字母字符串，注意-c和-d的用法
getthelengthofme
```

说明：

* %和#的区别是，删除字符的方向不一样，前者在右，后者在左，%%和%，##和#的方向是前者是最大匹配，后者是最小匹配。（好的记忆方法见网中人的键盘记忆法：#S%是键盘依次从左到右的三个键）
* tr的-c选项是complement的缩写，即invert，而-d选项是删除的意思，`tr -cd "[a-z]"`这样一来就变成保留所有的字母啦。

对于字符串的截取，实际上还有一些命令，如果head，tail等可以实现有意思的功能，可以截取某个字符串的前面、后面指定的行数或者字节数。例如：

```
$ echo "abcdefghijk" | head -c 4
abcd
$ echo -n "abcdefghijk" | tail -c 4
hijk
```

4.4.2 查询子串

子串查询包括：返回符合某个模式的子串本身和返回子串在目标串中的位置。

准备：在进行下面的操作之前，请准备一个文件text，里头有内容“consists of”，用于下面的操作。

范例：查询子串在目标串中的位置

貌似仅仅可以返回某个字符或者多个字符中第一个字符出现的位置

```
$ var="get the length of me"
$ expr index "$var" t
3
```

awk却能找出字串，match还可以匹配正则表达式

```
$ echo $var | awk '{printf("%d\n", match($0,"the"));}'
5
```

范例：查询子串，返回包含子串的行

awk，sed都可以实现这些功能，但是grep最擅长

```
$ grep "consists of" text   # 查询text文件包含consists of的行，并打印这些行
$ grep "consists[[:space:]]of" -n -H text # 打印文件名，子串所在行的行号和该行的内容
$ grep "consists[[:space:]]of" -n -o text # 仅仅打印行号和匹配到的子串本身的内容

$ awk '/consists of/{ printf("%s:%d:%s\n",FILENAME, FNR, $0)}' text  #看到没？和grep的结果一样
$ sed -n -e '/consists of/=;/consists of/p' text #同样可以打印行号
```

说明：

* awk,grep,sed都能通过模式匹配查找指定的字符串，但是它们各有擅长的领域，我们将在后续的章节中继续使用和比较它们，从而发现各自的优点。
* 在这里我们姑且把文件内容当成了一个大的字符串，在后面的章节中我们将专门介绍文件的操作，所以对文件内容中存放字符串的操作将会有更深入的分析和介绍。

4.4.3 子串替换

子串替换就是把某个指定的子串替换成其他的字符串，实际上这里就蕴含了“插入子串”和“删除子串”的操作。例如，你想插入某个字符串到某个子串之前，就可以把原来的子串替换成”子串+新的字符串“，如果想删除某个子串，就把子串替换成空串。不过有些工具提供了一些专门的用法来做插入子串和删除子串的操作，所以呆伙还是会专门介绍的。另外，要想替换掉某个子串，一般都是先找到子串（查询子串），然后再把它替换掉的，实质上很多工具在使用和设计上都体现了这么一点。

范例：把变量var中的空格替换成下划线

用{}运算符，还记得么？网中人的教程。

```
$ var="get the length of me"
$ echo ${var/ /_}        #把第一个空格替换成下划线
get_the length of me
$ echo ${var// /_}        #把所有空格都替换成了下划线了
get_the_length_of_me
```

用awk，awk提供了转换的最小替换函数sub和全局替换函数gsub，类似/和//

```
$ echo $var | awk '{sub(" ", "_", $0); printf("%s\n", $0);}'
get_the length of me
$ echo $var | awk '{gsub(" ", "_", $0); printf("%s\n", $0);}'
get_the_length_of_me
```

用sed了，子串替换可是sed的特长

```
$ echo $var | sed -e 's/ /_/'    #s <= substitude
get_the length of me
$ echo $var | sed -e 's/ /_/g'    #看到没有，简短两个命令就实现了最小匹配和最大匹配g <= global
get_the_length_of_me
```

有忘记tr命令么？可以用替换单个字符的

```
$ echo $var | tr " " "_"
get_the_length_of_me
$ echo $var | tr '[a-z]' '[A-Z]'    #这个可有意思了，把所有小写字母都替换为大写字母
GET THE LENGTH OF ME
```

说明：sed还有很有趣的标签用法呢，下面再介绍吧。

有一种比较有意思的字符串替换是，整个文件行的倒置，这个可以通过tac命令实现，它会把文件中所有的行全部倒转过来。在一定意义上来说，排序实际上也是一个字符串替换。

4.4.4 插入子串

就是在指定的位置插入子串，这个位置可能是某个子串的位置，也可能是从某个文件开头算起的某个长度。通过上面的练习，我们发现这两者之间实际上是类似的。

公式：插入子串=把“old子串”替换成“old子串+new子串”或者“new子串+old子串”

范例：在var字符串的空格之前或之后插入一个下划线

用{}

```
$ var="get the length of me"
$ echo ${var/ /_ }        #在指定字符串之前插入一个字符串
get_ the length of me
$ echo ${var// /_ }
get_ the_ length_ of_ me
$ echo ${var/ / _}        #在指定字符串之后插入一个字符串
get _the length of me
$ echo ${var// / _}
get _the _length _of _me
```

其他的还用演示么？这里主要介绍sed怎么用来插入字符吧，因为它的标签功能很有趣
说明：(和)将不匹配到的字符串存放为一个标签，按匹配顺序为\1,\2…

```
$ echo $var | sed -e 's/\( \)/_\1/'
get_ the length of me
$ echo $var | sed -e 's/\( \)/_\1/g'
get_ the_ length_ of_ me
$ echo $var | sed -e 's/\( \)/\1_/'
get _the length of me
$ echo $var | sed -e 's/\( \)/\1_/g'
get _the _length _of _me
```

看看sed的标签的顺序是不是\1,\2…., 看到没？\2和\1掉换位置后，the和get的位置掉换了

```
$ echo $var | sed -e 's/\([a-z]*\) \([a-z]*\) /\2 \1 /g'
the get of length me
```

sed还有专门的插入指令，a和i,分别表示在匹配的行后和行前插入指定字符

```
$ echo $var | sed '/get/a test'
get the length of me
test
$ echo $var | sed '/get/i test'
test
get the length of me
```

4.4.5 删除子串

删除子串：应该很简单了吧，把子串替换成“空”（什么都没有）不就变成了删除么。还是来简单复习一下替换吧。

范例：把var字符串中所有的空格给删除掉。 鼓励：这样一替换不知道变成什么单词啦，谁认得呢？但是中文却是连在一起的，所以中文有多难，你想到了么？原来你也是个语言天才，而英语并不可怕，你有学会它的天赋，只要你有这个打算。

再用{}

```
$ echo ${var// /}
getthelengthofme
```

再用awk

```
$ echo $var | awk '{gsub(" ","",$0); printf("%s\n", $0);}'
```

再用sed

```
$ echo $var | sed 's/ //g'
getthelengthofme
```

还有更简单的tr命令，tr也可以把” ”给删除掉，看

```
$ echo $var | tr -d " "
getthelengthofme
```

如果要删除掉第一个空格后面所有的字符串该怎么办呢？还记得{}的#和%用法么？如果不记得，回到这一节的还头开始复习吧。（实际上删除子串和取子串未尝不是两种互补的运算呢，删除掉某些不想要的子串，也就同时取得另外那些想要的子串——这个世界就是一个“二元”的世界，非常有趣）

4.4.6 子串比较

这个很简单：还记得test命令的用法么？man test。它可以用来判断两个字符串是否相等的。另外，你发现了“字符串是否相等”和“字符串能否跟另外一个字符串匹配”两个问题之间的关系吗？如果两个字符串完全匹配，那么这两个字符串就相等了。所以呢，上面用到的字符串匹配方法，也同样可以用到这里。

4.4.7 子串排序

差点忘记这个重要的内容了，子串排序可是经常用到的，常见的有按字母序、数字序等正序或反序排列。sort命令可以用来做这个工作，它和其他行处理命令一样，是按行操作的，另外，它类似cut和awk，可以指定分割符，并指定需要排序的列。

```
$ var="get the length of me"
$ echo $var | tr ' ' '\n' | sort   #正序排
get
length
me
of
the
$ echo $var | tr ' ' '\n' | sort  -r #反序排
the
of
me
length
get
$ cat > data.txt
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
41 45 44 44 26 44 42 20 20 38 37 25 45 45 45
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
44 20 30 39 35 38 38 28 25 30 36 20 24 32 33
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
41 33 51 39 20 20 44 37 38 39 42 40 37 50 50
46 47 48 49 50 51 52 53 54 55 56
42 43 41 42 45 42 19 39 75 17 17
$ cat data.txt | sort -k 2 -n
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
44 20 30 39 35 38 38 28 25 30 36 20 24 32 33
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
41 33 51 39 20 20 44 37 38 39 42 40 37 50 50
42 43 41 42 45 42 19 39 75 17 17
41 45 44 44 26 44 42 20 20 38 37 25 45 45 45
46 47 48 49 50 51 52 53 54 55 56
```

4.4.8 子串进制转换

如果字母和数字字符用来计数，那么就存在进制转换的问题。在数值计算一节的回复资料里，我们已经介绍了bc命令，这里再简单的复习一下。

```
$ echo "ibase=10;obase=16;10" | bc
A
```

说明：ibase指定输入进制，obase指出输出进制，这样通过调整ibase和obase，你想怎么转就怎么转啦！

4.4.9 子串编码转换

什么是字符编码？这个就不用介绍了吧，看过那些乱七八糟显示的网页么？大多是因为浏览器显示时的”编码“和网页实际采用的”编码“不一致导致的。字符编码通常是指把一序列”可打印“字符转换成二进制表示，而字符解码呢则是执行相反的过程，如果这两个过程不匹配，则出现了所谓的”乱码“。

为了解决”乱码“问题呢？就需要进行编码转换。在linux下，我们可以使用iconv这个工具来进行相关操作。这样的情况经常在不同的操作系统之间移动文件，不同的编辑器之间交换文件的时候遇到，目前在windows下常用的汉字编码是gb2312，而在linux下则大多采用utf8。

```
$ nihao_gb2312=$(echo "你好"
$ nihao_utf8=$(echo $nihao_gb2312 | iconv -f gb2312 -t utf8)
$ PS1="$ "
$ echo $nihao_utf8
你好
```

说明：我的终端默认编码是utf8，所以结果如上。

4.5 字符串操作进阶

实际上，在用Bash编程时，大部分时间都是在处理字符串，因此把这一节熟练掌握非常重要。

4.5.1 正则表达式

范例：处理URL地址

URL地址(URL（Uniform Resoure Locator：统一资源定位器）是WWW页的地址)几乎是我们日常生活的玩伴，我们已经到了无法离开它的地步啦，对它的操作很多，包括判断URL地址的有效性，截取地址的各个部分（服务器类型、服务器地址、端口、路径等）并对各个部分进行进一步的操作。

下面我们来具体处理这个URL地址：ftp://anonymous:ftp@mirror.lzu.edu.cn/software/scim-1.4.7.tar.gz

```
$ url="ftp://anonymous:ftp@mirror.lzu.edu.cn/software/scim-1.4.7.tar.gz"
```

匹配URL地址，判断URL地址的有效性

```
$ echo $url | grep "ftp://[a-z]*:[a-z]*@[a-z\./-]*"
```

截取服务器类型

```
$ echo ${url%%:*}
ftp
$ echo $url | cut -d":" -f 1
ftp
```

截取域名

```
$ tmp=${url##*@} ; echo ${tmp%%/*}
mirror.lzu.edu.cn
```

截取路径

```
$ tmp=${url##*@} ; echo ${tmp%/*}
mirror.lzu.edu.cn/software
```

截取文件名

```
$ basename $url
scim-1.4.7.tar.gz
$ echo ${url##*/}
scim-1.4.7.tar.gz
```

截取文件类型（扩展名）

```
$ echo $url | sed -e 's/.*[0-9].\(.*\)/\1/g'
tar.gz
```

范例：通过sed或者awk等命令匹配某个文件中的特定范围的行

先准备一个测试文件README

```
Chapter 7 -- Exercises

7.1 please execute the program: mainwithoutreturn, and print the return value
of it with the command "echo $?", and then compare the return of the printf
function, they are the same.

7.2 it will depend on the exection mode, interactive or redirection to a file,
if interactive, the "output" action will accur after the \n char with the line
buffer mode, else, it will be really "printed" after all of the strings have
been stayed in the buffer.

7.3 there is no another effective method in most OS. because argc and argv are
not global variables like environ.
```

然后开始实验，

打印出答案前指定行范围：第7行到第9行，刚好找出了第2题的答案

```
$ sed -n 7,9p README
7.2 it will depend on the exection mode, interactive or redirection to a file,
if interactive, the "output" action will accur after the \n char with the line
buffer mode, else, it will be really "printed" after all of the strings have
```

其实，因为这个文件内容格式很有特色，有更简单的办法

```
$ awk '/7.2/,/^$/ {printf("%s\n", $0);}' README
7.2 it will depend on the exection mode, interactive or redirection to a file,
if interactive, the "output" action will accur after the \n char with the line
buffer mode, else, it will be really "printed" after all of the strings have
been stayed in the buffer.
```

有了上面的知识，我们就可以非常容易地进行这些工作啦：修改某个文件的文件名，比如调整它的编码，下载某个网页里头的所有pdf文档等。这些就作为练习自己做吧，如果遇到问题，可以在回帖交流。

4.5.2 处理格式化的文本

平时做工作，大多数时候处理的都是一些“格式化”的文本，比如类似/etc/passwd这样的有固定行和列的文本，也有类似tree命令输出的那种具有树形结构的文本，当然还有其他具有特定结构的文本。

关于树状结构的文本的处理，可以考虑看看这这个例子：《用Graphviz进行可视化操作——绘制函数调用关系图》

实际上，只要把握好特性结构的一些特点，并根据具体的应用场合，处理起来就不会困难。

下面我们来介绍具体有固定行和列的文本的操作，以/etc/passwd文件为例。关于这个文件的帮忙和用户，请通过**man 5 passwd**查看。下面我们对这个文件以及相关的文件进行一些有意义的操作。

范例：选取指定列

选取/etc/passwd文件中的用户名和组ID两列

```
$ cat /etc/passwd | cut -d":" -f1,4
```

选取/etc/group文件中的组名和组ID两列

```
$ cat /etc/group | cut -d":" -f1,3
```

范例：文件关联操作

如果想找出所有用户所在的组，怎么办?

```
$ join -o 1.1,2.1 -t":" -1 4 -2 3 /etc/passwd /etc/group
root:root
bin:bin
daemon:daemon
adm:adm
lp:lp
pop:pop
nobody:nogroup
falcon:users
```

说明：join命令用来连接两个文件，有点类似于数据库的两个表的连接。-t指定分割符，-1 4 -2 3指定按照第一个文件的第4列和第二个文件的第3列，即组ID进行连接，-o 1.1,2.1表示仅仅输出第一个文件的第一列和第二个文件的第一列，这样就得到了我们要的结果，不过，可惜的是，这个结果并不准确，再进行下面的操作，你就会发现：

```
$ cat /etc/passwd | sort -t":" -n -k 4 > /tmp/passwd
$ cat /etc/group | sort -t":" -n -k 3 > /tmp/group
$ join -o 1.1,2.1 -t":" -1 4 -2 3 /tmp/passwd /tmp/group
halt:root
operator:root
root:root
shutdown:root
sync:root
bin:bin
daemon:daemon
adm:adm
lp:lp
pop:pop
nobody:nogroup
falcon:users
games:users
```

可以看到这个结果才是正确的，所以以后使用join千万要注意这个问题，否则采取更保守的做法似乎更能保证正确性，更多关于文件连接的讨论见参考资料[14]

上面涉及到了处理某格式化行中的指定列，包括截取（如SQL的select用法），连接（如SQL的join用法），排序（如SQL的order by用法），都可以通过指定分割符来拆分某个格式化的行，另外，“截取”的做法还有很多，不光是cut，awk，甚至通过IFS指定分割符的read命令也可以做到，例如：

```
$ IFS=":"; cat /etc/group | while read C1 C2 C3 C4; do echo $C1 $C3; done
```

因此，熟悉这些用法，我们的工作将变得非常灵活有趣。

到这里，需要做一个简单的练习，如何把按照列对应的用户名和用户ID转换成按照行对应的，即把类似下面的数据：

```
$ cat /etc/passwd | cut -d":" -f1,3 --output-delimiter=" "
root 0
bin 1
daemon 2
```

转换成：

```
$ cat a
root    bin     daemon
0       1       2
```

并转换回去，有什么办法呢？记得诸如tr，paste,split等命令都可以使用。

参考方法：

* 正转换：先截取用户名一列存入文件user，再截取用户ID存入id，再把两个文件用paste -s命令连在一起，这样就完成了正转换。
* 逆转换：先把正转换得到的结果用split -l拆分成两个文件，再把两个拆分后的文件用tr把分割符“\t”替换成“\n”，只有用paste命令把两个文件连在一起，这样就完成了逆转换。

4.6 参考资料

* 《高级Bash脚本编程指南》之操作字符串
* 《高级Bash脚本编程指南》之指定变量的类型
* 《Shell十三问》之$(( ))　()　有{ }差在哪?
* Regular Expressions - User guide
* Regular Expression Tutorial
* Grep Tutorial
* Sed Tutorial
* awk Tutorial
* sed Tutorial
* An awk Primer
* 一些奇怪的 unix 指令名字的由来
* 磨练构建正则表达式模式的技能
* 基础11：文件分类、合并和分割(sort,uniq,join,cut,paste,split)
* 使用Linux 文本工具简化数据的提取
* 如何控制终端：光标位置，字符颜色，背景，清屏…
* 在终端动态显示时间
* 用Shell写的五笔反查小工具
* SED单行脚本快速参考（Unix 流编辑器

4.7 后记

* 这一节本来是上个礼拜该弄好的，但是这些天太忙了，到现在才写好一个“初稿”，等到有时间再补充具体的范例。这一节的范例应该是最最有趣的，所有得好好研究一下几个有趣的范例。
* 写完上面的部分貌似是1点多，刚check了一下错别字和语法什么的，再添加了一节，即“字符串的存储结构”，到现在已经快half past 2啦，晚安，朋友们。
* 26号，添加“子串进制转换”和“子串编码转换”两小节以及一个处理URL地址的范例。

第 5 章

文件操作

5.1 前言

这一周我们来探讨文件操作。

在日常学习和工作中，我们总是在不断地和各种文件打交道，这些文件包括普通的文本文件，可以执行的程序文件，带有控制字符的文档、存放各种文件的目录文件、网络套接字文件、设备文件等。这些文件又具有诸如属主、大小、创建和修改日期等各种属性。文件对应文件系统的一些数据块，对应磁盘等存储设备的一片连续空间，对应于显示设备却是一些具有不同形状的字符集。

在这一节，为了把关注点定位在文件本身，我们不会深入探讨文件系统以及存储设备是如何组织文件的（在后续章节再深入探讨），而是探讨我们对它最熟悉的一面，即把文件当成是一序列的字符（一个byte）集合看待。因此之前介绍的《shell编程范例之字符串操作》在这里将会得到广泛的应用，关于普通文件的读写操作我想我们已经用得非常熟练啦，那就是“重定向”，在这里，我们会把这部分独立出来介绍。关于文件在Linux下的“数字化”（文件描述符）高度抽象，“一切皆为文件”的哲学在shell编程里也得到了深刻的体现。

下面我们先来介绍文件的各种属性，然后介绍普通文件的一般操作。

5.2 文件的各种属性

首先，我们通过文件的结构体来看看文件到底有哪些属性：

```
struct stat {
    dev_t st_dev; /* 设备□□ */
    ino_t st_ino; /* 节点□□ */
    mode_t st_mode; /* 模式□□ */
    nlink_t st_nlink; /* 硬连接 */
    uid_t st_uid; /* 用户ID */
    gid_t st_gid; /* 组ID□□ */
    dev_t st_rdev; /* 设备类型 */
    off_t st_off; /* 文件字节数 */
    unsigned long□ st_blksize; /* 块大小 */
    unsigned long st_blocks; /* 块数□□ */
```

```
    time_t st_atime; /* 最后一次访问时间 */
    time_t st_mtime; /* 最后一次修改时间 */
    time_t st_ctime; /* 最后一次改变时间(指属性) */
};
```

下面逐次来了解这些属性，如果需要查看某个文件的属性，用stat命令就可以，会按照上面的结构体把信息列出来。另外，ls命令在跟上一定的参数后也可以显示文件的相关属性，比如-l参数。

5.2.1 文件类型

文件类型对应于上面的st_mode，文件类型有很多，比如常规文件、符号链接（硬链接、软链接）、管道文件、设备文件(符号设备、块设备)、socket文件等，不同的文件类型对应不同的功能和作用。

范例：在命令行简单地区分各类文件

```
$ ls -l
total 12
drwxr-xr-x 2 root root 4096 2007-12-07 20:08 directory_file
prw-r-r-- 1 root root    0 2007-12-07 20:18 fifo_pipe
brw-r-r-- 1 root root 3, 1 2007-12-07 21:44 hda1_block_dev_file
crw-r-r-- 1 root root 1, 3 2007-12-07 21:43 null_char_dev_file
-rw-r-r-- 2 root root  506 2007-12-07 21:55 regular_file
-rw-r-r-- 2 root root  506 2007-12-07 21:55 regular_file_hard_link
lrwxrwxrwx 1 root root   12 2007-12-07 20:15 regular_file_soft_link -> regular_file
$ stat directory_file/
  File: `directory_file/'
  Size: 4096            Blocks: 8          IO Block: 4096   directory
Device: 301h/769d       Inode: 521521      Links: 2
Access: (0755/drwxr-xr-x)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2007-12-07 20:08:18.000000000 +0800
Modify: 2007-12-07 20:08:18.000000000 +0800
Change: 2007-12-07 20:08:18.000000000 +0800
$ stat null_char_dev_file
  File: `null_char_dev_file'
  Size: 0               Blocks: 0          IO Block: 4096   character special file
Device: 301h/769d       Inode: 521240      Links: 1     Device type: 1,3
Access: (0644/crw-r-r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2007-12-07 21:43:38.000000000 +0800
Modify: 2007-12-07 21:43:38.000000000 +0800
Change: 2007-12-07 21:43:38.000000000 +0800
```

说明： 在ls命令结果每行的第一个字符我们可以看到，它们之间都不相同，这正好反应了不同文件的类型。d表示目录，-表示普通文件（或者硬链接），l表示符号链接，p表示管道文件，b和c分别表示块设备和字符设备（另外s表示socket文件）。在stat命令的结

果中，我们可以在第二行的最后找到说明，从上面的操作可以看出，directory_file是目录，stat命令的结果中用directory表示，而null_char_dev_file它则用character special file说明。

范例：简单比较它们的异同

通常，我们只会用到目录、普通文件、以及符号链接，很少碰到其他类型的文件，不过这些文件还是各有用处的，如果要做嵌入式开发或者进程通信等，你可能会涉及到设备文件、有名管道(FIFO)。下面我们通过简单的操作来反应它们之间的区别（具体的原理可能会在下一节《shell编程范例之文件系统》介绍，如果感兴趣，也可以提前到网上找找设备文件的作用、快设备和字符设备的区别、以及驱动程序中如何编写相关设备驱动等）。

对于普通文件：就是一序列字符的集合，所以可以读、写等

```
$ echo "hello, world" > regular_file
$ cat regular_file
hello, world
```

目录文件下，我们可以创建新的文件，所以目录文件还有叫法：文件夹，到后面我们会分析目录文件的结构体，它实际上存放了它下面的各个文件的文件名。

```
$ cd directory_file
$ touch file1 file2 file3
```

对于有名管道，操作起来比较有意思：如果你要读它，除非有内容，否则阻塞；如果你要写它，除非有人来读，否则阻塞。它常用于进程通信中。你可以打开两个终端terminal1和terminal2，试试看：

```
terminal1$ cat fifo_pipe   #刚开始阻塞在这里，直到下面的写动作发生，才打印test字符串
terminal2$ echo "test" > fifo_pipe
```

关于块设备，字符设备，上面的两个设备文件对应于/dev/hda1和/dev/null，如果你用过u盘，或者是写过简单的脚本的话，这样的用法你应该用过 :-)

```
$ mount hda1_block_dev_file /mnt   #挂载硬盘的第一个分区到/mnt下（关于挂载的原理，我们在下一节讨论）
$ echo "fewfewfef" > /dev/null  #/dev/null像个黑洞，什么东西丢进去都消失殆尽
```

最后两个文件分别是regular_file文件的硬链接和软链接，你去读写它们，他们的内容是相同的，不过你去删除它们，他们去互不相干，硬链接和软链接又有什么不同呢？前者可以说就是原文件，后者呢只是有那么一个inode，但没有实际的存储空间，建议用stat命令查看它们之间的区别，包括它们的Blocks,inode等值，也可以考虑用diff比较它们的大小。

```
$ ls regular_file*
ls regular_file* -l
-rw-r-r-- 2 root root 204800 2007-12-07 22:30 regular_file
-rw-r-r-- 2 root root 204800 2007-12-07 22:30 regular_file_hard_link
```

```
lrwxrwxrwx 1 root root     12 2007-12-07 20:15 regular_file_soft_link -> regular_file
$ rm regular_file      # 删除原文件
$ cat regular_file_hard_link    # 硬链接还在，而且里头的内容还有呢
fefe
$ cat regular_file_soft_link
cat: regular_file_soft_link: No such file or directory
```

虽然软链接文件本身还在，不过因为它本身不存储内容，所以读不到东西拉，这就是软链接和硬链接的区别，该知道则么用它们了吧。

另外，需要注意的是，硬链接不可以跨文件系统，而软链接则可以。另外，也不不允许给目录创建硬链接。

范例：普通文件再分类

文件类型从Linux文件系统那么一个级别分了以上那么多类型，不过普通文件还是可以再分的（根据文件内容的”数据结构“分），比如常见的文本文件，可执行的ELF文件，odt文档，jpg图片格式，swap分区文件，pdf文件。除了文本文件外，它们大多是二进制文件，有特定的结构，因此需要有专门的工具来创建和编辑它们。关于各类文件的格式，可以参考相关文档标准。不过如果能够了解Linux下可执行的ELF文件的工作原理，可能对你非常有用。所以，如果有兴趣，建议阅读一下参考资料中和ELF文件相关部分。

虽然各类普通文件都有专属的操作工具，但是我们还是可以直接读、写它们，这里先提到这么几个工具，回头讨论细节。

od：以八进制或者其他格式“导出”文件内容。 strings：读出文件中的字符(可打印的字符) gcc,gdb,readelf,objdump等：ELF文件分析、处理工具（gcc编译器、gdb调试器、readelf分析elf文件，objdump反编译工具）

这里补充一个非常重要的命令，file，这个命令用来查看各类文件的属性。和stat命令相比，它可以进一步识别普通文件，即stat命令显示的regular file。因为regular file可以有各种不同的结构，因此在操作系统的支持下得到不同的解释，执行不同的动作。虽然，Linux下，文件也会加上特定的后缀以便用户能够方便地识别文件的类型，但是Linux操作系统根据文件头识别各类文件，而不是文件后缀，这样，在解释相应的文件时就更不容易出错。下面我们简单介绍file命令的用法。

```
$ file ./
./: directory
$ file /etc/profile
/etc/profile: ASCII English text
$ file /lib/libc-2.5.so
/lib/libc-2.5.so: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), not stripped
$ file /bin/test
/bin/test: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared
$ file /dev/hda
/dev/hda: block special (3/0)
$ file /dev/console
/dev/console: character special (5/1)
$ cp /etc/profile .
$ tar zcf profile.tar.gz profile
```

```
$ file profile.tar.gz
profile.tar.gz: gzip compressed data, from Unix, last modified: Tue Jan  4 18:53:53 2000
$ mkfifo fifo_test
$ file fifo_test
fifo_test: fifo (named pipe)
```

更多用法见file命令的手册，关于file命令的实现原理，请参考magic的手册（看看/etc/file/magic文件，了解什么是文件的magic number等）。

5.2.2 文件属主

Linux作为一个多用户系统，为多用户使用同一个系统提供了极大的方便，比如对于系统上的文件，它通过属主来区分不同的用户，以便分配它们对不同文件的操作权限。为了更方便地管理，文件属主包括该文件所属用户，以及该文件所属的用户组，因为用户可以属于多个组。我们先来简单介绍Linux下用户和组的管理。

Linux下提供了一组命令用于管理用户和组，比如用户创建它们的useradd和groupadd，用户删除它们的userdel和groupdel，另外，passwd命令用于修改用户密码。当然，Linux还提供了两个相应的配置，即/etc/passwd和/etc/group，另外，有些系统还把密码单独放到了配置文件/etc/shadow中。关于它们的详细用法请参考后面的资料，在这里我们不介绍了，仅介绍文件和用户之间的一些关系。

范例：修改文件的属主

```
$ chown 用户名:组名 文件名
```

如果要递归地修改某个目录下所有文件的属主，可以添加-R选项。

在本节开头我们列的文件结构体中，可以看到仅仅有用户ID和组ID的信息，但ls -l的结果却显示了用户名和组名信息，这个是怎么实现的呢？下面先看看-n的结果：

范例：查看文件的属主

```
$ ls -n regular_file
-rw-r--r-- 1 0 0 115 2007-12-07 23:45 regular_file
$ ls -l regular_file
-rw-r--r-- 1 root root 115 2007-12-07 23:45 regular_file
```

范例：分析文件属主实现的背后原理

可以看到，ls -n显示了用户ID和组ID，而ls -l显示了它们的名字。还记得上面提到的两个配置文件/etc/passwd和/etc/group文件么？它们分别存放了用户ID和用户名，组ID和组名的对应关系，因此很容易想到ls -l命令在实现时是如何通过文件结构体的ID信息找到它们对应的名字信息的。如果想对ls -l命令的实现有更进一步的了解，可以用strace跟踪看看它是否读取了/etc/passwd和/etc/group这个两个文件。

```
$ strace -f -o strace.log ls -l regular_file
$ cat strace.log | egrep "passwd|group|shadow"
```

```
2989  open("/etc/passwd", O_RDONLY)      = 3
2989  open("/etc/group", O_RDONLY)       = 3
```

说明：strace是一个非常有用的工具，可以用来跟踪系统调用和信号。如同gdb等其他强大的工具一样，它基于系统的ptrace系统调用实现，所以如果你感兴趣，可以好好研究一下这个工具。

实际上，把属主和权限分开介绍不太好，因为只有它们两者结合才使得多用户系统成为可能，否则无法隔离不同用户对某个文件的操作，所以下面来介绍文件操作权限。

5.2.3 文件权限

从ls -l命令的结果的第一列的后9个字符中，我们可以看到类似这样的信息rwxr-xr-x，它们对应于文件结构体的st_mode部分（st_mode包含文件类型信息和文件权限信息两部分）。这类信息可以分成三部分，即rwx,r-x,r-x,分别对应该文件所属用户，所属组，其他组对该文件的操作权限，如果有rwx中任何一个表示可读、可写、可执行，如果为-表示没有这个权限。对应的，可以用八进制来表示它，比如rwxr-xr-x就可表示成二进制111101101，对应的八进制则为755。正是因为这样，我们要修改文件的操作权限时，也可以有多种方式来实现，它们都可通过chmod命令来修改。

范例：给文件添加读、写、可执行权限

比如，把regular_file的文件权限修改为所有用户都可读、可写、可执行，即rwxrwxrwx，也可表示为111111111，翻译成八进制，则为777。这样就可以通过两种方式修改这个权限。

```
$ chmod a+rwx regular_file
```

或

```
$ chmod 777 regular_file
```

说明：a指所用用户，如果只想给用户本身可读可写可执行权限，那么可以把a换成u；而+就是添加权限，相反的，如果想去掉某个权限，用-，而rwx则对应可读、可写、可执行。更多用法见chmod命令的帮助。

实际上除了这些权限外，还有两个涉及到安全方面的权限，即setuid/setgid和只读控制等。

如果设置了文件（程序或者命令）的setuid/setgid权限，那么用户将可用root身份去执行该文件，因此，这将可能带来安全隐患；如果设置了文件的只读权限，那么用户将仅仅对该文件将有可读权限，这为避免诸如rm -rf的“可恶”操作带来一定的庇佑。

范例：授权普通用户执行root所属命令

默认情况下，系统是不允许普通用户执行passwd命令的，通过setuid/setgid，可以授权普通用户执行它。

```
$ ls -l /usr/bin/passwd
-rwx-x--x 1 root root 36092 2007-06-19 14:59 /usr/bin/passwd
```

```
$ su      #切换到root用户，给程序或者命令添加“粘着位”
$ chmod +s /usr/bin/passwd
$ ls -l /usr/bin/passwd
-rws-s--x 1 root root 36092 2007-06-19 14:59 /usr/bin/passwd
$ exit
$ passwd #普通用户通过执行该命令，修改自己的密码
```

说明：

> setuid和setgid位是让普通用户可以以root用户的角色运行只有root帐号才能运行的程序或命令。

虽然这在一定程度上为管理提供了方便，比如上面的操作让普通用户可以修改自己的帐号，而不是要root帐号去为每个用户做这些工作。关于setuid/setgid的更多详细解释，请参考最后推荐的资料。

范例：给重要文件加锁

只读权限示例：给重要文件加锁(添加不可修改位[immutable]))，以避免各种误操作带来的灾难性后果（例如：rm -rf）

```
$ chattr +i regular_file
$ lsattr regular_file
----i-------- regular_file
$ rm regular_file      #加了immutable位以后，你无法对文件进行任何“破坏性”的活动啦
rm: remove write-protected regular file `regular_file'? y
rm: cannot remove `regular_file': Operation not permitted
$ chattr -i regular_file #如果想对它进行常规操作，那么可以把这个位去掉
$ rm regular_file
```

说明：chattr可以用设置文件的特殊权限，更多用法请参考chattr的帮助。

5.2.4 文件大小

普通文件是文件内容的大小，而目录作为一个特殊的文件，它存放的内容是以目录结构体组织的各类文件信息，所以目录的大小一般都是固定的，它存放的文件个数自然也就有上限，即少于它的大小除以文件名的长度。设备文件的文件大小则对应设备的主、次设备号，而有名管道文件因为特殊的读写性质，所以大小常是0。硬链接（目录文件不能创建硬链接）实质上是原文件的一个完整的拷比，因此，它的大小就是原文件的大小。而软链接只是一个inode，存放了一个指向原文件的指针，因此它的大小仅仅是原文件名的字节数。下面我们通过演示增加记忆。

范例：查看普通文件和链接文件

原文件，链接文件文件大小的示例：

```
$ echo -n "abcde" > regular_file   #往regular_file写入5字节
$ ls -l regular_file*
```

```
-rw-r-r-- 2 root root  5 2007-12-08 15:28 regular_file
-rw-r-r-- 2 root root  5 2007-12-08 15:28 regular_file_hard_file
lrwxrwxrwx 1 root root 12 2007-12-07 20:15 regular_file_soft_link -> regular_file
lrwxrwxrwx 1 root root 22 2007-12-08 15:21 regular_file_soft_link_link -> regular_file_soft_link
$ i="regular_file"
$ j="regular_file_soft_link"
$ echo ${#i} ${#j}   #可以参考，软链接存放的刚好是它们指向的原文件的文件名的字节数
12 22
```

范例：查看设备文件

设备号对应的文件大小：主、次设备号

```
$ ls -l hda1_block_dev_file
brw-r-r-- 1 root root 3, 1 2007-12-07 21:44 hda1_block_dev_file
$ ls -l null_char_dev_file
crw-r-r-- 1 root root 1, 3 2007-12-07 21:43 null_char_dev_file
```

补充：主(major)、次(minor)设备号的作用有不同。当一个设备文件被打开时，内核会根据主设备号（major number）去查找在内核中已经以主设备号注册的驱动（可以cat /proc/devices查看已经注册的驱动号和主设备号的对应情况），而次设备号（minor number）则是通过内核传递给了驱动本身（参考《The Linux Primer》第十章）。因此，对于内核而言，通过主设备号就可以找到对应的驱动去识别某个设备，而对于驱动而言，为了能够更复杂地访问设备，比如访问设备的不同部分（如硬件通过分区分成不同部分，而出现hda1,hda2,hda3等），比如产生不同要求的随机数（如/dev/random和/dev/urandom等）。

范例：查看目录

目录文件的大小，为什么是这样呢？看看下面的目录结构体的大小，目录文件的Block中存放的该目录下所有文件名的入口。

```
$ ls -ld directory_file/
drwxr-xr-x 2 root root 4096 2007-12-07 23:14 directory_file/
```

目录的结构体如下：

```
struct dirent {
    long d_ino;
    off_t d_off;
    unsigned short d_reclen;
    char d_name[NAME_MAX+1]; /* 文件名称 */
}
```

5.2.5 文件访问、更新、修改时间

文件的时间属性可以记录用户对文件的操作信息，在系统管理、判断文件版本信息等情况下将为管理员提供参考。因此，在阅读文件时，建议用cat等阅读工具，不要用编辑工具vim去阅读，因为即使你没有做任何修改操作，一旦你执行了保存命令，你将修改文件的时间戳信息。

5.2.6 文件名

文件名并没有存放在文件结构体里，而是存放在它所在的目录结构体中。所以，在目录的同一级别中，文件名必须是唯一的。

5.3 文件的基本操作

对于文件，常见的操作包括创建、删除、修改、读、写等。关于各种操作对应的“背后动作”我们将在下一章《shell编程范例之文件系统操作》详细分析。

5.3.1 范例：创建文件

socket文件是一类特殊的文件，可以通过C语言创建，这里不做介绍（暂时不知道是否可以用命令直接创建），其他文件我们将通过命令创建。

```
$ touch regular_file       #创建普通文件
$ mkdir directory_file     #创建目录文件，目录文件里头可以包含更多文件
$ ln regular_file regular_file_hard_link  #硬链接，是原文件的一个完整拷比
$ ln -s regular_file regular_file_soft_link  #类似一个文件指针，指向原文件
$ mkfifo fifo_pipe   #或者通过 "mknod fifo_pipe p" 来创建，FIFO满足先进先出的特点
$ mknod hda1_block_dev_file b 3 1  #块设备
$ mknod null_char_dev_file c 1 3   #字符设备
```

创建一个文件实际上是在文件系统中添加了一个节点（inode），该节点信息将保存到文件系统的节点表中。更形象地说，就是在一颗树上长了一颗新的叶子（文件）或者枝条（目录文件，上面还可以长叶子的那种），这些可以通过tree命令或者ls命令形象地呈现出来。文件系统从日常使用的角度，完全可以当成一颗倒立的树来看，因为它们太像了，太容易记忆啦。

```
$ tree 当前目录
```

或者

```
$ ls 当前目录
```

5.3.2 范例：删除文件

删除文件最直接的印象是这个文件再也不存在啦，这同样可以通过ls或者tree命令呈现出来，就像树木被砍掉一个分支或者摘掉一片叶子一样。实际上，这些文件删除之后，并不是立即消失了，而是仅仅做了删除标记，因此，如果删除之后，没有相关的磁盘写操作把相应的磁盘空间“覆盖”，那么原理上是可以恢复的（虽然如此，但是这样的工作往往是很麻烦的，所以在删除一些重要数据时，请务必三思而后行，比如做好备份工作），相应的做法可以参考后续资料。

具体删除文件的命令有rm，如果要删除空目录，可以用rmdir命令。例如：

```
$ rm regular_file
$ rmdir directory_file
$ rm -r directory_file_not_empty
```

rm有两个非常重要的参数，一个是-f，这个命令是非常“野蛮的”（关于**rm -rf**的那些钟爱者轻强烈建议您阅读的相应章节），它估计给很多linux user带来了痛苦，另外一个是-i，这个命令是非常“温柔的”，它估计让很多用户感觉烦躁不已过。用哪个还是根据您的“心情”吧，如果做好了充分的备份工作，或者采取了一些有效避免灾难性后果的动作的话，您在做这些工作的时候就可以放心一些啦。

5.3.3 范例：复制文件

文件的复制通常是指文件内容的“临时”复制。通过这一节开头的介绍，我们应该了解到，文件的硬链接和软链接在某种意义上说也是“文件的复制”，前者同步复制文件内容，后者在读写的情况下同步“复制”文件内容。例如：

用cp命令常规地复制文件（复制目录需要-r选项）

```
$ cp regular_file regular_file_copy
$ cp -r diretory_file directory_file_copy
```

创建硬链接(link和copy不同之处是后者是同步更新，前者则不然，复制之后两者不再相关)

```
$ ln regular_file regular_file_hard_link
```

创建软链接

```
$ ln -s regular_file regluar_file_soft_link
```

5.3.4 范例：修改文件名

修改文件名实际上仅仅修改了文件名标识符。我们可以通过mv命令来实现修改文件名操作（即重命名）。

```
$ mv regular_file regular_file_new_name
```

5.3.5 范例：编辑文件

编辑文件实际上是操作文件的内容，对应普通文本文件的编辑，这里主要涉及到文件内容的读、写、追加、删除等。这些工作通常会通过专门的编辑器来做，这类编辑器有命令行下的vim、emacs和图形界面下的gedit,kedit等。如果是一些特定的文件，会有专门的编辑和处理工具，比如图像处理软件gimp，文档编辑软件OpenOffice等。这些工具一般都会有专门的教程。关于vim的用法，建议阅读这篇帖子：VIM高级命令集锦

下面主要简单介绍Linux下通过重定向来实现文件的这些常规的编辑操作。

创建一个文件并写入abcde

```
$ echo "abcde" > new_regular_file
```

再往上面的文件中追加一行abcde

```
$ echo "abcde" >> new_regular_file
```

按行读一个文件

```
$ while read LINE; do echo $LINE; done < test.sh
```

提示：如果要把包含重定向的字符串变量当作命令来执行，请使用eval命令，否则无法解释重定向。例如，

```
$ redirect="echo \"abcde\" >test_redirect_file"
$ $redirect   #这里会把>当作字符 > 打印出来，而不会当作 重定向 解释
"abcde" >test_redirect_file
$ eval $redirect    #这样才会把 > 解释成 重定向
$ cat test_redirect_file
abcde
```

5.3.6 范例：压缩 / 解压缩文件

压缩和解压缩文件在一定意义上来说是为了方便文件内容的传输，不过也可能有一些特定的用途，比如内核和文件系统的映像文件等（更多相关的知识请参考后续资料）。

这里仅介绍几种常见的压缩和解压缩方法：

tar

```
$ tar -cf file.tar file   #压缩
$ tar -xf file.tar    #解压
```

gz

```
$ gzip  -9 file
$ gunzip file
```

tar.gz

```
$ tar -zcf file.tar.gz file
$ tar -zxf file.tar.gz
```

bz2

```
$ bzip2 file
$ bunzip2 file
```

tar.bz2

```
$ tar -jcf file.tar.bz2 file
$ tar -jxf file.tar.bz2
```

通过上面的演示，我们应该已经非常清楚tar命令，bzip2/bunzip2,gzip/gunzip命令的角色了吧？如果还不清楚，多操作和比较一些上面的命令，并查看它们的手册：man tar…

5.3.7 范例：文件搜索（文件定位）

文件搜索是指在某个目录层次中找出具有某些属性的文件在文件系统中的位置，这个位置如果扩展到整个网络，那么可以表示为一个URL地址，对于本地的地址，可以表示为file://+本地路径。本地路径在Linux系统下是以/开头，例如，每个用户的家目录可以表示为：file:///home/。下面仅仅介绍本地文件搜索的一些办法。

find命令提供了一种“及时的”搜索办法，它根据用户的请求，在指定的目录层次中遍历所有文件直到找到需要的文件为止。而updatedb+locate提供了一种“快速的”的搜索策略，updatedb更新并产生一个本地文件数据库，而locate通过文件名检索这个数据库以便快速找到相应的文件。前者支持通过各种文件属性进行搜索，并且提供了一个接口(-exec选项)用于处理搜索后的文件。因此为“单条命令”脚本的爱好者提供了极大的方便，不过对于根据文件名的搜索而言，updatedb+locate的方式在搜索效率上会有明显提高。下面简单介绍这两种方法：

find命令基本使用演示

```
$ find ./ -name "*.c" -o -name "*.h"  #找出所有的C语言文件，-o是或者
$ find ./ \( -name "*.c" -o -name "*.h" \) -exec mv '{}' ./c_files/ \;
# 把找到的文件移到c_files下，这种用法非常有趣
```

上面的用法可以用xargs命令替代

```
$ find ./ -name "*.c" -o -name "*.h" | xargs -i mv '{}' ./c_files/
# 如果要对文件做更复杂的操作，可以考虑把mv改写为你自己的处理命令，例如，我需要修
```

改所有的文件名后缀为大写。

```
$ find ./ -name "*.c" -o -name "*.h" | xargs -i ./toupper.sh '{}' ./c_files/
```

toupper.sh就是我们需要实现的转换小写为大写的一个处理文件，具体实现如下：

```
$ cat toupper.sh
#!/bin/bash

# the {} will be expended to the current line and becomen the first argument of this script
FROM=$1
BASENAME=${FROM##*/}

BASE=${BASENAME%.*}
SUFFIX=${BASENAME##*.}

TOSUFFIX="$(echo $SUFFIX | tr '[a-z]' '[A-Z]')"
TO=$2/$BASE.$TOSUFFIX
COM="mv $FROM $TO"
echo $COM
eval $COM
```

updatedb+locate基本使用演示

```
$ updatedb #更新库
$ locate find*.gz #查找包含find字符串的所有gz压缩包
```

实际上，除了上面两种命令外，Linux下还有命令查找工具：which和whereis，前者用于返回某个命令的全路径，而后者用于返回某个命令、源文件、man文件的路径。例如，我们需要查找find命令的绝对路径：

```
$ which find
/usr/bin/find
$ whereis find
find: /usr/bin/find /usr/X11R6/bin/find /usr/bin/X11/find /usr/X11/bin/find /usr/man/man1/find.1.gz
```

需要提到的是，如果想根据文件的内容搜索文件，那么find和updatedb+locate以及which,whereis都无能为例啦，可选的方法是 grep，sed等命令，前者在加上-r参数以后可以在指定目录下文件中搜索指定的文件内容，后者再使用-i参数后，可以对文件内容进行替换。它们的基本 用法在前面的章节中我们已经详细介绍了，所以这里就不叙述。

值得强调的是，这些命令对文件的操作是非常有意义的。它们在某个程度上把文件 系统结构给抽象了，使得对整个文件系统的操作简化为对单个文件的操作，而单个文件如果仅仅考虑文本部分，那么最终却转化成了我们之前的字符串操作，即我们 上一节讨论过的内容。为了更清楚的了解文件的组织结构，文件之间的关系，在下一节我们将深入探讨文件系统。

5.4 参考资料

* 从文件 I/O 看 Linux 的虚拟文件系统

* Linux 文件系统剖析
* 《Linux 核心》第九章 文件系统
* Linux Device Drivers, 3rd Edition
* 技巧：Linux I/O重定向的一些小技巧
* Intel平台下Linux中ELF文件动态链接的加载、解析及实例分析： part1, part2
* Shell脚本调试技术
* ELF文件格式及程序加载执行过程总汇
* Linux下C语言编程——文件的操作
* “Linux下C语言编程” 的 文件操作 部分
* Filesystem Hierarchy Standard
* 学会恢复 Linux系统里被删除的 Ext3文件
* 使用mc恢复被删除文件
* linux ext3误删除及恢复原理
* Linux压缩／解压缩方式大全
* Everything is a byte

5.5 后记

* 考虑到文件和文件系统的重要性，我们将把它分成三个小节来介绍：文件、文件系统、程序与进程。在文件这一部分，我们主要介绍文件的基本属性和常规操作，在 文件系统那部分，将深入探讨Linux 文件系统的各个部分（包括Linux 文件系统的结构、具体某个文件系统的大体结构分析、底层驱动的工作原理），在程序与进程一节将专门讨论可执行文件的相关内容（包括不同的程序类型、加载执 行过程、不同进程之间的交互[命令管道和无名管道、信号通信]、对进程的控制等）。
* 有必要讨论清楚 目录大小 的含义，另外，最好把一些常规的文件操作全部考虑到，包括文件的读、写、执行、删除、修改、复制、压缩／解压缩等。
* 下午刚从上海回来，比赛结果很“糟糕”，不过到现在已经不重要了，关键是通过决赛发现了很多不足，发现了设计在系统开发中的关键角色，并且发现了上海是个美丽的城市，上交也是个美丽的大学。回来就开始整理这个因为比赛落下了两周的blog。
* 12月15日，添加文件搜索部分内容。

第 6 章

文件系统操作

6.1 前言

准备了很久，找了好多天的资料，还不知道应该如何开始动笔写：因为担心拿捏不住，所以一方面继续查找资料，一方面思考如何来写。作为《shell编程范例序列》的一部分，希望它能够很好地帮助shell程序员理解如何用shell命令来完成和Linux系统关系非常之大的文件系统的各种操作，希望让Shell程序员中对文件系统“混沌”的状态从此消失，希望文件系统以一种更为清晰的样子呈现在我们的眼前。

6.2 文件系统在Linux操作系统中的位置

如何来认识文件系统呢？从shell程序员的角度来看，文件系统就是一个用来组织各种文件的方法。但是文件系统无法独立于硬件存储设备和操作系统而存在，因此我们还是有必要来弄清楚硬件存储设备、分区、操作系统、逻辑卷、文件系统等各种概念之间的联系，以便理解我们对文件系统的常规操作的一些“细节”。这个联系或许（也许会有一些问题）可以通过这样一种方式来呈现，如下图：

从该图中，我们可以清晰地看到各个“概念”之间的关系，它们以不同层次分布，覆盖硬件设备、系统内核空间、系统用户空间。在用户空间，用户可以不管内核是如何操作具体硬件设备的，仅仅使用程序员设计的各种界面就可以了，而普通程序员也仅仅需要利用内核提供的各种接口（System Call）或者一些C库来和内核进行交互，而无须关心具体的实现细节。不过对于操作系统开发人员，他们需要在内核空间设计特定的数据结构来管理和组织底层的硬件设备。

下面我们从下到上的方式（即从底层硬件开始），用工具来分析和理解图中几个重要的概念。（如果有兴趣，可以先看看下面的几则资料）

参考资料：

* 从文件 I/O 看 Linux 的虚拟文件系统
* Linux 文件系统剖析
* 第九章 文件系统
* Linux逻辑盘卷管理LVM详解

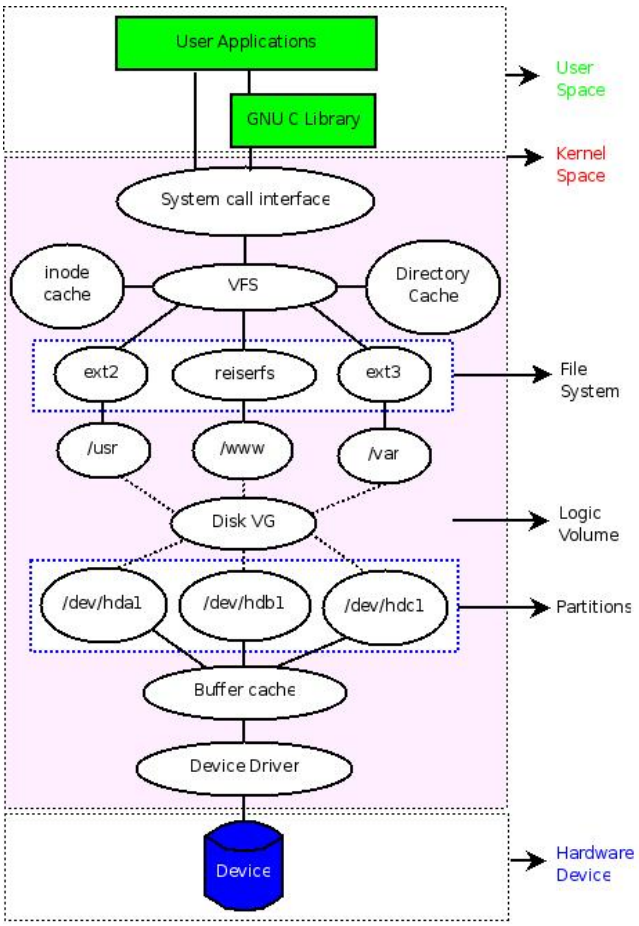

图 6.1: Linux FileSystem Architecture

6.3 硬件管理和设备驱动

Linux系统通过不同的设备驱动模块管理不同的硬件设备。如果添加了新的硬件设备，那么需要编写相应的硬件驱动模块来管理它。对于一些常见的硬件设备，系统已经自带了相应的驱动，编译内核时，选中它们，可以把它们编译成内核的一部分，也可以以模块的方式编译。如果以模块的方式编译，那么可以在系统的/lib/modules/`uname -r`目录下找到对应的模块文件。

6.3.1 范例：查找设备所需的驱动文件

比如，可以这样找到相应的scsi驱动和usb驱动的模块：
更新系统中文件索引数据库(有点慢，不耐烦就按下CTRL+C取消掉)

```
$ updatedb
```

查找scsi相关的驱动

```
$ locate scsi*.ko
```

查找usb相关的驱动

```
$ locate usb*.ko
```

这些驱动的名字以.ko为后缀，在安装系统时默认编译为了模块。实际上可以把它们编译为内核的一部分，仅仅需要在编译内核时选择为[*]即可。但是，很多情况下会以模块的方式编译它们，这样可以减少内核的大小，并根据需要灵活地加载和卸载它们。下面简单地演示如何查看已加载模块的状态，卸载模块，加载模块。

可通过查看/proc文件系统的modules文件检查内核中已加载的各个模块的状态，也可以通过lsmod命令直接查看它们。

```
$ cat /proc/modules
```

或者

```
$ lsmod
```

6.3.2 范例：查看已经加载的设备驱动

例如，查看scsi和usb相关驱动模块如下，结果各列为模块名、模块大小、被其他模块的引用情况（引用次数、引用它们的模块）

```
$ lsmod | egrep "scsi|usb"
usbhid                 29536  0
hid                    28928  1 usbhid
usbcore               138632  4 usbhid,ehci_hcd,ohci_hcd
scsi_mod              147084  4 sg,sr_mod,sd_mod,libata
```

6.3.3 范例：卸载设备驱动

下面卸载usbhid模块看看（呵呵，小心卸载scsi的驱动哦！因为你的系统就跑在上面，如果确实想玩玩，卸载前记得保存数据），通过rmmod命令就可以实现，先切换到root用户：

```
$ sudo -s
# rmmod usbhid
```

再查看该模块的信息，已经看不到了吧

```
$ lsmod | grep ^usbhid
```

6.3.4 范例：挂载设备驱动

如果你有个usb鼠标，那么移动一下，是不是发现动不了啦？因为设备驱动都没有了，设备自然就没法用罗。不过不要紧张，既然知道是什么原因，那么把设备驱动重新加载上就可以啦，下面用insmod把usbhid模块重新加载上。

```
$ sudo -s
# insmod `locate usbhid.ko`
```

locate usbhid.ko是为了找出usbhid.ko模块的路径，如果你之前没有updatedb，估计用它是找不到了，不过你可以直接到/lib/modules目录下把usbhid.ko文件找到。

okay,现在鼠标又可以用啦，不信再动一下鼠标 :-)

到这里，硬件设备和设备驱动之间关系应该是比较清楚了吧。如果没有，那么继续下面的内容。

6.3.5 范例：查看设备驱动对应的设备文件

在Linux下，设备驱动关联着相应的设备文件，而设备文件则和硬件设备一一对应。这些设备文件都统一存放在系统的/dev/目录下。

例如，scsi设备对应的/dev/sda,/dev/sda1,/dev/sda2…下面查看这些设备文件的信息。

```
$ ls -l /dev/sda*
brw-rw---- 1 root disk 8, 0 2007-12-28 22:49 /dev/sda
brw-rw---- 1 root disk 8, 1 2007-12-28 22:50 /dev/sda1
brw-rw---- 1 root disk 8, 3 2007-12-28 22:49 /dev/sda3
brw-rw---- 1 root disk 8, 4 2007-12-28 22:49 /dev/sda4
brw-rw---- 1 root disk 8, 5 2007-12-28 22:50 /dev/sda5
brw-rw---- 1 root disk 8, 6 2007-12-28 22:50 /dev/sda6
brw-rw---- 1 root disk 8, 7 2007-12-28 22:50 /dev/sda7
brw-rw---- 1 root disk 8, 8 2007-12-28 22:50 /dev/sda8
```

可以看到第一列第一个字符都是b，第五列都是数字8。b表示该文件是一个块设备文件，对应地，如果是c则表示字符设备（例如/dev/ttyS0），关于块设备和字符设备的区别，可以看这里：

> 字符设备：字符设备就是能够像字节流一样访问的设备，字符终端和串口就属于字符设备。 块设备：块设备上可以容纳文件系统。与字符设备不同，在读写操作时，块设备每次只能传输一个或多个完整的块。在Linux操作系统中，应用程序可以像访问字符设备一样读写块设备（一次读取或写入任意的字节数据）。因此，块设备和字符设备的区别仅仅是在内核中对于数据的管理不同。

数字8则是该硬件设备在内核中对应的设备编号，可以在内核的Documentation/devices.txt文件中找到设备号分配情况。但是为什么同一个设备会对应不同的设备文件（/dev/sda后面为什么还有不同的数字，而且ls结果中的第6列貌似和它们对应起来的）。这实际上是为了区分不同设备的不同部分。对于硬盘，这样可以处理硬盘内部的不同分区。就内核而言，它仅仅需要通过第5列的设备号就可以找到对应的硬件设备，但是对于驱动模块来说，它还需要知道如何处理不同的分区，于是就多了一个辅设备号，即第6列对应的内容。这样一个设备就有了主设备号（第5列）和辅设备号（第6列），从而方便的实现对各种硬件设备的管理。

因为设备文件和硬件是对应的，这样我们可以直接从/dev/sda（如果是IDE的硬盘，那么对应的设备就是/dev/hda啦）设备中读出硬盘的信息，例如：

6.3.6 范例：访问设备文件

用dd命令复制出硬盘的前512个字节，要root用户哦

```
$ sudo dd if=/dev/sda of=mbr.bin bs=512 count=1
```

用file命令查看相应的信息

```
$ file mbr.bin
mbr.bin: x86 boot sector, LInux i386 boot LOader; partition 3: ID=0x82, starthead 254, startsector 1
```

也可以用od命令以16进制的形式读取并进行分析

```
$ od -x mbr.bin
```

bs是块的大小（以字节bytes为单位），count是块数

因为这些信息并不是很直观（而且下面我们会进一步深入的分析），那么我们来看看另外一个设备文件，将可以非常直观的演示设备文件和硬件的对应关系。还是以鼠标为例吧，下面来读取鼠标对应的设备文件的信息。

```
$ sudo -s
# cat /dev/input/mouse1 | od -x
```

你的鼠标驱动可能不太一样，所以设备文件可能是其他的，但是都会在/dev/input下

移动鼠标看看，是不是发现有不同信息输出。基于这一原理，我们经常通过在一端读取设备文件/dev/ttyS0中的内容，而在另一端往设备文件/dev/ttyS0中写入内容来检查串口线是否被损坏。

到这里，对设备驱动、设备文件和硬件设备之间的关联应该是印象更深刻了。如果想深入了解设备驱动的工作原理和设备驱动模块的编写，那么看看下面列出的相关资料，开始你的设备驱动模块的编写历程吧。

参考资料:

* Compile linux kernel 2.6
* Linux系统的硬件驱动程序编写原理
* Linux下USB设备的原理、配置、 常见问题
* The Linux Kernel Module Programming Guide
* Linux设备驱动开发

6.4 理解、查看磁盘分区

实际上内存、u盘等都可以作为文件系统底层的“存储”设备，但是这里我们仅用硬盘作为实例来介绍磁盘和分区的关系。

目前Linux的分区依然采用第一台PC硬盘所使用的分区原理，下面逐步分析和演示这一分区原理。

6.4.1 磁盘分区基本原理

先来看看几个概念:

* 设备管理和分区

在Linux下，每一个存储设备对应一个系统的设备文件，对于硬盘等IDE和SCSI设备，在系统的/dev目录下可以找到对应的包含字符hd和sd的设备文件。而根据硬盘连接的主板设备接口和数据线接口的不同，在hd或者sd字符后面可以添加一个从a到z的字符，例如hda,hdb,hdc和sda,sdb,sdc等，另外为了区别同一个硬件设备的不同分区，在后面还可以添加了一个数字，例如hda1,hda2,hda3…和sda1,sda2,sda3，所以你在/dev目录下，可以看到很多类似的设备文件。

* 各分区的作用

在分区的时候常遇到主分区和逻辑分区的问题，这实际上是为了方便扩展分区，正如后面的逻辑卷的引入是为了更好地管理多个硬盘一样，引入主分区和逻辑分区可以方便地进行分区的管理。

在Linux系统中，每一个硬盘设备最多由4个主分区（包括扩展分区）构成。

主分区的作用是计算机用来进行启动操作系统的，因此每一个操作系统的启动程序或者称作是引导程序，都应该存放在主分区上。Linux规定主分区（或者扩展分区）占用分区编号中的前4个。所以你会看到主分区对应的设备文件为/dev/hda1-4或者/dev/sda1-4，而不会是hda5或者sda5。

扩展分区则是为了扩展更多的逻辑分区的，在Linux下，逻辑分区占用了hda5-16或者sda5-16等12个编号。

* 分区类型

它规定了这个分区上的文件系统的类型。Linux支持诸如msdoc,vfat,ext2,ext3等诸多的文件系统类型，更多信息在下一小节进行进一步的介绍。

6.4.2 通过分析MBR来理解分区原理

下面通过分析硬盘的前512个字节（即MBR）来分析和理解分区。

先来看看这张图:

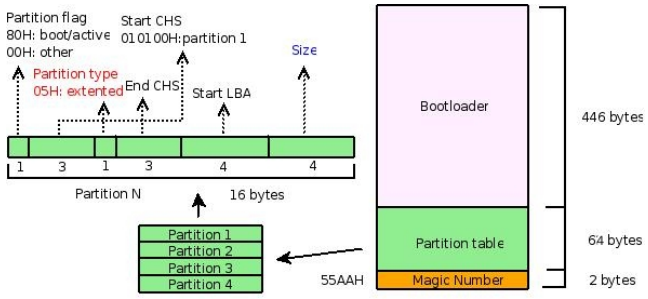

图 6.2: MBR Architecture

它用来描述MBR的结构。MBR包括引导部分、分区表、以及结束标记(55AAH)，分别占用了512字节中446字节、64字节和2字节。这里仅仅关注分区表部分，即中间的64字节以及图中左边的部分。

由于我用的是SCSI的硬盘，下面从/dev/sda设备中把硬盘的前512个字节拷贝到文件mbr.bin中。

```
$ sudo -s
# dd if=/dev/sda of=mbr.bin bs=512 count=1
```

下面用file,od,fdisk等命令来分析这段MBR的数据，并对照上图以便加深理解。

```
$ file mbr.bin
mbr.bin: x86 boot sector, LInux i386 boot LOader; partition 3: ID=0x82, starthead 254, startsector 1
$ od -x mbr.bin | tail -6   #仅关注中间的64字节，所以截取了结果中后6行
0000660 0000 0000 0000 0000 a666 a666 0000 0180
0000700 0001 fe83 ffff 003f 0000 1481 012a 0000
0000720 0000 0000 0000 0000 0000 0000 0000 fe00
0000740 ffff fe82 ffff 14c0 012a e7fa 001d fe00
0000760 ffff fe05 ffff fcba 0147 9507 0360 aa55
$ sudo -s
# fdisk -l | grep ^/  #仅分析MBR相关的部分，不分析逻辑分区部分
/dev/sda1   *           1        1216     9767488+  83  Linux
/dev/sda3            1217        1338      979965   82  Linux swap / Solaris
/dev/sda4            1339        4865    28330627+   5  Extended
```

file命令的结果显示，刚拷比的512字节是启动扇区，用分号分开的几个部分分别是bootloader，分区3和分区4。分区3的类型是82，即swap分区（可以通过fdisk命令的l命令列出相关信息），它对应fdisk的结果中/dev/sda3所在行的第5列，分区3的扇区数是

1959930，转换成字节数是1959930*512（目前，硬盘的默认扇区大小是512字节），而swap分区的默认块大小是1024字节，这样块数就是：

```
$ echo 1959930*512/1024 | bc
979965
```

正好是fdisk结果中/dev/sda3所在行的第四列对应的块数，同样地，可以对照fdisk和file的结果分析分区4。

再来看看od命令以十六进制显示的结果，同样考虑分区3，计算一下发现，分区3对应的od命令的结果为：

```
 fe00 ffff fe82 ffff 14c0 012a e7fa 001d
```

首先是分区标记，00H，从上图中，看出它就不是引导分区（80H标记的才是引导分区），而分区类型呢？为82H，和file显示结果一致，现在再来关注一下分区大小，即file结果中的扇区数。

```
$ echo "ibase=10;obase=16;1959930" | bc
1DE7FA
```

刚好对应e7fa 001d，同样地考虑引导分区的结果：

```
 0180 0001 fe83 ffff 003f 0000 1481 012a
```

分区标记：80H，正好反应了这个分区是引导分区，随后是引导分区所在的磁盘扇区情况，010100，即1面0道1扇区。其他内容可以对照分析。

考虑到时间关系，更多细节请参考下面的资料或者查看看系统的相关手册。

补充：安装系统时，可以用fdisk,cfdisk等命令进行分区。如果要想从某个分区启动，那么需要打上80H标记，例如可通过cfdisk把某个分区设置为bootable来实现。

参考资料：

* Inside the linux boot process
* Develop your own OS: booting
* Redhat 9磁盘分区简介
* Linux partition HOWTO

6.5 分区和文件系统的关系

在没有引入逻辑卷之前，分区类型和文件系统类型几乎可以同等对待，设置分区类型的过程就是格式化分区，建立相应的文件系统类型的过程。

下面主要介绍如何建立分区和文件系统类型的联系，即如何格式化分区为指定的文件系统类型。

6.5.1 常见分区类型

先来看看Linux下文件系统的常见类型（如果要查看所有Linux支持的文件类型，可以用fdisk命令的l命令查看，或者通过man fs查看，也可通过/proc/filesystems查看到当前内核支持的文件系统类型）

* ext2,ext3：这两个是Linux根文件系统通常采用的类型
* swap:这个是具体实现Linux虚拟内存时采用的一种文件系统，安装时一般需要建立一个专门的分区，并格式化为swap文件系统（如果想添加更多的swap分区，那么可以参考本节的资料[1]，熟悉dd,mkswap,swapon,swapoff等命令的用法）
* proc:这是一种比较特别的文件系统，作为内核和用户之间的一个接口存在，建立在内存中（你可以通过cat命令查看/proc系统下的文件，甚至可以通过修改/proc/sys下的文件实时调整内核的配置，当前前提是你需要把proc文件系统挂载上：`mount -t proc proc /proc`

除了这三个最常见的文件系统类型外，Linux支持包括vfat,iso,xfs,nfs在内各种常见的文件系统类型，在linux下，你可以自由地查看和操作windows等其他操作系统使用的文件系统。

那么如何建立磁盘和这些文件系统类型的关联呢？格式化。

格式化的过程实际上就是重新组织分区的过程，可通过mkfs命令来实现，当然也可以通过fdisk等命令来实现。这里仅介绍mkfs，mkfs可用来对一个已有的分区进行格式化，不能实现分区操作（如果要对一个磁盘进行分区和格式化，那么可以用fdisk就可以啦）。格式化后，相应的分区上的数据就通过某种特别的文件系统类型进行组织了。

6.5.2 范例：格式化文件系统

例如：把/dev/sda9分区格式化为ext3的文件系统。

```
$ sudo -s
# mkfs -t ext3 /dev/sda9
```

如果要列出各个分区的文件系统类型，那么可以用`fdisk -l`命令。

更多信息请参考下列资料。

参考资料：

* Linux下加载swap分区的步骤
* Linux下ISO镜像文件的制作与刻录
* RAM磁盘分区解释：1，2
* 高级文件系统实现者指南

6.6 分区、逻辑卷和文件系统的关系

在上一节中，我们直接把分区格式化为某种文件系统类型，但是考虑到扩展新的存储设备的需要，开发人员在文件系统和分区之间引入了逻辑卷。考虑到时间关系，这里不再详述，请参考资料：Linux逻辑卷管理详解

6.7 文件系统的可视化结构

文件系统最终呈现出来的是一种可视化的结构，我们可用ls,find,tree等命令把它呈现出来。它就像一颗倒挂的“树”，在树的节点上还可以挂载新的“树”。

下面简单介绍文件系统的挂载。

一个文件系统可以通过一个设备挂载（mount）到某个目录下，这个目录被称为挂载点。有趣的是，在Linux下，一个目录本身还可以挂载到另外一个目录下，一个格式化了的

文件也可以通过一个特殊的设备/dev/loop进行挂载（如iso文件）。另外，就文件系统而言，Linux不仅支持本地文件系统，还支持远程文件系统（如nfs）。

6.7.1 范例：挂载文件系统

下面简单介绍文件系统挂载的几个实例。

* 根文件系统的挂载

挂载需要root权限，先切换到root用户，例如，挂载系统根文件系统/dev/sda1到一个新的目录下

```
$ sudo -s
# mount -t ext3 /dev/sda1 /mnt/
```

查看/dev/sda1的挂载情况，可以看到，一个设备可以多次挂载

```
$ mount | grep sda1
/dev/sda1 on / type ext3 (rw,errors=remount-ro)
/dev/sda1 on /mnt type ext3 (rw)
```

对于一个已经挂载的文件系统，为支持不同的属性可以重新挂载

```
$ mount -n -o remount, rw /
```

* 挂载一个新增设备

如果内核已经支持了USB接口，那么在插入u盘的时候，我们可以通过dmesg命令查看它对应的设备号，并挂载它。

查看dmesg结果中的最后几行内容，找到类似/dev/sdN的信息，找出u盘对应的设备号

```
$ dmesg
```

这里假设u盘是vfat格式的，以便在一些打印店里的windows上也可使用

```
# mount -t vfat /dev/sdN /path/to/mountpoint_directory
```

* 挂载一个iso文件或者是光盘

对于一些iso文件或者是iso格式的光盘，同样可以通过mount命令挂载。

对于iso文件：

```
# mount -t iso9660 /path/to/isofile /path/to/mountpoint_directory
```

对于光盘：

```
# mount -t iso9660 /dev/cdrom /path/to/mountpoint_directory
```

* 挂载一个远程文件系统

```
# mount -t nfs remote_ip:/path/to/share_directory /path/to/local_directory
```

* 挂载一个proc文件系统

```
# mount -t proc proc /proc
```

proc文件系统组织在内存中，但是你可以把它挂载到某个目录下。通常把它挂载在/proc目录下，以便一些系统管理和配置工具使用它。例如top命令用它分析内存的使用情况（读取/proc/meminfo和/proc/stat等文件中的内容），lsmod命令通过它获取内核模块的状态（读取/proc/modules），netstat命令通过它获取网络的状态（读取/proc/net/dev等文件），当然，你也可以编写自己的相关工具。除此之外，通过调整/proc/sys目录下的文件，你可以动态的调整系统的配置，比如通过往/proc/sys/net/ipv4/ip_forward文件中写入数字1就可以让内核支持数据包的转发。（更多信息请参考proc的帮助，man proc）

* 挂载一个目录

```
$ mount --bind /path/to/needtomount_directory /path/to/mountpoint_directory
```

这个非常有意思，比如你可以把某个目录挂载到ftp服务的根目录下，而无须把内容复制过去，就可以把相应目录中的资源提供给别人共享。

6.7.2 范例：卸载某个分区

以上都只提到了挂载，那怎么卸载呢？用umount命令跟上挂载的源地址或者挂载点（设备，文件，远程目录等）就可以。例如：

```
$ umount /path/to/mountpoint_directory
```

或者

```
$ umount /path/to/mount_source
```

如果想管理大量的或者经常性的挂载服务，那么每次手动挂载是很糟糕的事情。这个时候就可以利用mount的配置文件/etc/fstab，把mount对应的参数写到/etc/fstab文件对应的列中即可实现批量挂载(**mount -a**)和卸载(**umount -a**)。/etc/fstab中各列分别为文件系统、挂载点、类型、相关选项。更多信息可参考fstab的帮助(**man fstab**)。

参考资料：

* Linux硬盘分区以及其挂载原理
* 从文件I/O看linux的虚拟文件系统
* 用Graphviz进行可视化操作——绘制函数调用关系图

6.8 如何制作一个文件系统

Linux的文件系统下有一些最基本的目录，不同的目录下存放着不同作用的各类文件。最基本的目录有/etc，/lib，/dev，/bin等，它们分别存放着系统配置文件，库文件，设备文件和可执行程序。这些目录一般情况下是必须的，在做嵌入式开发的时候，我们需要手动或者是用busybox等工具来创建这样一个基本的文件系统。这里我们仅制作一个非常简单的文件系统，并对该文件系统进行各种常规的操作，以便加深对文件系统的理解。

6.8.1 范例：用dd创建一个固定大小的文件

还记得dd命令么？我们就用它来产生一个固定大小的文件，这个为1M(1024*1024 bytes)的文件

```
$ dd if=/dev/zero of=minifs bs=1024 count=1024
```

查看文件类型，这里的minifs是一个充满\0的文件，没有任何特定的数据结构

```
$ file minifs
minifs: data
```

说明：/dev/zero是一个非常特殊的设备，如果读取它，可以获取任意多个\0。

接着把该文件格式化为某个指定文件类型的文件系统。（是不是觉得不可思议，文件也可以格式化？是的，不光是设备可以，文件也可以以某种文件系统类型进行组织，但是需要注意的是，某些文件系统（如ext3）要求被格式化的目标最少有64M的空间）。

6.8.2 范例：用mkfs格式化文件

```
$ mkfs.ext2 minifs
```

查看此时的文件类型，这个时候文件minifs就以ext2文件系统的格式组织了

```
$ file minifs
minifs: Linux rev 1.0 ext2 filesystem data
```

6.8.3 范例：挂载刚创建的文件系统

因为该文件以文件系统的类型组织了，那么可以用mount命令挂载并使用它。

请切换到root用户挂载它，并通过-o loop选项把它关联到一个特殊设备/dev/loop

```
$ sudo -s
# mount minifs /mnt/ -o loop
```

查看该文件系统的信息，仅可以看到一个目录文件lost+found

```
$ ls /mnt/
lost+found
```

6.8.4 范例：对文件系统进行读、写、删除等操作

在该文件系统下进行各种常规操作，包括读、写、删除等。（每次操作前先把minifs文件保存一份，以便比较，结合相关资料就可以深入地分析各种操作对文件系统的改变情况，从而深入理解文件系统作为一种组织数据的方式的实现原理等）

```
$ cp minifs minifs.bak
$ cd /mnt
$ touch hello
$ cd -
$ cp minifs minifs-touch.bak
$ od -x minifs.bak > orig.od
$ od -x minifs-touch.bak > touch.od
```

创建一个文件后，比较此时文件系统和之前文件系统的异同

```
$ diff orig.od touch.od
diff orig.od touch.od
61,63c61,64
< 0060020 000c 0202 2e2e 0000 000b 0000 03e8 020a
< 0060040 6f6c 7473 662b 756f 646e 0000 0000 0000
< 0060060 0000 0000 0000 0000 0000 0000 0000 0000
---
> 0060020 000c 0202 2e2e 0000 000b 0000 0014 020a
> 0060040 6f6c 7473 662b 756f 646e 0000 000c 0000
> 0060060 03d4 0105 6568 6c6c 006f 0000 0000 0000
> 0060100 0000 0000 0000 0000 0000 0000 0000 0000
```

通过比较发现：添加一个文件后，文件系统的相应位置发生了明显的变化

```
$ echo "hello, world" > /mnt/hello
```

执行sync命令，确保缓存中的数据已经写入磁盘（还记得附图[1]的buffer cache吧，这里就是把cache中的数据写到磁盘中）

```
$ sync
$ cp minifs minifs-echo.bak
$ od -x minifs-echo.bak > echo.od
```

写入文件内容后，比较文件系统和之前的异同

```
$ diff touch.od echo.od
```

查看文件系统中的字符串

```
$ strings minifs
lost+found
hello
hello, world
```

删除hello文件，查看文件系统变化

```
$ rm /mnt/hello
$ cp minifs minifs-rm.bak
$ od -x minifs-rm.bak > rm.od
$ diff echo.od rm.od
```

通过查看文件系统的字符串们发现：删除文件时并没有覆盖文件的内容，所以从理论上说内容此时还是可恢复的

```
$ strings minifs
lost+found
hello
hello, world
```

上面仅仅演示了一些分析文件系统的常用工具，并分析了几个常规的操作，如果你想非常深入地理解文件系统的实现原理，请熟悉使用上述工具并阅读相关资料。

参考资料：

* Build a mini filesystem in linux from scratch
* Build a mini filesystem in linux with BusyBox
* ext2 文件系统

6.9 如何开发自己的文件系统

随着fuse的出现，在用户空间开发文件系统成为可能，如果想开发自己的文件系统，那么阅读：使用fuse开发自己的文件系统。

6.10 后记

* 2007年12月22日，收集了很多资料，写了整体的框架。
* 2007年12月28日下午，完成初稿，考虑到时间关系，很多细节也没有进一步分析，另外有些部分可能存在理解上的问题，欢迎批评指正。
* 2007年12月28日晚，修改部分资料，并正式公开该篇文档。
* 29号，添加设备驱动和硬件设备一小节。

第 7 章

进程操作

7.1 前言

这一小节写了很久，到现在才写完。本来关注的内容比较多，包括程序开发过程的细节、ELF格式的分析、进程的内存映像等，后来搞得“雪球越滚越大”，甚至脱离了shell编程关注的内容。

所以呢，想了个小办法，“大事化小，小事化了”，把涉及到的内容分成如下几个部分：

* 《把VIM打造成源代码编辑器》（源代码编辑过程：用VIM编辑代码的一些技巧）
* 《GCC编译的背后》： 第一部分：《预处理和编译》 第二部分：《汇编和链接》（编译过程：预处理、编译、汇编、链接）
* 《程序执行的那一刹那 》（执行过程：当我们从命令行输入一个命令之后）
* 《进程的内存映像》 （进程加载过程：程序在内存里是个什么样子）
* 《动态符号链接的细节》（动态链接过程：函数puts/printf的地址在哪里）
* 《代码测试、调试与优化小结》（程序开发过后：内存溢出了吗？有缓冲区溢出？代码覆盖率如何测试呢？怎么调试汇编代码？有哪些代码优化技巧和方法呢？）
* 《为你的可执行文件“减肥”》（从“减肥”的角度一层一层剖开ELF文件）
* 《进程和进程的基本操作》（本章）

呵呵，好多。终于可以一部分一部分地完成了，不会再有一种对着一个大蛋糕却不知道如何下口的尴尬了。

进程作为程序真正发挥作用时的“形态”，我们有必要对它的一些相关操作非常熟悉，这一节主要描述进程相关的概念和操作，将介绍包括程序、进程、作业等基本概念以及进程状态查询、进程通信等相关的基本操作等。

说明：上面的整个序列都已经编写完成，在网络中都可以搜索到，作者计划在近期整理成册，陆续更新到TinyLab.org。

7.2 什么是程序，什么又是进程

程序是指令的集合，而进程则是程序执行的基本单元。为了让程序完成它的工作，我们必须让程序运行起来成为进程，进而利用处理器资源，内存资源，进行各种I/O操作，从而完成某项指定的工作。

在这个意思上说，程序是静态的，而进程则是动态的。

而进程有区别于程序的地方还有，进程除了包含程序文件中的指令数据以外，还需要在内核中有一个数据结构用以存放特定进程的相关属性，以便内核更好的管理和调度进程，从而完成多进程协作的任务。因此，从这个意义上可以说“高于”程序，超出了程序指令本身。

如果进行过多进程程序的开发，你又会发现，一个程序可能创建多个进程，通过多个进程的交互完成任务。在Linux下，多进程的创建通常是通过fork系统调用实现的。从这个意义上来说程序则”包含”了进程。

另外一个需要明确的是，程序可以由多种不同的程序语言描述，包括C语言程序、汇编语言程序和最后编译产生的机器指令等。

下面我们简单讨论一下Linux下面如何通过shell进行进程的相关操作。

7.3 进程的创建

通常在命令行键入某个程序文件名以后，一个进程就被创建了。例如，

7.3.1 范例：让程序在后台运行

```
$ sleep 100 &
[1] 9298
```

7.3.2 范例：查看进程ID

用pidof可以查看指定程序名的进程ID

```
$ pidof sleep
9298
```

7.3.3 范例：查看进程的内存映像

```
$ cat /proc/9298/maps
08048000-0804b000 r-xp 00000000 08:01 977399     /bin/sleep
0804b000-0804c000 rw-p 00003000 08:01 977399     /bin/sleep
0804c000-0806d000 rw-p 0804c000 00:00 0          [heap]
b7c8b000-b7cca000 r-p 00000000 08:01 443354
...
bfbd8000-bfbed000 rw-p bfbd8000 00:00 0          [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0          [vdso]
```

当一个程序被执行以后，程序被加载到内存中，成为了一个进程。上面显示了该进程的内存映像(虚拟内存)，包括程序指令、数据，以及一些用于存放程命令行参数、环境变量的栈空间，用于动态内存申请的堆空间都被分配好了。

关于程序在命令行执行过程的细节，请参考《Linux命令行下程序执行的那一刹那》。

实际上，创建一个进程，也就是说让程序运行，还有其他的办法，比如，通过一些配置让系统启动时自动启动我们的程序（具体参考man init），或者是通过配置crond（或者

at）让它定时启动我们的程序。除此之外，还有一个方式，那就是编写shell脚本，把程序写入一个脚本文件，当执行脚本文件时，文件中的程序将被执行而成为进程。这些方式的细节就不介绍了，下面介绍如何查看进程的属性。

需要补充一点的是，在命令行下执行程序时，我们可以通过ulimit内置命令来设置进程可以利用的资源，比如进程可以打开的最大文件描述符个数，最大的栈空间，虚拟内存空间等。具体用法见`help ulimit`。

7.4 查看进程的属性和状态

我们可以通过ps命令查看进程的相关属性和状态，这些信息包括进程所属用户，进程对应的程序，进程对cpu和内存的使用情况等信息。熟悉如何查看它们有助于我们进行相关的统计分析和进一步的操作。

7.4.1 范例：通过ps命令查看进程属性

查看系统所有当前所有进程的属性

```
$ ps -ef
```

查看命令中包含某个指定字符的程序对应的进程，进程ID是1，TTY为？表示和终端没有关联

```
$ ps -C init
  PID TTY          TIME CMD
    1 ?        00:00:01 init
```

选择某个特定用户启动的进程

```
$ ps -U falcon
```

可以按照指定格式输出指定内容，这里会输出命令名和cpu使用率

```
$ ps -e -o "%C %c"
```

这样则会打印cpu使用率最高的前4个程序

```
$ ps -e -o "%C %c" | sort -u -k1 -r | head -5
 7.5 firefox-bin
 1.1 Xorg
 0.8 scim-panel-gtk
 0.2 scim-bridge
```

使用虚拟内存最大的5个进程

```
$ ps -e -o "%z %c" | sort -n -k1 -r | head -5
349588 firefox-bin
 96612 xfce4-terminal
 88840 xfdesktop
 76332 gedit
 58920 scim-panel-gtk
```

7.4.2 范例：通过pstree查看进程亲缘关系

由于系统所有进程之间都有“亲缘”关系，所以可以通过pstree查看这种关系，

```
$ pstree
```

打印系统进程调用树，可以非常清楚地看到当前系统中所有活动进程之间的调用关系

7.4.3 范例：用top动态查看进程信息

```
$ top
```

该命令最大的特点是可以动态地查看进程的信息，当然，它还提供了一些有用的参数，比如-S可以按照累计执行时间的大小排序查看，也可以通过-u查看指定用户启动的进程等。

补充：top命令支持交互式，比如它支持u命令显示用户的所有进程，支持通过k命令杀掉某个进程；如果使用-n 1选项可以采用它的批处理模式，具体用法为：

```
$ top -n 1 -b
```

7.4.4 范例：确保特定程序只有一个副本在运行

感觉有上面几个命令来查看进程的信息就差不多了，下面来讨论一个有趣的问题：如何让一个程序在同一时间只有一个在运行。

这意味着当一个程序正在被执行时，它将不能再被启动。那该怎么做呢?

假如一份相同的程序被复制成了很多份，并且具有不同的文件名被放在不同的位置，这个将比较糟糕，所以我们考虑最简单的情况，那就是这份程序在整个系统上是唯一的，而且名字也是唯一的。这样的话，我们有哪些办法来回答上面的问题呢?

总的机理是：在这个程序的开头检查自己有没有执行，如果执行了则停止否则继续执行后续代码。 策略则是多样的，由于前面的假设已经保证程序文件名和代码的唯一性，所以通过ps命令打印找出当前的所有进程对应的程序名，逐个与自己的程序名比较，如果已经有，那么说明自己已经运行了。

```
ps -e -o "%c" | tr -d " " | grep -q ^init$ # #查看当前程序是否执行
[ $? -eq 0 ] && exit  #如果在，那么退出，$?表示上一条指令是否执行成功
```

每次运行时先在指定位置检查是否存在一个保存自己进程ID的文件，如果不存在，那么继续执行，如果存在，那么查看该进程ID是否正在运行，如果在，那么退出，否则往该文件重新写的新的进程ID，并继续。

```
pidfile=/tmp/$0".pid"
if [ -f $pidfile ]; then
       OLDPID=$(cat $pidfile)
    ps -e -o "%p" | tr -d " " | grep -q "^$OLDPID$"
    [ $? -eq 0 ] && exit
fi

echo $$ > $pidfile

#... 代码主体

#设置信号0的动作，当程序退出时触发该信号从而删除掉临时文件
trap "rm $pidfile"□□□□□ 0
```

更多实现策略自己尽情的发挥吧！

7.5 调整进程的优先级

在保证每个进程都能够顺利执行外，为了让某些任务优先完成，那么系统在进行进程调度时就会采用一定的调度办法，比如常见的有按照优先级的时间片轮转的调度算法。这种情况下，我们可以通过renice调整正在运行的程序的优先级，例如，

7.5.1 范例：获取进程的优先级

```
$ ps -e -o "%p %c %n" | grep xfs
 5089 xfs              0
```

7.5.2 范例：调整进程的优先级

```
$ renice 1 -p 5089
renice: 5089: setpriority: Operation not permitted
$ sudo renice 1 -p 5089   #需要权限才行
[sudo] password for falcon:
5089: old priority 0, new priority 1
$ ps -e -o "%p %c %n" | grep xfs  #再看看，优先级已经被调整过来了
 5089 xfs              1
```

7.6 结束进程

既然可以通过命令行执行程序，创建进程，那么也有办法结束它。我们可以通过kill命令给用户自己启动的进程发送一定信号让进程终止，当然“万能”的root几乎可以kill所有进程（除了init之外）。例如，

7.6.1 范例：结束进程

```
$ sleep 50 &   #启动一个进程
[1] 11347
$ kill 11347
```

kill命令默认会发送终止信号(SIGTERM)给程序，让程序退出，但是kill还可以发送其他的信号，这些信号的定义我们可以通过**man 7 signal**查看到，也可以通过**kill -l**列出来。

```
$ man 7 signal
$ 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
```

7.6.2 范例：暂停某个进程

例如，我们用kill命令发送SIGSTOP信号给某个程序，让它暂停，然后发送SIGCONT信号让它继续运行。

```
$ sleep 50 &
[1] 11441
$ jobs
[1]+  Running                 sleep 50 &
```

```
$ kill -s SIGSTOP 11441   #这个等同于我们对一个前台进程执行CTRL+Z操作
$ jobs
[1]+  Stopped                 sleep 50
$ kill -s SIGCONT 11441   #这个等同于之前我们使用bg %1操作让一个后台进程运行起来
$ jobs
[1]+  Running                 sleep 50 &
$ kill %1                  #在当前会话(session)下，也可以通过作业号控制进程
$ jobs
[1]+  Terminated              sleep 50
```

可见kill命令为我们提供了非常好的功能，不过kill命令只能根据进程的ID或者作业来控制进程，所以pkill和killall给我们提供了更多选择，它们扩展了通过程序名甚至是进程的用户名来控制进程的方法。更多用法请参考它们的手册。

7.6.3 范例：查看进程退出状态

当一个程序退出以后，如何判断这个程序是正常退出还是异常退出呢？还记得Linux下，那个经典“hello,world”程序吗？在代码的最后总是有条**return 0**语句。这个**return 0**实际上是为了让程序员来检查进程是否正常退出的。如果进程返回了一个其他的数值，那么我们可以肯定的说这个进程异常退出了，因为它都没有执行到**return 0**这条语句就退出了。

那怎么检查进程退出的状态，即那个返回的数值呢？

在shell程序中，我们可以检查这个特殊的变量$?，它存放了上一条命令执行后的退出状态。

```
$ test1
bash: test1: command not found
$ echo $?
127
$ cat ./test.c | grep hello
$ echo $?
1
$ cat ./test.c | grep hi
    printf("hi, myself!\n");
$ echo $?
0
```

貌似返回0成为了一个潜规则，虽然没有标准明确规定，不过当程序正常返回时，我们总是可以从**$?**中检测到0，但是异常时，我们总是检测到一个非0的值。这就告诉我们在程序的最后我们最好是跟上一个**exit 0**以便任何人都可以通过检测$?确定你的程序是否正常结束。如果有一天，有人偶尔用到你的程序，试图检查你的程序的退出状态，而你却在程序的末尾莫名的返回了一个-1或者1，那么他将会很苦恼，会怀疑自己的程序哪个地方出了问题，检查半天却不知所措，因为他太信任你了，竟然从头至尾都没有怀疑你的编程习惯可能会与众不同。

7.7 进程通信

为了便于设计和实现，通常一个大型的任务都被划分成较小的模块。不同模块之间启动后成为进程，它们之间如何通信以便交互数据，协同工作呢？在《UNIX环境高级编程》一书中提到很多方法，诸如管道（无名管道和有名管道）、信号（signal）、报文（Message）队列（消息队列）、共享内存（mmap/munmap）、信号量(semaphore，主要是同步用，进程之间，进程的不同线程之间)、套接口(Socket，支持不同机器之间的进程通信）等，而在shell编程里头，我们通常直接用到的就有管道和信号等。下面主要介绍管道和信号机制在shell编程时候的一些用法。

7.7.1 范例：无名管道(pipe)

在Linux下，你可以通过|连接两个程序，这样就可以用它来连接后一个程序的输入和前一个程序的输出，因此被形象地叫做个管道。在C语言里头，创建无名管道非常简单方便，用pipe函数，传入一个具有两个元素的int型的数组就可以。这个数组实际上保存的是两个文件描述符，父进程往第一个文件描述符里头写入东西后，子进程可以从第一个文件描述符中读出来。

如果用多了命令行，这个管子|应该会经常用。比如我们在上面的演示中把ps命令的输出作为grep命令的输入，从而可以过滤掉一些我们感兴趣的信息：

```
$ ps -ef | grep init
```

也许你会觉得这个“管子”好有魔法，竟然真地能够链接两个程序的输入和输出，它们到底是怎么实现的呢？实际上当我们输入这样一组命令的时候，当前解释程序会进行适当的解析，把前面一个进程的输出关联到管道的输出文件描述符，把后面一个进程的输入关联到管道的输入文件描述符，这个关联过程通过输入输出重定向函数dup(或者fcntl)来实现。

7.7.2 范例：有名管道(named pipe)

有名管道实际上是一个文件（无名管道也像一个文件，虽然关系到两个文件描述符，不过只能一边读另外一边写），不过这个文件比较特别，操作时要满足先进先出，而且，如果试图读一个没有内容的有名管道，那么就会被阻塞，同样地，如果试图往一个有名管道里头写东西，而当前没有程序试图读它，也会被阻塞。下面看看效果。

```
$ mkfifo fifo_test     #通过mkfifo命令可以创建一个有名管道
$ echo "fewfefe" > fifo_test     #试图往fifo_test文件中写入内容，但是被阻塞，要另开一个终端继续下面的
$ cat fifo_test         #另开一个终端，记得，另开一个。试图读出fifo_test的内容
fewfefe
```

在这里echo和cat是两个不同的程序，在这种情况下，通过echo和cat启动的两个进程之间并没有父子关系。不过它们依然可以通过有名管道通信。这样一种通信方式非常适合某些情况：例如有这样一个架构，这个架构由两个应用程序构成，其中一个通过一个循环不断读取fifo_test中的内容，以便判断，它下一步要做什么。如果这个管道没有内容，那么它就会被阻塞在那里，而不会死循环而耗费资源，另外一个则作为一个控制程序不断地往fifo_test中写入一些控制信息，以便告诉之前的那个程序该做什么。下面写一个非常简单的例子。我们可以设计一些控制码，然控制程序不断的往fifo_test里头写入，然后应用程

序根据这些控制码完成不同的动作。当然，也可以往fifo_test传入除控制码外的不同的数据。

应用程序的代码：

```
$ cat app.sh
#!/bin/bash

FIFO=fifo_test
while :;
do
    CI=`cat $FIFO`  #CI --> Control Info
    case $CI in
        0) echo "The CONTROL number is ZERO, do something ..."
            ;;
        1) echo "The CONTROL number is ONE, do something ..."
            ;;
        *) echo "The CONTROL number not recognized, do something else..."
            ;;
    esac
done
```

控制程序的代码：

```
$ cat control.sh
#!/bin/bash

FIFO=fifo_test
CI=$1

[ -z "$CI" ] && echo "the control info should not be empty" && exit

echo $CI > $FIFO
```

一个程序通过管道控制另外一个程序的工作：

```
$ chmod +x app.sh control.sh    #修改这两个程序的可执行权限，以便用户可以执行它们
$ ./app.sh            #在一个终端启动这个应用程序，在通过./control.sh发送控制码以后查看输出
The CONTROL number is ONE, do something ...    #发送1以后
The CONTROL number is ZERO, do something ...    #发送0以后
The CONTROL number not recognized, do something else...    #发送一个未知的控制码以后
$ ./control.sh 1            #在另外一个终端，发送控制信息，控制应用程序的工作
$ ./control.sh 0
$ ./control.sh 4343
```

这样一种应用架构非常适合本地的多程序任务的设计，如果结合web cgi，那么也将适合远程控制的要求。引入web cgi的唯一改变是，要把控制程序./control.sh放到web的cgi

目录下，并对它作一些修改，以使它符合CGI的规范，这些规范包括文档输出格式的表示(在文件开头需要输出**content-tpye: text/html**以及一个空白行)和输入参数的获取(web输入参数都存放在QUERY_STRING环境变量里头)。因此一个非常简单的CGI形式控制程序将类似下面。

```
#!/bin/bash

FIFO=./fifo_test
CI=$QUERY_STRING

[ -z "$CI" ] && echo "the control info should not be empty" && exit

echo -e "content-type: text/html\n\n"
echo $CI > $FIFO
```

在实际使用的时候，请确保control.sh能够访问到fifo_test管道，并且有写权限。这样我们在浏览器上就可以这样控制app.sh了。

http://ipaddress_or_dns/cgi-bin/control.sh?0

问号(?)后面的内容即QUERY_STRING，类似之前的**$1**。

这样一种应用对于远程控制，特别是嵌入式系统的远程控制很有实际意义。在去年的暑期课程上，我们就通过这样一种方式来实现马达的远程控制。首先，我们实现了一个简单的应用程序以便控制马达的转动，包括转速，方向等的控制。为了实现远程控制，我们设计了一些控制码，以便控制马达转动相关的不同属性。

在C语言中，如果要用有名管道，和shell下的类似，只不过在读写数据的时候用read,write调用，在创建fifo的时候用mkfifo函数调用。

7.7.3 范例：信号(Signal)

信号是软件中断，在Linux下面用户可以通过kill命令给某个进程发送一个特定的信号，也可以通过键盘发送一些信号，比如CTRL+C可能触发CGIINT信号，而CTRL+\可能触发SGIQUIT信号等，除此之外，内核在某些情况下也会给进程发送信号，比如在访问内存越界时产生SGISEGV信号，当然，进程本身也可以通过kill,raise等函数给自己发送信号。对于Linux下支持的信号类型，大家可以通过**man 7 signal**或者**kill -l**查看到相关列表和说明。

对于有些信号，进程会有默认的响应动作，而有些信号，进程可能直接会忽略，当然，用户还可以对某些信号设定专门的处理函数。在shell程序中，我们可以通过trap命令（shell的内置命令）来设定响应某个信号的动作(某个命令或者是你定义的某个函数)，而在C语言里头可以通过signal调用注册某个信号的处理函数。这里仅仅演示trap命令的用法。

```
$ function signal_handler {    #定一个signal_handler的函数,>是按下换行符号自动出现的
> echo "hello, world"
> }
$ trap signal_handle SIGINT    #执行该命令设定：当发生SIGINT信号时将打印hello。
$ hello, world            #按下CTRL+C，可以看到屏幕上输出了hello, world字符串
```

类似地，如果设定信号0的响应动作，那么就可以用trap来模拟C语言程序中的atexit程序终止函数的登记，即通过**trap signal_handler SIGQUIT**设定的signal_handler函数将在程序退出的时候被执行。信号0是一个特别的信号，在POSIX.1中把信号编号0定义为空信号，这将常被用来确定一个特定进程是否仍旧存在。当一个程序退出时会触发该信号。

```
$ cat sigexit.sh
#!/bin/bash

function signal_handler {
    echo "hello, world"
}
trap signal_handler 0
$ chmod +x sigexit.sh
$ ./sigexit.sh    #实际上在shell编程时，会用这种方式在程序退出时来做一些清理临时文件的收尾工作
hello, world
```

7.8 作业和作业控制

当我们为完成一些复杂的任务而将多个命令通过|,\>,<, ;, (,)等组合在一起的时候，通常这样个命令序列会启动多个进程，它们之间通过管道等进行通信。而有些时候，我们在执行一个任务的同时，还有其他的任务需要处理，那么就经常会在命令序列的最后加上一个&，或者在执行命令以后，按下CTRL+Z让前一个命令暂停。以便做其他的任务。等做完其他一些任务以后，再通过fg命令把后台的任务切换到前台。这样一种控制过程通常被成为作业控制，而那些命令序列则被成为作业，这个作业可能涉及一个或者多个程序，一个或者多个进程。下面演示一下几个常用的作业控制操作。

7.8.1 范例：创建后台进程，获取进程的作业号和进程号

```
$ sleep 50 &
[1] 11137
```

7.8.2 范例：把作业调到前台并暂停

使用shell内置命令fg把作业1调到前台运行，然后按下CTRL+Z让该进程暂停

```
$ fg %1
sleep 50
^Z
[1]+  Stopped                 sleep 50
```

7.8.3 范例：查看当前作业情况

```
$ jobs            #查看当前作业情况，有一个作业停止
[1]+  Stopped                 sleep 50
$ sleep 100 &       #让另外一个作业在后台运行
[2] 11138
$ jobs            #查看当前作业情况，一个正在运行，一个停止
[1]+  Stopped                 sleep 50
[2]-  Running                 sleep 100 &
```

7.8.4 范例：启动停止的进程并运行在后台

```
$ bg %1
[2]+ sleep 50 &
```

不过，要在命令行下使用作业控制，需要当前shell，内核终端驱动等对作业控制支持才行。

7.9 参考资料

* 《UNIX环境高级编程》

第 8 章

网络操作

8.1 前言

之前已经介绍了shell编程范例之数值、布尔值、字符串、文件、文件系统、进程等的操作。这些内容基本覆盖了网络中某个独立机器正常工作的“方方面面”，现在需要把视角从单一的机器延伸到这些机器通过各种网络设备和协议连接起来的网络世界，分析网络拓扑结构、网络工作原理、了解各种常见网络协议、各种常见硬件工作原理、网络通信与安全相关软件以及工作原理分析等。

不过网络相关的问题确实太复杂了，这里不可能介绍具体，因此如果想了解更多的细节，还是建议参考相关的资料，例如后面的参考资料。但是Linux是一个网络原理学习和实践的好平台，不仅因为它本身对网络体系结构的实现是开放源代码的，而且各种相关的分析工具和函数库数不胜数，因此，如果你是学生，千万不要错过通过它来做相关的实践工作。

8.2 网络原理介绍

8.2.1 我们的网络世界

在进行所有的介绍之前，来让我们直观地感受一下那个真真实实存在的网络世界吧。当我在Linux下通过WEB编辑器写这篇blog时，一边用mplayer听着远程音乐，等累了时则打开兰大的网络TV频道开始看看凤凰卫视……这些“现代化”的生活，我想，如果没有网络，将变得无法想象。

下面来构想一下这样一个网络世界的优美图画：

我一边盯着显示器，一边敲击着键盘，一边挂着耳机。

我的主机电源灯灿烂得很，发着绿光，这个时候我很容易想象主机背后的那个网卡位置肯定有两个不同颜色的灯光在闪烁，这个告诉我，它正在与计算机网络世界打着交道。

就在实验室的某个角落，有一个交换机上的一个网口的网线连到我的主机上，这个交换机接到了一个局域网的网关上，然后这个网关再接到了信息楼的某个路由器上，再转接到学校网络中心的另外一个路由器上……

期间，有一个路由器连接到了这个blog的服务器上，而另外一个则可能连到了那个网络TV服务器上，还有呢，另外一些则连接到了电信网络里头的某个音乐服务器上……

下面用dia绘制一个简单的“网络地图”：

在这个图中，把一些最常见的网络设备和网络服务基本都呈现出来了，包括本地主机、路由、交换机、网桥，域名服务器，万维网服务，视频服务，防火墙服务，动态IP地址服务

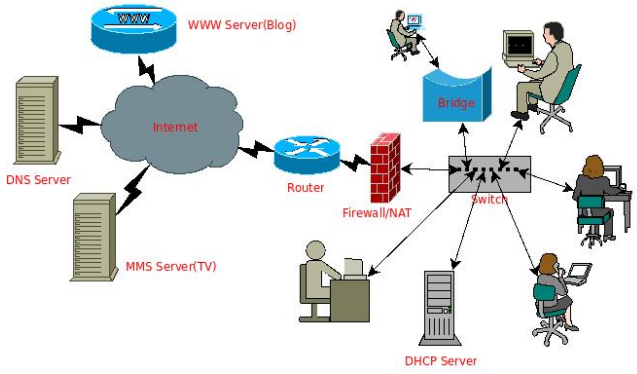

图 8.1: Network Architecture

等。其中各种设备构成了整个物理网络，而网络服务则是构建在这些设备上的各种网络应用。

现在的网络应用越来越丰富多样，比如即时聊天（IM）、p2p资源共享、网络搜索等，它们是如何实现的，它们如何构建在各种各样的网络设备之上，并且能够安全有效的工作呢？这取决于这背后逐步完善的网络体系结构和各种相关网络协议的开发、实现和应用。

8.2.2 网络体系结构和网络协议介绍

那么网络体系结构是怎么样的呢？涉及到哪些相关的网络协议呢？什么又是网络协议呢？

在《计算机网络——自顶向下的方法》一书中非常巧妙地给出了网络体系结构分层的比喻，把网络中各层跟交通运输体系中的各个环节对照起来，让人更通俗易懂。在交通运输体系中，运输的是人和物品，在计算机网络体系中，运输的是电子数据。考虑到交通运输网络和计算机网络中最终都可以划归为点对点的信息传输。这里考虑两点之间的信息传递过程，得到这样一个对照关系，见下图：

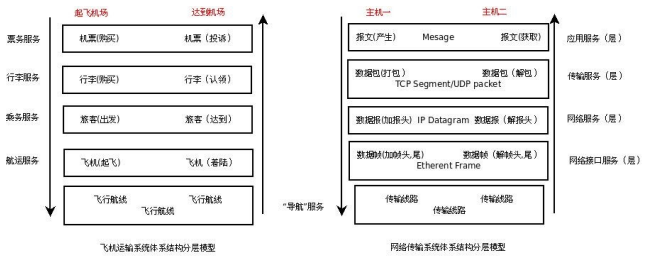

图 8.2: Network Layer与交通运输体系比较

对照上图，更容易理解右侧网络体系结构的分层原理（如果比照一封信发出到收到的这一中间过程可能更容易理解），上图右侧是TCP/IP网络体系结构的一个网络分层示意图，在把数据发送到网络之前，在各层中需要进行各种“打包”的操作，而从网络中接收到数据以后，就需要进行“解包”操作，最终把纯粹的数据信息给提取出来。这种分层的方式是为了传输数据的需要，也是两个主机之间如何建立连接以及如何保证数据传输的完整性和可靠性的需要。通过把各种需要分散在不同的层次，使得整个体系结构更加清晰和明了。这些“需求”具体通过各种对应的协议来规范，这些规范统成为网络协议。

关于OSI模型（7层）比照TCP/IP模型（4层）的协议栈可以从下图（来自网络）看个明了：

而下图（来自网络）则更清晰地体现了TCP/IP分层模型。

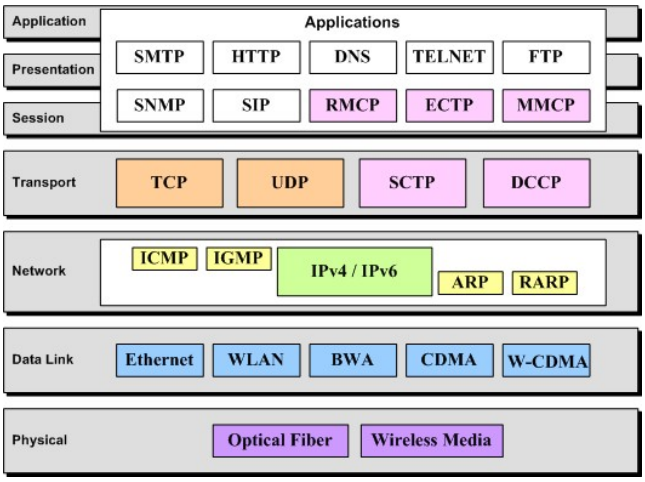

图 8.3: OSI Network Layer

上面介绍了网络原理方面的基本内容，如果想了解更多网络原理和操作系统对网络支持的实现，可以考虑阅读后面的参考资料。下面将做一些相关的实践，即在Linux下如何联网，如何用Linux搭建各种网络服务，并进行网络安全方面的考量以及基本的网络编程和开发的介绍。

8.3 Linux下网络“实战”

8.3.1 如何把我们的Linux主机接入网络

如果要让一个系统能够联网，首先当然是搭建好物理网络了。接入网络的物理方式还是蛮多的，比如直接用网线接入以太网，用无线网卡上网，用ADSL拨号上网……

对于用以太网网卡接入网络的常见方式，在搭建好物理网络并确保连接正常后，可以通过配置IP地址和默认网关来接入网络，这个可以通过手工配置和动态获取两种方式。

范例：通过dhclient获取IP地址 如果所在的局域网有DHCP服务，那么可以这么获取，N是设备名称，如果只有一块网卡，一般是0或者1。

```
$ dhclient ethN
```

范例：静态配置IP地址 当然，也可以考虑采用静态配置的方式，ip_address本地主机的IP地址，gw_ip_address是接入网络的网关的IP地址。

```
$ ifconfig eth0 ip_address on
$ route add deafult gw gw_ip_address
```

如果这个不工作，记得通过ifconfig/mii-tool/ethtool等工具检查网卡是否被驱动起来了，然后通过lspci/dmesg等检查网卡类型（或者通过主板手册和独立网卡自带的手册查看），接着安装或者编译相关驱动，最后把驱动通过insmod/modprobe等工具加载到内核中。

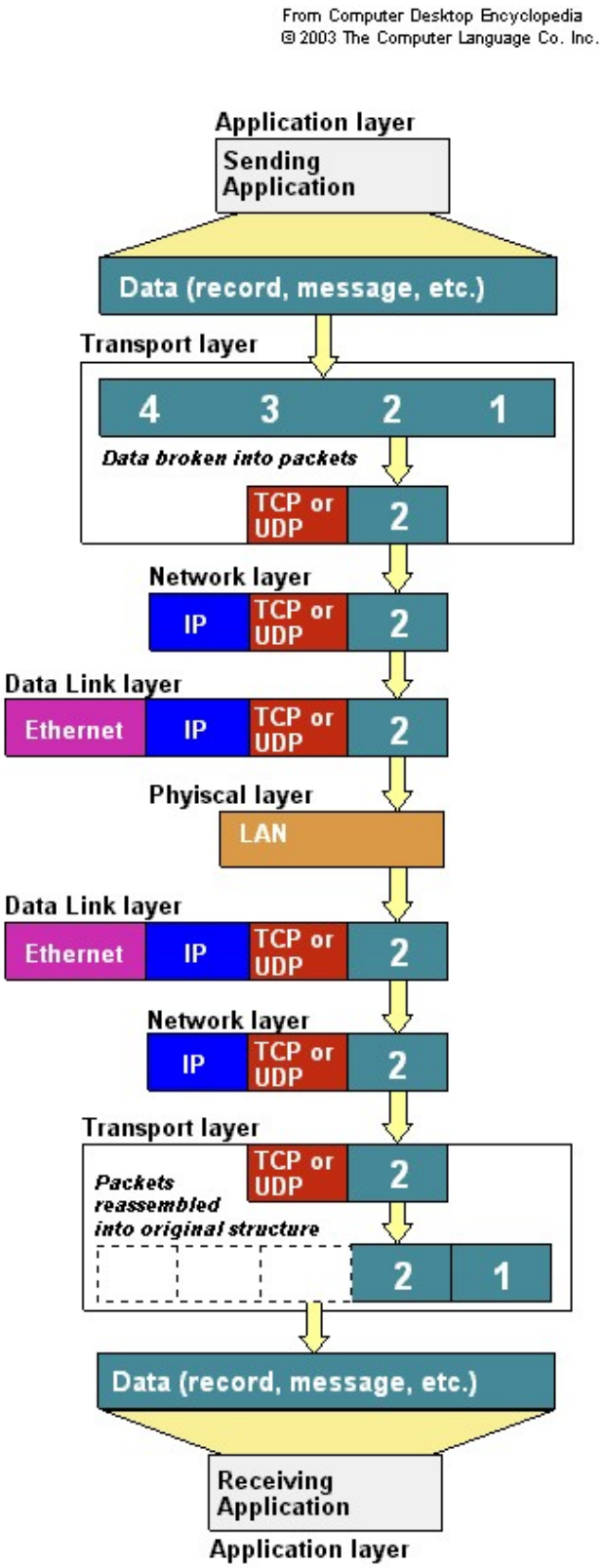

图 8.4: TCP IP Network Layer

8.3.2 用Linux搭建网桥

网桥工作在OSI模型的第二层，即数据链路层，它只需要知道目标主机的MAC地址就可以工作。Linux内核在2.2开始就已经支持了这个功能了，具体怎么配置看看后续相关参考资料吧。如果要把Linux主机配置成一个网桥，至少需要两个网卡。

网桥的作用相当于一根网线，用户无须关心里头有什么东西，把它的两个网口连接到两个主机上就可以让这两个主机支持相互通信。不过它比网线可厉害多了，如果在起上配置防火墙，就可以隔离连接在它两端的网段（注意这里是网络，因为它不识别IP），另外，如果这个网桥有多个网口，那么可以实现一个功能复杂的交换机了，而如果有效组合多个网桥，则有可能实现一个复杂的可现实现流量控制和负载平衡的防火墙系统了。

8.3.3 用Linux做路由

路由工作在OSI模型的第三层，即网络层，通过router可以配置Linux的路由，当然，Linux下也有很多工具支持动态路由的。相关的资料在网路中铺天盖地，由于时间关系，这里不做介绍，自己找找合适的去看吧。

8.3.4 用Linux搭建各种常规的网络服务

你需要什么服务呢？给你的局域网弄个DHCP服务器，那就弄个吧，看看参考资料；如果想弄个邮件发送服务器，那就弄个sendmail或者exim4吧，如果再想弄个邮件列表服务器呢，那就装个mailman，如果想弄个接收邮件的服务器呢，那就安装个pop3服务器吧，如果想弄个web站点，那就弄个apache服务器，如果想弄上防火墙服务，那么通过iptables工具配置netfilter就可以，what’s more？如果你能想到，Linux上应该都有相应的实现。

8.3.5 Linux下网络问题诊断与维护

如果出现网络问题，不要惊慌，逐步检查网络的各个层次：物理链接、链路层、网络层直到应用层，熟悉使用各种如下的工具，包括ethereal/tcpdump，hping，nmap，net-stat,netpipe,netperf,vnstat,ntop等等。

关于这些工具的详细用法和网络问题诊断和维护的相关知识，看看后续资料吧。

8.4 Linux下网络编程与开发

如果想自己做一些相关的网络编程开发呢，比如实现一个客户端/服务器架构的应用，那么就可以采用linux下的socket编程了，如果想写一个数据包抓获和协议分析的程序呢，那么可以采用libpap等函数库了，如果想实现某个协议呢，那就可以参考相关的RFC文档，并通过socket编程来实现了。

这个可以参考相关的Linux socket编程等资料。

8.5 后记

本来积累了很多相关的技巧，但是因为时间关系，暂且不详述了，更多细节你可能能够从本blog中搜索到，或者直接到网络中找找。

到这里，整个《shell编程范例序列》算是很粗略地完成了，你可以从这里访问到这份范例序列的列表。不过“范例”却缺少实例，特别是这一节。因此，如果有时间我会逐步补充一些实例，并在sf.net上维护一份范例列表的。

如果有任何疑问和建议，欢迎在后面回帖交流。

8.6 参考资料

* 计算机网络——自上而下的分析方法
* Linux 网络体系结构（清华大学出版社出版）
* Linux 系统故障诊断与排除 第13章 网络问题（人民邮电出版社）
* 在Linux下用ADSL拨号上网
* Linux下无线网络相关资料收集
* Linux网桥的实现分析与使用
* DHCP mini howto
* 最佳的75个安全工具
* 网络管理员必须掌握的知识
* Linux上检测rootkit的两种工具： Rootkit Hunter和Chkrootkit
* 数据包抓获与ip协议的简单分析（基于pcap库）
* RFC
* HTTP协议的C语言编程实现实例

第 9 章

总结

9.1 前言

到这里，整个shell编程序列就要结束了，作为总结篇，主要回顾一下各个小节的主要内容，并总结出shell编程的一些常用框架和相关注意事项等。

9.2 shell编程范例回顾

主要回顾各小节的内容

9.3 常用shell编程“框架”

通过总结和分析一些实例总结各种常见的问题的解决办法，比如如何保证同一时刻每个程序只有一个运行实体（进程）。

9.4 程序优化技巧

多思考，总会有更简洁和高效的方式。

9.5 其他注意事项

比如小心rm -rf的用法，如何查看系统帮助等。
（有待补充）

A

附录

A.1 Shell编程学习笔记

A.1.1 前言

这里只是个人学习笔记哦，主要包括Shell概述、Shell变量、位置参数、特殊符号、别名、各种控制语句、函数等Shell编程知识。

要是想系统的学shell，应该找一些比较系统的资料，例如：《Shell编程范例序列》和《鸟哥学习Shell Scripts》。

A.1.2 执行Shell脚本的方式

范例：输入重定向到bash

```
$ bash < ex1
```

可以读入ex1中的程序，并执行

范例：以脚本名作为参数

其一般形式是：

```
$ bash 脚本名 [参数]
```

例如：

```
$ bash ex2 /usr/meng /usr/zhang
```

其执行过程与上一种方式一样，但这种方式的好处是能在脚本名后面带有参数，从而将参数值传递给程序中的命令，使一个Shell脚本可以处理多种情况，就如同函数调用时可根据具体问题给定相应的实参。

范例：以 . 来执行

如果以目前Shell（以 · 表示）执行一个Shell脚本，则可以使用如下简便形式：

```
$ · ex3 [参数]
```

以Shell脚本作为Shell的命令行参数，这种方式可用来进行程序调试。

将Shell脚本的权限设置为可执行，然后在提示符下直接执行它。

具体办法：

```
$ chmod a+x ex4
$ ./ex4
```

A.1.3 Shell的执行原理

Shell 接收用户输入的命令（脚本名），并进行分析。如果文件被标记为可执行的，但不是被编译过的程序，Shell就认为它是一个Shell脚本。Shell将读 取其中的内容，并加以解释执行。所以,从用户的观点看，执行Shell脚本的方式与执行一般的可执行文件的方式相似。

因此，用户开发的Shell脚本可以驻留在命令搜索路径的目录之下（通常是“/bin”、“/usr/bin”等），像普通命令一样使用。这样，也就开发出自己的新命令。如果打算反复使用编好的Shell脚本，那么采用这种方式就比较方便。

A.1.4 变量赋值

可以将一个命令的执行结果赋值给变量。有两种形式的命令替换：一种是使用倒引号引用命令，其一般形式是： **命令表**。

例如：将当前工作目录的全路径名存放到变量dir中，输入以下命令行：

```
$ dir=`pwd`
```

另一种形式是：$(命令表)。上面的命令行也可以改写为：

```
$ dir=$(pwd)
```

A.1.5 数组

bash只提供一维数组，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。用户可以使用赋值语句对数组变量赋值。

范例：对数组元素赋值

对数组元素赋值的一般形式是：数组名[下标]＝值，例如：

```
$ city[0]=Beijing
$ city[1]=Shanghai
$ city[2]=Tianjin
```

也可以用declare命令显式声明一个数组，一般形式是：

```
$ declare -a 数组名
```

范例：访问某个数组元素

读取数组元素值的一般格式是： ${数组名[下标]} ，例如：

```
$ echo ${city[0]}
Beijing
```

范例：数组组合赋值

一个数组的各个元素可以利用上述方式一个元素一个元素地赋值，也可以组合赋值。定义一个数组并为其赋初值的一般形式是：

数组名=(值1 值2 … 值n)

其中，各个值之间以空格分开。

例如：

```
$ A=(this is an example of shell script)
$ echo ${A[0]} ${A[2]} ${A[3]} ${A[6]}
this an example script
$ echo ${A[8]}
```

由于值表中初值共有7个，所以A的元素个数也是7。A[8]超出了已赋值的数组A的范围，就认为它是一个新元素，由于预先没有赋值，所以它的值是空串。

若没有给出数组元素的下标，则数组名表示下标为0的数组元素，如city就等价于city[0]。

范例：列出数组中所有内容

使用*或@做下标，则会以数组中所有元素取代。

```
$ echo ${A[*]}
this is an example of shell script
```

范例：获取数组元素个数

```
$ echo ${#A[*]}
7
```

A.1.6 参数传递

假如要编写一个shell来求两个数的和，可以怎么实现呢？ 为了介绍参数传递的使用，我们用vim编写一个这样的脚本：

```
$ cat > add
let sum=$1+$2
echo $sum
```

保存后，我们执行一下：

```
$ chmod a+x ./add
$ ./add 5 10
15
```

可以看出5和10分别传给了$1和$2，其实这是shell自己设定的参数顺序，我们可以先定义好变量，然后传递进去。

例如，修改上面的add文件得到：

```
let sum=$X+$Y
echo $sum
```

现在我们，这样执行：

```
$ X=5 Y=10 ./add
15
```

我们同样可以得到正确结果哦。

A.1.7 设置环境变量

export一个环境变量：

```
$ export opid=True
```

这样子就可以拉，如果要登陆以后都生效，可以直接添加到/etc/profile或者~/.bashrc里头。

A.1.8 键盘读起变量值

我们可以通过read来读取变量值，下面我们来等待用户输入一个值并且显示出来

```
$ read -p "请输入一个值 : " input ; echo "你输入了一个值为 :" $input
请输入一个值 : 21500
你输入了一个值为 : 21500
```

A.1.9 设置变量的只读属性

有些重要的shell变量，赋值后不应该修改，那么我们可设置他为readonly:

```
$ oracle_home=/usr/oracle7/bin
$ readonly oracle_home
```

A.1.10 条件测试命令test

语法: test 表达式 如果表达式为真，则返回真，否则，返回假

范例：数值比较

先给出数值比较时常见的比较符:

-eg =; -ne !=; -gt >; -ge >=; -lt <; -le <=

```
$ test var1 -gt var2
```

A.1.11 范例：测试文件属性

文件的可读、可写、可执行，是否为普通文件，是否为目录分别对应:

-r -w -x -f -d

```
$ test -r filename
```

范例：字符传属性以及比较

串的长度为零: -z; 非零:-n，如:

```
$ test -z s1
```

如果串s1长度为零，返回真

范例：串比较

相等“s1”=“s2”； 不相等 “s1”!=“s2”

我们还有一种比较串的方法(可以按字典序来比较哦)：

```
$ if [[ 'abcde' <'abcdf' ]]; then  echo "yeah,果然是诶"; fi
yeah,果然是诶
```

A.1.12 整数算术或关系运算expr

可用该命令进行的运算有：

算术运算：+ - * / %；逻辑运算：= ! < <= > >=

如：

```
$ i=5;expr $i+5
```

另外，bc是linux下的一个计算器，可以进行一些算术计算

A.1.13 控制执行流程命令

范例：条件分支命令if

if命令举例：

下面的shell作用：判断输入的第一个参数的值是一个普通文件名，那么分页打印该文件；否则，判断它是否为目录名，若是则进入该目录并打印该目录下的所有文件，如果也不是目录名，那么提示相关信息。

```
if test -f $1
then
    pr $1>/dev/lp0
elif
    test-d $1
then
    (cd $1;pr *>/dev/lp0)
else
    echo $1 is neither a file nor a directory
fi
```

范例：case命令举例

case命令是一个基于模式匹配的多路分之命令，下面的shell将根据用户键盘输入情况决定下一步将执行那一组命令。

```
while  [ $reply!="y" ]  && [ $reply!="Y" ]                    #下面将学习的循环语句
do
    echo "\nAre you want to continue?(Y/N)\c"
    read reply             #读取键盘
    case $replay in
        (y|Y) break;;         #退出循环
        (n|N) echo "\n\nTerminating\n"
              exit 0;;
           *) echo "\n\nPlease answer y or n"
           continue;       #直接返回内层循环开始出继续
    esac
done
```

范例：循环语句while, until

语法:

```
while/until 命令表1
do
    命令表2
done
```

区别是，前者在执行完命令表1后，如果出口状态为零，那么执行do后面的命令表2，然后回到起始处,而后者执行命令表1后，如果出口状态非零，才执行类似操作。 例子同上。

范例：有限循环命令for

语法:

```
for 变量名 in 字符串表
do
    命令表
done
```

举例:

```
FILE="test1.c myfile1.f pccn.h"
for i in $FILE
do
    cd ./tmp
    cp $i $i.old
    echo "$i copied"
done
```

A.1.14 函数

先在我们来看看shell里头的函数怎么用

先看个例子:我们写一个函数，然后调用它显示“Hello,World!”

```
$ cat > show
# 函数定义
function show
{
    echo $1$2;
}
H="Hello,"
W="World!"
show $H $W
```

演示:

```
$ chmod 770 show
$./show
Hello,World!
```

呵呵，看出什么蹊跷了吗?

$ show $H $W

我们可以通过直接在函数名后面跟实参哦

实参顺序对应“虚参”的$1,$2,$3…….

注意一个地方哦，假如要传进去一个参数，这个参数中间带空格，怎么办呢? 你先试试看。

我们来显示“Hello World”(两个单词之间有个空格哦)

```
function show
{
    echo $1
}
HW="Hello World"
show "$HW"
```

看看，是不是ok拉，如果直接show $HW，肯定是不行的，因为$1只接受到了Hello，所以结果之显示Hello，原因是字符串变量必须用”包含起来。

A.1.15 后记

感兴趣的话继续学习哦! 还有好多强大的东西值得你去学习呢，呵呵，比如cut，expr，sed，awk等等。